Overview of Forecasting Methods

Lesson 1.3

Introduction

The ultimate end goal of time series is forecasting. Forecasting refers to predicting what will happen in the future by taking into consideration the past and present events. Forecasting methods can be generally classified into two: **Quantitative** and **Qualitative**.

Quantitative forecasting methods include, among others, naive method, moving averages, exponential smoothing, and regression approach.

1. **Average method**: The forecasts of all future values are equal to the average (or "mean") of the historical data.
2. **Naive method**: The forecast is set as the last value of the series.
3. **Seasonal naïve method**: A similar method is useful for highly seasonal data. In this case, we set each forecast to be equal to the last observed value from the same season (e.g., the same month of the previous year).
4. **Drift method**: A variation on the naive method is to allow the forecasts to increase or decrease over time, where the amount of change over time (called the drift) is set to be the average change seen in the historical data. This is equivalent to drawing a line between the first and last observations, and extrapolating it into the future.
5. **Forecasting with linear regression**
6. **Forecasting with exponential smoothing models**
7. **Forecasting with ARIMA models**

The Forecasting Workflow

Traditional time series forecasting follows the same workflow as most of the fields of predictive analysis, such as regression or classification, and typically includes the following steps:

1. **Data preparation**: Here, we prepare the data for the training and testing process of the model. This step includes splitting the series into training (in-sample) and testing (out-sample) partitions, creating new features (when applicable), and applying a transformation if needed (for example, log transformation, scaling, and so on).
2. **Train the model**: Here, we used the training partition to train a statistical model. The main goal of this step is to utilize the training set to train, tune, and estimate the model coefficients that minimize the selected error criteria (later on in this course, we will discuss common error metrics in detail). The fitted values and the model estimation of the training partition observations will be used later on to evaluate the overall performance of the model.
3. **Test the model**: Here, we utilize the trained model to forecast the corresponding observations of the testing partition. The main idea here is to evaluate the performance of the model with a new dataset (that the model did not see during the training process)
4. **Model evaluation**: Last but not least, after the model was trained and tested, it is time to evaluate the overall performance of the model on both the training and testing partitions.

Training Approaches

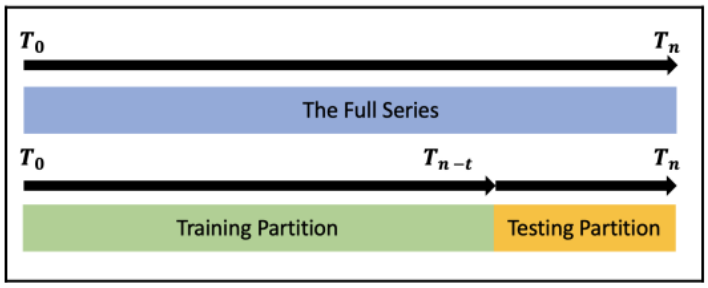
One of the core elements of the forecasting workflow is the model training process. The quality of the model's training will have a direct impact on the forecast output. The main goals of this process are as follows:

- Formalize the relationship of the series with other factors, such as seasonal and trend patterns, correlation with past lags, and external variables in a predictive manner
- Tune the model parameters (when applicable)
- The model is scalable on new data, or in other words, avoids overfitting.

As we mentioned previously, prior to the training process, the series is split into training and testing partitions, where the model is being trained on the training partition and tested on the testing partition. These partitions must be in chronological order, regardless of the training approach that has been used. The main reason for this is that most of the time series models establish a mathematical relationship between the series in terms of its past lags and error terms.

Training with single training and testing partitions

One of the most common training approaches is using single training and testing (or single out-of-sample) partitions. This simplistic approach is based on splitting the series into training and testing partitions (or in-sample and out-sample partitions, respectively), training the model on the training partition, and testing its performance on the testing set:



As you can see in the diagram, the training and testing partitions are ordered and organized in chronological order. This approach has a single parameter—the length of the out-of-sample (or the length of the testing partition). Typically, the length of the testing partition is derived from the following rules of thumb:

- The length of the testing partition should be up to 30% of the total length of the series in order to have enough observation data for the training process.
- The length of the forecasting horizon (as long it is not violating the previous term). This is mainly related to the fact that the level of uncertainty increases as the forecast horizon increases. Therefore, aligning the testing set to the forecast horizon could, potentially, provide a closer estimate of the forecast's expected error.

For example, if we have a monthly series with 72 observations (or 6 years) and the goal is to forecast the next year (or 12 months), it would make sense to use the first 60 observations for training and test the performance using the last 12 observations.

The simplicity of this method is its main advantage, as it is fairly fast to train and test a model while using (relatively) cheap computeing power. On the other hand, it isn't possible to come to a conclusion about the stability and scalability of the model's performance based on a single test unit. It is feasible that a model, only by chance, will have relatively good performance on the testing set but do poorly on the actual forecast as it isn't stable over time. One way to mitigate that risk is with the backtesting approach, which is based on training a model with multiple training and testing partitions.

Example

```
library(TSstudio)
data(USgas)
ts_info(USgas)
```

```
The USgas series is a ts object with 1 variable and 238 observations
Frequency: 12
Start time: 2000 1
End time: 2019 10
```

We can use the *window()* function in the **stats** package to split the time series into training and testing sets.

```
train <- window(USgas,
                start = time(USgas)[1],
                end = time(USgas)[length(USgas) - 12])
test <- window(USgas,
               start = time(USgas)[length(USgas) - 12 + 1],
               end = time(USgas)[length(USgas)])
ts_info(train)
```

```
The train series is a ts object with 1 variable and 226 observations
Frequency: 12
Start time: 2000 1
End time: 2018 10
```

```
ts_info(test)
```

```
The test series is a ts object with 1 variable and 12 observations
Frequency: 12
Start time: 2018 11
End time: 2019 10
```

Alternatively, we can use the *ts_split()* function from the **TSstudio** package which provides a customized way for creating training and testing partitions for time series data.

```
USgas_partitions <- ts_split(USgas, sample.out = 12)
train1 <- USgas_partitions$train
test1 <- USgas_partitions$test
ts_info(train1)
```

```
The train1 series is a ts object with 1 variable and 226 observations
Frequency: 12
Start time: 2000 1
End time: 2018 10
```

```
ts_info(test1)
```

```
The test1 series is a ts object with 1 variable and 12 observations
Frequency: 12
Start time: 2018 11
End time: 2019 10
```

The *sample.out* argument in *ts_split()* set the size of the testing partition.

Forecasting with backtesting

The backtesting approach for training a forecasting model is an advanced version of the single out-of-sample approach we saw previously. It is based on the use of a rolling window to split the series into multiple pairs of training and testing partitions. A basic backtesting training process includes the following steps:

1. *Data preparation*: Create multiple pairs of training and testing partitions.
2. *Train a model*: This is done on each one of the training partitions.
3. *Test the model*: Score its performance on the corresponding testing partitions.
4. *Evaluate the model*: Evaluate the model's accuracy, scalability, and stability based on the testing score. Based on the evaluation, you would do one of the following:
    - Generate the final forecast to check whether the model score meets a specific threshold or criteria
    - Apply additional tuning and optimization for the model and repeat the training and evaluations steps

The use of scoring methodology allows us to assess the model's stability by examining the model's error rate on the different testing sets. We would consider a model as stable whenever the model's error distribution on the testing sets is fairly narrow. In this case, the error rate of the actual forecast should be within the same range of the testing sets (assuming there are no abnormal events that impact the forecast error rate).

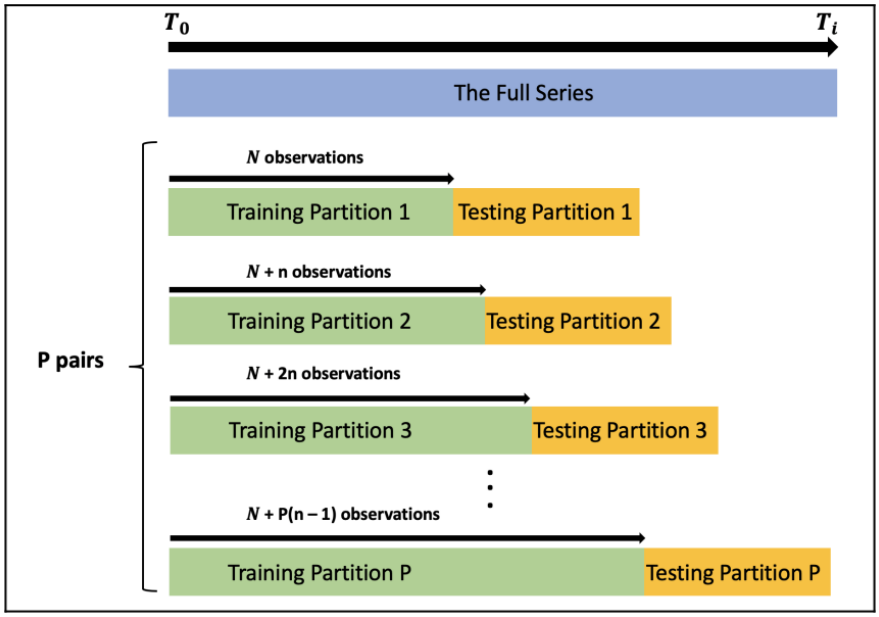
This method is, conceptually, similar to the cross-validation approach for training machine learning models. The main distinction between the two approaches is related to how their partitions are set. While the backtesting partitions are ordered chronologically, the ones of the cross-validation approach are based on random sampling.

The main setting parameters of a backtesting training model are as follows:

The length of the training partitions: It is set by the window setting. There are two common types of rolling windows:

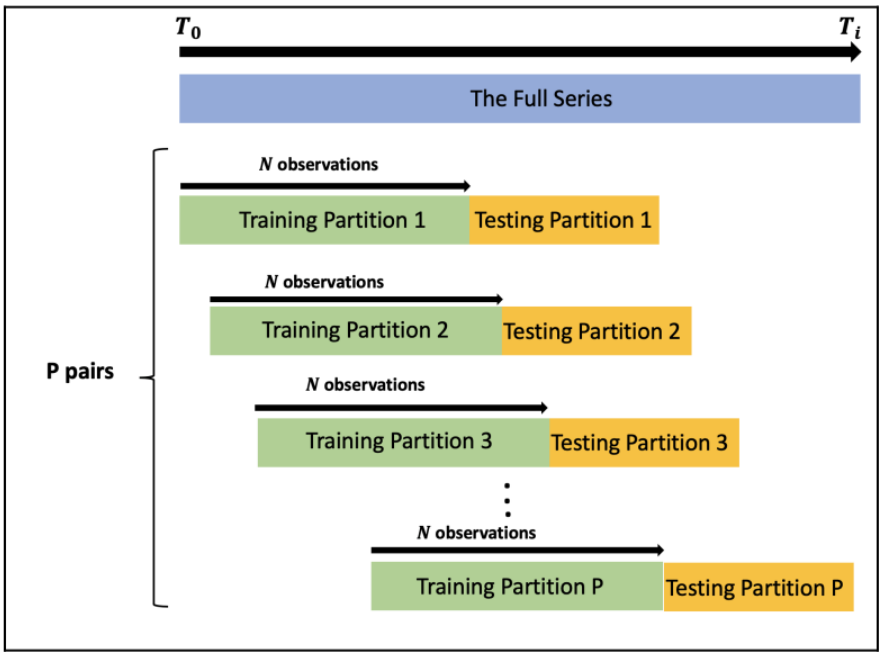
- Expanding window: Using the first N observations as the initial training partition, add the following n observations to create the following training partition. The length of the training partition grows as the window rolls over the series
- Sliding window: Set the first N observations as the initial training partition and shift the window by n observations to create the following training partition. The length of the training partitions remains the same as the window rolls over the series:
  - The length of the testing partitions—a constant value, typically aligned with the length of the forecasting horizon (under the limitation of the minimum number of observations required to train the model)
  - The space between each training partition—defines the pace of the rolling window
  - The number of training and testing partition

The following diagram demonstrates the structure of the backtesting with an expanding window:



As you can see in the preceding diagram, all of the training partitions of the expanding window method start at the same index point, T_0, where each training partition is a combination of the previous training partition with the following n observations (where n is a constant that represents the expanded rate of the window). It would make sense to use this method when the series has a strong seasonal pattern and stable trend. In this case, the first observations of the series could, potentially, have predictive information that can be utilized by the model. The downside of this method is training the model with different length partitions, as typically a model tends to learn more and therefore perform better when more data is available. Therefore, we may observe that the performance of models that are trained on the latest partitions perform better than the ones that are trained with the first partitions.

This bias does not exist in the second training approach—the sliding window, as all of training partitions, are of the same length, as you can see in the following diagram:



It would make more sense to use the sliding window whenever the input series has structural change or high volatility, or when most of the predictive power is linked to the most recent history (or high correlation with the most recent lags of the series). For example, as we saw in the previous chapter, monthly crude oil prices have a strong relationship with the most recent lags of the series, and the far history doesn't contain powerful predictive information about the series' future direction.

While the backtesting approach provides us with intuitive information about the stability and scalability of the model, it comes with a higher computation cost compared to the single training and testing partition approach. Hence, this trade-off between the two approaches should be taken into consideration when selecting the training approach.

Forecast evaluation

The primary goal of the evaluation step is to assess the ability of the trained model to forecast (or based on other criteria) the future observations of the series accurately. This process includes doing the following:

- **Residual analysis**: This focuses on the quality of the model, with fitted values in the training partition
- **Scoring the forecast**: This is based on the ability of the model to forecast the actual values of the testing set

Residual analysis

Residual analysis tests how well the model captured and identified the series patterns. In addition, it provides information about the residuals distributions, which are required to build confidence intervals for the forecast.

The mathematical definition of a residual is the difference between the actual observation and the corresponding fitted value of the model, which was observed in the training process, or as the following equation:

$$\epsilon_t = Y_t - \hat{Y}_t$$

Here ϵ_t, Y_t, and $\hat{Y}_t$ represent the residual, actual, and fitted values, respectively, at time t.

This process includes the use of data visualizations tools and statistical test to assess the following:

- **Test the goodness of the fit against the actual values**: You do this by plotting the residuals values over time in chronological order. The plot indicates how well the model was able to capture the oscillation of the series in the training partition. Residuals with random oscillation around the zero and with constant variation (white noise) indicate that the model is able to capture the majority of the series variation. On the other hand, if the residual oscillation does not have the white noise characteristics, it is an indication that the model failed to capture the series patterns. Here are some potential interpretations of the possible output:
  - **All or most of the residuals are above the zero lines**: This is an indication that the model tends to underestimate the actual values
  - **All or most of the residuals are below the zero lines**: This is an indication that the model tends to overestimate the actual values
  - **Random spikes**: This is an indication for potential outliers in the training partition:
    - **The residual autocorrelation**: This indicates how well the model was able to capture the patterns of the series. Non-correlated lags indicate that there were no patterns that the model did not capture. Similarly, the existence of correlated lags indicates patterns that the model did not capture
    - **The residuals distribution**: This is required to conclude about the reliability of the forecast confidence interval. If the residuals are not normally distributed, we cannot use it to create confidence intervals as it is based on the assumption that the residuals are normally distributed.

To examine the residuals, we will use the *checkresiduals()* function from the **forecast** package, which returns the following four outputs:

- Time series plot of the residuals

- ACF plot of the residuals
- Distribution plot of the residuals
- The output of the Ljung-Box test

In an ideal world, you should end this process with white noise and independent residuals. Yet, in reality, in some cases, it will be harder to achieve this goal due to the series' structure (outliers, structural breaks, and so on), and you may have to select a model that brings you closer to this goal.

Scoring the forecast

Once you finalize the model tuning, it is time to test the ability of the model to predict observations that the model didn't see before (as opposed to the fitted values that the model saw throughout the training process). The most common method for evaluating the success of the forecast in order to predict the actual values is to use accuracy or error metrics. The most common method for evaluating the forecast's success is to predict the actual values with the use of an error metric to quantify the forecast's overall accuracy. The selection of a specific error metric depends on the forecast accuracy's goals. An example of common error metrics are as follows:

- **Mean Squared Error (MSE)**: This quantifies the average squared distance between the actual and forecasted values

$$MSE = \frac{1}{n}\sum_{t=1}^{n}(Y_t - \hat{Y}_t)^2$$

- **Root Mean Squared Error (RMSE)**: This is the root of the average squared distance of the actual and forecasted values

$$RMSE = MSE = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(Y_t - \hat{Y}_t)^2}$$

- **Mean Absolute Error (MAE)**: This measures the absolute error rate of the forecast

$$MAE = \sum_{t=1}^{n}\frac{|Y_t - \hat{Y}_t|}{n}$$

- **Mean Absolute Percentage Error (MAPE)**: This measures the average percentage absolute error

$$MAPE = \frac{1}{n}\sum_{t=1}^{n}\left|\frac{Y_t - \hat{Y}_t}{Y_t}\right|$$

Forecast benchmark

How can we assess whether these results are too high or low? The most common method is to benchmark the model's performance to some baseline forecast or to some legacy method that we wish to replace. A popular benchmark approach would be to use a simplistic forecasting approach as a baseline.

Finalizing the forecast

Now that the model has been trained, tested, tuned (if required), and evaluated successfully, we can move forward to the last step and finalize the forecast. This step is based on recalibrating the model's weights or coefficients with the full series. There are two approaches to using the model parameter setting:

- If the model was tuned manually, you should use the exact tuning parameters that were used on the trained model
- If the model was tuned automatically by an algorithm, you can do either of the following:
  - Extract the parameter setting that was used by with the training partition
  - Let the algorithm retune the model parameters using the full series, under the assumption that the algorithm has the ability to adjust the model parameters correctly when training the model with new data

The use of algorithms to automate the model tuning process is recommended when the model's ability to tune the model is tested with backtesting. This allows you to review whether the algorithm has the ability to adjust the model parameters correctly, based on the backtesting results.

Handling forecast uncertainty

The main goal of the forecasting process, as we saw previously, is to minimize the level of uncertainty around the future values of the series. Although we cannot completely eliminate this uncertainty, we can quantify it and provide some range around the point estimate of the forecast (which is nothing but the model's expected value of each point in the future). This

can be done by using either the confidence interval (or a credible interval, when using the Bayesian model) or by using simulation.

Confidence interval

The confidence interval is a statistical approximation method that's used to express the range of possible values that contain the true value with some degree of confidence (or probability). There are two parameters that determine the range of confidence interval:

- The level of confidence or the probability that the true value will be in that range. The higher the level of confidence is, the wider the interval range.
- The estimated standard deviation of the forecast at time $T + i$, where T represents the length of the series and i represents the i forecasted value. The lower the error rate, the shorter the range of the prediction interval.

By default, the *forecast()* function generates a prediction interval with a level of confidence of 80% and 95%, but you can modify it using the *level* argument.

Simulation

An alternative approach is to use the model distribution to simulate possible paths for the forecast. This method can only be used when the model distribution is available. The *forecast_sim()* function from the **TSstudio** package provides a built-in function for simulating possible forecasting paths. This estimate can be used to calculate the forecast point estimate (for example, using the mean or median of all of the paths), or to calculate probabilities of getting different values.

Horse race approach

Last but not least, we will end this chapter with a robust forecasting approach that combines what we've learned so far in this chapter. The horse race approach is based on training, testing, and evaluating multiple forecasting models and selecting the model that performs the best on the testing partitions.

The *ts_backtesting()* function from the **TSstudio** package conducts the full process of training, testing, evaluating, and then forecasting, using the model that performed the best on the backtesting testing partitions.