Tutorial - time series and exp smoothing

Tutorial - Time Series Models With Exponential Smoothing

1. What are time series?

a. Definitions

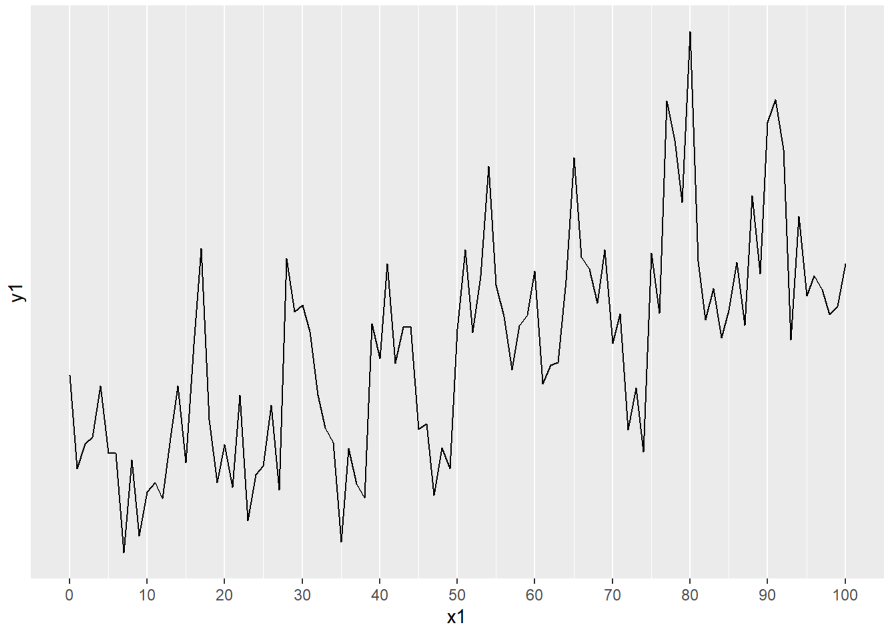
- Time Series - a time series is a dataset that captures the same variables at different intervals. data taken at a snapshot in time or where time data is not included is referred to as cross-sectional. Typically time series have a specified interval at which data is collected and data are usually modeled as time series in cases where the time of measurement can be hypothesized to have an impact on the level of the outcome variable.
- Autocorrelation - when datapoints are consistently correlated with other points from the same series at a set interval, we refer to them as autocorrelated. For example, the temperature at 9AM is highly correlated with the temperature at 8AM on the same day. It is also correlated (less strongly) with the temperature at 9AM the previous day. We will see how to identify the specific time correlation in the next section.
- Trend - trend refers to the movement of data values to either higher or lower values over a medium to long-term time horizon. In contrast to cross-sectional regressions, trend in time series data typically refers specifically to the movement of the outcome variable over time. High autocorrelation with the period $t-1$ might indicate strong trend components.
- Seasonality - this refers to predictable and repeated cycles and patterns in the outcome variable over time. An example might be monthly rainfall, where more rain typically falls in the winter and spring and less rain falls in summer. There can also be daily seasonality, for instance road traffic which has a daily cycle corresponding to rush hour, and may have weekly seasonality corresponding to weekdays vs weekend days.
- Time lag - when discussing time series, it is common to refer to “lags”. These simply refer to prior time periods; lag 1 refers to the datapoint at time $t-1$. Lag 10 would refer to the datapoint from 10 time periods ago at $t-10$.

b. How to identify autocorrelation in time series

i. ACF plots

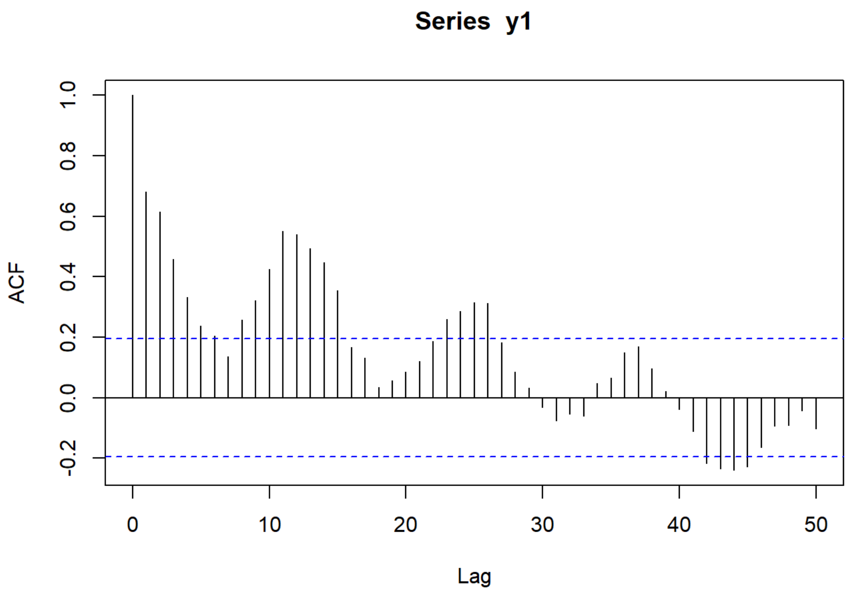
Perhaps the simplest way to identify autocorrelation is to look at a line chart with time on the x-axis. However, picking out cycles in the data can be challenging, especially when there is a lot of noise in the data. To be more scientific, we can use an ACF (Autocorrelation Function) plot. Let’s construct some data with trend and seasonality:

```r
#-- Settings
sim_x1 <- function(n) seq(0,n,1)
sim_y1 <- function(x, sd){               # generate Y|X from N{f(x),sd}
  n = length(x)
  f <- function(x) 50 + .1* x+ 3*sin(x/2)   # true mean function
  f(x) + rnorm(n, sd=sd)
}

#-- Generate data
set.seed(825)                           # set seed for reproducibility
n = 100                              # number of observations
sd = 2                             # stdev for error

x1= sim_x1(n)                            # get x values
y1 = sim_y1(x1, sd=sd)                     # get y values

ggplot(tibble(x1,y1), aes(x1,y1)) +
  geom_line() +
  scale_x_continuous(breaks=seq(0, n, by=n/10)) +
  scale_y_continuous(breaks=seq(-6, 50, by=n/10))
```



The plot shows the correlation of the response variable y to itself at different time lags. Of course, at time $t = t$ the datapoint $y(t)$ is fully correlated with itself. At time $t - 1$, the ACF will show the correlation between all values $y(t)$ and $y(t-1)$. If the data is independent with respect to time, there will be little correlation between these two series, but if $y(t)$ can be in part predicted by the level of $y(t-1)$ then the ACF will show a spike. The ACF plot repeats this analysis to n periods where $y(t)$ is correlated with $y(t-n)$. A lag is considered significant if it rises above a 95% confidence level, shown as a dotted line on the chart. Note that negative correlation is also valid and useful.

```
acf(y1, lag.max = 50)
```



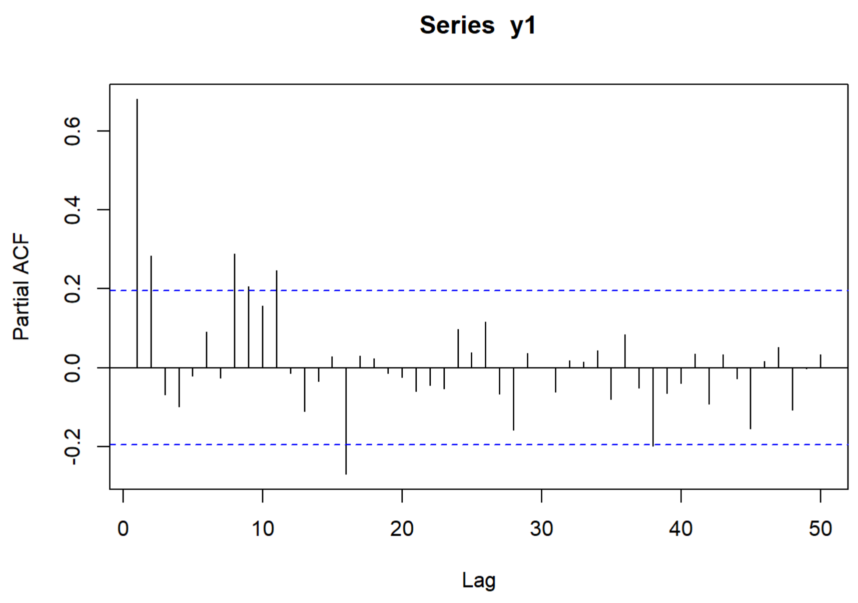
Notice how we see many significant lags in this plot. Should we model all of these? Probably not; estimating parameters out to 43+ lags would likely be a bad idea considering we only have 100 datapoints. How can we narrow this down?

ii. PACF plots

A useful extension of the ACF is the PACF or Partial Autocorrelation Function. The problem with the ACF is that if a value is correlated with itself ($y(t) \approx y(t-1)$) then logically it will also be correlated with $y(t-2)$ since $y(t-2)$ is correlated with $y(t-1)$ which is in turn correlated with $y(t)$. This leads to long chains of significant lags which could be replicating the same explanatory factor of a single time lag. As each lag we add into any model will increase the number of parameters, it is important to select the fewest number of lags to generate the time series to avoid overfit.

When examining $y(t-2)$ PACF predicts the parts of $y(t)$ NOT predicted by $y(t-1)$. By removing the seasonal factors already explainable by earlier lags, we can see more pronounced spikes when looking at lags with specific impact on $y(t)$ in the longer time horizon. For example, in the following PACF, we see high correlation at lag 1, then at lag 2, and again at lags 8 and 9. Considering the data was generated with a sine function, we would expect this behavior, and perhaps we would model the data using time lags of 1, 2, 8, and 16. There is random noise in the data, so we will still have to experiment to find the best model.

```
pacf(y1, lag.max = 50)
```



## 2. Simple Exponential Smoothing

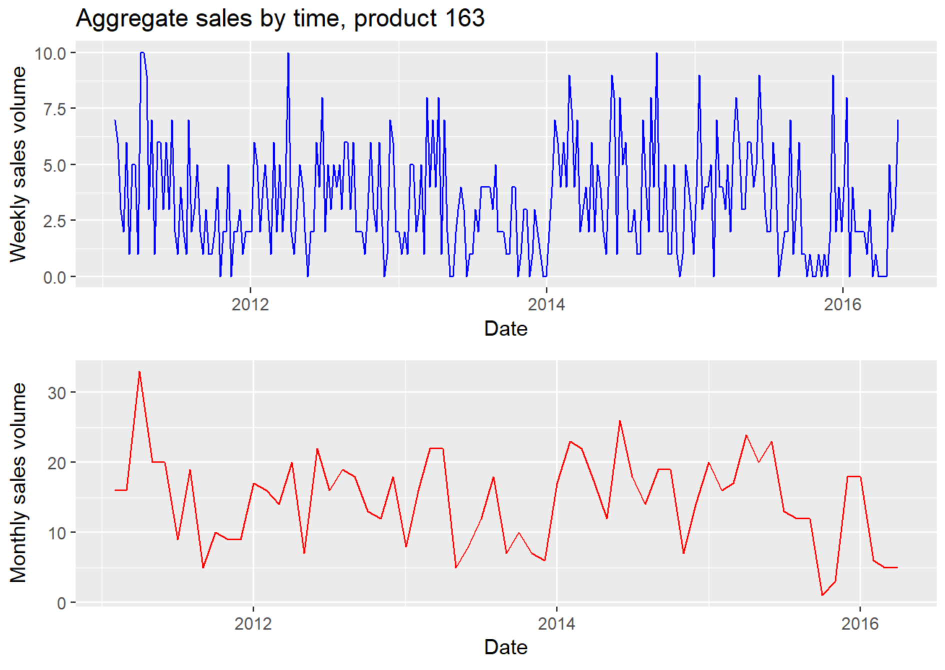
a. Simple moving average

Possibly the simplest method of forecasting is a moving average. As it sounds, a moving average just takes the average of the set of points preceding the desired prediction point. For the forecast F_t, simple moving average (SMA) is formulated as follows:

$$F_t = \frac{Y_{t-1} + Y_{t-2} + \ldots + Y_{t-k}}{k}$$

Where… F_t is the desired forecast for time period t Y_t is the observed demand for time t k is the tuning parameter specifying the number of periods to average over

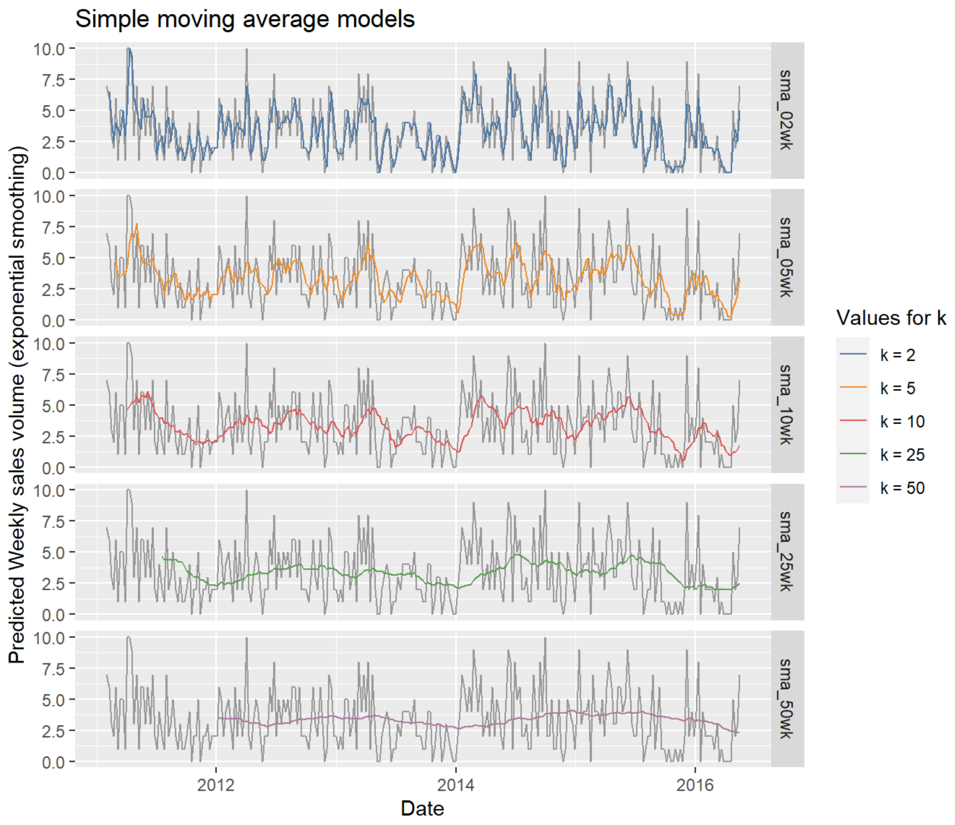
A larger value of k will mean a smoother curve, incorporating more past data, while a smaller value for k will result in a more responsive model. Let's see how the model looks on some data taken from the Kaggle M5 Walmart dataset. This data is the sales of a particular product in a single store. We can aggregate to the weekly or monthly level as shown.



Using the weekly data, lets build some simple moving average models with different values of k.

```
## Warning: package 'zoo' was built under R version 4.1.3
```

```
## Warning: Removed 4 row(s) containing missing values (geom_path).
## Warning: Removed 107 row(s) containing missing values (geom_path).
```



Notice that the level of smoothing applied to the forecast line changes as the value of k changes. A larger value for k will create a smoother, less responsive demand forecast that will not be as affected by spikes in demand. However, with a very large value for k, we potentially lose too much responsiveness, meaning the model will not react to a general increase in demand.

The case where $k = 1$ is a special case where we simply take the demand for the previous period as our prediction for the next period's demand: this is called the Naive forecast. This may be appropriate, but it is unlikely to be ideal for most products.

Additionally, the future forecast is not dynamic. Once the moving average reaches the end of the data, it will not change for future predictions, no matter how far ahead.

b. Simple exponential smoothing (SES)

SMA assumes that the previous k periods all have equal predictive power when used to predict demand for our next period. Logically for any k value (say $k = 5$), the most recent demand figures likely have more relevance to future demand than demand numbers from 5 weeks ago. We can correct for this using a different class of models called exponential smoothing models. Essentially, they act in a similar way to SMA models, but place higher weight on more recent information about demand.

We will first see how the exponential smoothing calculation is is made and then return to show why it is a weighted average of old data Let's assume that we somehow made a forecast at the end of the previous period $t-1$ for the current period t. That is, we calculated L_{t-1} and used that as $F_{t-1,t}$. Now, we want to apply exponential smoothing to make the forecast now (at the end of period t) for the next period. To calculate my new estimate of the level with exponential smoothing, I place some amount of weight (10% for example) on the most recent sales observation A_t and the remaining weight (90%) on the previous forecast. Let α represent the weight that I place on the most recent sales observation We will call α the

$$L_t = \alpha A_t + (1-\alpha)L_{t-1}.$$

Since there is no systematic variation, the forecast for a future period is equal to my current estimate of the level, $F_{t,t+k} = L_t$.

Notice that a higher level of alpha results in less smoothing as we reduce the impact of older data. The case where $\alpha = 1$ is equivalent to the special case of SMA where $k = 1$ and the forecast is assumed to be the same as demand in period $t-1$.

Recall that we defined the forecast error above as $E_t^k = A_t - F_{t-k,t}$. With exponential smoothing, our lag-1 forecast $F_{t-1,t}$ is given by L_{t-1}, so the lag-1 error is $E_t^1 = A_t - L_{t-1}$. Using this, we can rewrite the exponential smoothing forecast calculation:

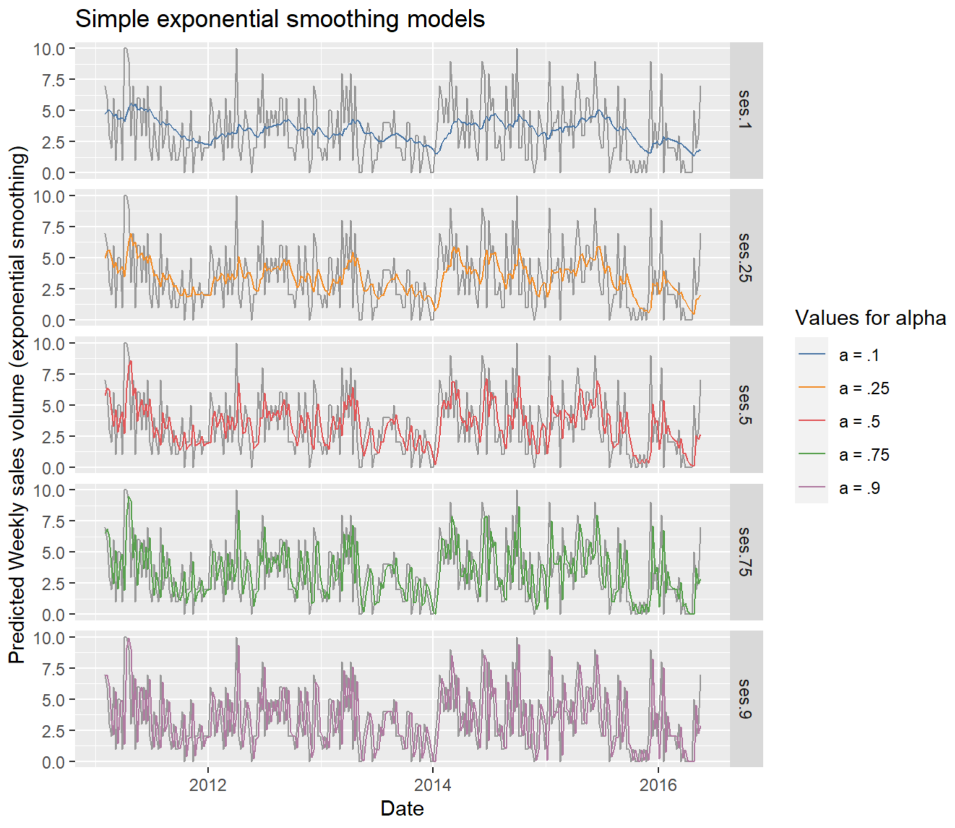
$$\begin{aligned} L_t &= \alpha A_t + (1-\alpha)L_{t-1} \\ L_t &= L_{t-1} + \alpha(A_t - L_{t-1}) \\ L_t &= L_{t-1} + \alpha E_t^1. \end{aligned}$$

Viewed this way, we see that the exponential smoothing calculation adjusts the old forecast in the direction of the most recent error.If our previous forecast was too high, the error will be negative,and the exponential smoothing calculation will lower the old forecast to produce the new forecast. The smoothing constantα dictates how strongly we react to this most recent error observation. We can rewrite the exponential smoothing calculation to see that the exponential smoothing forecast is just a weighted average of past data. To do this, we write the equation for L_t and then substitute in the equation for L_{t-1} and so on.

$$\begin{aligned}
L_t &= \alpha A_t + (1-\alpha)L_{t-1} \\
L_t &= \alpha A_t + (1-\alpha)(\alpha A_{t-1} + (1-\alpha)L_{t-2}) \\
L_t &= \alpha A_t + \alpha(1-\alpha)A_{t-1} + (1-\alpha)^2(\alpha A_{t-2} + (1-\alpha)L_{t-3}) \\
L_t &= \alpha A_t + \alpha(1-\alpha)A_{t-1} + \alpha(1-\alpha)^2 A_{t-2} + \dots \\
L_t &= \sum_{n=0}^{\infty} \alpha(1-\alpha)^n A_{t-n}.
\end{aligned}$$

So, we place weight of α on the most recent observation, weight of $\alpha(1-\alpha)$ on the second most recent, weight of $\alpha(1-\alpha)^2$ on the third most recent and so on. Now, most forecasters do not have an infinite number of old observations (at least not yet), so this process must be initialized somehow. A common approach is to take a simple average of the first few data points to initialize the process.

Let's see how exponential smoothing looks for different levels of α:



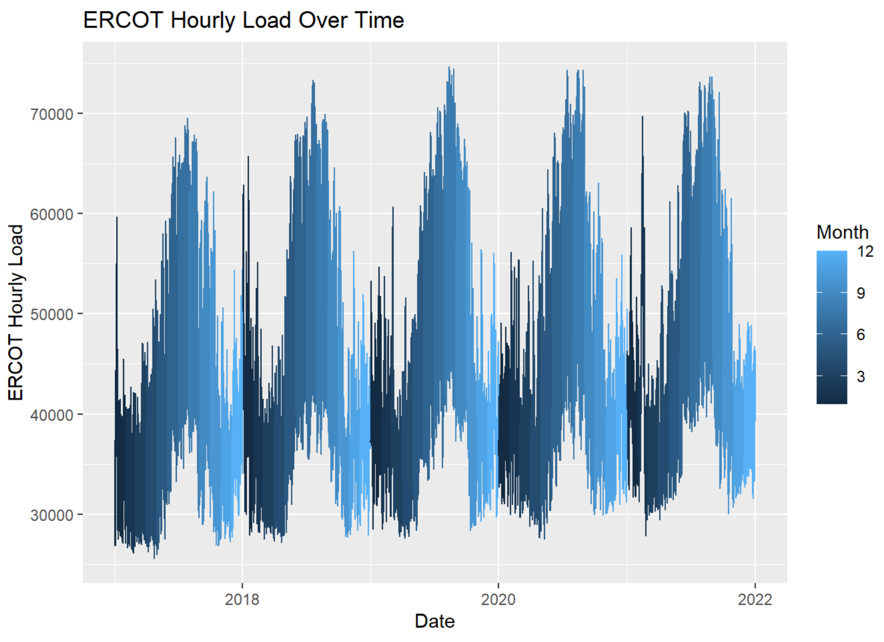
3. Deseasonalization

Although simple exponential smoothing provides a basic framework for accounting for the influence of data points stretching further into the past, often times when dealing with time series data we will also need to be able to appropriately handle instances of seasonality. Since seasonality assumes that there are patterns of fluctuation with fixed length and predictable repetition, there are a number of ways to try and address it and account for its impacts through our modeling. More advanced and adaptive methods will be discussed in the following section, but one of the most straightforward and easily implemented approaches is known as seasonal adjustment or deseasonalization.

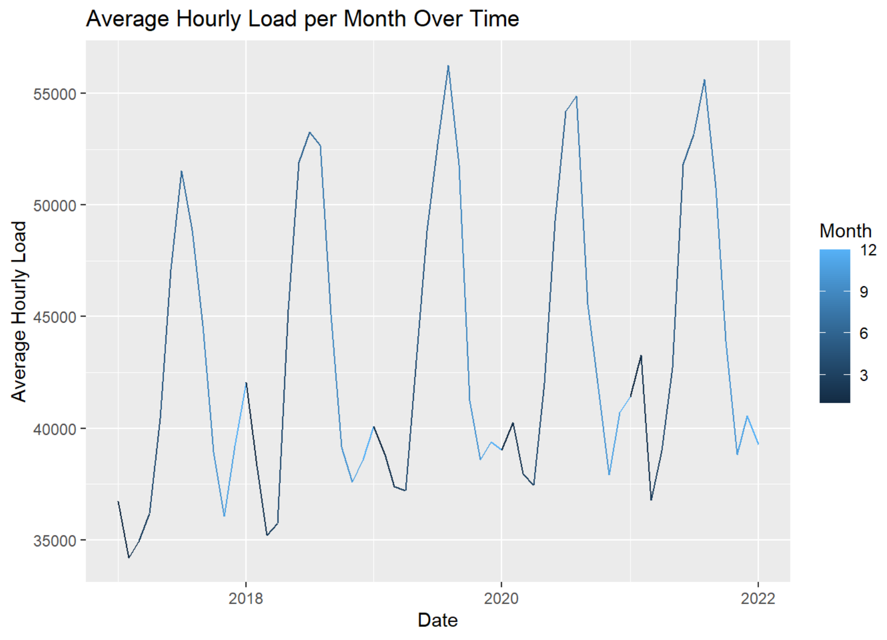
The basic premise behind this method is that, once we have identified a seasonal trend through either inspection of the ACF/PACF plots or by directly observing the lineplot of the time series data, we can attempt to normalize and standardize the data by removing the increase or decrease in the response variable induced by the seasonal component. This process involves determining a multiplicative seasonal factor that will scale the values within that season relative to an expected, baseline value, and, to provide a grounded example of this issue and illustrate how the deseasonalization solution can be implemented in code, we will offer a demonstration using the data Hourly Load data which can be publicly accessed here: https://www.data.com/gridinfo/load/load_hist (https://www.data.com/gridinfo/load/load_hist). This data tracks the hourly electrical energy consumption in Texas and can clearly be seen to contain both a 24 hour seasonality and an annual seasonality.

Once the data has been cleaned, we can start by visualizing the issue at hand using a simple lineplot.

```r
# plot the energy consumption over time
ggplot(ercot, aes(x = HourEnding, y = ERCOT)) +
  geom_line(aes(color=Month)) +
  labs(title = 'ERCOT Hourly Load Over Time', x = 'Date', y = 'ERCOT Hourly Load')
```



Since the significant daily fluctuations generate quite a lot of noise in the data when we try to analyze it over a longer timeframe, we can aggregate the hourly data such that we treat each month as a single data point. This can be achieved by simply taking the average energy consumption across every hour in the month as the one representative value for that month. Performing this transformation results in the following plot.



From the plot above, it can be readily observed that there are indeed strong seasonal trends which seem to fluctuate roughly with the four yearly seasons as can be noted by the prominent peaks during the summer months and relative troughs during the fall and winter periods. At a more granular level, there are also potentially meaningful daily trends, but, before we can address this issue of layered or stacked seasonality, let us first try to address these more macroscopic seasonal trends.

In order to capture the seasonal fluctuations we can treat each month as its own "season" and use these as the basis upon which we will perform the deseasonalization. In this setting, we now have that all 12 months taken together form a single cycle.

To tease out the relative impact each month has on the energy consumption within it, we will need to first calculate the centered yearly moving average at each data point to determine what baseline value we would expect if there was no seasonal influence, and then we can divide the actual values by these expected values and average these derived ratios over the months to uncover what multiplicative factor each month has on its energy values.

To consider this process one component at a time, let us begin by computing the centered yearly moving average. Since our data is collected hourly, this makes it so that each value in the stream of centered moving averages should be computed by summing up all of the data points that lie within a half year radius about point of interest and then dividing this value by the total number of data points that occurred within that year long span. If the number of periods in our seasonal pattern were odd, this would make is so that we could simply center our summation window over the point of interest and calculate a single average to get the centered moving average at that location. Unfortunately, in our case, our period is 8760 entries long which means the center of this interval would be at position $\frac{1+8760}{2} = 4380.5$.

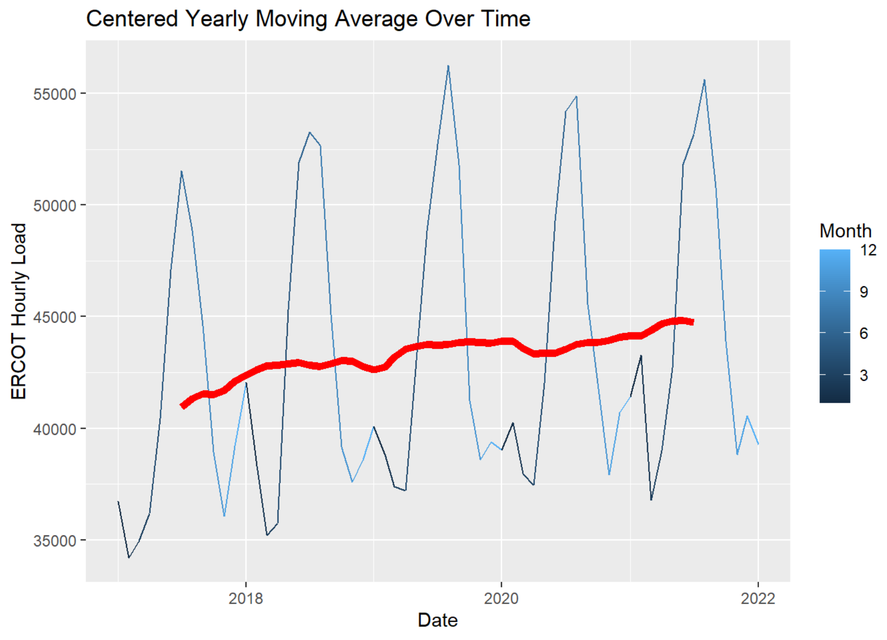
To remedy this issue and make it so that integer positions can be the center points of the moving average, we will need to take the mean of two averages rather than just a single average; explicitly, we will take one yearly moving average of the interval containing the point of interest at index 4380 and another of the interval containing the point of interest at index 4381, and, by finding the mean of these two yearly moving averages, we will get a good approximation of what the centered yearly moving average should be at the desired point.

Since we need a roughly half year buffer on either side of the first and last entries we compute the centered yearly moving averages at, this makes it so that we will not get baseline, expected values for the the entire domain of our data, but we should still have enough values per season to calculate reasonable seasonal factors.

With the baseline averaging scheme established, we can now implement that process and visualize the results.

```
MovingAvg <- rep(NA, length(MonthlyData$MonthAvg))
for (i in ((12/2)+1):(length(MonthlyData$MonthAvg)-(12/2))) {
  MovingAvg[i] <- (sum(MonthlyData$MonthAvg[(i-(12/2)):(i+(12/2)-1)])/(12) + sum(MonthlyData$MonthAvg[(i-(12/2)+1):(i+(12/
2))])/(12))/2
}
MonthlyData$MovingAvg <- MovingAvg
# plot the energy consumption over time
ggplot(data = MonthlyData) +
  geom_line(aes(x = Date, y = MonthAvg, color = Month)) +
  geom_line(aes(x = Date, y = MovingAvg), color = 'red', size = 2) +
  labs(title = 'Centered Yearly Moving Average Over Time', x = 'Date', y = 'ERCOT Hourly Load')
```

```
## Warning: Removed 12 row(s) containing missing values (geom_path).
```

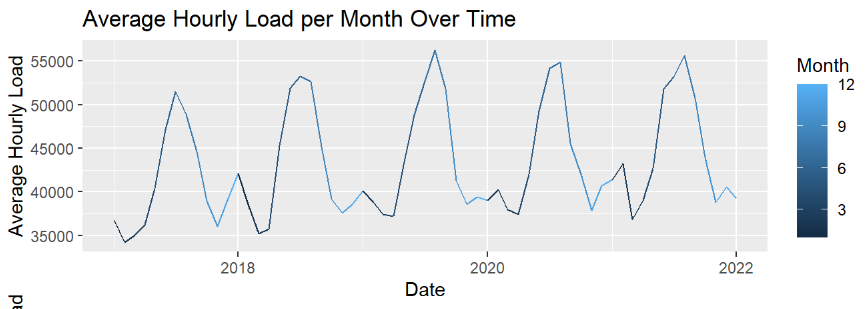


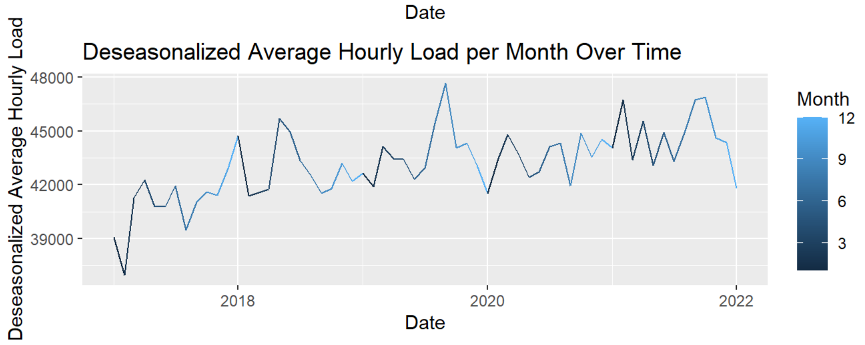
As can be seen from the plot above, the centered yearly moving average has successfully smoothed out the seasonality effects within the data and provides us with a basis upon which we will be able to determine how each month influences the energy consumption within it. This can be achieved by dividing every datapoint by the expected, smoothed value and then grouping by and averaging these ratios over each month.

```
MonthlySeasonalFactors <- MonthlyData %>% group_by(Month) %>% summarise(SeasonalFactor = mean(MonthAvg/MovingAvg, na.rm=TRU
E))
MonthlyData <- MonthlyData %>% group_by(Month) %>% mutate(SeasonalFactor = mean(MonthAvg/MovingAvg, na.rm=TRUE))
MonthlySeasonalFactors
```

```
## # A tibble: 12 x 2
##    Month SeasonalFactor
##    <dbl>          <dbl>
##  1     1          0.940
##  2     2          0.926
##  3     3          0.847
##  4     4          0.856
##  5     5          0.991
##  6     6          1.15
##  7     7          1.23
##  8     8          1.24
##  9     9          1.09
## 10    10          0.937
## 11    11          0.870
## 12    12          0.914
```

So for month 1 (Jan), the energy consumption is typically 93% of the long term average energy consumption. For month 7 (July) energy consumption is 123.7% of the long run average. With these seasonal factors computed, we are able to now deseasonalize the data by dividing the original energy consumption values by their respective seasonal factors.





Based on the plot above, we can see that the blatant monthly seasonality has been removed from the data, and we are now left with a set of time series data that can be more appropriately modeled using the previously discussed simple exponential smoothing techniques. To show this method of forecasting in action, we can use the ERCOT data from 2022 as a reference point against which we can compare the results of our predictive model.

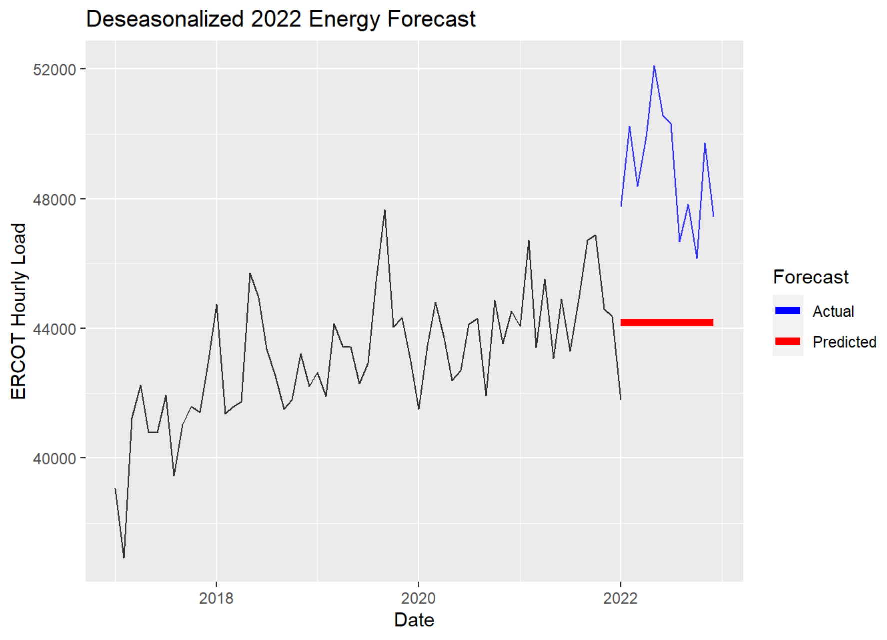
We can load and clean this new validation data in the same manner as we did earlier, primarily making sure that the date-time data is in the correct format.

```
# aggregate the data by month
futuremonthly <- future %>% group_by(Year = year(HourEnding), Month) %>% summarise(MonthAvg = mean(ERCOT), .groups = 'rowwis
e')
# convert the monthly dates into POSIXct format
Dates <- rep(NA, length(futuremonthly$Year))
for (i in 1:length(futuremonthly$Year)) {
  if (futuremonthly$Month[i] < 10) {
    Dates[i] <- paste(toString(futuremonthly$Year[i]), toString(futuremonthly$Month[i]), sep='-0')
  } else {
    Dates[i] <- paste(toString(futuremonthly$Year[i]), toString(futuremonthly$Month[i]), sep='-')
  }
}
futuremonthly$Date <- ym(Dates)
```

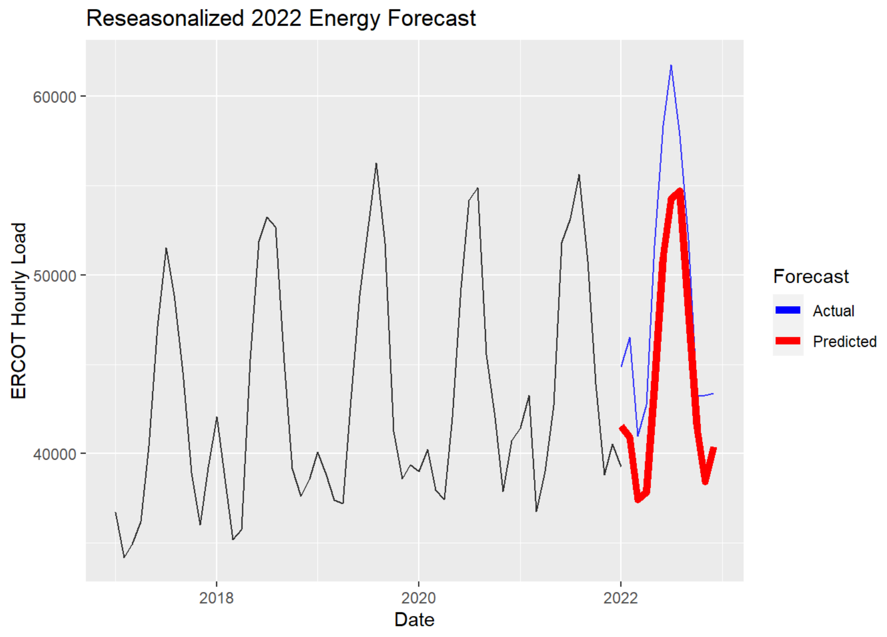
Now that the data has been configured properly, we are able to use the ses() function from the forecast package to derive the future forecast estimates for the deseasonalized data over the the entire domain of the 2022 dataset. We can then plot the results to see how this approach has performed.

```
fcast_model <- ses(MonthlyData$Deseasonalized, h = nrow(futuremonthly))
fcast <- as.data.frame(fcast_model)
futuremonthly$DeseasonalizedForecast <- fcast$`Point Forecast`
futuremonthly <- inner_join(futuremonthly, MonthlySeasonalFactors, by = 'Month')
futuremonthly <- futuremonthly %>% mutate(Deseasonalized = MonthAvg/SeasonalFactor)
futuremonthly <- futuremonthly %>% mutate(Forecast = DeseasonalizedForecast*SeasonalFactor)
```

```
ggplot() +
  geom_line(aes(x = MonthlyData$Date, y = MonthlyData$Deseasonalized), alpha=0.8) +
  geom_line(aes(x = futuremonthly$Date, y = futuremonthly$Deseasonalized, color = 'Actual'), alpha=0.8) +
  geom_line(aes(x = futuremonthly$Date, y = futuremonthly$DeseasonalizedForecast, color = 'Predicted'), size = 2) +
  labs(title = 'Deseasonalized 2022 Energy Forecast', x = 'Date', y = 'ERCOT Hourly Load', color = 'Forecast') +
  scale_color_manual(values = c('Actual' = 'blue', 'Predicted' = 'red'))
```



While the predicted forecast for the deseasonalized 2022 ERCOT load data does not appear to match the actual values that well and tends to be consistently underestimating the true energy volumes, this is primarily due to the fact that the energy consumption at the end of 2021 was lower than normal, thus setting the point forecast for all future dates to be below where they normally might be. To truly appreciate the benefits of the deseasonalization method and see how well it does at handling the issue of seasonality in our forecasting models, we must now reseasonalize the data by multiplying all of the deseasonalized values and predicted forecasts by their respective seasonal scaling factors. This transformation back into the original frame of reference unveils how powerful this technique has actually been considering its simplicity. However, this does not seem to be an ideal result since the exponential smoothing forecast was flat and the seasonal factors are predetermined. We will discuss in the next section how to make all these factors dynamic.



The plot depicted above clearly shows how these seasonal factors produced through deseasonalization have allowed us to take into consideration the seasonal variation and influence on energy consumption and has made it so that our predicted forecast is now able to adhere far more closely to the actual values we observe in the world. While this method of deseasonalization has helped us take solid strides towards accounting for more factors that play a role in time series forecasting, there still a few crucial omissions in features it can handle: most notably, although this process has alleviated some of the seasonality concerns, it has done nothing to address the possible linear trends that are underlying the evolution of the data, and is also fails to manage the seasonal fluctuations in a fully adaptive way. In order to model time-series data more precisely and accurately with all of its potentially nuanced characteristics and interactions, we will need to rely on more sophisticated and advanced methods such as the Holt-Winters method of exponential smoothing.

References: Seasonality in Time Series Forecasting article https://r4ds.had.co.nz/dates-and-times.html (https://r4ds.had.co.nz/dates-and-times.html) https://www.pluralsight.com/guides/time-series-forecasting-using-r (https://www.pluralsight.com/guides/time-series-forecasting-using-r) ## 4. Holt-Winters models

a. Dealing with trend

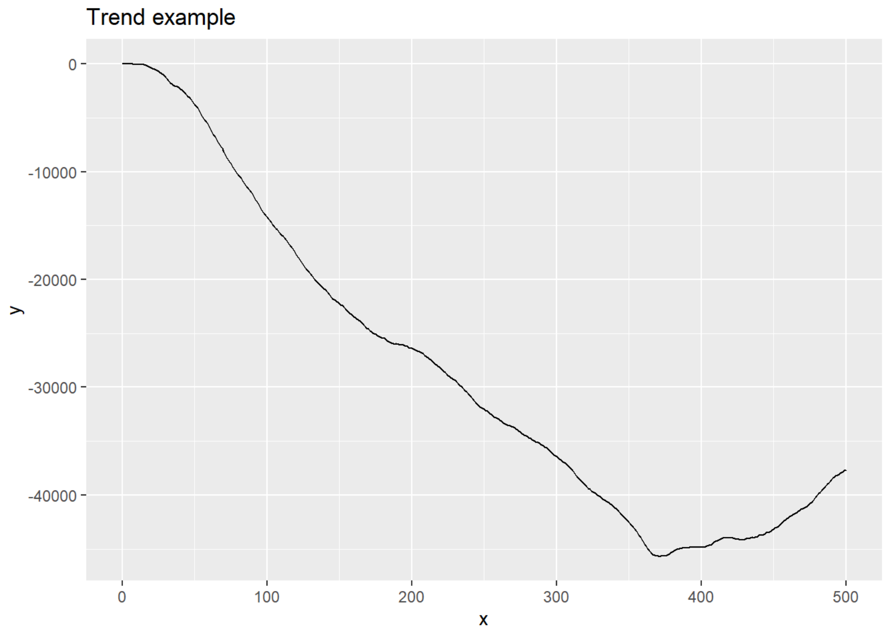
In 1957, Charles Holt built on the existing simple exponential smoothing model by designing a set of equations to handle trended data:

$$\text{Forecast: } \hat{y}_{t+h|t} = l_t + b_t * h$$
$$\text{Current Level: } l_t = \alpha_t * y_t + (1-\alpha) * (l_{t-1} + b_{t-1})$$
$$\text{Current Slope: } b_t = \beta * (l_t - l_{t-1}) + (1-\beta) * b_{t-1}$$

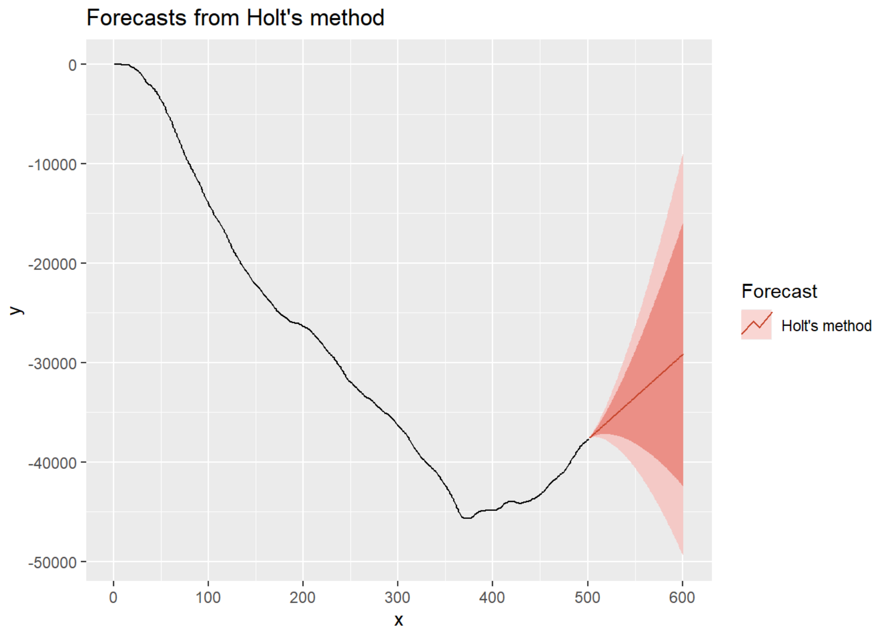
The new addition to Holt's formulation is β, the slope or trend smoothing parameter. Similar to simple exponential smoothing, the current level is a weighted average of two terms. However, Holt's major addition was to have the second term by a simple addition of the previous level and the previous slope. The slope estimate is also a weighted average of the change in level AND the previous slope. The resulting forecast equation then combines the level and the slope multiplied by h. Forecasts are therefore a linear function of h, sometimes referred to as the step, and the model now leverages trend.

i. Simple trend example

First we will construct some data with trend. Note that the slope is constructed using a random walk and changes over time. It is not a fixed value. This concept helps us to handle issues like sales forecasting, where the slope will not be a constant function of time, but may slowly change from positive to negative and back again. Holt-Winters will both capture this trend and, if tuned correctly, an estimate on the rate of change in the slope.



```
y_time_trend = ts(y_trend)
holt_model = holt(y_time_trend, h=100, damped = FALSE)
autoplot(y_time_trend) +
  autolayer(holt_model, series="Holt's method") +
  ggtitle("Forecasts from Holt's method") + xlab("x") +
  ylab("y") +
  guides(colour=guide_legend(title="Forecast"))
```



Now we can take a weighted estimate of past slopes in order to estimate the trend at a given point in time. The trend estimate is essentially looking at a change in level between t and $t-1$ and an estimate of trend at t. The estimated parameter β^* is the trend smoothing parameter in the same way that α was the weigting parameter for the level in SES.

```
holt_model[["model"]]
```

```
## Holt's method
##
## Call:
##  holt(y = y_time_trend, h = 100, damped = FALSE)
##
##   Smoothing parameters:
##     alpha = 0.911
##     beta  = 0.5973
##
##   Initial states:
##     l = 0.1804
##     b = 8.1599
##
##   sigma:  29.7511
##
##      AIC     AICc      BIC
## 6520.155 6520.276 6541.238
```

b. Damped Trend

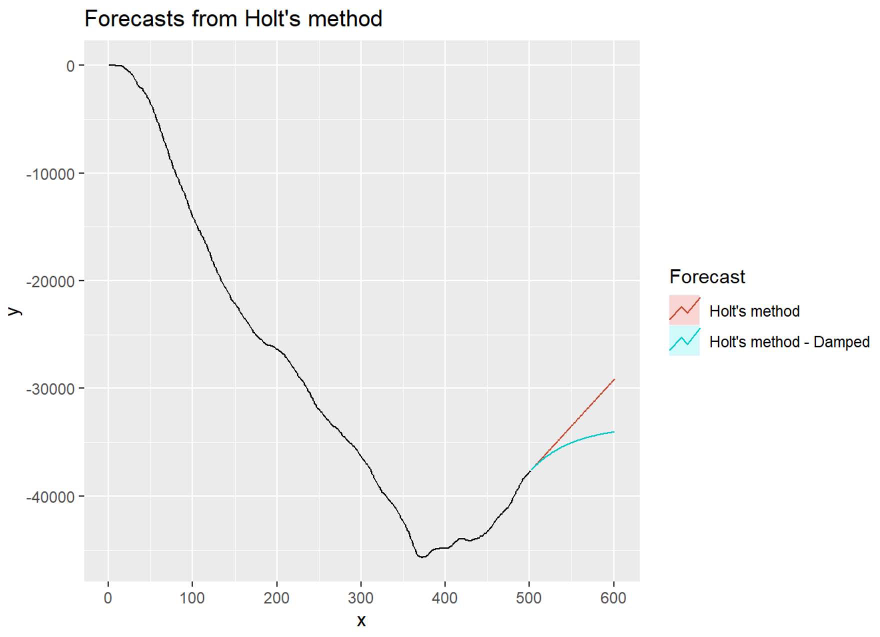
While linear trend is simple to implement, its practicality is of some concern. In fact, linear trend models have a tendency to overpredict the slope into the future. This motivated Gardner and McKenzie to develop their own method to better match observed phenomena. Garden and McKenzie intuited that decreasing trend of a forecast period may curtail many of the overestimates seen in practice. The idea is simple, if not based in pure mathematics: many real world trends will be strong for a time, then weaken. Using empirical evidence they proposed a damped trend model defined as follows:

$$\begin{aligned}\text{Forecast: } \hat{y}_{t+h|t} &= l_t + b_t * (\phi + \phi^2 + \phi^3 \cdots + \phi^h) \\ \text{Current Level: } l_t &= \alpha * y_t + (1-\alpha) * (l_{t-1} + \phi * b_{t-1}) \\ \text{Current Slope: } b_t &= \beta * (l_t - l_{t-1}) + (1-\beta) * \phi * b_{t-1}\end{aligned}$$

If ϕ = 1, Gardner and McKenzie's method reduces to Holt's model. However, as ϕ decreases the trend falls off more quickly. In fact, it can be show that as h, the forecast period, moves to ∞ the forecast becomes constant. Since low ϕ values can have significant effects on small valued data, ϕ is generally restricted to be between .8 and .98 (to allow discernible difference between an undamped model).

The below example fits a "damped Holt model" to the same data as generated above and compares the damped and undamped predictions:

```
y_time_trend = ts(y_trend)
holt_model_damped = holt(y_time_trend, h=100, damped = TRUE)
autoplot(y_time_trend) +
  autolayer(holt_model, series="Holt's method", PI = FALSE) +
  autolayer(holt_model_damped, series="Holt's method - Damped", PI = FALSE) +
  ggtitle("Forecasts from Holt's method") + xlab("x") +
  ylab("y") +
  guides(colour=guide_legend(title="Forecast"))
```



```
holt_model_damped[["model"]]
```

```
## Damped Holt's method
##
## Call:
##  holt(y = y_time_trend, h = 100, damped = TRUE)
##
##   Smoothing parameters:
##     alpha = 0.8851
##     beta  = 0.6298
##     phi   = 0.98
##
##   Initial states:
##     l = 28.7072
##     b = -7.5948
##
##   sigma:  29.7961
##
##      AIC     AICc      BIC
## 6522.662 6522.832 6547.962
```

```
holt_model[["model"]]$mse
```

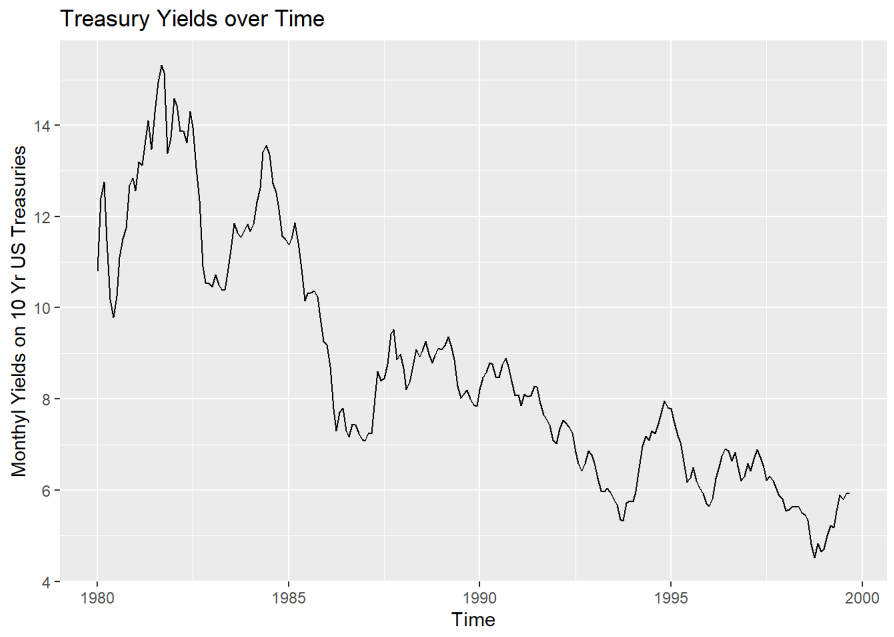
```
## [1] 878.0607
```

```
holt_model_damped[["model"]]$mse
```

```
## [1] 878.9497
```

In this circumstance, the MSE from the undamped model actually is slightly better than the damped model. This makes sense since the data was generated from an undamped, high stylized example. What happens on real data though?

The below example fits the damped and undamped Holt models to US Treasury Yield data



```
holt_treas_un = holt(tcm10y_1980, h = 50, damped = FALSE)
holt_treas_damped = holt(tcm10y_1980, h = 50, damped = TRUE)
holt_treas_un[["model"]]$mse
```

```
## [1] 0.1537669
```

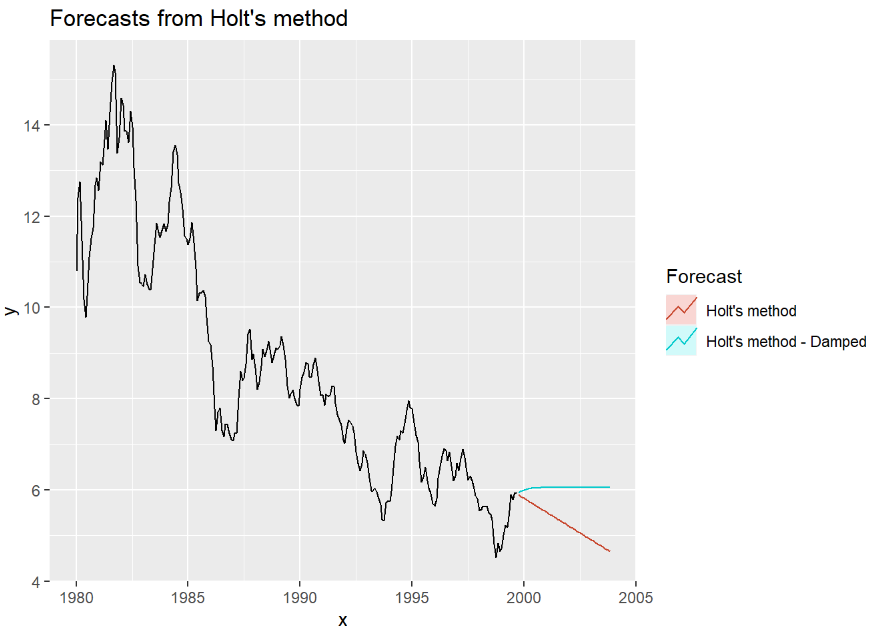
```
holt_treas_damped[["model"]]$mse
```

```
## [1] 0.1491149
```

```
holt_treas_damped[["model"]]
```

```
## Damped Holt's method
## 
## Call:
##  holt(y = tcm10y_1980, h = 50, damped = TRUE) 
## 
##   Smoothing parameters:
##     alpha = 0.9999 
##     beta  = 0.1319 
##     phi   = 0.8 
## 
##   Initial states:
##     l = 11.56 
##     b = -0.1557 
## 
##   sigma:  0.3903
## 
##      AIC     AICc      BIC 
## 856.9103 857.2755 877.7186
```

In this situation, the damped estimate produces a lower MSE. The ϕ value in this fit is .8 indicating a very strong damping effect. This can be seen more clearly in the below plot:



4.2 Winters Method

a. Dealing with seasonality

To this point, seasonality has not been captured any exponential smoothing models. In 1960, Peter Winters extended Holt's model to incorporate yet another important element of time series - the seasonal component. Winters' additions to the equation was a new term γ representing the seasonal smoothing coefficient and s_t the seasonal component of the forecast. To begin, here is Winters' (additive - more on this later) method accounting for seasonality with no inclusion of trend:

$$\begin{aligned}\text{Forecast: } \hat{y}_{t+h|t} &= l_t + s_{t+h-m(k+1)} \\ \text{Current Level: } l_t &= \alpha * (y_t - s_{t-m}) + (1-\alpha) * l_{t-1} \\ \text{Current Seasonal Component: } s_t &= \gamma * (y_t - l_{t-1}) + (1-\gamma) * s_{t-m}\end{aligned}$$

where k = integer of $(h-1)/m$ and m denotes the number of seasons in a year. This equation is defining the current forecast as a function of the current level and some additional seasonal component.

b. Additive vs. Multiplicative

As mentioned, the above form shows the additive version of the equation, but there is also a multiplicative form. The additive form is preferred when the seasonal component of the forecast is simply changing the forecast by a constant value. For example, temperature typically increases by ~20 degrees in the summer when compared to spring. A muliplicative form, as shown below, is more advantageous when the forecast is changing proportional to its level. For example, holiday spending is often proportional to the amount of disposable income families have. In other words, families spend a percentage of their earnings, not a fixed dollar amount. The multiplicative seasonal form can be seen below:

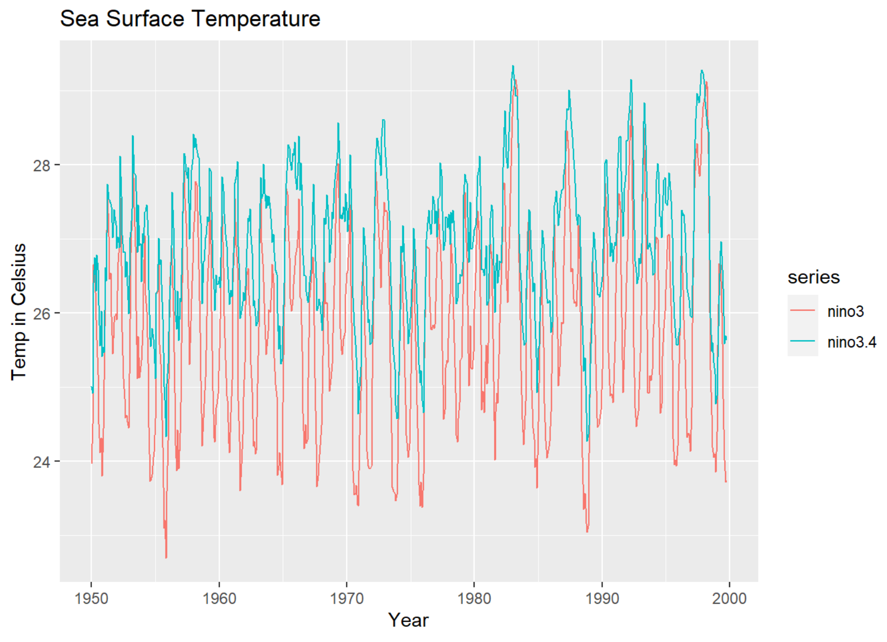
$$\begin{aligned}\text{Forecast: } \hat{y}_{t+h|t} &= l_t * s_{t+h-m(k+1)} \\ \text{Current Level: } l_t &= \alpha * (y_t / s_{t-m}) + (1-\alpha) * l_{t-1} \\ \text{Current Seasonal Component: } s_t &= \gamma * (y_t / l_{t-1}) + (1-\gamma) * s_{t-m}\end{aligned}$$

where k = integer of $(h-1)/m$ and m denotes the number of seasons in a year.

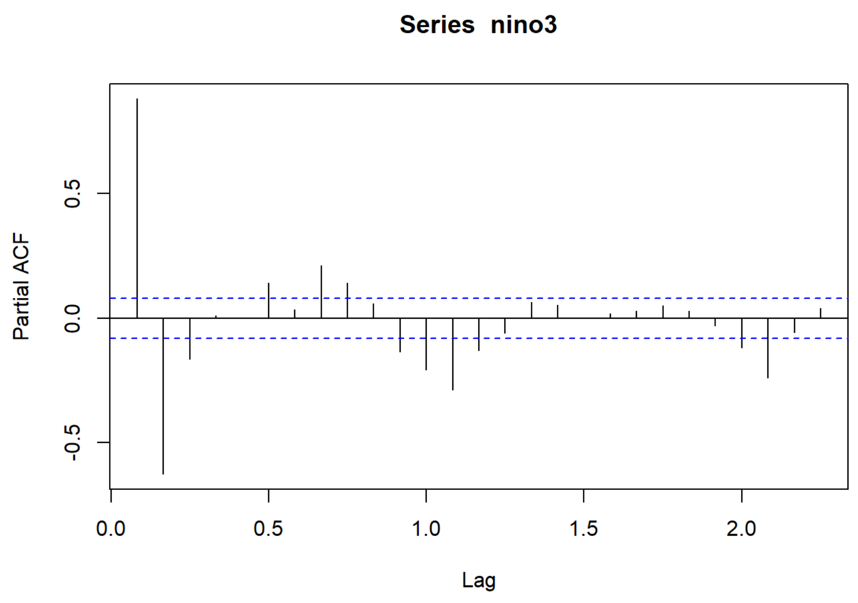
Let's see how this can be used in practice.

The following data set contains montly sea surface temperatures from the Nino 3.4 region of the Pacific Ocean as gathered by NOAA (https://www.cpc.ncep.noaa.gov/data/indices/ (https://www.cpc.ncep.noaa.gov/data/indices/)).

```
data(nino)
autoplot(nino)+
  ggtitle("Sea Surface Temperature") + xlab("Year") +
  ylab("Temp in Celsius")
```

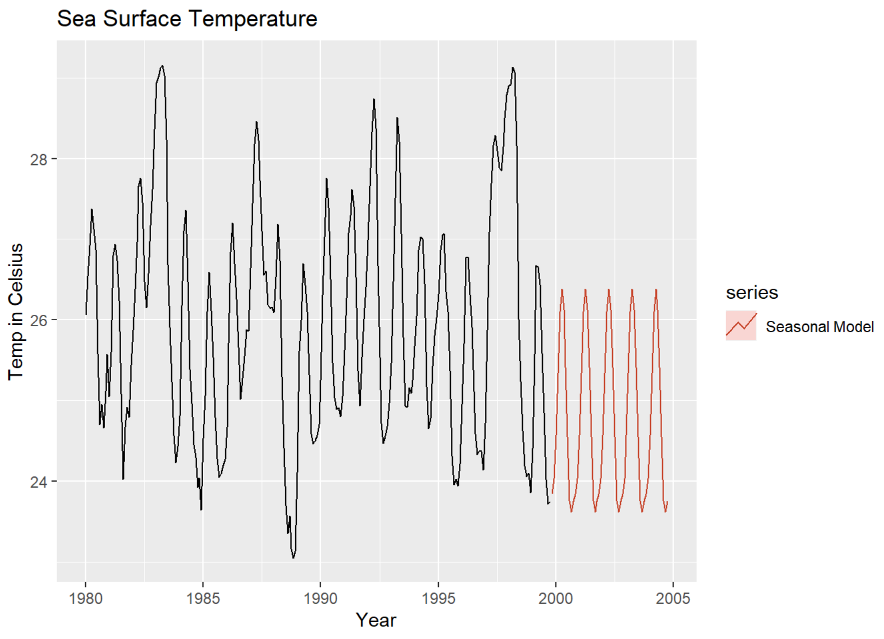


```
pacf(nino3)
```



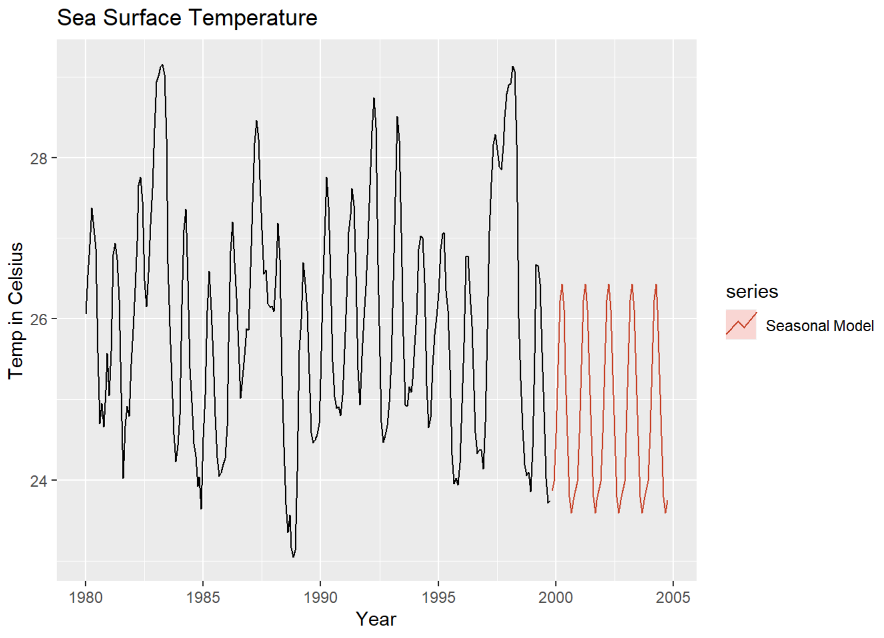
It is clear that this data shows strong seasonality, as would be expected with temperature data over the course of a year. Let's see how a simple additive seasonal model fits the data. We will forecast 5 years into the future to show the corresponding plot. The forecast function around the ets function from the tseries library in R allows direct specification of the type of model to be fit. For this scenario, "ANA" indicates additive:

```
temp_1980 = window(nino3, start = 1980)
temp_model = forecast(ets(temp_1980, model ="ANA"), h= 60)
autoplot(temp_1980) +
  autolayer(temp_model, series="Seasonal Model", PI = FALSE) +
  ggtitle("Sea Surface Temperature") + xlab("Year") +
  ylab("Temp in Celsius")
```



This looks pretty good considering the randomness of the data, but how does a multiplicative model fare?

```
temp_1980 = window(nino3, start = 1980)
temp_model = forecast(ets(temp_1980, model ="MNM"), h= 60)
autoplot(temp_1980) +
  autolayer(temp_model, series="Seasonal Model", PI = FALSE) +
  ggtitle("Sea Surface Temperature") + xlab("Year") +
  ylab("Temp in Celsius")
```



Not particularly different. This makes sense since the temperature data we would expect to be best approximated from an additive model, but the variability is small over the data set. As scale increases, additive vs. multiplicative may become important.

4.3 Full Holt-Winters

With the seasonal component accounted for, the next logical piece is to combine trend and seasonality. Holt and Winters worked together to arrive at both additive and multiplicative models that fully account for trend and seasonality:

a. Additive Holt-Winters

```
\begin{align}
\text{Forecast: } \hat{y}_{t+h|t} = l_t + h*b_t + s_{t+h-m(k+1)}\\
\text{Current Level: } l_t = \alpha*(y_t -s_{t-m})+ (1-\alpha)*(l_{t-1}+b_{t-1})\\
\text{Current Slope: } b_t = \beta*(l_t-l_{t-1})+ (1-\beta)*b_{t-1}\\
\text{Current Seasonal Component: } s_t = \gamma*(y_t-l_{t-1}-b_{t-1})+ (1-\gamma)*s_{t-m}\\
\end{align}
```

where h = forecast period, k = integer of $(h-1)/m$ and m denotes the number of seasons in a year.

b. Multiplicative Holt-Winters

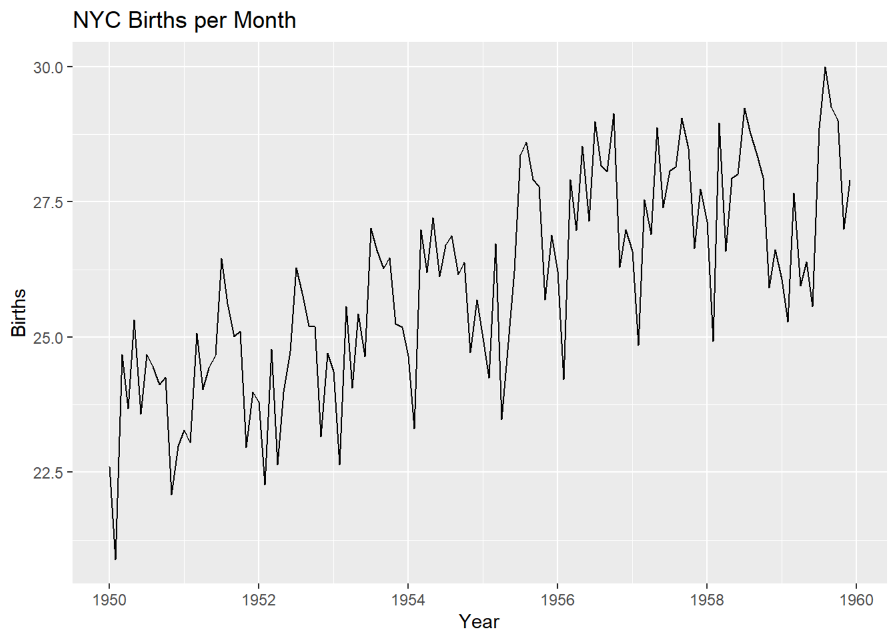
```
\begin{align}
\text{Forecast: } \hat{y}_{t+h|t} = (l_t + h*b_t) * s_{t+h-m(k+1)}\\
\text{Current Level: } l_t = \alpha*(y_t/s_{t-m})+ (1-\alpha)*(l_{t-1}+b_{t-1})\\
\text{Current Slope: } b_t = \beta*(l_t-l_{t-1})+ (1-\beta)*b_{t-1}\\
\text{Current Seasonal Component: } s_t = \gamma*(y_t/(l_{t-1}-b_{t-1}))+ (1-\gamma)*s_{t-m}\\
\end{align}
```

where h = forecast period, k = integer of $(h-1)/m$ and m denotes the number of seasons in a year.

Both the additive and multiplicative models can even be extended to include ϕ, the damping constant, to create damped additive and damped multiplicative Holt-Winters.

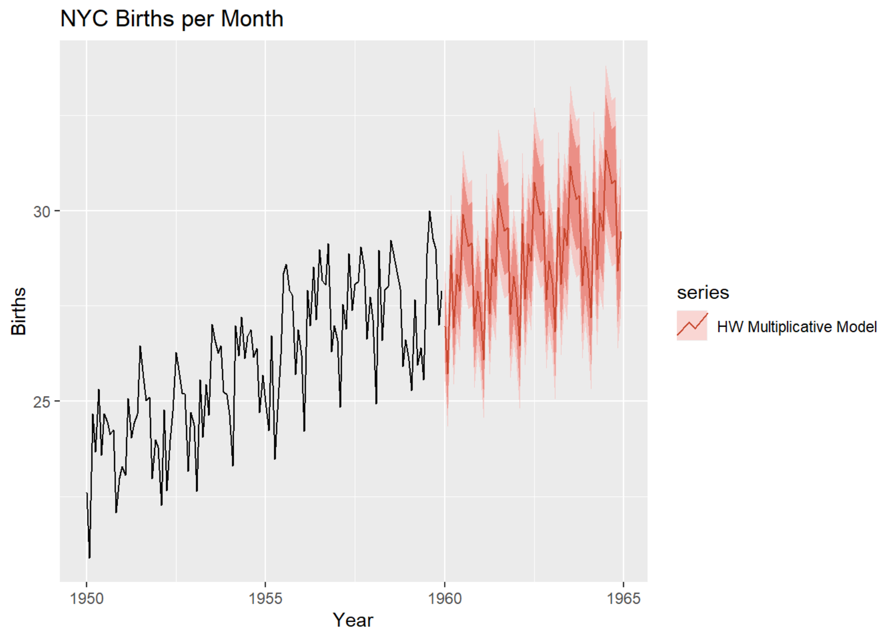
Let's take a look at medical data from NYC on the number of births per month (http://robjhyndman.com/tsdldata/data/nybirths.dat (http://robjhyndman.com/tsdldata/data/nybirths.dat)):

```
births = scan("http://robjhyndman.com/tsdldata/data/nybirths.dat")
birth_ts <- ts(births, frequency=12, start=c(1946,1))
birth_ts = window(birth_ts, start = 1950)
autoplot(birth_ts) +
  ggtitle("NYC Births per Month") + xlab("Year") +
  ylab("Births")
```

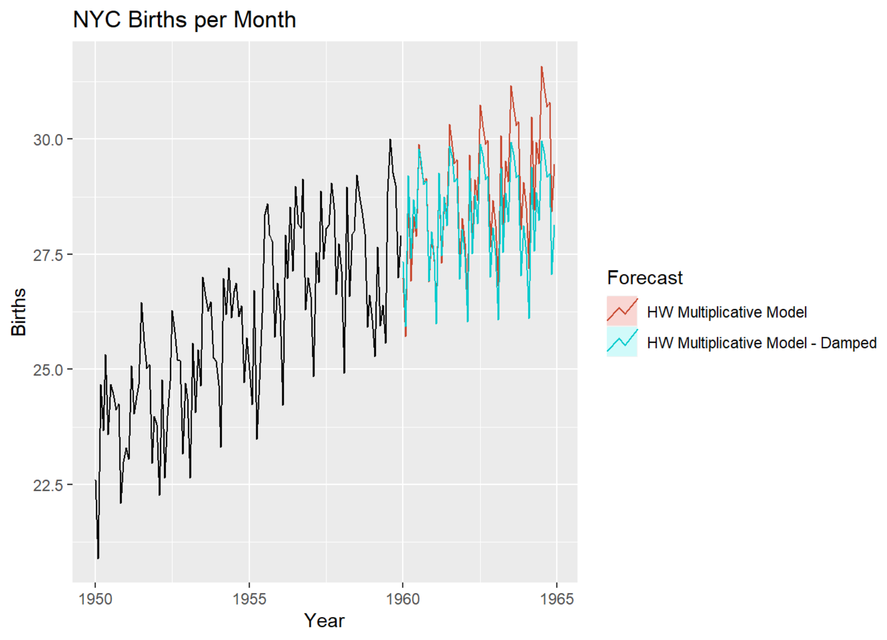


Clearly, this data has seasonality and trend. Let's see how a Holt-Winter's multiplicative model fits the data. Note, the use of {} conveniently removes the forecast wrapper on the ets function and is preferable in most circumstances. Unless a specific type of model is required, {} is the method of choice.

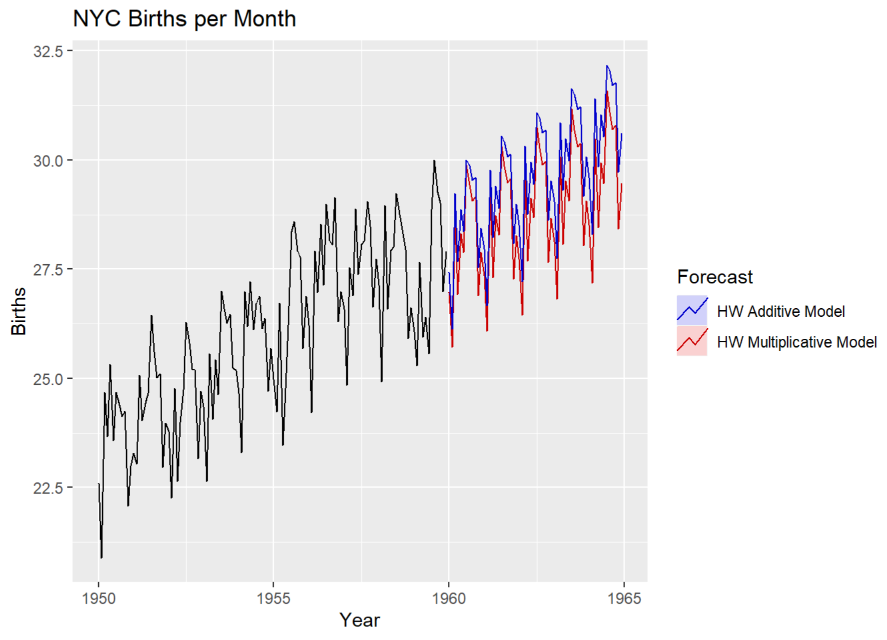
```
births_model = hw(birth_ts, h=60, seasonal = "multiplicative")
autoplot(birth_ts) +
  autolayer(births_model, series="HW Multiplicative Model", PI = TRUE) +
  ggtitle("NYC Births per Month") + xlab("Year") +
  ylab("Births")
```



As seen, this method generates a much better forecast than could be accomplished with only trend or seasonality. The below plots variations on the multiplicative Holt-Winters model for comparison:



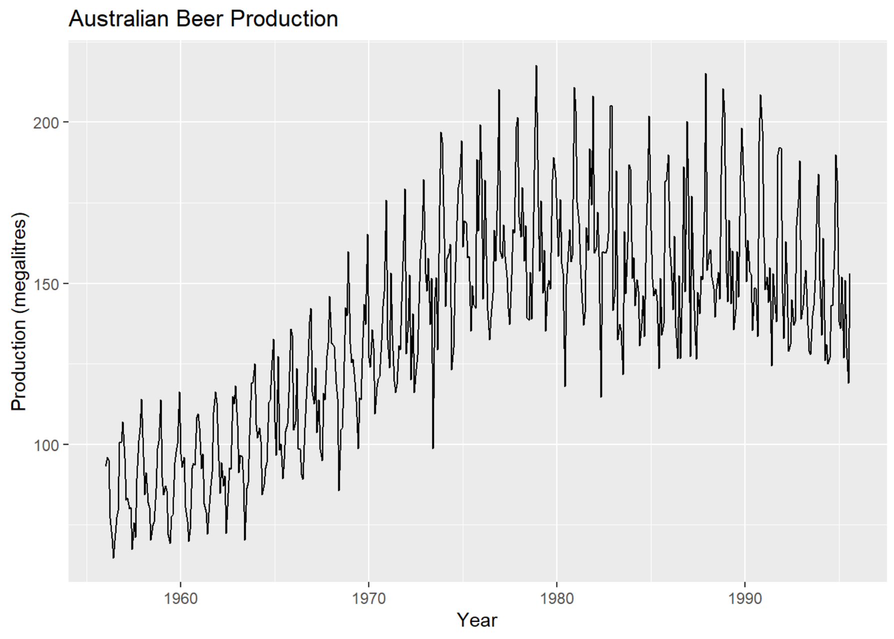
Note how the damped model increasingly estimates lower in comparison to the regular model as expected.



c. Model Tuning

To this point, Holt and Holt-Winters models have been fit using all of the training data and the one-step ahead forecast error. Specifically, R is choosing the model parameters that minimize MSE in the training set. However, in practice, holdout data should be used for hyperparameter tuning. With a full, damped Holt-Winters model ϕ - the damping parameter, γ - the seasonal parameter, β - the trend parameter, and α - the smoothing parameter all need to be estimated. The most straightforward (and recommended) strategy is a grid search over values of those parameters. Keep in mind that ϕ should be between .8 and 1. Below we will use hyperparameter tuning to build a model for Austrailian Beer Production (https://www.kaggle.com/datasets/shenba/time-series-datasets?select=monthly-beer-production-in-austr.csv (https://www.kaggle.com/datasets/shenba/time-series-datasets?select=monthly-beer-production-in-austr.csv))

```
beer = read.csv("monthly-beer-production-in-austr.csv")
beer_ts = ts(beer$Monthly.beer.production, frequency=12, start=c(1956,1))
autoplot(beer_ts)+
ggtitle("Australian Beer Production") +
  xlab("Year")+
  ylab("Production (megalitres)")
```



Now, we will hold out data after 1990:

```
beer_ts_train = window(beer_ts, end = c(1989,12))
beer_ts_test = window(beer_ts, start = 1990)
```

And set up a grid of values over which to fit multiple models (note this is a limited grid for explanatory purposes:

```
phi = c(.8,.9, 1)
gamma = c(.001, .2, .6,  .99)
beta = c(.001, .2, .6, .99)
alpha = c(.001, .2, .6, .99)
```

Next, a grid search over all of these values can generate the most accurate model. Note, MSE will be used in this example but other metrics can be evaluated. Finally, some combinations of parameters are intractable and cannot be fit together. A “Try-Catch” block allows code to execute without having to manually work through possible combinations of parameters. For more information, consult Holt-Winters original formulation.

```
eval_df = tibble()

for (p in phi){
  for (g in gamma){
    for (b in beta){
      for (a in alpha){
        result = try({
        curr_model = hw(beer_ts_train, seasonal = "multiplicative", damped = TRUE, alpha = a, beta = b, gamma = g, phi = p,
h = 68)
        }, silent = TRUE)
        if (class(result) == "try-error") {
          next
        }
        else {

          eval_df = rbind(eval_df, c(a, b, g, p, mean((curr_model$mean-as.numeric(beer_ts_test))^2)))
        }
    }
  }
}
}

colnames(eval_df) = c("alpha", "beta", "gamma", "phi", "MSE")
```

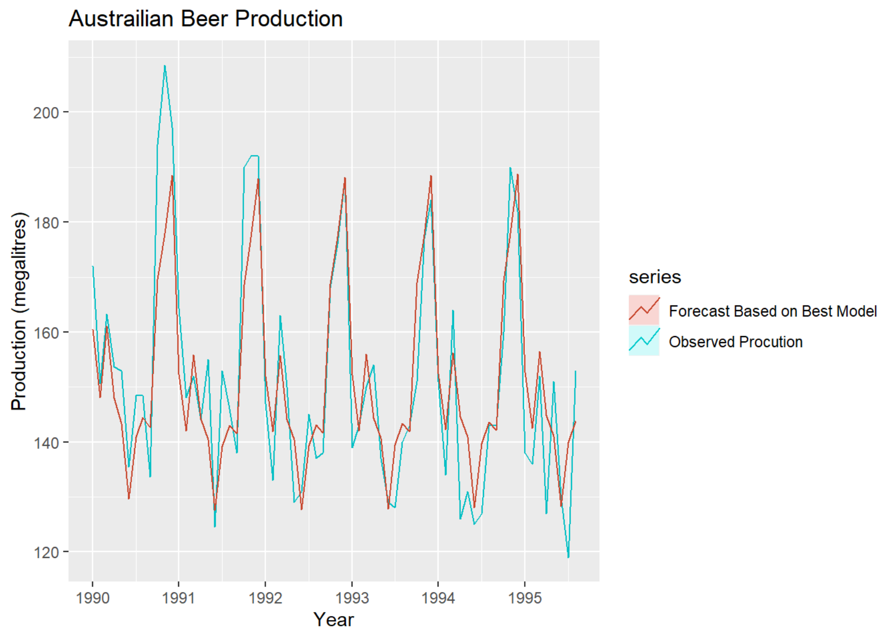
Here is the best model

```
eval_df[which.min(eval_df$MSE),]
```

```
##   alpha beta gamma phi      MSE
## 9   0.2  0.2   0.2 0.8 101.1459
```

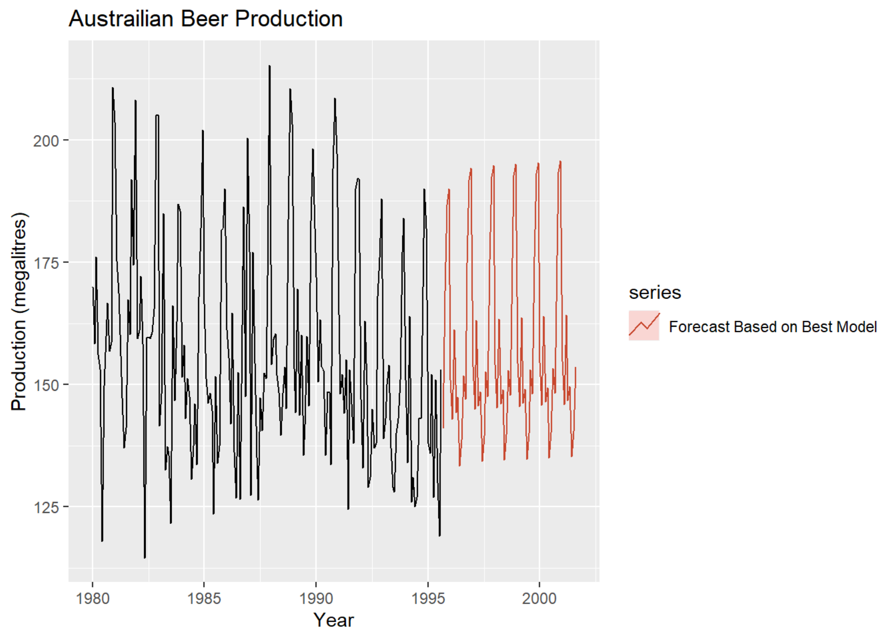
Here is how that best model looks against the test data:

```
best_alpha = eval_df[which.min(eval_df$MSE),]$alpha
best_beta = eval_df[which.min(eval_df$MSE),]$beta
best_gamma = eval_df[which.min(eval_df$MSE),]$gamma
best_phi = eval_df[which.min(eval_df$MSE),]$phi

best_model = hw(beer_ts_train, seasonal = "multiplicative", damped = TRUE, alpha = best_alpha, beta = best_beta, gamma = bes
t_gamma, phi = best_phi, h = 68)

autoplot(beer_ts_test, series="Observed Procution") +
  autolayer(best_model, series="Forecast Based on Best Model", PI = FALSE)+
  ggtitle("Austrailian Beer Production") +
  xlab("Year")+
  ylab("Production (megalitres)")
```



The last step is to refit over the whole data set using the best parameters and examine the resulting forecast

```
best_model_full = hw(beer_ts, seasonal = "multiplicative", damped = TRUE, alpha = best_alpha, beta = best_beta, gamma = best
_gamma, phi = best_phi, h = 72)

autoplot(window(beer_ts, start = 1980)) +
  autolayer(best_model_full, series="Forecast Based on Best Model", PI = FALSE)+
  ggtitle("Austrailian Beer Production") +
  xlab("Year")+
  ylab("Production (megalitres)")
```

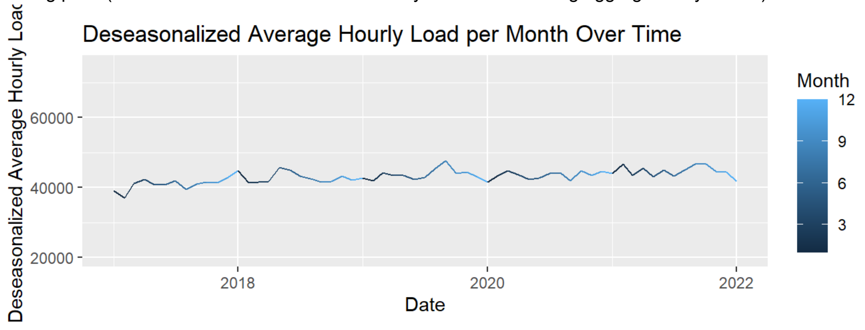


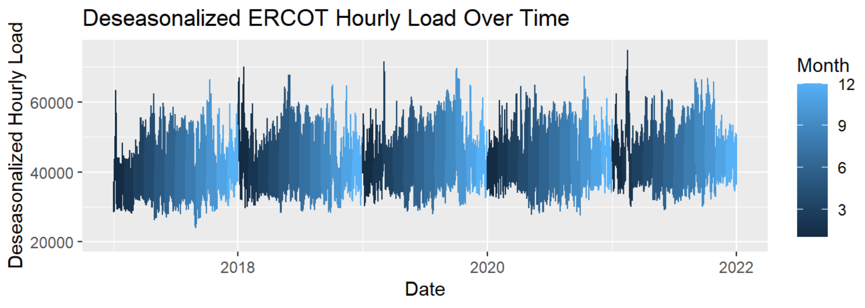
5. Compound seasonality

a. Problem description

The Holt-Winters method of exponential smoothing has demonstrated robust versatility in its ability to adaptively fit forecasts for time series data that have both linear trends and seasonal components. There are, however, still situations where this model fails to capture the full story being expressed by the data, and one important example of this is when it comes to compound seasonality. As has been mentioned before, we will sometimes be faced with situations where there are not only global seasonal trends that might cycle on the magnitude of years but also more frequent, shorter interval seasons that might cycle at the daily scale. This phenomenon was present in the ERCOT Hourly Load data example presented in the former section, and we will return to this case study to illustrate how deseasonalization paired with Holt-Winters' methods and reseasonalization can be used to address the issue of compound seasonality that has yet to be resolved by any techniques mentioned until this point. Even though Holt-Winters has been shown to be able to handle instances of seasonality, it is inadequate to address the situation of compound seasonality on its own since, when we have both daily and monthly seasonality, this would require that we approximate seasonal parameters for every hour of every day in the year, thus resulting in $24 * 365 = 8760$ parameters needing to be estimated. This would be quite infeasible and not the most meaningful approach to solve the problem, so that is why using deseasonalization in conjunction with Holt-Winters is a more appropriate option.

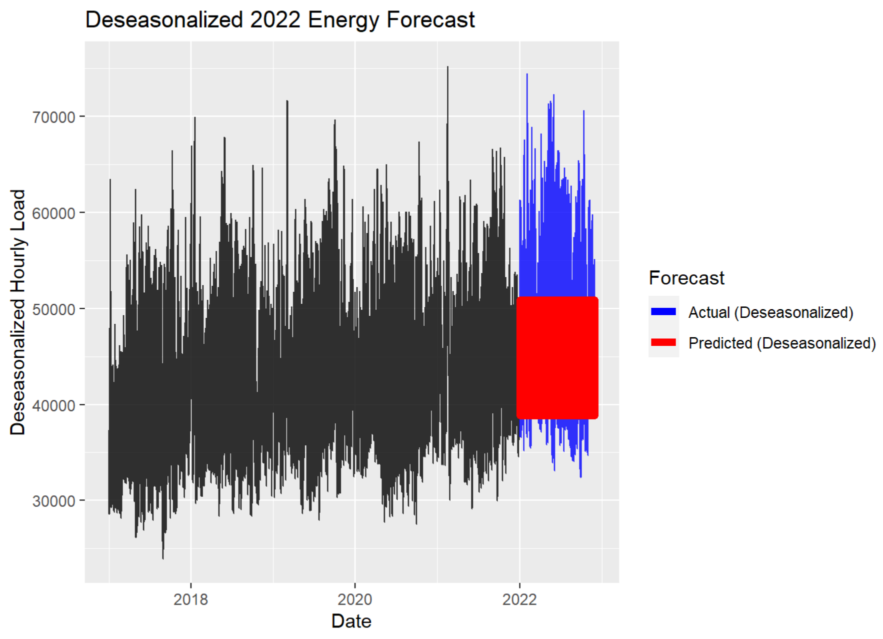
As you may recall, once we had deseasonalized the original ERCOT Hourly Load data by accounting for the monthly scaling factors, we found ourselves with the following plots (the second of which shows the hourly data without being aggregated by month):



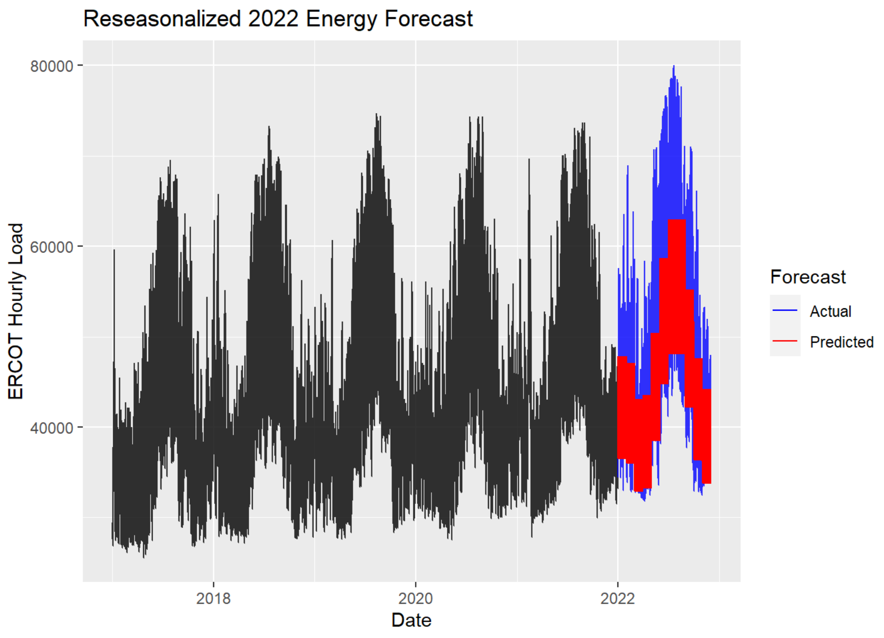


Although the monthly seasonality had been addressed relatively well by deseasonalization, we can still see high volatility occurring on a daily cycle whereby energy consumption spikes during the day and is reduced overnight. This can be noticed in the plot above by the large variance in the y-axis of the second plot which shows that each day can fluctuate in hourly energy consumption by almost $30,000$ MW. In order to try and handle this second layer of seasonality present in the data, we can now fit a Holt-Winters seasonal model on the monthly deseasonalized data using a 24 period daily cycle. This will make it so that our new model is able to capture the daily seasonality through a more adaptive and parameterized approach without having to consider the seasonality that is occurring on a monthly scale.

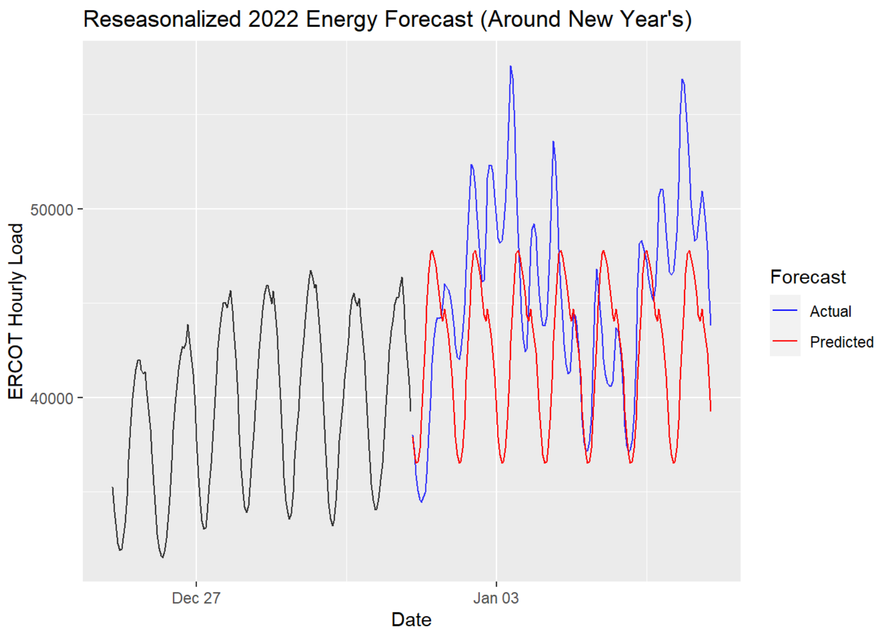
With the process of deseasonalization followed by fitting a Holt-Winters model complete, we are now able to attempt to forecast the 2022 data as we have done in the past.



As was seen previously, the predicted forecast for the double deseasonalized data appears to be occasionally underestimating the true values observed in 2022, but this is mainly due to the abnormally low energy consumption at the end of 2021 combined with a systematic rise in energy consumption in 2022 rather than an issue with the model, and, in order to see the full effects of this stacked seasonality approach, we can now reseasonalize the data and forecasted values by multiplying each point by their respective monthly seasonal factors. Visualizing the result of this will allow us to better appreciate the positive influence this technique has on our final predictions.



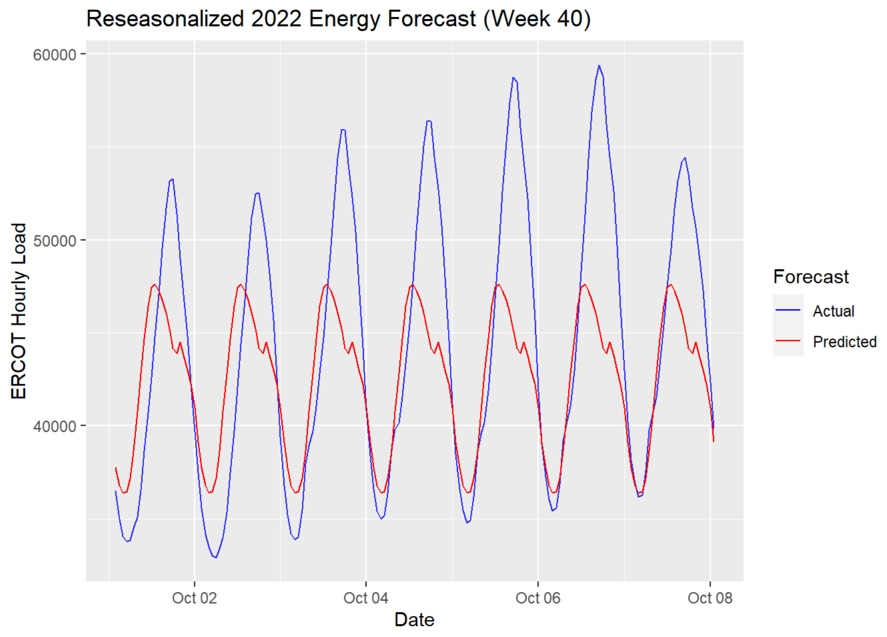
While the plot above allows us to see that the forecasted energy consumption projected for 2022 better matches the actual values, due to the fact that the second seasonal pattern we have identified is cycling on a daily interval rather than a more macroscopic scale, this causes it is be difficult to see how well our predictions are doing at estimating the values of any given day or week. In order to visualize our model's performance with a finer level of detail, we can zoom in and just examine the forecast as we cross over from 2021 to 2022.



This plot does a much better job of illustrating how this stacked seasonality approach has allowed us to capture both the monthly seasonal trends and daily seasonal patterns. Unfortunately, while the graphic above makes it clear that the cyclical daily fluctuations are being correctly modeled by this method and that the predictions closely resemble the data at the very end of 2021, the true energy consumption at the start of 2022 behaves quite erratically and does not follow the expected oscillations very well.

To get a better sense of whether our model is performing highly and tracking multiple levels of seasonality as desired, we can split the 2022 data into week long chunks and then plot an arbitrary week during the year to assess how close our predictions match the actual values at that time.

Once we have split our data up, we can now choose any week in 2022 to visually inspect our forecast over. Let us take week 40 for example.



Observing the plot above allows us to more fully understand and appreciate the value which combining seasonality methods can provide since these predictions are clearly adhering very closely to the actual energy consumption values during this week, and the multiple levels of seasonality have made it so that we are not only able to measure and account for the larger scale monthly trends but can also precisely approximate even the hourly fluctuations we expect to occur over the course of a day. This example does a fantastic job of demonstrating how compound seasonality can appear in real world data sets and how properly modeling it can allow our predictions perform extremely well and adhere very closely to the actual data even up to 40 weeks out!

Instances of compound seasonality are just another feature which may be present in time series data, and, even though some adaptive and more parametric approaches may struggle to handle these situations due to the excessively high number of parameters required to model them, this sections illustrates how sometimes returning to the more basic and intuitive methods can end up providing very strong approximations and results.

Once we begin layering methods and models on top of one another, we can also begin to see how more complicated and iterative techniques can be applied to time series data, and this is exactly the subject matter of the final section of this tutorial.

6. Fitting more complex models

a. Problem description

We have discussed several methods of handling forecasts and predictions on future time series values using the historic data and trends. However, all these models are univariate, only relying on the past values of the target variable. What if we have other data available? How can we combine these powerful univariate methods with other explanatory factors that could help predict whether the target variable will be above or below the exponential smoothing estimate.

b. Concept - boosting using time series

One possible approach is to use boosted models. First, we remove the time series correlation in the data using seasonal factors or exponential smoothing. Then we can model the residuals using other data. Once we have fitted the residuals of our time series model, we can use any method to predict residuals using other data, including random forest, OLS, XGBoost, etc.

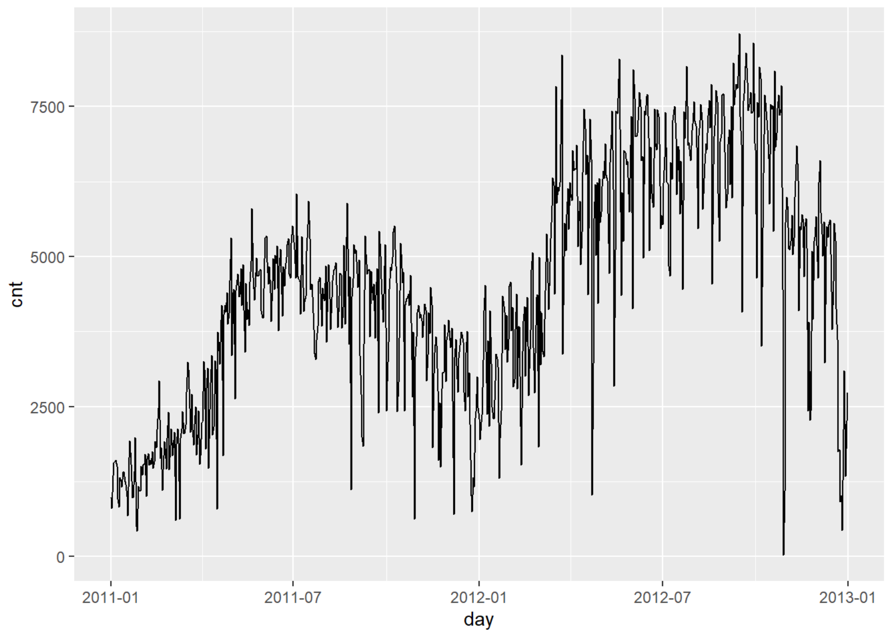
To generate predictions, we would generate a prediction using the times series forecasting model. Then, using other data, we would calculate a predicted residual and add that to our pure time series forecast to get a prediction of the overall level. This approach has flaws, as by fitting time and the data separately, we miss out on any interactions between the time variable and the other data. ARIMA models and other time series methods allow us to fit simultaneously, but for some models, the flexibility of being able to combine any multivariate approach to an exponential smoothing approach is valuable.

c. Example - bikesharing

Data from https://www.kaggle.com/datasets/lakshmi25npathi/bike-sharing-dataset?select=day.csv (https://www.kaggle.com/datasets/lakshmi25npathi/bike-sharing-dataset?select=day.csv)

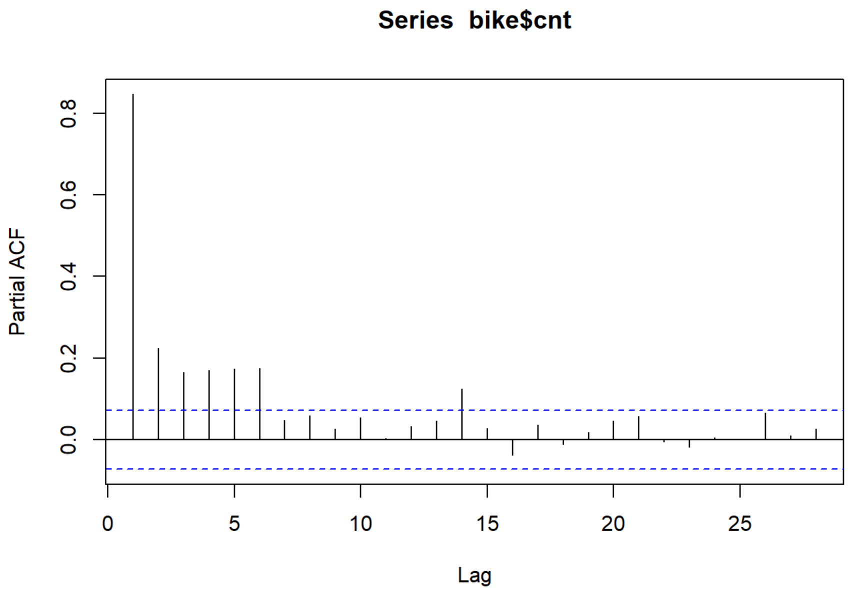
This data asks us to predict the level of usage seen by a bike sharing program on any given day. First, let's see the output variable:

```r
bike = read.csv('bikeshare_daily.csv')
# convert the dates to POSIXct format
bike$day = ymd(bike$dteday)

ggplot() + geom_line(data = bike, aes(x = day, y = cnt))
```



Let’s check the PACF to see if there is any autocorrelation

```
pacf(bike$cnt)
```



Knowing there is a time series element to this data, we can fit Holt Winters to calculate the prediction for count of riders using just prior information

7. Appendix - measure of model fit

DO NOT GRADE - borrowed from Prof. Doug Thomas (with permission) as useful information for the tutorial

a. Model metrics

No matter how forecasts are generated, we are interested in evaluating the quality of the forecasts for essentially two reasons. First, recall that a forecast is a description, not just a number. The historical performance of a forecasting process helps us describe the possible future performance. Second, by tracking the performance of a forecasting process over time, we may be able to identify ways to improve that process.

We define some basic measures of forecast performance:

$$\bar{E} = \text{Average Error} = \sum_{t=1}^{N} E_t^k / N.$$

Similarly, define the average error as, often called the mean absolute error (abbreviated MAE) or mean absolute deviation (abbreviated MAD), as

$$MAE = (AE_1^k + \cdots + AE_N^k)/N.$$

The average absolute percent error is often called the mean absolute percent error and abbreviated MAPE.

$$MAPE = \sum_{t=1}^{N} APE_t^k / N.$$

2.2 Forecast Bias

One dimension of forecast performance is the of the forecast. Measures of bias tell us if the point-estimate forecast is higher or lower than the actual value. Unlike averaging errors over time, this measure allows positive and negative errors to cancel each other out over time. For example, suppose one forecasting process produced point-estimates that alternated between being 10 units too high and 10 units too low, and another forecasting process produced point-estimates with no errors at all. The average error for both of these processes would be zero. So, the average error does not tell us about the magnitude of the error the way MAE does. Instead, the average error can tell us if we are forecasting too high or too low over time. A closely related measure sometimes used in practice is the (rather than the average) of forecast errors. This is sometimes referred to as the running sum of forecast errors (RSFE).

$$RSFE = \text{Sum of Errors} = \sum_{t=1}^{N} E_t^k.$$

With either of these measures, if their value is close to zero, we say that the forecasting process is .

```
Close to zero'' is not a terribly precise term obviously, so we need to be concerned with scaling or interpreting the measures to try to decide wh
```

signal." There are multiple versions of tracking signals used in practice. Here we consider one that (hopefully) has some intuitive appeal. We will use one or more tracking signals later to try to identify bias in a forecasting process.

$$TS = \frac{\sum_{t=1}^{N} E_t^k / N}{\sum_{t=1}^{N} AE_t^k / N} = \frac{\bar{E}}{MAE}.$$

In this case, we try to determine whether the average error is large or not by comparing it to the average error (MAE). If our errors are always negative, this tracking signal will equal -1. If our errors are always positive, this tracking signal will equal 1. A reasonable rule of thumb is that if this value is between -0.5 and 0.5, we can conclude with some level of comfort that the forecasting process is unbiased.

Another measure of bias commonly used in practice is the average of actual values divided by forecasts. If over time this value is close to 100%, we conclude our forecasting process is unbiased. This kind of measure is more frequently used as a summary performance measure for many forecasts aggregated together whereas tracking signals are often embedded in software models.

b. Forecast Accuracy

When we speak of the of a forecast, we generally mean some measure of how far away our point-estimate forecast is from the actual value. A challenge is trying to separate the of forecast errors from the . Recall that measures of bias tells us how far off the forecasts are from the target , but they do not tell us how widely or narrowly scattered the forecast errors are.

The standard deviation of forecast errors (SDFE) gives us a measure of dispersion of forecast errors:

$$SDFE = \sqrt{\sum_{t=1}^{N} (E_t^k - \bar{E})^2 / (N-1)},$$

where $\bar{E}$ is the average error. SDFE can be calculated in Excel by using the function on the column of errors. It is important to note here that bias does not affect this measure of forecast dispersion. SDFE tells us if the forecast errors are narrowly or widely grouped but cannot by itself tell us if our forecasting process is working well. We could have narrowly grouped forecast errors (low SDFE) that are very biased (high average error). Thus, we need to consider both bias and dispersion when evaluating a forecasting process.

c. Measures that combine accuracy and bias

Rather than calculate many different metrics for measuring accuracy and bias of forecasts, it can be helpful to calculate a single metric to compare both

Mean absolute error (MAE, defined above) is a common measure of forecast performance that combines both bias and dispersion. As an example, consider the case where all forecasts where exactly 2 units too low. It is easily verified that the average error is 2, the mean absolute error is also 2, but the standard deviation of forecast errors would be zero. That is, the forecasts are biased but (extremely) narrowly dispersed.

Another common measure of forecast performance is the mean squared error (MSE) and the square root of MSE, often called root mean squared error (rMSE). Similar to MAE, the mean squared error is affected by both bias and dispersion.

$$MSE = \sum_{t=1}^{N} (E_t^k)^2 / N rMSE \; = \sqrt{\sum_{t=1}^{N} (E_t^k)^2 / N}$$

After a bit of algebra, MSE can be rewritten as the variance of forecast errors (the SDFE squared) plus the average error squared.

$$MSE = SDFE^2 + \bar{E}^2 rMSE \; = \sqrt{SDFE^2 + \bar{E}^2}$$

This implies that when forecasts are not biased (average error close to zero), rMSE will be quite close in value to SDFE.

Some organizations use the term to mean a specific measurement of forecast performance. Usually this is defined as 100% minus MAPE.