In this chapter, we discuss some general tools that are useful for many different forecasting situations. We will describe some benchmark forecasting methods, procedures for checking whether a forecasting method has adequately utilised the available information, techniques for computing prediction intervals, and methods for evaluating forecast accuracy.

Each of the tools discussed in this chapter will be used repeatedly in subsequent chapters as we develop and explore a range of forecasting methods.

5.1 A tidy forecasting workflow

The process of producing forecasts for time series data can be broken down into a few steps.

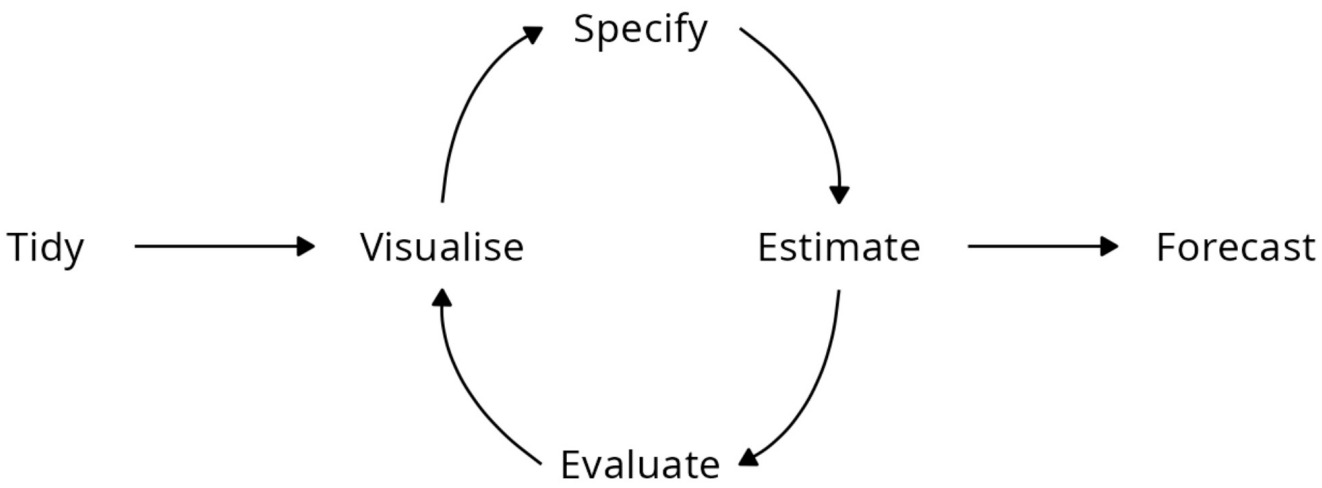

Figure 5.1: Steps to produce forecasts for time series

To illustrate the process, we will fit linear trend models to national GDP data stored in `global_economy`.

Data preparation (tidy)

The first step in forecasting is to prepare data in the correct format. This process may involve loading in data, identifying missing values, filtering the time series, and other pre-processing tasks. The functionality provided by `pandas` substantially simplifies this step.

Many models have different data requirements; some require the series to be in time order, others require no missing values. Checking your data is an essential step to understanding its features and should always be done before models are estimated.

We will model GDP per capita over time; so first, we must compute the relevant variable.

```
gdp_df = pd.read_csv("../data/global_economy.csv", parse_dates=["ds"])
gdp_df[["GDP", "Population"]] = gdp_df[["GDP", "Population"]].interpolate()
gdp_df["y"] = gdp_df["GDP"] / gdp_df["Population"]
gdp_df = gdp_df.drop(["Code", "Growth", "CPI", "Imports", "Exports"],
  axis=1)
gdp_df.head()
```

	unique_id	ds	GDP	Population	y
0	Afghanistan	1960-01-01	5.378e+08	8.996e+06	59.777
1	Afghanistan	1961-01-01	5.489e+08	9.167e+06	59.878
2	Afghanistan	1962-01-01	5.467e+08	9.346e+06	58.493
3	Afghanistan	1963-01-01	7.511e+08	9.534e+06	78.783
4	Afghanistan	1964-01-01	8.000e+08	9.731e+06	82.208

Plot the data (visualise)

As we have seen in Chapter 2, visualisation is an essential step in understanding the data. Looking at your data allows you to identify common patterns, and subsequently specify an appropriate model.

The data for one country in our example are plotted in Figure 5.2.

```
plot_series(gdp_df, ids=["Sweden"],
            xlabel="Year [1Y]", ylabel="$US",
            title="GDP per capita for Sweden")
```

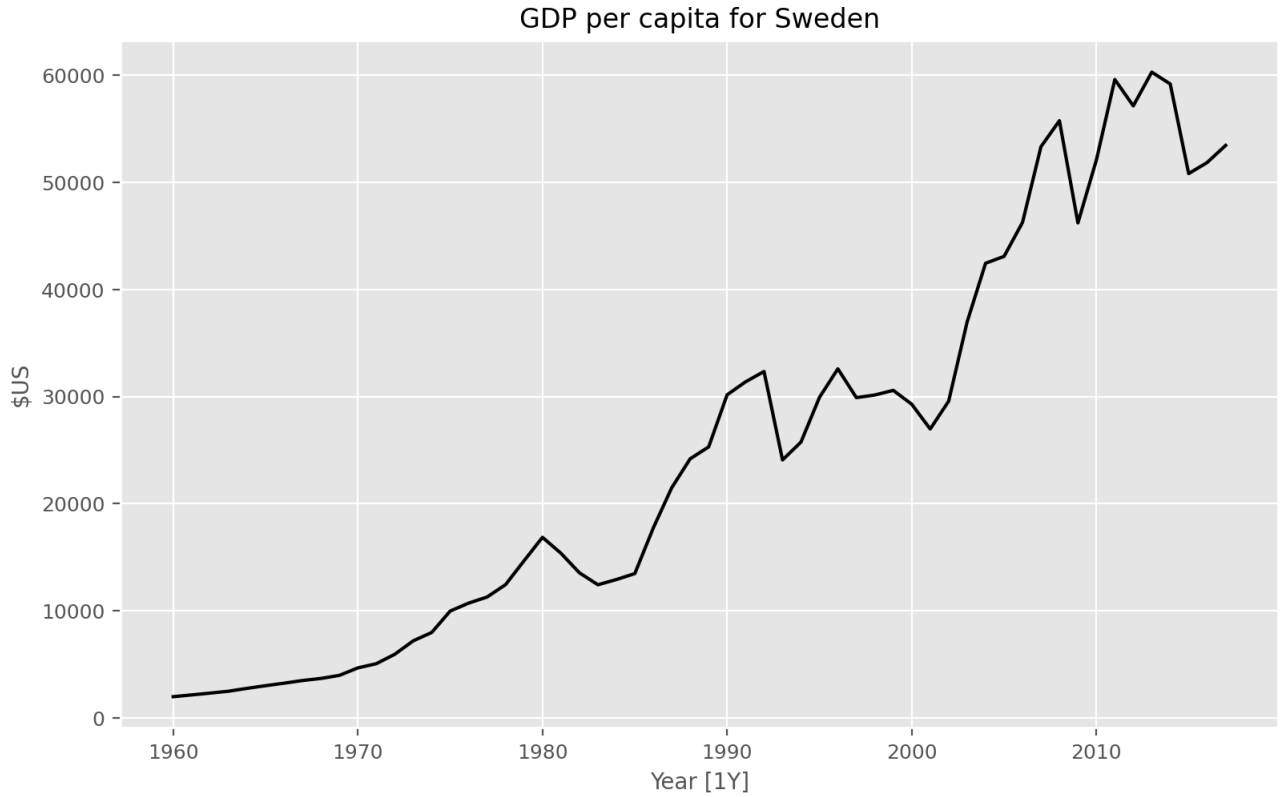

Figure 5.2: GDP per capita data for Sweden from 1960 to 2017.

Define a model (specify)

There are many different time series models that can be used for forecasting, and much of this book is dedicated to describing various models. Specifying an appropriate model for the data is essential for producing appropriate forecasts.

`Statsforecast` comes with many statistical models already implemented and it is also possible to use machine learning models from `scikit-learn` with `Statsforecast`.

For example, a linear trend model (to be discussed in Chapter 7) for GDP per capita can be specified with with the corresponding exogenous variables (as we will be specified later) and using the following to specify the models,

```
SklearnModel(LinearRegression()).
```

In this case the model function is `LinearRegression()` and the response variable is being modelled using a linear trend. We will

be taking a closer look at how each model can be specified in their respective sections.

Train the model (estimate)

Once an appropriate model is specified, we next train the model on some data. One or more model specifications can be estimated using the `sf` object.

To estimate the model in our example, we first remove the unused columns with

```
train_df = gdp_df.copy()
train_df.drop(["GDP", "Population"], axis=1, inplace=True)
sweden_df = train_df.query("unique_id == 'Sweden'")
```

Then, we fit a linear trend model to the GDP per capita data for each combination of key variables. We create these variables using the `pipeline()` function where we specify that we want the `trend` feature. We can then initialize a `StatsForecast` object that will handle the forecasting process. In this example, it will fit a model to each of the countries in the dataset. The resulting object is an instance of `Statsforecast`.

```
train_features, valid_features = pipeline(
    sweden_df,
    features=[trend],
    freq="Y",
    h=3,
)
sf = StatsForecast(
    models=[SklearnModel(LinearRegression())],
    freq="Y",
)
```

Check model performance (evaluate)

Once a model has been fitted, it is important to check how well it has performed on the data. There are several diagnostic tools available to check model behaviour, and also accuracy measures that allow one model to be compared against another. Sections 5.8 and 5.9 go into further details.

Produce forecasts (forecast)

With an appropriate model specified, estimated and checked, it is time to produce the forecasts using the `forecast()` method. The easiest way to use this function is by specifying the number of future observations to forecast. For example, forecasts for the next 10 observations can be generated using `h = 10`.

In other situations, it may be more convenient to provide a dataset of future time periods to forecast. This is commonly required when your model uses additional information from the data, such as exogenous regressors. Additional data required by the model can be included in the dataset of observations to forecast.

```
sf.fit(df=train_features)
fcasts = sf.predict(
    h=3,
    X_df=valid_features,
)
fcasts = sf.fitted_[0][0].model_["model"].add_prediction_intervals(fcasts,
  valid_features.rename(columns={"trend": "x1"}))
```

This is a forecast table. Each row corresponds to one forecast period for each country. The `LinearRegression` column contains the point forecast. The point forecast is the mean (or average) of the forecast distribution.

The forecasts can be plotted along with the historical data using `plot_series()` as follows.

```
plot_series(train_df, fcasts,
            level=[80, 95],
            ids=["Sweden"],
            xlabel="Year",
            ylabel="$US",
            title="GDP per capita for Sweden",
            rm_legend=False
            )
```

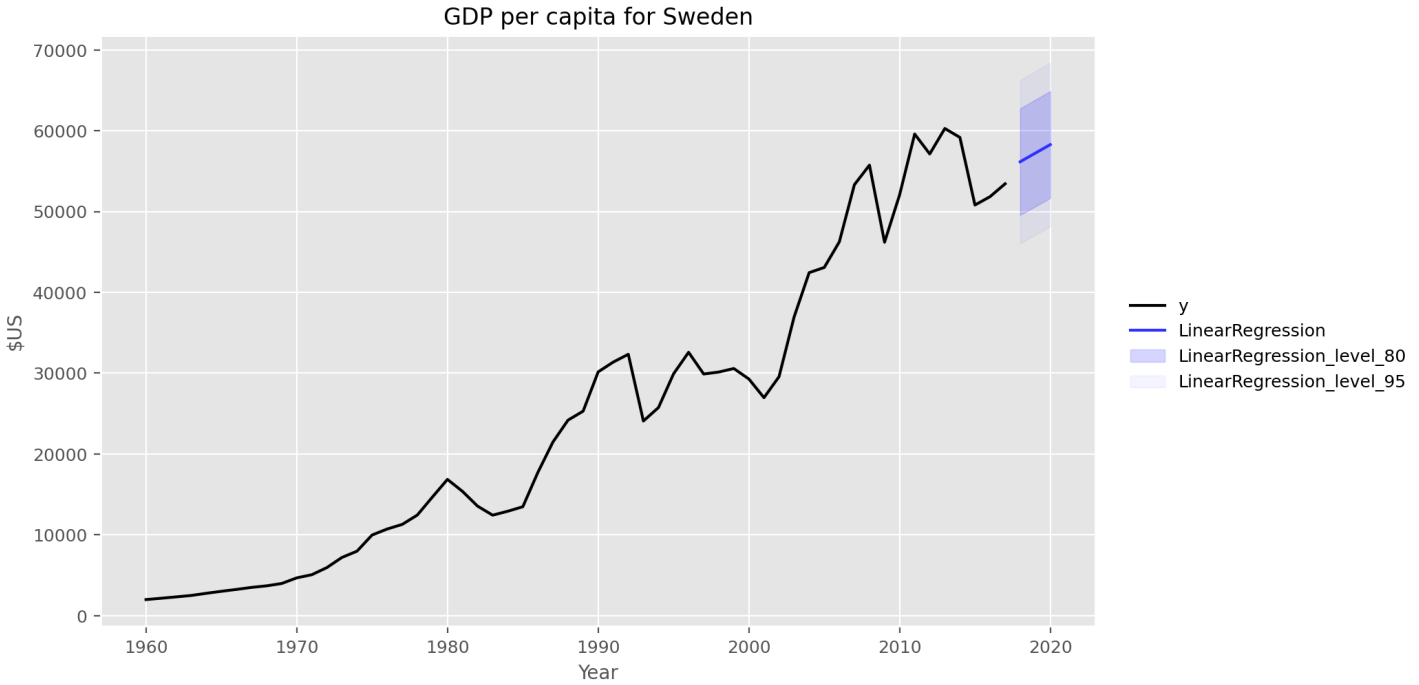

Figure 5.3: Forecasts of GDP per capita for Sweden using a simple trend model.

5.2 Some simple forecasting methods

Some forecasting methods are extremely simple and surprisingly effective. We will use four simple forecasting methods as benchmarks throughout this book. To illustrate them, we will use quarterly Australian clay brick production between 1970 and 2004.

```
production_df = pd.read_csv("../data/aus_production_formatted.csv",
  parse_dates=["ds"])
production_df.head()
```

	unique_id	ds	y
0	Beer	1956-03-01	284.0
1	Beer	1956-06-01	213.0
2	Beer	1956-09-01	227.0
3	Beer	1956-12-01	308.0
4	Beer	1957-03-01	262.0

Mean method

Here, the forecasts of all future values are equal to the average (or “mean”) of the historical data. If we let the historical data be denoted by y_{1},\dots,y_{T}, then we can write the forecasts as \hat{y}_{T+h|T} = \bar{y} = (y_{1}+\dots+y_{T})/T. The notation \hat{y}_{T+h|T} is a short-hand for the estimate of y_{T+h} based on the data y_1,\dots,y_T.

```
avg_method = HistoricAverage()
sf = StatsForecast(models=[avg_method], freq='Q')
```

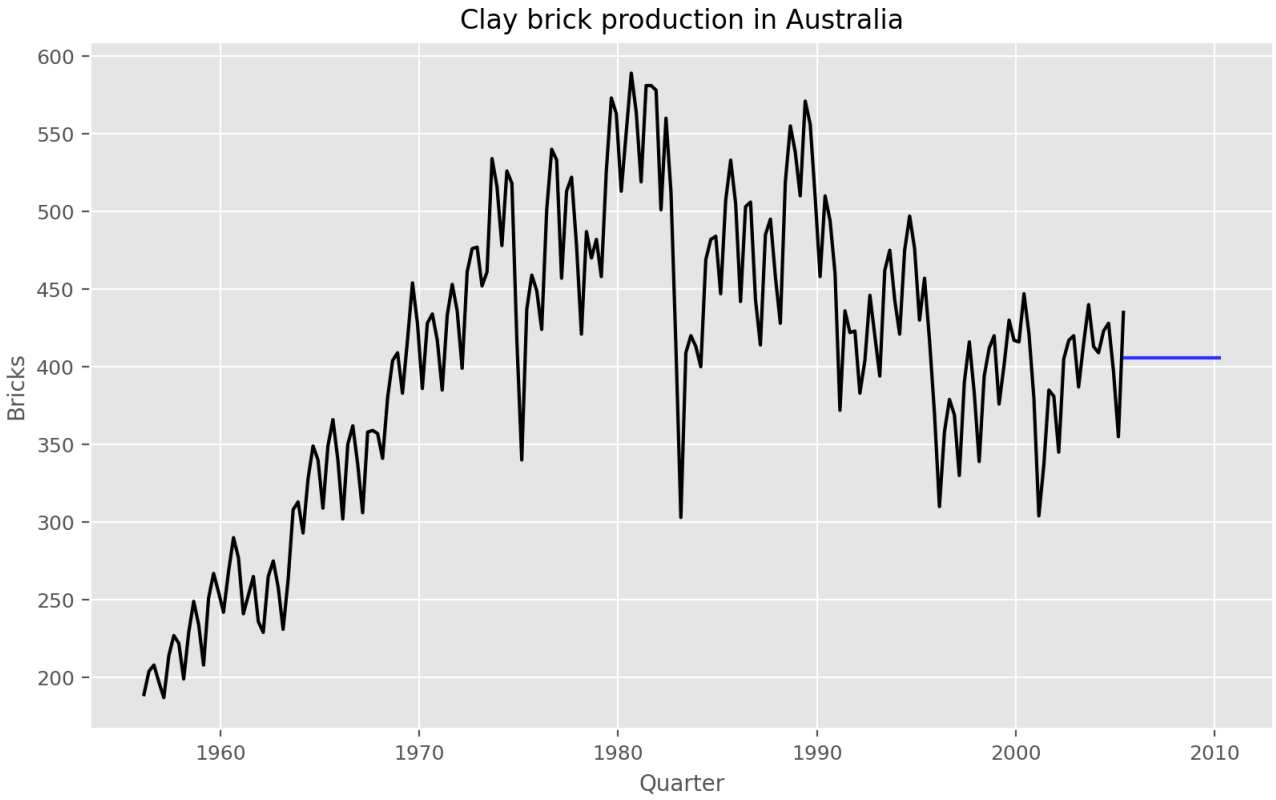

Figure 5.4: Mean (or average) forecasts applied to clay brick production in Australia.

Naïve method

For naïve forecasts, we simply set all forecasts to be the value of the last observation. That is, \hat{y}_{T+h|T} = y_{T}. This method works remarkably well for many economic and financial time series.

```
naive_method = Naive()
sf = StatsForecast(models=[naive_method], freq='Q')
```

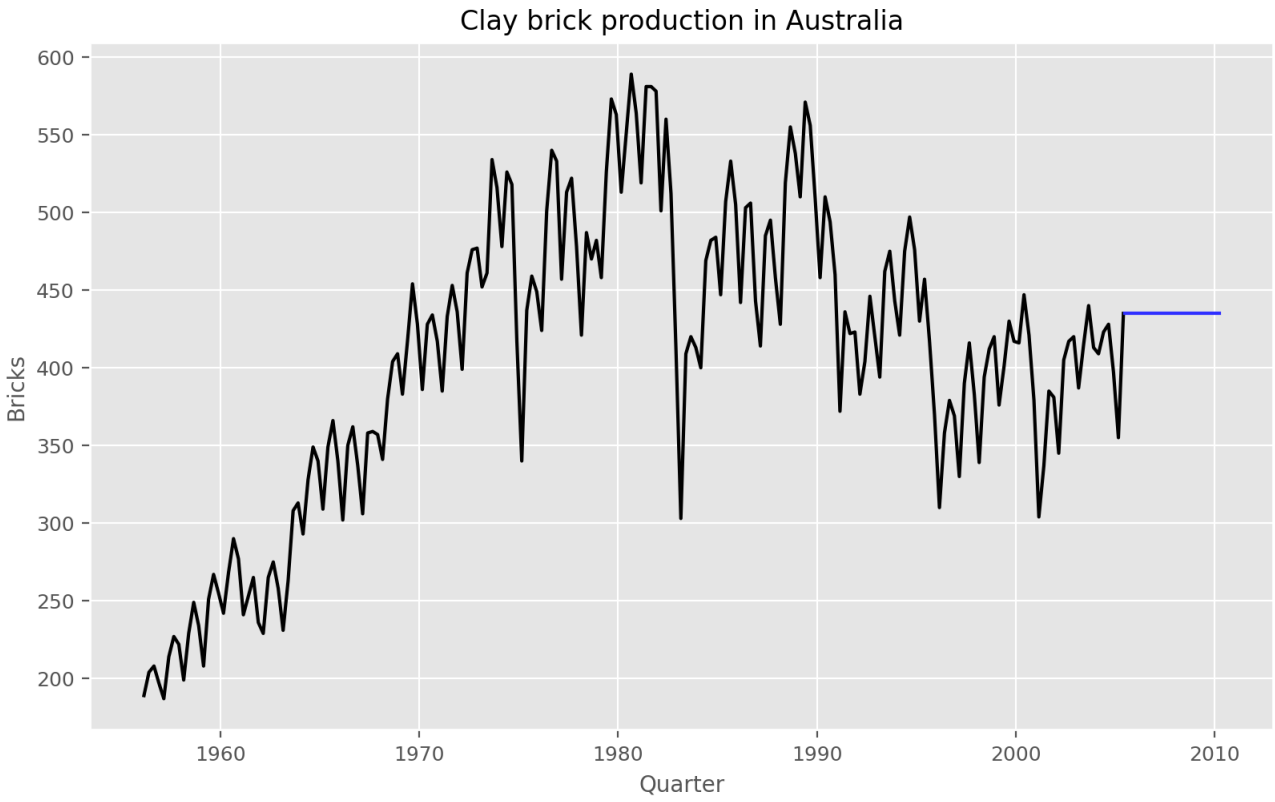

Figure 5.5: Naïve forecasts applied to clay brick production in Australia.

Because a naïve forecast is optimal when data follow a random walk (see Section 9.1), these are also called **random walk forecasts**.

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). Formally, the forecast for time T+h is written as \hat{y}_{T+h|T} = y_{T+h-m(k+1)}, where m= the seasonal period, and k is the integer part of (h-1)/m (i.e., the number of complete years in the forecast period prior to time T+h). This looks more complicated than it really is. For example, with monthly data, the forecast for all future February values is equal to the last observed February value. With quarterly data, the forecast of all future Q2 values is equal to the last observed Q2 value (where Q2 means the second quarter). Similar rules apply for other months and quarters, and for other seasonal periods.

```
seasonal_naive_method = SeasonalNaive(4)
sf = StatsForecast(models=[seasonal_naive_method], freq='Q')
```

Make sure to specify the seasonal length when initializing the `SeasonalNaive` class. Since the data is quarterly, the period is 4.

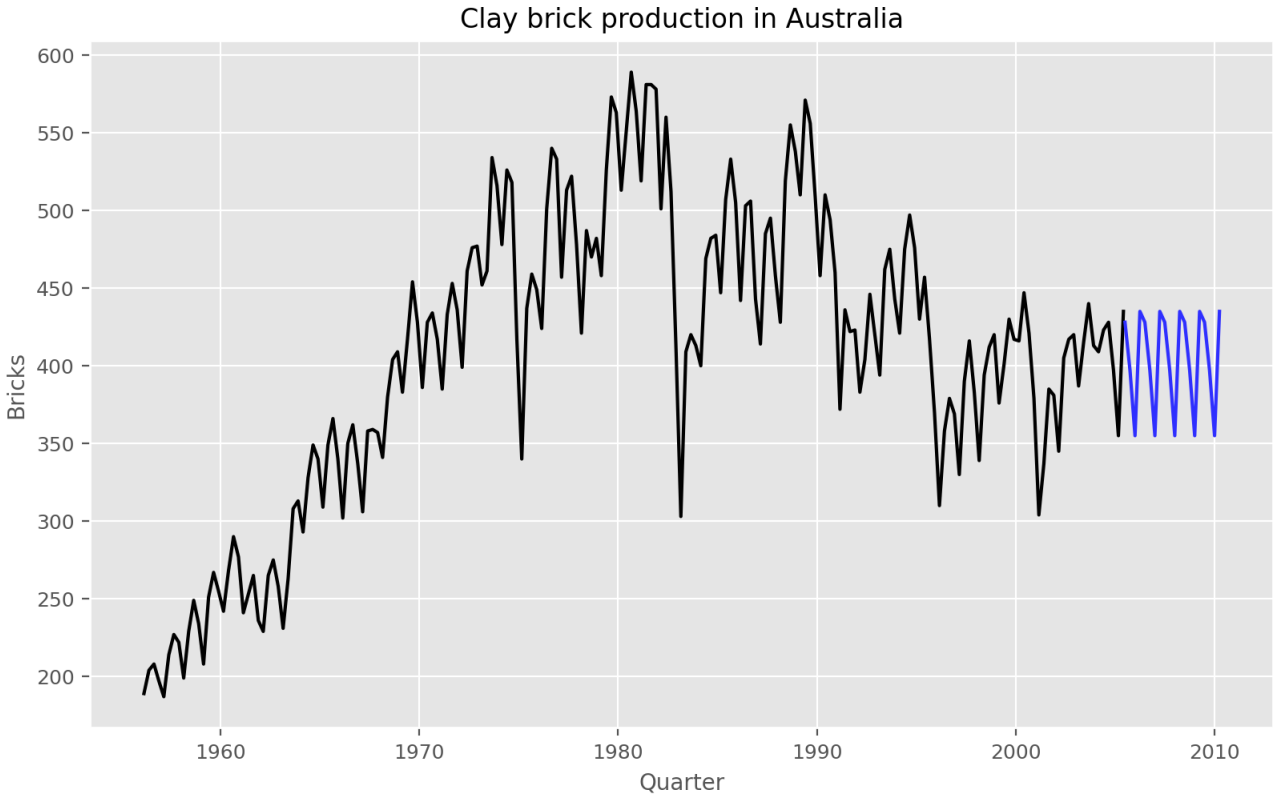

Figure 5.6: Seasonal naïve forecasts applied to clay brick production in Australia.

Drift method

A variation on the naïve 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. Thus the forecast for time T+h is given by \hat{y}_{T+h|T} = y_{T} + \frac{h}{T-1}\sum_{t=2}^T (y_{t}-y_{t-1}) = y_{T} + h \left(\frac{y_{T} -y_{1}}{T-1}\right). This is equivalent to drawing a line between the first and last observations, and extrapolating it into the future.

```
drift_method = RandomWalkWithDrift()
sf = StatsForecast(models=[drift_method], freq='Q')
```

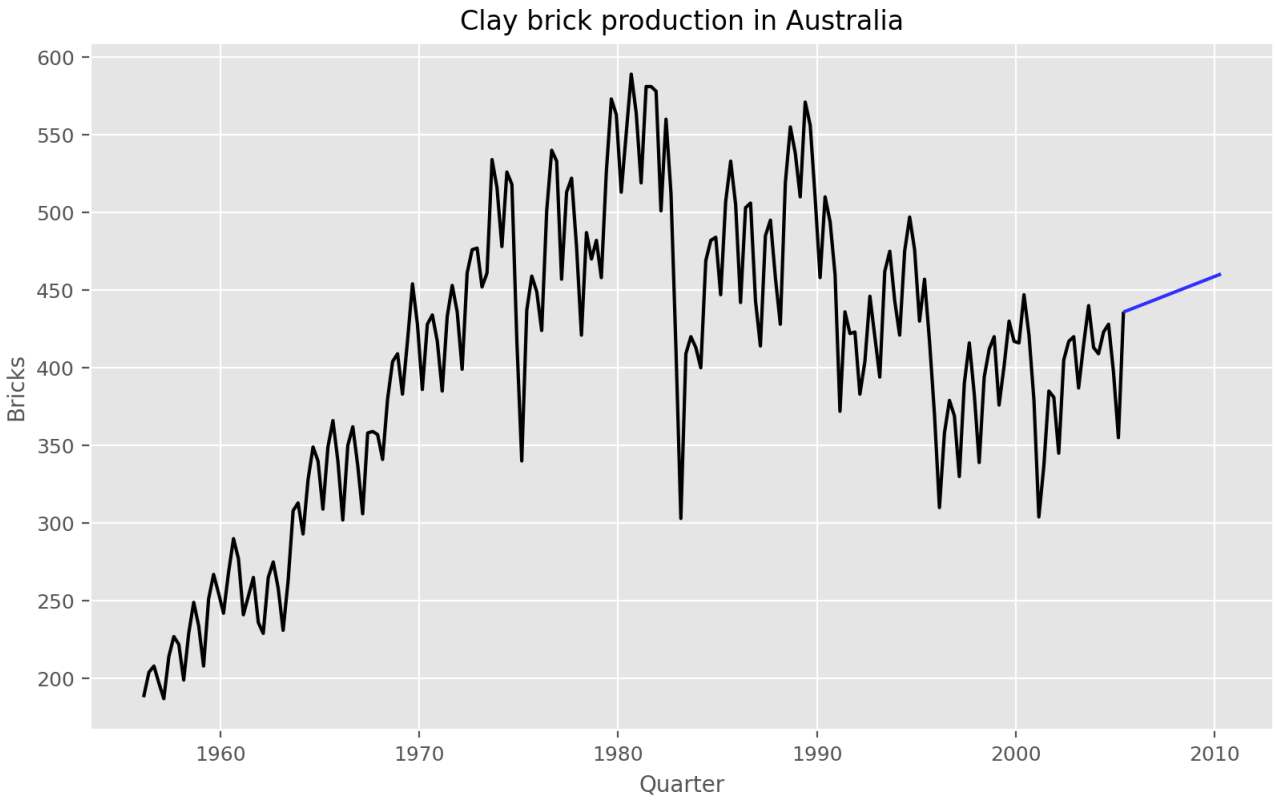

Figure 5.7: Drift forecasts applied to clay brick production in Australia.

Example: Australian quarterly beer production

Figure 5.8 shows the first three methods applied to Australian quarterly beer production from 1992 to 2006, with the forecasts compared against actual values in the next 3.5 years.

```
beers_df = production_df.query("unique_id == 'Beer' and ds >= '1992-01-01'")
train = beers_df[:-14]
test = beers_df[-14:]

avg_method = HistoricAverage()
naive_method = Naive()
seasonal_naive_method = SeasonalNaive(4)

sf = StatsForecast(
    models=[avg_method, naive_method, seasonal_naive_method],
    freq=pd.offsets.QuarterBegin(1)
)
sf.fit(train)

fcasts = sf.predict(h=14)
fcasts["y"] = test["y"].values

plot_series(train, fcasts,
            xlabel="Quarter",
            ylabel="Megalitres",
            title="Forecasts for quarterly beer production",
            rm_legend=False)
```

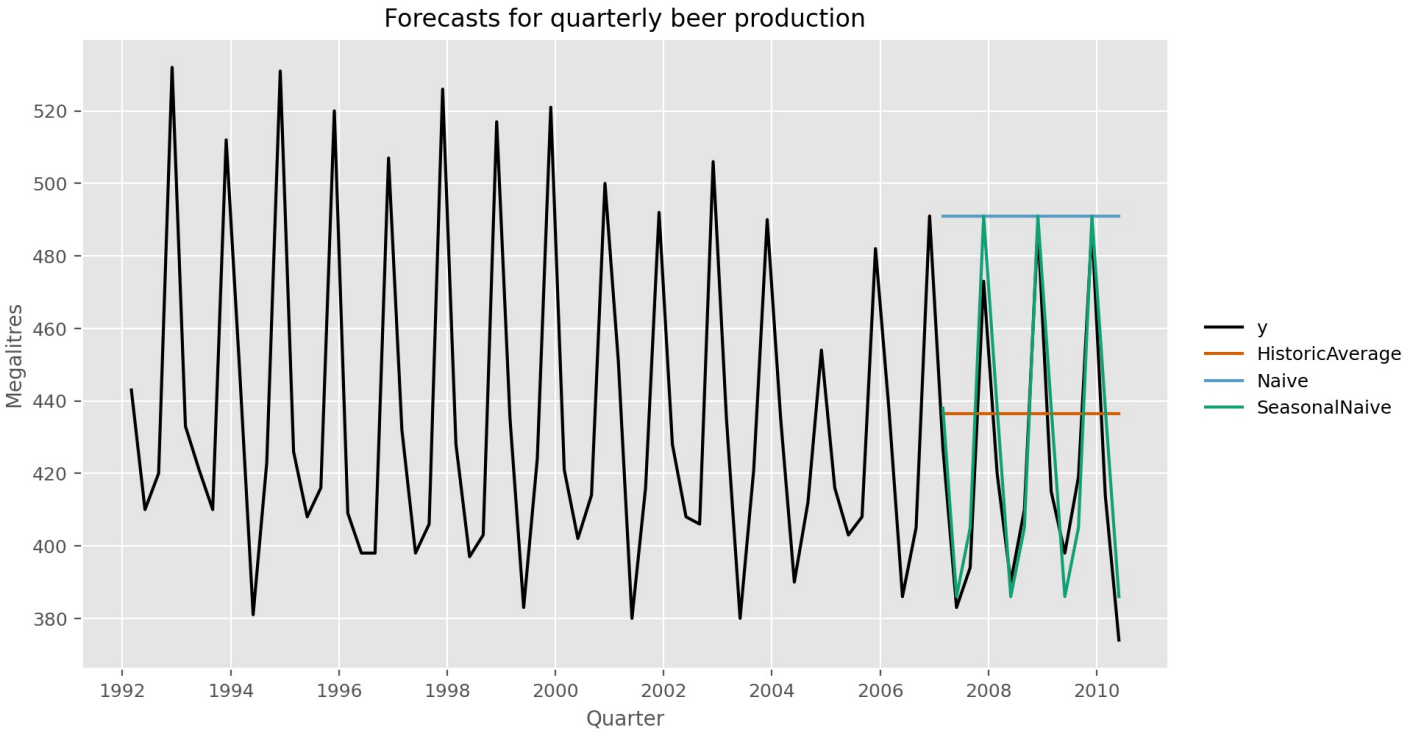

Figure 5.8: Forecasts of Australian quarterly beer production.

In this case, only the seasonal naïve forecasts are close to the observed values from 2007 onwards.

Example: Google's daily closing stock price

In Figure 5.9, the non-seasonal methods are applied to Google's daily closing stock price in 2015, and used to forecast one month ahead. Because stock prices are not observed every day, we first set up a new time index based on the trading days rather than calendar days.

```
gafa_df = pd.read_csv("../data/gafa_stock.csv", parse_dates=["ds"])
goog_df = gafa_df.query(
    "unique_id == 'GOOG_Close' and '2015-01-01' <= ds <= '2016-01-31'"
)

train = goog_df.query("ds.dt.year == 2015")
test = goog_df.query("ds.dt.year == 2016")

train["ds"] = np.arange(len(train))
test["ds"] = np.arange(len(test))

avg_method = HistoricAverage()
naive_method = Naive()
drift_method = RandomWalkWithDrift()
sf = StatsForecast(models=[drift_method, avg_method, naive_method], freq=1)
sf.fit(train)

fcasts = sf.predict(h=len(test))
fcasts["y"] = test["y"].values

plot_series(train, fcasts,
            xlabel="day",
            ylabel="$US",
            title="Google daily closing stock prices (Jan 2015-Jan 2016)",
            rm_legend=False)
```

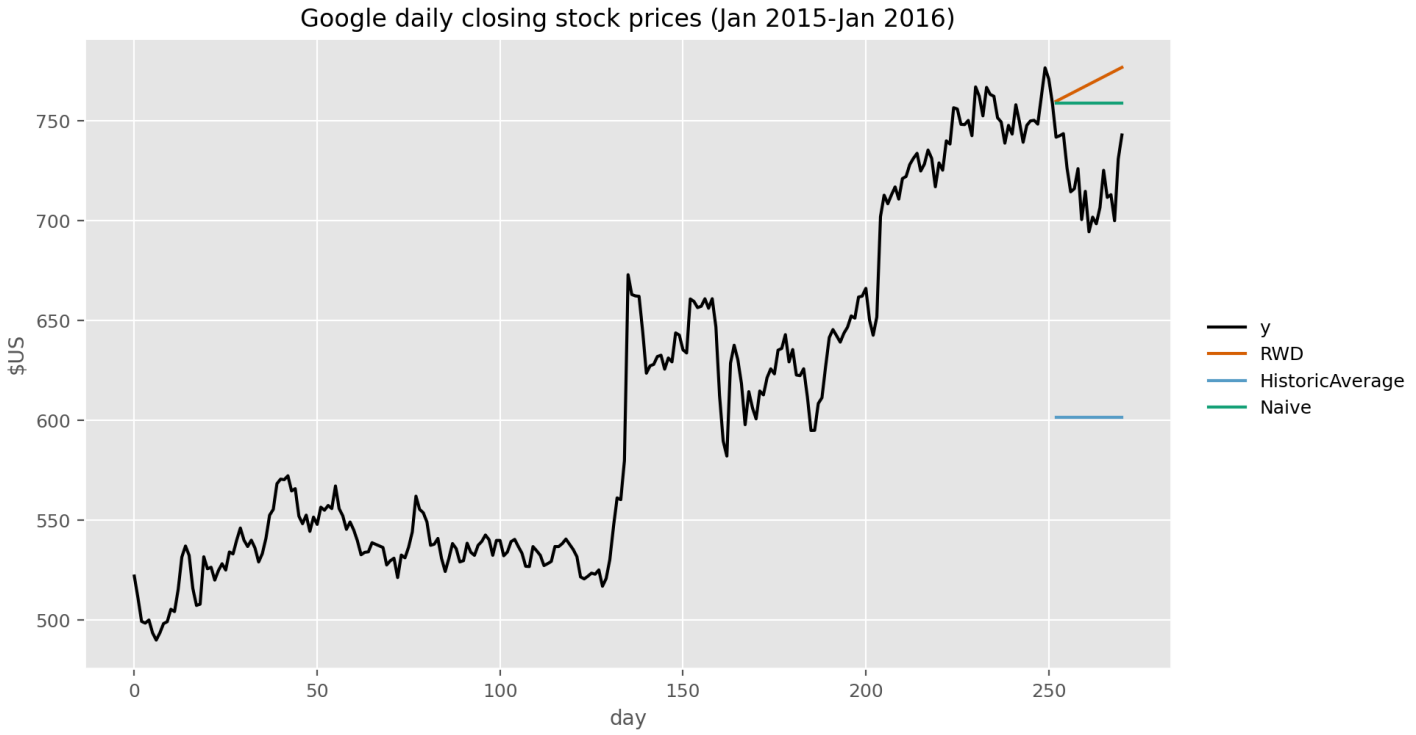

Figure 5.9: Forecasts based on Google's daily closing stock price in 2015.

Sometimes one of these simple methods will be the best forecasting method available; but in many cases, these methods will serve as benchmarks rather than the method of choice. That is, any forecasting methods we develop will be compared to these simple methods to ensure that the new method is better than these simple alternatives. If not, the new method is not worth considering.

5.3 Fitted values and residuals

Fitted values

Each observation in a time series can be forecast using all previous observations. We call these **fitted values** and they are denoted by \hat{y}_{t|t-1}, meaning the forecast of y_t based on observations y_{1},\dots,y_{t-1} . We use these so often, we sometimes drop part of the subscript and just write \hat{y}_t instead of \hat{y}_{t|t-1}. Fitted values almost always involve one-step forecasts.

Actually, fitted values are often not true forecasts because any parameters involved in the forecasting method are estimated using all available observations in the time series, including future observations. For example, if we use the mean method, the fitted values are given by \hat{y}_t = \hat{c} where \hat{c} is the average computed over all available observations, including those at times *after* t. Similarly, for the drift method, the drift parameter is estimated using all available observations. In this case, the fitted values are given by \hat{y}_t = y_{t-1} + \hat{c} where \hat{c} = (y_T-y_1)/(T-1). In both cases, there is a parameter to be estimated from the data. The "hat" above the c reminds us that this is an estimate. When the estimate of c involves observations after time t, the fitted values are not true forecasts. On the other hand, naïve or seasonal naïve forecasts do not involve any parameters, and so fitted values are true forecasts in such cases.

Residuals

The "residuals" in a time series model are what is left over after fitting a model. The residuals are equal to the difference between the observations and the corresponding fitted values: e_{t} = y_{t}-\hat{y}_{t}.

If a transformation has been used in the model, then it is often useful to look at residuals on the transformed scale. We call these "**innovation residuals**". For example, suppose we modelled the logarithms of the data, w_t = \log(y_t). Then the innovation residuals are given by w_t - \hat{w}_t whereas the regular residuals are given by y_t - \hat{y}_t. (See Section 5.6 for how to use transformations when forecasting.) If no transformation has been used then the innovation residuals are identical to the regular residuals, and in such cases we will simply call them "residuals".

The fitted valued can be obtained using the `forecast_fitted_values()` function after running `forecast()` specifying `fitted=True`. We then store those values in `fitted` column and then we can compute the residuals by taking the difference betwwen the `y` column and the `fitted` column.

```
train = beers_df[:-14]
test = beers_df[-14:]

mean_method = HistoricAverage()
sf = StatsForecast(models=[mean_method], freq="Q")
sf.forecast(h=14, df=train, fitted=True)
fitted_values = sf.forecast_fitted_values()
train["fitted"] = fitted_values["HistoricAverage"].values
train["resid"] = train["y"] - train["fitted"]
train["innov"] = train["y"] - train["fitted"]
train.head()
```

	unique_id	ds	y	fitted	resid	innov
144	Beer	1992-03-01	443.0	436.45	6.55	6.55
145	Beer	1992-06-01	410.0	436.45	-26.45	-26.45
146	Beer	1992-09-01	420.0	436.45	-16.45	-16.45
147	Beer	1992-12-01	532.0	436.45	95.55	95.55
148	Beer	1993-03-01	433.0	436.45	-3.45	-3.45

There are three new columns added to the original data:

- `fitted` contains the fitted values;
- `resid` contains the residuals;
- `innov` contains the "innovation residuals" which, in this case, are identical to the regular residuals.

Residuals are useful in checking whether a model has adequately captured the information in the data. For this purpose, we use innovation residuals.

If patterns are observable in the innovation residuals, the model can probably be improved. We will look at some tools for exploring patterns in residuals in the next section.

5.4 Residual diagnostics

A good forecasting method will yield innovation residuals with the following properties:

1. The innovation residuals are uncorrelated. If there are correlations between innovation residuals, then there is information left in the residuals which should be used in computing forecasts.
2. The innovation residuals have zero mean. If they have a mean other than zero, then the forecasts are biased.

Any forecasting method that does not satisfy these properties can be improved. However, that does not mean that forecasting methods that satisfy these properties cannot be improved. It is possible to have several different forecasting methods for the same data set, all of which satisfy these properties. Checking these properties is important in order to see whether a method is using all of the available information, but it is not a good way to select a forecasting method.

If either of these properties is not satisfied, then the forecasting method can be modified to give better forecasts. Adjusting for bias is easy: if the residuals have mean m, then simply add m to all forecasts and the bias problem is solved. Fixing the correlation problem is harder, and we will not address it until Chapter 10.

In addition to these essential properties, it is useful (but not necessary) for the residuals to also have the following two properties.

3. The innovation residuals have constant variance. This is known as "homoscedasticity".
4. The innovation residuals are normally distributed.

These two properties make the calculation of prediction intervals easier (see Section 5.5 for an example). However, a forecasting method that does not satisfy these properties cannot necessarily be improved. Sometimes applying a Box-Cox transformation may assist with these properties, but otherwise there is usually little that you can do to ensure that your innovation residuals have constant variance and a normal distribution. Instead, an alternative approach to obtaining prediction intervals is necessary. We will show how to deal with non-normal innovation residuals in Section 5.5.

Example: Forecasting Google daily closing stock prices

We will continue with the Google daily closing stock price example from Section 5.2. For stock market prices and indexes, the best forecasting method is often the naïve method. That is, each forecast is simply equal to the last observed value, or \hat{y}_{t} = y_{t-1}. Hence, the residuals are simply equal to the difference between consecutive observations: e_{t} = y_{t} - \hat{y}_{t} = y_{t} - y_{t-1}.

The following graph shows the Google daily closing stock price for trading days during 2015. The large jump corresponds to 17 July 2015 when the price jumped 16% due to unexpectedly strong second quarter results. (The `goog_df` object was created in Section 5.2.)

```
train = goog_df.query("ds.dt.year == 2015")
plot_series(train,
            xlabel="day [1]",
            ylabel="$US",
            title="Google daily closing stock prices in 2015")
```

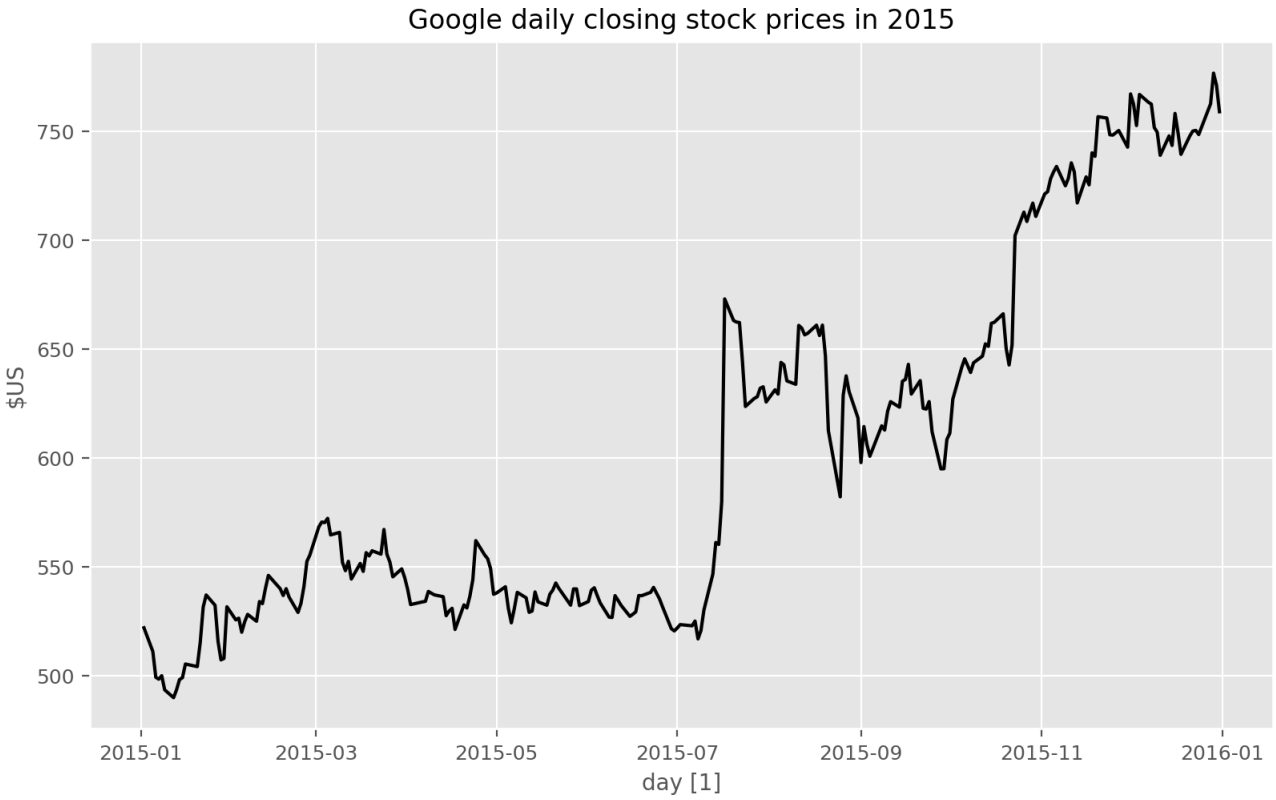

Figure 5.10: Daily Google stock prices in 2015.

The residuals obtained from forecasting this series using the naïve method are shown in Figure 5.11. The large positive residual is a result of the unexpected price jump in July.

```
train["resid"] = train["y"].diff().values
plot_series(train, target_col="resid",
            xlabel="day [1]",
            ylabel="$US",
            title="Residuals from the naive method")
```

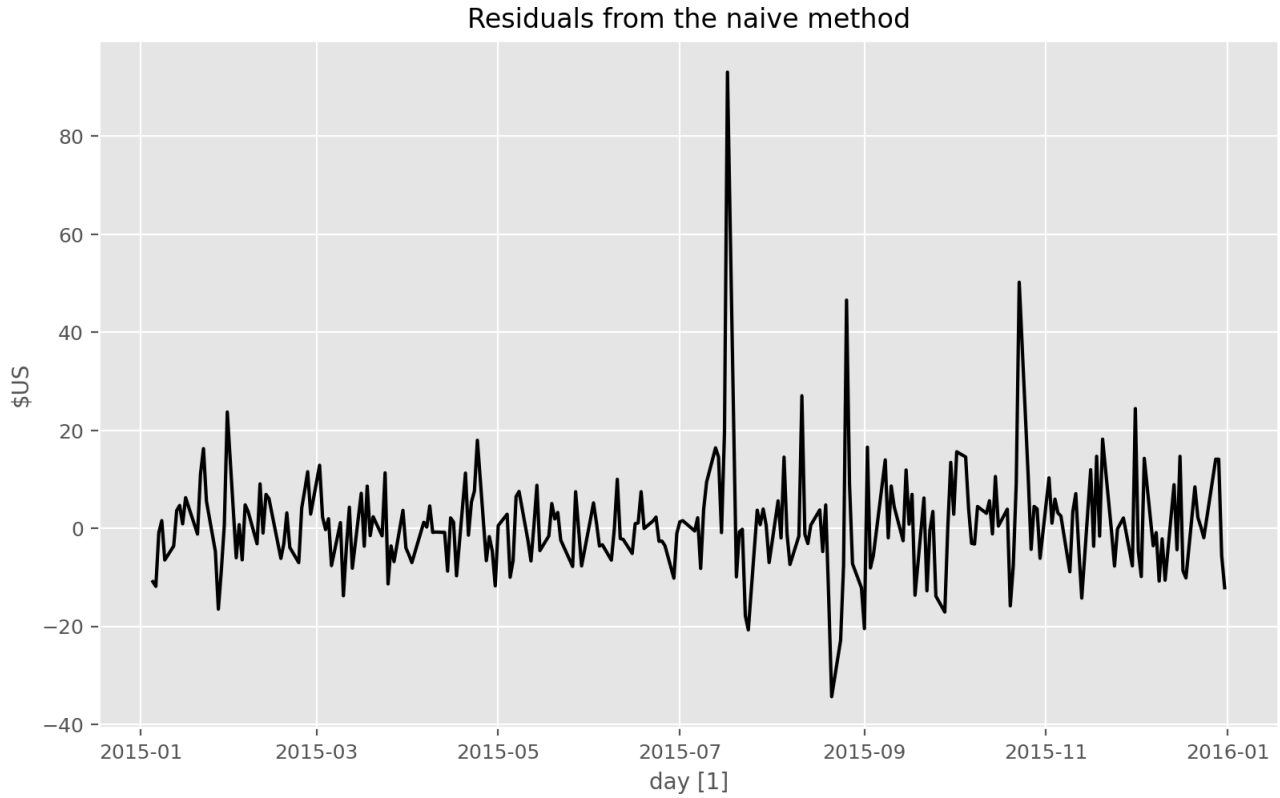

Figure 5.11: Residuals from forecasting the Google stock price using the naïve method.

```
ax = train["resid"].hist()
ax.set_title("Histogram of residuals")
plt.show()
```

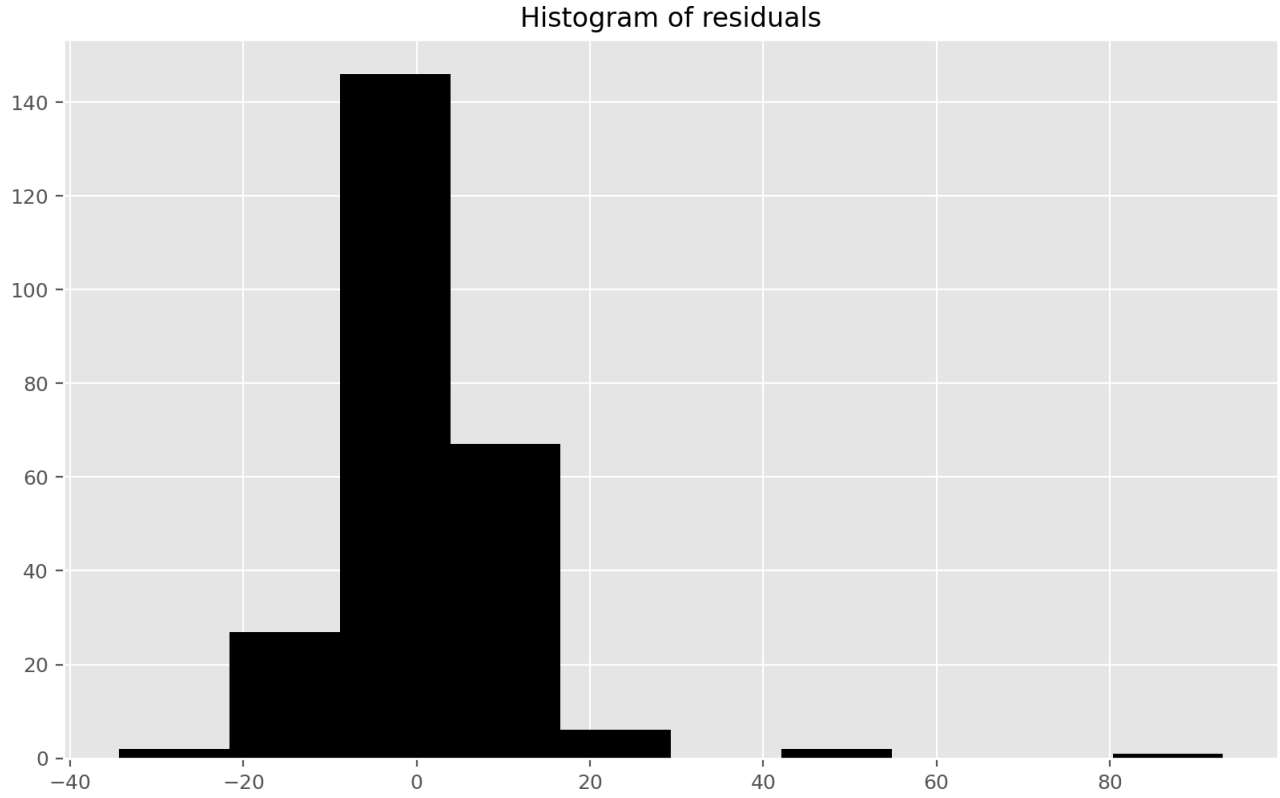

Figure 5.12: Histogram of the residuals from the naïve method applied to the Google stock price. The right tail seems a little too long for a normal distribution.

```
fig = plot_acf(train["resid"][1:], zero=False, auto_ylims=True,
         bartlett_confint=False, title="Residuals from the naïve method")
```

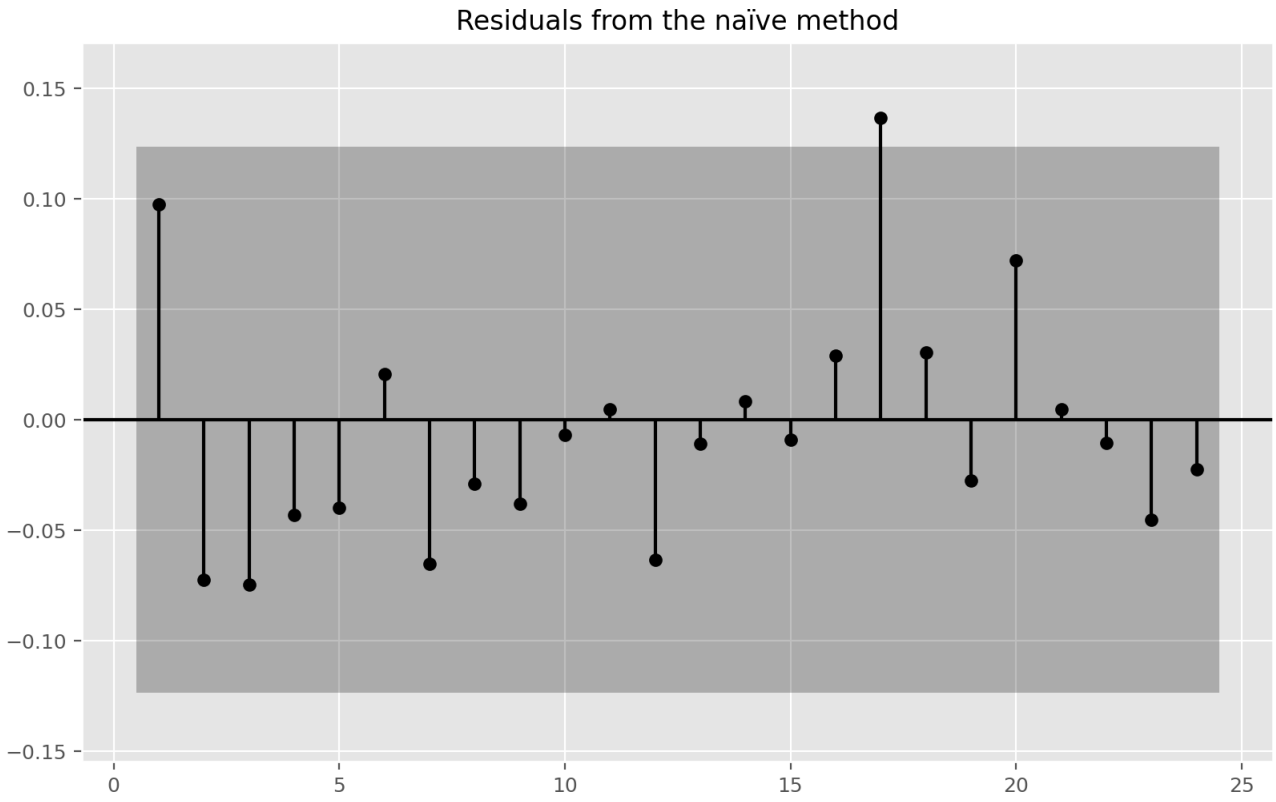

Figure 5.13: ACF of the residuals from the naïve method applied to the Google stock price. The lack of correlation suggesting the forecasts are good.

These graphs show that the naïve method produces forecasts that appear to account for all available information. The mean of the residuals is close to zero and there is no significant correlation in the residuals series. The time plot of the residuals shows that the variation of the residuals stays much the same across the historical data, apart from the one outlier, and therefore the residual variance can be treated as constant. This can also be seen on the histogram of the residuals. The histogram suggests that the residuals may not be normal — the right tail seems a little too long, even when we ignore the outlier. Consequently, forecasts from this method will probably be quite good, but prediction intervals that are computed assuming a normal distribution may be inaccurate.

A convenient shortcut for producing these residual diagnostic graphs is the `plot_diagnostics()` function, which will produce a time plot, ACF plot and histogram of the residuals.

```
def plot_diagnostics(data):
    fig = plt.figure(figsize=(8, 5))

    ax1 = fig.add_subplot(2, 2, (1, 2))
    ax1.plot(data['ds'], data["resid"])
    ax1.set_title("Innovation Residuals")

    ax2 = fig.add_subplot(2, 2, 3)
    plot_acf(data["resid"].dropna(), ax=ax2, zero=False,
      bartlett_confint=False, auto_ylims=True)
    ax2.set_title("ACF Plot")
    ax2.set_xlabel('lag[1]')

    ax3 = fig.add_subplot(2, 2, 4)
    ax3.hist(data["resid"], bins=20)
    ax3.set_title("Histogram")
    ax3.set_xlabel(".resid")
    ax3.set_ylabel("Count")

    plt.tight_layout()
    plt.show()


plot_diagnostics(train)
```

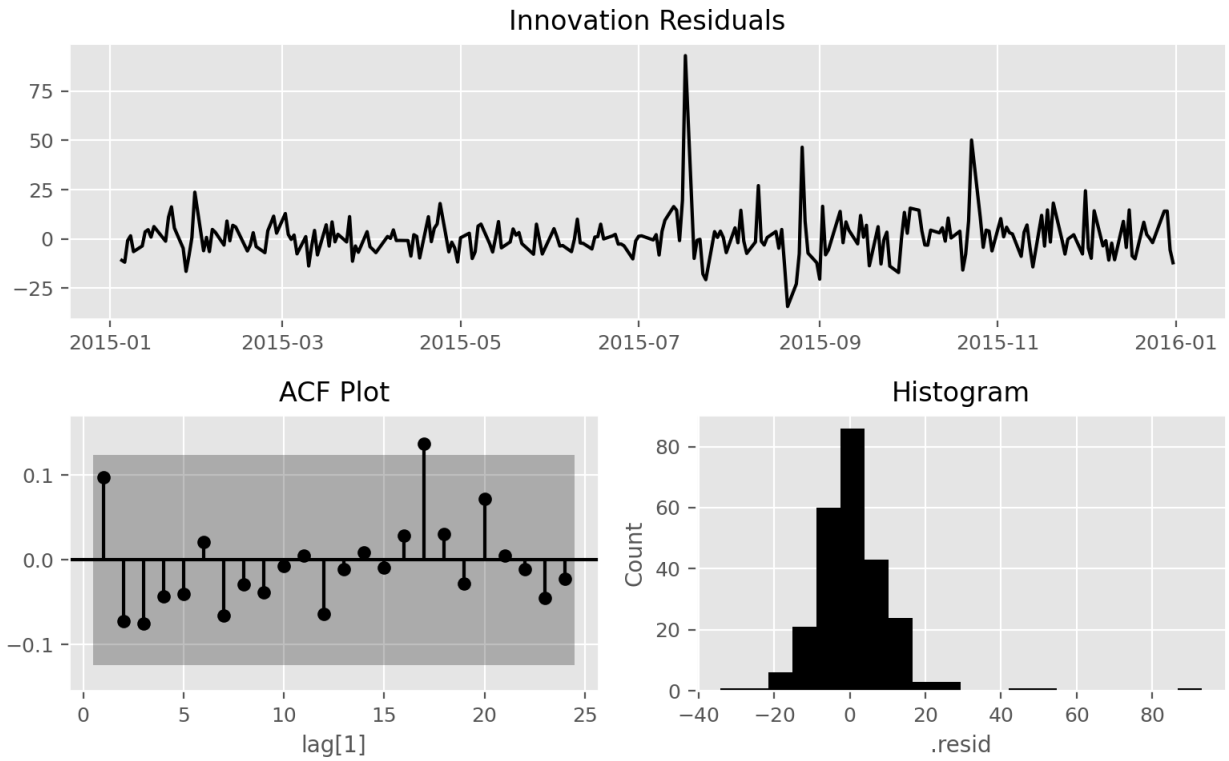

Figure 5.14: Residual diagnostic graphs for the naïve method applied to the Google stock price.

Portmanteau tests for autocorrelation

In addition to looking at the ACF plot, we can also do a more formal test for autocorrelation by considering a whole set of r_k values as a group, rather than treating each one separately.

Recall that r_k is the autocorrelation for lag k. When we look at the ACF plot to see whether each spike is within the required limits, we are implicitly carrying out multiple hypothesis tests, each one with a small probability of giving a false positive. When enough of these tests are done, it is likely that at least one will give a false positive, and so we may conclude that the residuals have some remaining autocorrelation, when in fact they do not.

In order to overcome this problem, we test whether the first \ell autocorrelations are significantly different from what would be expected from a white noise process. A test for a group of autocorrelations is called a **portmanteau test**, from a French word describing a suitcase or coat rack carrying several items of clothing.

One such test is the **Box-Pierce test**, based on the following statistic Q = T \sum_{k=1}^\ell r_k^2, where \ell is the maximum lag being considered and T is the number of observations. If each r_k is close to zero, then Q will be small. If some r_k values are large (positive or negative), then Q will be large. We suggest using \ell=10 for non-seasonal data and \ell=2m for seasonal data, where m is the period of seasonality. However, the test is not good when \ell is large, so if these values are larger than T/5, then use \ell=T/5

A related (and more accurate) test is the **Ljung-Box test**, based on Q^* = T(T+2) \sum_{k=1}^\ell (T-k)^{-1}r_k^2.

Again, large values of Q^* suggest that the autocorrelations do not come from a white noise series.

How large is too large? If the autocorrelations did come from a white noise series, then both Q and Q^* would have a \chi^2 distribution with \ell degrees of freedom.[1].

In the following code, `lag`=\ell.

```
resid_test = acorr_ljungbox(train["resid"].dropna(),
  boxpierce=True)
resid_test
```

	lb_stat	lb_pvalue	bp_stat	bp_pvalue
1	2.417	0.120	2.389	0.122
2	3.761	0.153	3.711	0.156
3	5.193	0.158	5.115	0.164
4	5.675	0.225	5.585	0.232
5	6.084	0.298	5.983	0.308
6	6.195	0.402	6.090	0.413
7	7.303	0.398	7.159	0.413
8	7.525	0.481	7.372	0.497
9	7.902	0.544	7.733	0.561
10	7.914	0.637	7.745	0.654

For both Q and Q^*, the results are not significant (i.e., the p-values are relatively large). Thus, we can conclude that the residuals are not distinguishable from a white noise series.

An alternative simple approach that may be appropriate for forecasting the Google daily closing stock price is the drift method. The `RandomWalkWithDrift()` function is used and we also return the fitted values to analyze the residuals.

```
train = goog_df.query("ds.dt.year == 2015")
drift_method = RandomWalkWithDrift()
sf = StatsForecast(models=[drift_method], freq="B")
sf.fit(train)
fcasts = sf.forecast(df=train, h=10, fitted=True)
insample_preds = sf.forecast_fitted_values()
insample_preds["resid"] = insample_preds["y"] - insample_preds["RWD"]
insample_preds
```

	unique_id	ds	y	RWD	resid
0	GOOG_Close	2015-01-02	521.938	NaN	NaN
1	GOOG_Close	2015-01-05	511.058	522.882	-11.824
2	GOOG_Close	2015-01-06	499.213	512.002	-12.789
3	GOOG_Close	2015-01-07	498.358	500.157	-1.799
4	GOOG_Close	2015-01-08	499.929	499.302	0.627
...	...	...	...	...	...
247	GOOG_Close	2015-12-24	748.400	751.254	-2.854
248	GOOG_Close	2015-12-28	762.510	749.344	13.166
249	GOOG_Close	2015-12-29	776.600	763.454	13.146
250	GOOG_Close	2015-12-30	771.000	777.544	-6.544
251	GOOG_Close	2015-12-31	758.880	771.944	-13.064

252 rows × 5 columns

Applying the Ljung-Box test, we obtain the following result.

```
resid_test = acorr_ljungbox(insample_preds["resid"].dropna(),
  boxpierce=True)
resid_test
```

	lb_stat	lb_pvalue	bp_stat	bp_pvalue
1	2.417	0.120	2.389	0.122
2	3.761	0.153	3.711	0.156
3	5.193	0.158	5.115	0.164
4	5.675	0.225	5.585	0.232
5	6.084	0.298	5.983	0.308
6	6.195	0.402	6.090	0.413
7	7.303	0.398	7.159	0.413
8	7.525	0.481	7.372	0.497
9	7.902	0.544	7.733	0.561
10	7.914	0.637	7.745	0.654

As with the naïve method, the residuals from the drift method are indistinguishable from a white noise series.

5.5 Distributional forecasts and prediction intervals

Forecast distributions

As discussed in Section 1.7, we express the uncertainty in our forecasts using a probability distribution. It describes the probability of observing possible future values using the fitted model. The point forecast is the mean of this distribution. Most time series models produce normally distributed forecasts — that is, we assume that the distribution of possible future values follows a normal distribution. We will look at a couple of alternatives to normal distributions later in this section.

Prediction intervals

A prediction interval gives an interval within which we expect y_{t} to lie with a specified probability. For example, assuming that distribution of future observations is normal, a 95% prediction interval for the h-step forecast is \hat{y}_{T+h|T} \pm 1.96 \hat\sigma_h, where \hat\sigma_h is an estimate of the standard deviation of the h-step forecast distribution.

More generally, a prediction interval can be written as \hat{y}_{T+h|T} \pm c \hat\sigma_h where the multiplier c depends on the coverage probability. In this book we usually calculate 80% intervals and 95% intervals, although any percentage may be used. Table 5.1 gives the value of c for a range of coverage probabilities assuming a normal forecast distribution.

Table 5.1: Multipliers to be used for prediction intervals.

Percentage	Multiplier
50	0.67
55	0.76
60	0.84
65	0.93
70	1.04
75	1.15
80	1.28
85	1.44
90	1.64
95	1.96
96	2.05
97	2.17
98	2.33
99	2.58

The value of prediction intervals is that they express the uncertainty in the forecasts. If we only produce point forecasts, there is no way of telling how accurate the forecasts are. However, if we also produce prediction intervals, then it is clear how much uncertainty is associated with each forecast. For this reason, point forecasts can be of almost no value without the accompanying prediction intervals.

One-step prediction intervals

When forecasting one step ahead, the standard deviation of the forecast distribution can be estimated using the standard deviation of the residuals given by \hat{\sigma} = \sqrt{\frac{1}{T-K-M}\sum_{t=1}^T e_t^2}, \tag{5.1} where K is the number of parameters estimated in the forecasting method, and M is the number of missing values in the residuals. (For example, M=1 for a naive forecast, because we can't forecast the first observation.)

For example, consider a naïve forecast for the Google stock price data `google_2015` (shown in Figure 5.9). The last value of the observed series is 758.88, so the forecast of the next value of the price is 758.88. The standard deviation of the residuals from the naïve method, as given by Equation 5.1, is 11.19. Hence, a 95% prediction interval for the next value of the GSP is

\begin{equation*} 758.88 \pm 1.96(11.19) = [736.97, 780.8]. \end{equation*}

Similarly, an 80% prediction interval is given by

\begin{equation*} 758.88 \pm 1.28(11.19) = [744.5, 773.2]. \end{equation*}

The value of the multiplier (1.96 or 1.28) is taken from Table 5.1.

Multi-step prediction intervals

A common feature of prediction intervals is that they usually increase in length as the forecast horizon increases. The further ahead we forecast, the more uncertainty is associated with the forecast, and thus the wider the prediction intervals. That is, \sigma_h usually increases with h (although there are some non-linear forecasting methods which do not have this property).

To produce a prediction interval, it is necessary to have an estimate of \sigma_h. As already noted, for one-step forecasts (h=1), Equation 5.1 provides a good estimate of the forecast standard deviation \sigma_1. For multi-step forecasts, a more complicated method of calculation is required. These calculations assume that the residuals are uncorrelated.

Benchmark methods

For the four benchmark methods, it is possible to mathematically derive the forecast standard deviation under the assumption of uncorrelated residuals. If \hat{\sigma}_h denotes the standard deviation of the h-step forecast distribution, and \hat{\sigma} is the residual standard deviation given by Equation 5.1, then we can use the expressions shown in Table 5.2. Note that when h=1 and T is large, these all give the same approximate value \hat\sigma.

Table 5.2: Multi-step forecast standard deviation for the four benchmark methods, where \sigma is the residual standard deviation, m is the seasonal period, and k is the integer part of \left(\frac{h-1}{m} \right) (i.e., the number of complete years in the forecast period prior to time T + h)

Benchmark method	h-step forecast standard deviation
Mean	\hat{\sigma}_h = \hat{\sigma} \sqrt{1 + \frac{1}{T}}
Naïve	\hat{\sigma}_h = \hat{\sigma} \sqrt(h)
Seasonal naïve	\hat{\sigma}_h = \hat{\sigma} \sqrt(k + 1)
Drift	\hat{\sigma}_h = \hat{\sigma} \sqrt{h \left(1 + \frac{h}{T - 1}\right)}

Prediction intervals can easily be computed for you when using the `statsforecast` package. For example, here is the output when using the naïve method for the Google stock price.

```
train = goog_df.query("ds.dt.year == 2015")
naive_method = Naive()
sf = StatsForecast(models=[naive_method], freq="B")
fcasts = sf.forecast(df=train, h=10, level=[80, 95])
fcasts
```

	unique_id	ds	Naive	Naive-lo-80	Naive-lo-95	Naive-hi-80	Naive-hi-95
0	GOOG_Close	2016-01-01	758.88	744.540	736.949	773.220	780.811
1	GOOG_Close	2016-01-04	758.88	738.600	727.865	779.160	789.895
2	GOOG_Close	2016-01-05	758.88	734.042	720.894	783.718	796.866
3	GOOG_Close	2016-01-06	758.88	730.200	715.018	787.560	802.742
4	GOOG_Close	2016-01-07	758.88	726.815	709.840	790.945	807.920
5	GOOG_Close	2016-01-08	758.88	723.754	705.160	794.006	812.600
6	GOOG_Close	2016-01-11	758.88	720.940	700.856	796.820	816.904
7	GOOG_Close	2016-01-12	758.88	718.320	696.849	799.440	820.911
8	GOOG_Close	2016-01-13	758.88	715.860	693.086	801.900	824.674
9	GOOG_Close	2016-01-14	758.88	713.533	689.528	804.227	828.232

The `level` argument allows us to specify the prediction intervals. Here, 80% and 95% prediction intervals are returned.

When plotted, the prediction intervals are shown as shaded regions, with the strength of colour indicating the probability associated with the interval. Again, 80% and 95% intervals are shown by default, with other options available via the `level` argument.

```
train["ds"] = np.arange(len(train))
fcasts["ds"] = train["ds"].max() + np.arange(start=1, stop=len(fcasts) + 1)
plot_series(train, fcasts, level=[80, 95],
            xlabel="day",
            ylabel="$US",
            title="Google daily closing stock price",
            rm_legend=False)
```

Figure 5.15: 80% and 95% prediction intervals for the Google closing stock price based on a naïve method.

```
sf = StatsForecast(
    models = [Naive()],
    freq = 1
)
fc = sf.forecast(df=train, h=30, fitted=True)
fitted = sf.forecast_fitted_values()
errors = fitted['y']-fitted['Naive']
errors = errors[1:].to_list() # first value is NaN
last_obs = train.iloc[-1, -1]

num_sims = 5000
sim = np.zeros((30,num_sims))

for j in range(num_sims):
    e = random.sample(errors, 1).pop()
    yhat = last_obs+e

    sim[0,j] = yhat

    for i in range(1,30):
        e = random.sample(errors, 1).pop()
        sim[i,j] = sim[i-1,j]+e

sim_df = pd.DataFrame(sim)
sim_df.columns = [f"sim_{i}" for i in range(num_sims)]
sim_df['unique_id'] = 'GOOG_Close'
sim_df['ds'] = train["ds"].max() + np.arange(start=1, stop=len(sim_df) + 1)
plot_series(train, sim_df,
            models = ['sim_0', 'sim_1', 'sim_2', 'sim_3', 'sim_4'],
            xlabel="day",
            ylabel="$US",
            title="Google daily closing stock price")
```

Figure 5.16: Five simulated future sample paths of the Google closing stock price based on a naïve method with bootstrapped residuals.

Then we can compute prediction intervals by calculating percentiles of the future sample paths for each forecast horizon. The result is called a **bootstrapped** prediction interval. The name "bootstrap" is a reference to pulling ourselves up by our bootstraps, because the process allows us to measure future uncertainty by only using the historical data.

Notice that the forecast distribution is now represented as a simulation with 5000 sample paths. Because there is no normality assumption, the prediction intervals are not symmetric. The `bootstrap` column is the mean of the bootstrap samples, so it may be slightly different from the results obtained without a bootstrap (the `Naive` method from `Statsforecast`).

```
sim_df['bootstrap'] = sim_df.drop(['unique_id', 'ds'], axis=1).mean(axis=1)
sim_df['bootstrap-lo-80'] = \
  sim_df.drop(['unique_id', 'ds'], axis=1).quantile(0.1, axis=1)
sim_df['bootstrap-hi-80'] = \
  sim_df.drop(['unique_id', 'ds'], axis=1).quantile(0.9, axis=1)
sim_df['bootstrap-lo-95'] = \
  sim_df.drop(['unique_id', 'ds'], axis=1).quantile(0.025, axis=1)
sim_df['bootstrap-hi-95'] = \
  sim_df.drop(['unique_id', 'ds'], axis=1).quantile(0.975, axis=1)

plot_series(train, sim_df, models=['bootstrap'], level=[80, 95],
            xlabel="day",
            ylabel="$US",
            title="Google daily closing stock price",
            rm_legend=False)
```

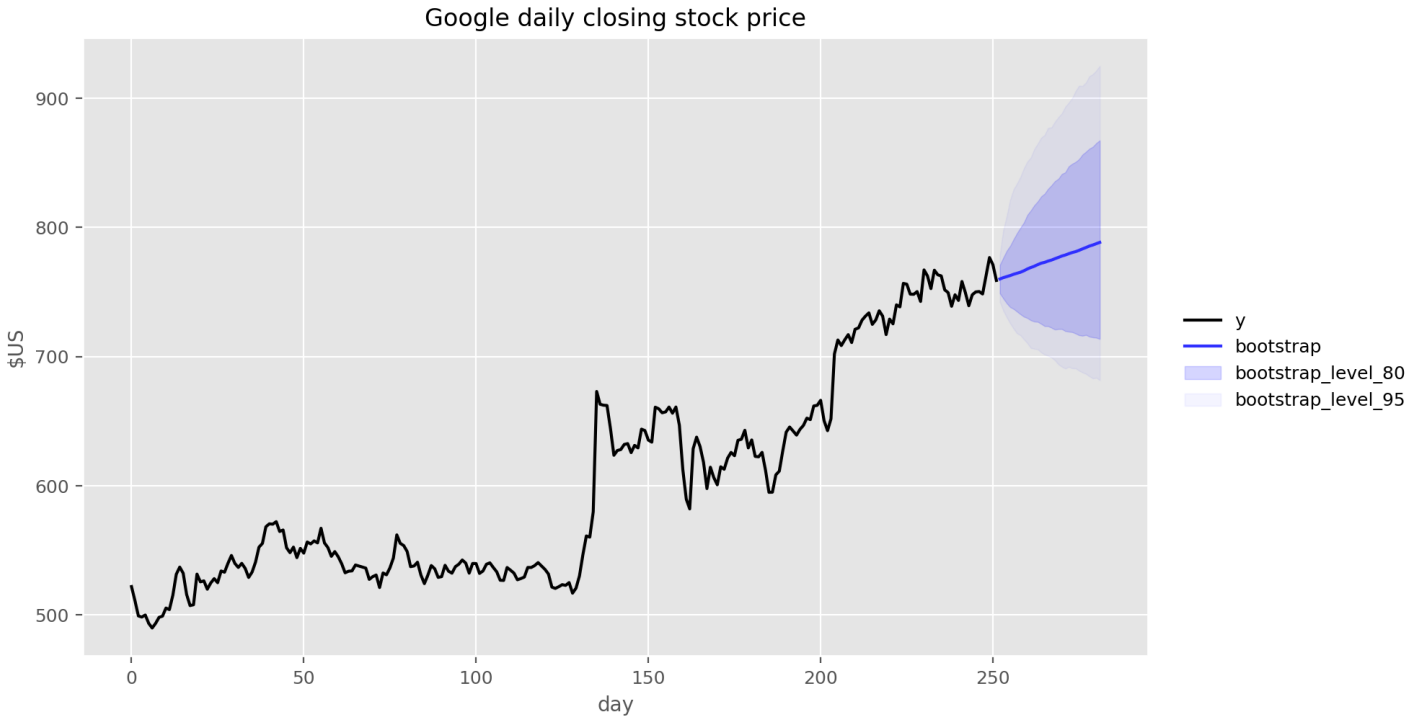

Figure 5.17: Forecasts of the Google closing stock price based on a naïve method with bootstrapped residuals.

Conformal prediction for distribution-free prediction intervals

While traditional prediction intervals rely on assumptions about the forecast distribution—typically assuming normality—**conformal prediction** offers a flexible, model-agnostic alternative that does not require strong distributional assumptions. Instead of estimating prediction intervals based on a predefined probabilistic model, conformal prediction constructs intervals using past forecast errors, ensuring that the empirical coverage closely matches the desired confidence level.

Constructing conformal prediction intervals

Conformal prediction intervals are typically built using past residuals to calibrate future uncertainty. These residuals are calculated from a **calibration set**, which consists of h-step-ahead errors, defined as e_{t+h|t} = y_{t+h} - \hat{y}_{t+h|t}, that serve as a basis for estimating future uncertainty of h-step-ahead forecasts. For h=1, these are the same as the residuals discussed in Section 5.3.

A key assumption in conformal prediction as presented in this section is **exchangeability**—that is, the distribution of past residuals is representative of future errors. This assumption allows us to estimate the distribution of forecast errors using their **absolute values**, ensuring that the constructed intervals provide valid empirical coverage even when the underlying distribution is unknown. While this exchangeability assumption underpins most classical conformal approaches, recent research has explored methods that go beyond exchangeability, enabling applications in settings with data distribution drifts (Barber et al. 2023).

A **simple split conformal method** for constructing a (1-\alpha) prediction interval at horizon h is:

1. **Fit a forecasting model** to historical data and compute residuals from past observations.
2. **Estimate a prediction interval** using the empirical quantiles of past absolute residuals. For a given confidence level (1 - \alpha), the conformal prediction interval is: \hat{y}_{T+h|T} \pm Q_{1-\alpha} (|e_{t+h|t}|),

where Q_{1-\alpha}(|e_{t+h|t}|) is the (1-\alpha) quantile of the past absolute residuals for the h-step-ahead error.

This approach ensures that the constructed prediction intervals have empirical coverage close to (1-\alpha), even when the underlying forecast distribution is unknown or does not follow a normal distribution (Stankeviciute, Alaa, and Schaar 2021).

5.6 Forecasting using transformations

Some common transformations which can be used when modelling were discussed in Section 3.1. When forecasting from a model with transformations, we first produce forecasts of the transformed data. Then, we need to reverse the transformation (or *back-transform*) to obtain forecasts on the original scale. For Box-Cox transformations given by Equation 3.1, the reverse transformation is given by y_{t} = \begin{cases} \exp(w_{t}) & \text{if $\lambda=0$};\\ \text{sign}(\lambda w_t+1)|\lambda w_t+1|^{1/\lambda} & \text{otherwise}. \end{cases} \tag{5.2}

Prediction intervals with transformations

If a transformation has been used, then the prediction interval is first computed on the transformed scale, and the end points are back-transformed to give a prediction interval on the original scale. This approach preserves the probability coverage of the prediction interval, although it will no longer be symmetric around the point forecast.

Transformations sometimes make little difference to the point forecasts but have a large effect on prediction intervals.

Bias adjustments

One issue with using mathematical transformations such as Box-Cox transformations is that the back-transformed point forecast will not be the mean of the forecast distribution. In fact, it will usually be the median of the forecast distribution (assuming that the distribution on the transformed space is symmetric). For many purposes, this is acceptable, although the mean is usually preferable. For example, you may wish to add up sales forecasts from various regions to form a forecast for the whole country. But medians do not add up, whereas means do.

For a Box-Cox transformation, the back-transformed mean is given (approximately) by \hat{y}_{T+h|T} = \begin{cases} \exp(\hat{w}_{T+h|T})\left[1 + \frac{\sigma_h^2}{2}\right] & \text{if $\lambda=0$;}\\ (\lambda \hat{w}_{T+h|T}+1)^{1/\lambda}\left[1 + \frac{\sigma_h^2(1-\lambda)}{2(\lambda \hat{w}_{T+h|T}+1)^{2}}\right] & \text{otherwise;} \end{cases} \tag{5.3} where \hat{w}_{T+h|T} is the h-step forecast mean and \sigma_h^2 is the h-step forecast variance on the transformed scale. The larger the forecast variance, the bigger the difference between the mean and the median.

The difference between the simple back-transformed forecast given by Equation 5.2 and the mean given by Equation 5.3 is called the **bias**. When we use the mean, rather than the median, we say the point forecasts have been **bias-adjusted**.

To see how much difference this bias-adjustment makes, consider the following example, where we forecast the average annual price of eggs using the drift method with a log transformation (\lambda=0). The log transformation is useful in this case to ensure the forecasts and the prediction intervals stay positive.

```
egg_df = pd.read_csv("../data/eggs.csv", parse_dates=["ds"])
egg_df_log = egg_df.copy()
egg_df_log['y'] = np.log(egg_df_log['y'])

rwd_method = RandomWalkWithDrift()
sf = StatsForecast(models=[rwd_method], freq="Y")
sf.fit(egg_df_log)
fcasts = sf.forecast(df=egg_df_log, h=50, level=[80, 95])
sigma_h = (fcasts['RWD-hi-80'] - fcasts['RWD-lo-80']) / (2 * 1.28)
sigma_h_squared = sigma_h**2
bias_adjusted = np.exp(fcasts['RWD']) * (1 + sigma_h_squared/2)
fcasts_original = fcasts.copy()
columns_to_transform = \
  ['RWD', 'RWD-lo-80', 'RWD-lo-95', 'RWD-hi-80', 'RWD-hi-95']
for col in columns_to_transform:
    fcasts_original[col] = np.exp(fcasts[col])
fcasts_original['RWD-adjusted'] = bias_adjusted

fig, ax = plt.subplots()
ax.plot(egg_df['ds'], egg_df['y'], color='black', label='Historical',
  linewidth=1)
ax.fill_between(fcasts_original['ds'],
                fcasts_original['RWD-lo-80'],
                fcasts_original['RWD-hi-80'],
                alpha=0.3, color='#B8BFFF', label='80% CI')
ax.plot(fcasts_original['ds'], fcasts_original['RWD'],
        color='blue', linestyle='--', label='Median forecast', linewidth=1)
ax.plot(fcasts_original['ds'], fcasts_original['RWD-adjusted'],
        color='blue', linestyle='-', label='Bias-adjusted mean', linewidth=1)
ax.set_title('Annual egg prices')
ax.set_xlabel('year')
ax.set_ylabel('$US (in cents adjusted for inflation)')
handles, labels = fig.axes[0].get_legend_handles_labels()
fig.legend(
  handles,
  labels,
  loc="center left",
  bbox_to_anchor=(1.02, 0.5),
  borderaxespad=0.0,
  frameon=False,
)
plt.show()
```

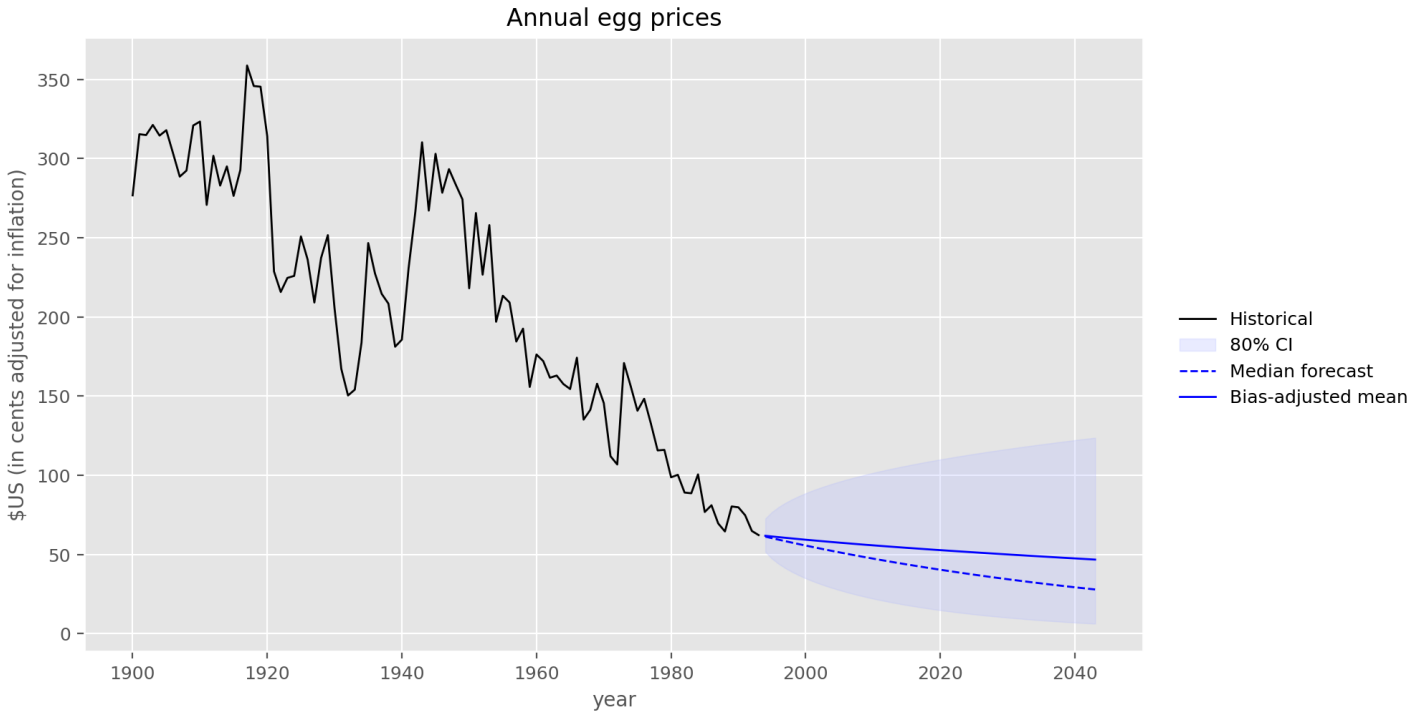

Figure 5.18: Forecasts of egg prices using the drift method applied to the logged data.

Notice how the skewed forecast distribution pulls up the forecast distribution's mean.

5.7 Forecasting with decomposition

Time series decomposition (discussed in Chapter 3) can be a useful step in producing forecasts.

Assuming an additive decomposition, the decomposed time series can be written as y_t = \hat{S}_t + \hat{A}_t, where \hat{A}_t = \hat{T}_t+\hat{R}_{t} is the seasonally adjusted component. Or, if a multiplicative decomposition has been used, we can write y_t = \hat{S}_t\hat{A}_t, where \hat{A}_t = \hat{T}_t\hat{R}_{t}.

To forecast a decomposed time series, we forecast the seasonal component, \hat{S}_t, and the seasonally adjusted component \hat{A}_t, separately. It is usually assumed that the seasonal component is unchanging, or changing extremely slowly, so it is forecast by simply taking the last year of the estimated component. In other words, a seasonal naïve method is used for the seasonal component.

To forecast the seasonally adjusted component, any non-seasonal forecasting method may be used. For example, the drift method, or Holt's method (discussed in Chapter 8), or a non-seasonal ARIMA model (discussed in Chapter 9), may be used.

Example: Employment in the US retail sector

```
us_employment_df = pd.read_csv("../data/us_employment_formatted.csv",
  parse_dates=["ds"])
us_retail_employment_df = us_employment_df.query(
    "Title == 'Retail Trade' and ds.dt.year >= 1992"
)

train_df = us_retail_employment_df.drop(["Title"], axis=1)

stl = STL(train_df["y"].values, period=12, robust=True, trend_deg=7)
res = stl.fit()

train_df["seasonal"] = res.seasonal
train_df["y_adjusted"] = train_df["y"] - train_df["seasonal"]

naive_method = Naive()
sf = StatsForecast(models=[naive_method], freq="M")
adjusted_fcasts = sf.forecast(h=24, level=[80, 95], df=train_df,
                              target_col="y_adjusted", fitted=True)

adjusted_fcasts_fitted = sf.forecast_fitted_values()
plot_series(train_df, adjusted_fcasts, target_col="y_adjusted",
            level=[80, 95],
            xlabel="Month",
            ylabel="Number of people",
            title="US retail employment",
            rm_legend=False)
```

Figure 5.19: Naïve forecasts of the seasonally adjusted data obtained from an STL decomposition of the total US retail employment.

Figure 5.19 shows naïve forecasts of the seasonally adjusted US retail employment data. These are then "reseasonalised" by adding in the seasonal naïve forecasts of the seasonal component.

This is made easy with the `STL()` function, which allows you to compute forecasts via any additive decomposition, using other model functions to forecast each of the decomposition's components. Seasonal components of the model will be forecast automatically using `SeasonalNaive()` if a different model isn't specified. The function will also do the reseasonalising for you, ensuring that the resulting forecasts of the original data are obtained. These are shown in Figure 5.20.

```
seasonal_naive = SeasonalNaive(12)
sf = StatsForecast(models=[seasonal_naive], freq="M")
seasonal_fcasts = sf.forecast(h=24, level=[80, 95], df=train_df,
  target_col="seasonal", fitted=True)
merged_df = adjusted_fcasts.merge(seasonal_fcasts, on=['unique_id', 'ds'],
  how='inner')
merged_df_fitted = adjusted_fcasts_fitted.merge(sf.forecast_fitted_values(),
  on=['unique_id', 'ds'], how='inner')
merged_df_fitted["combined"] = merged_df_fitted['Naive'] + \
  merged_df_fitted['SeasonalNaive']
merged_df_fitted = merged_df_fitted.merge(train_df, on=["unique_id", "ds"],
  how="right")
final_df = merged_df[['unique_id', 'ds']].copy()  # Start with identifiers
final_df['combined'] = merged_df['Naive'] + merged_df['SeasonalNaive']
train_df["resid"] = merged_df_fitted["y"].values - \
  merged_df_fitted["Naive"].values - \
  merged_df_fitted["SeasonalNaive"].values
final_df['combined-lo-80'] = merged_df['Naive-lo-80'] + \
  merged_df['SeasonalNaive-lo-80']
final_df['combined-lo-95'] = merged_df['Naive-lo-95'] + \
  merged_df['SeasonalNaive-lo-95']
final_df['combined-hi-80'] = merged_df['Naive-hi-80'] + \
  merged_df['SeasonalNaive-hi-80']
final_df['combined-hi-95'] = merged_df['Naive-hi-95'] + \
  merged_df['SeasonalNaive-hi-95']

plot_series(train_df, final_df, target_col="y", level=[80, 95],
            xlabel="Month",
            ylabel="Number of people",
            title="US retail employment",
            rm_legend=False)
```

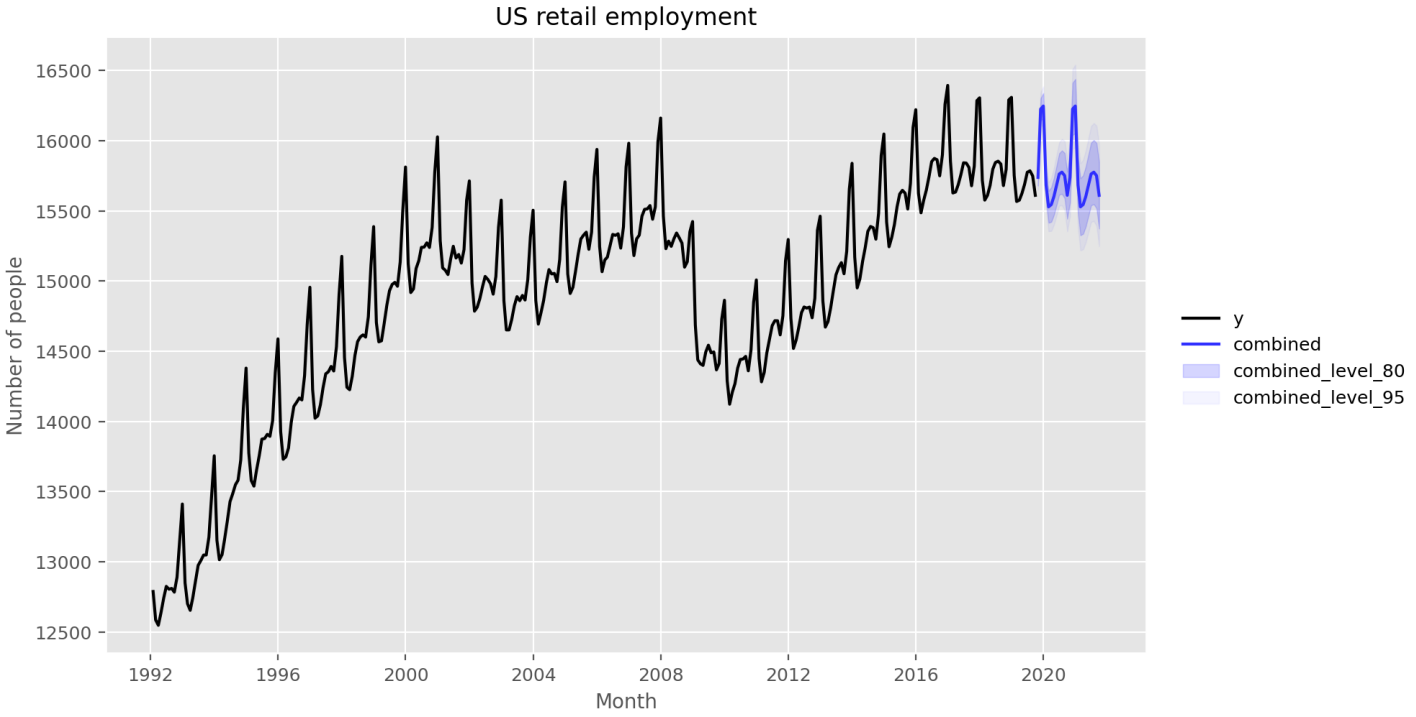

Figure 5.20: Forecasts of the total US retail employment data based on a naïve forecast of the seasonally adjusted data and a seasonal naïve forecast of the seasonal component, after an STL decomposition of the data.

The prediction intervals shown in this graph are constructed in the same way as the point forecasts. That is, the upper and lower limits of the prediction intervals on the seasonally adjusted data are "reseasonalised" by adding in the forecasts of the seasonal component.

The ACF of the residuals, shown in Figure 5.21, displays significant autocorrelations. These are due to the naïve method not

capturing the changing trend in the seasonally adjusted series.

```
plot_diagnostics(train_df)
```

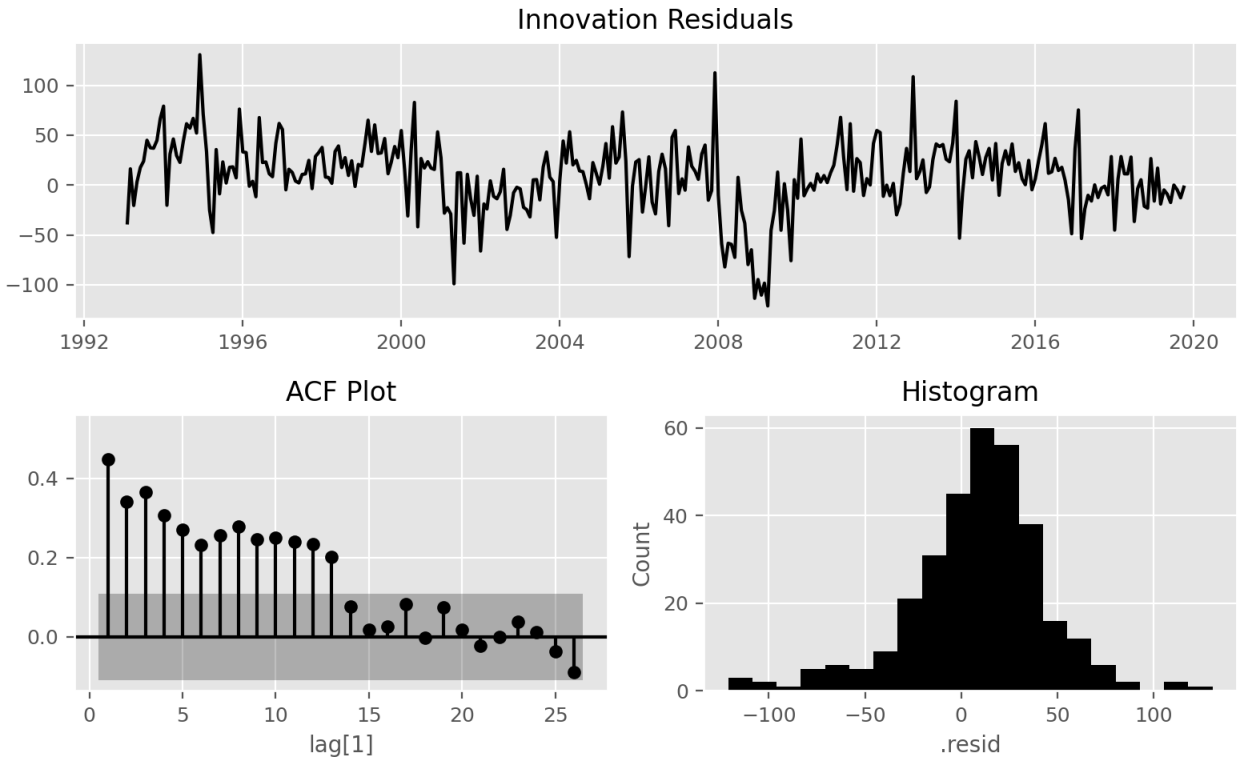

Figure 5.21: Checking the residuals.

In subsequent chapters we study more suitable methods that can be used to forecast the seasonally adjusted component instead of the naïve method.

5.8 Evaluating point forecast accuracy

Training and test sets

It is important to evaluate forecast accuracy using genuine forecasts. Consequently, the size of the residuals is not a reliable indication of how large true forecast errors are likely to be. The accuracy of forecasts can only be determined by considering how well a model performs on new data that were not used when fitting the model.

When choosing models, it is common practice to separate the available data into two portions, **training** and **test** data, where the training data is used to estimate any parameters of a forecasting method and the test data is used to evaluate its accuracy. Because the test data is not used in determining the forecasts, it should provide a reliable indication of how well the model is likely to forecast on new data.

Figure 5.22: Train/test split

The size of the test set is typically about 20% of the total sample, although this value depends on how long the sample is and how far ahead you want to forecast. The test set should ideally be at least as large as the maximum forecast horizon required. The following points should be noted.

- A model which fits the training data well will not necessarily forecast well.
- A perfect fit can always be obtained by using a model with enough parameters.
- Over-fitting a model to data is just as bad as failing to identify a systematic pattern in the data.

Some references describe the test set as the “hold-out set” because these data are “held out” of the data used for fitting. Other references call the training set the “in-sample data” and the test set the “out-of-sample data”. We prefer to use “training data” and

“test data” in this book.

Functions to subset a time series

The `query()` function is useful when extracting a portion of a time series, such as we need when creating training and test sets. When splitting data into evaluation sets, filtering the index of the data is particularly useful. For example,

```
filtered_df = aus_production.query("ds.dt.year >= 1995")
```

Another useful function is slicing, which allows the use of indices to choose a subset from each group. For example,

```
filtered_df = aus_production[:-20]
```

extracts the last 20 observations (5 years).

Forecast errors

A forecast “error” is the difference between an observed value and its forecast. Here “error” does not mean a mistake, it means the unpredictable part of an observation. It can be written as e_{T+h} = y_{T+h} - \hat{y}_{T+h|T}, where the training data is given by \{y_1,\dots,y_T\} and the test data is given by \{y_{T+1},y_{T+2},\dots\}.

Note that forecast errors are different from residuals in two ways. First, residuals are calculated on the *training* set while forecast errors are calculated on the *test* set. Second, residuals are based on *one-step* forecasts while forecast errors can involve *multi-step* forecasts.

We can measure forecast accuracy by summarising the forecast errors in different ways.

Scale-dependent errors

The forecast errors are on the same scale as the data. Accuracy measures that are based only on e_{t} are therefore scale-dependent and cannot be used to make comparisons between series that involve different units.

The two most commonly used scale-dependent measures are based on the absolute errors or squared errors: \begin{align*} \text{Mean absolute error: MAE} & = \text{mean}(|e_{t}|),\\ \text{Root mean squared error: RMSE} & = \sqrt{\text{mean}(e_{t}^2)}. \end{align*} When comparing forecast methods applied to a single time series, or to several time series with the same units, the MAE is popular as it is easy to both understand and compute. A forecast method that minimises the MAE will lead to forecasts of the median, while minimising the RMSE will lead to forecasts of the mean. Consequently, the RMSE is also widely used, despite being more difficult to interpret.

Percentage errors

The percentage error is given by p_{t} = 100 e_{t}/y_{t}. Percentage errors have the advantage of being unit-free, and so are frequently used to compare forecast performances between data sets. The most commonly used measure is: \text{Mean absolute percentage error: MAPE} = \text{mean}(|p_{t}|). Measures based on percentage errors have the disadvantage of being infinite or undefined if y_{t}=0 for any t in the period of interest, and having extreme values if any y_{t} is close to zero. Another problem with percentage errors that is often overlooked is that they assume the unit of measurement has a meaningful zero.[2] For example, a percentage error makes no sense when measuring the accuracy of temperature forecasts on either the Fahrenheit or Celsius scales, because temperature has an arbitrary zero point.

They also have the disadvantage that they put a heavier penalty on negative errors than on positive errors. This observation led to the use of the so-called “symmetric” MAPE (sMAPE) proposed by Armstrong (1978, 348), which was used in the M3 forecasting competition. It is defined by \text{sMAPE} = \text{mean}\left(200|y_{t} - \hat{y}_{t}|/(y_{t}+\hat{y}_{t})\right). However, if y_{t} is close to zero, \hat{y}_{t} is also likely to be close to zero. Thus, the measure still involves division by a number close to zero, making the calculation unstable. Also, the value of sMAPE can be negative, so it is not really a measure of “absolute percentage errors” at all.

Hyndman and Koehler (2006) recommend that the sMAPE not be used. It is included here only because it is widely used, although we will not use it in this book.

Scaled errors

Scaled errors were proposed by Hyndman and Koehler (2006) as an alternative to using percentage errors when comparing forecast accuracy across series with different units. They proposed scaling the errors based on the *training* MAE from a simple forecast method.

For a non-seasonal time series, a useful way to define a scaled error uses naïve forecasts: q_{j} = \frac{\displaystyle e_{j}}{\displaystyle\frac{1}{T-1}\sum_{t=2}^T |y_{t}-y_{t-1}|}. Because the numerator and denominator both involve values on the scale of the original data, q_{j} is independent of the scale of the data. A scaled error is less than one if it arises from a better forecast than the average one-step naïve forecast computed on the training data. Conversely, it is greater than one if the forecast is worse than the average one-step naïve forecast computed on the training data.

For seasonal time series, a scaled error can be defined using seasonal naïve forecasts: q_{j} = \frac{\displaystyle e_{j}}{\displaystyle\frac{1}{T-m}\sum_{t=m+1}^T |y_{t}-y_{t-m}|}.

The *mean absolute scaled error* is simply \text{MASE} = \text{mean}(|q_{j}|). Similarly, the *root mean squared scaled error* is given by \text{RMSSE} = \sqrt{\text{mean}(q_{j}^2)}, where q^2_{j} = \frac{\displaystyle e^2_{j}}{\displaystyle\frac{1}{T-m}\sum_{t=m+1}^T (y_{t}-y_{t-m})^2}, and we set m=1 for non-seasonal data.

Examples

```
train_df = beers_df.query("ds.dt.year <= 2007")
test_df = beers_df.query("ds.dt.year > 2007")

mean_method = HistoricAverage()
naive_method = Naive()
drift_method = RandomWalkWithDrift()
seasonal_naive = SeasonalNaive(4)

sf = StatsForecast(
    models=[drift_method, mean_method, naive_method, seasonal_naive],
    freq="Q"
)
preds = sf.forecast(df=train_df, h=10)

plot_series(beers_df, preds,
            xlabel="Quarter",
            ylabel="Megalitres",
            title="Forecasts for quarterly beer production",
            rm_legend=False)
```

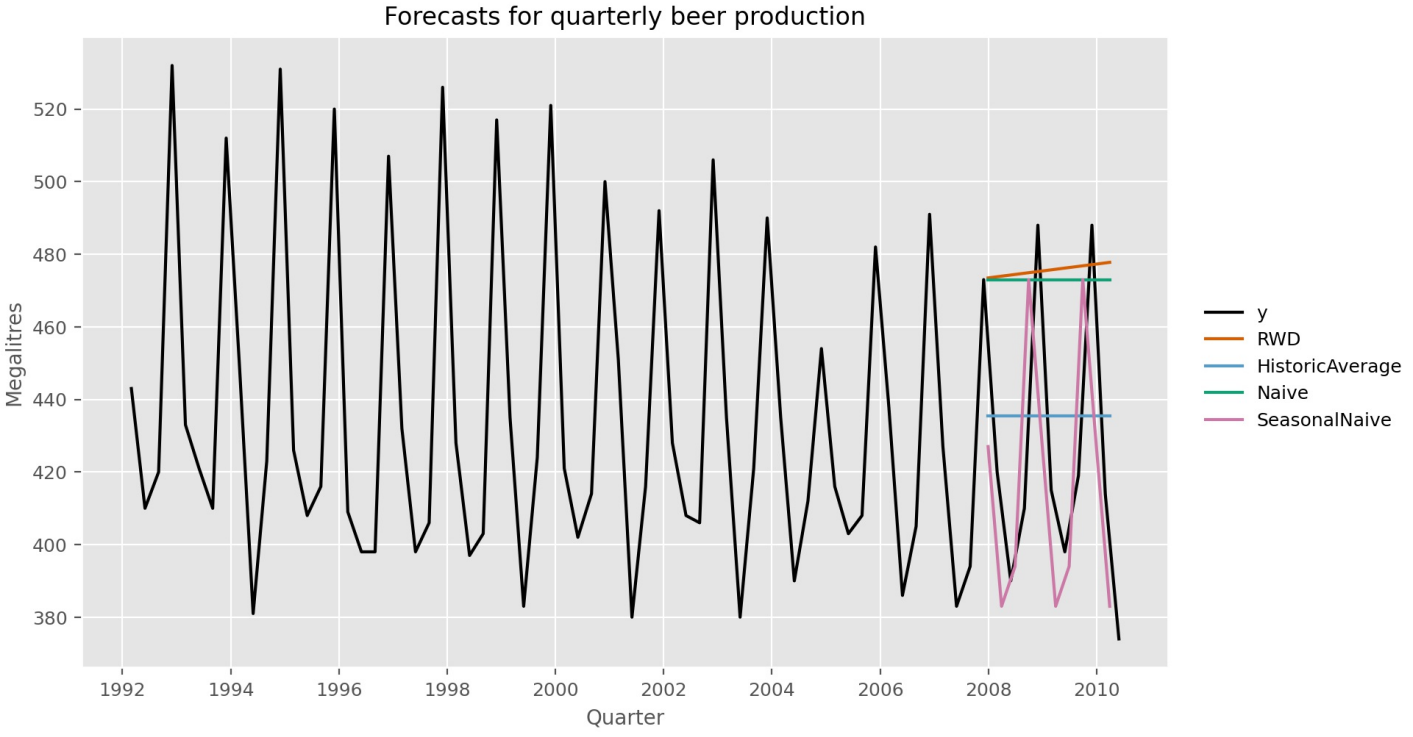

Figure 5.23: Forecasts of Australian quarterly beer production using data up to the end of 2007.

Figure 5.23 shows four forecast methods applied to the quarterly Australian beer production using data only to the end of 2007. The actual values for the period 2008–2010 are also shown. We compute the forecast accuracy measures for this period.

```
evaluation = evaluate(preds, metrics=[rmse, mae, mape,
  partial(mase, seasonality=4)], train_df=train_df)
```

	Method	RMSE	MAE	MAPE	MASE
0	RWD	64.901	58.876	14.577	4.117
1	HistoricAverage	38.447	34.825	8.283	2.435
2	Naive	62.693	57.400	14.184	4.014
3	SeasonalNaive	14.311	13.400	3.169	0.937

The `evaluate()` function will automatically extract the relevant periods from the data (`production` in this example) to match the forecasts when computing the various accuracy measures.

It is obvious from the graph that the seasonal naïve method is best for these data, although it can still be improved, as we will discover later. Sometimes, different accuracy measures will lead to different results as to which forecast method is best. However, in this case, all of the results point to the seasonal naïve method as the best of these four methods for this data set.

To take a non-seasonal example, consider the Google stock price. The following graph shows the closing stock prices from 2015, along with forecasts for January 2016 obtained from three different methods.

```
train_df = goog_df.query("ds.dt.year == 2015")
test_df = goog_df.query("ds.dt.year == 2016")
train_df["ds"] = np.arange(len(train_df["ds"]))
test_df["ds"] = np.arange(len(test_df["ds"]))

mean_method = HistoricAverage()
naive_method = Naive()
drift_method = RandomWalkWithDrift()

sf = StatsForecast(models=[drift_method, mean_method, naive_method], freq=1)
preds = sf.forecast(df=train_df, h=len(test_df))
preds["y"] = test_df["y"].values

plot_series(train_df, preds,
            xlabel="day",
            ylabel="$US",
            title="Google closing stock prices from Jan 2015",
            rm_legend=False,)
```

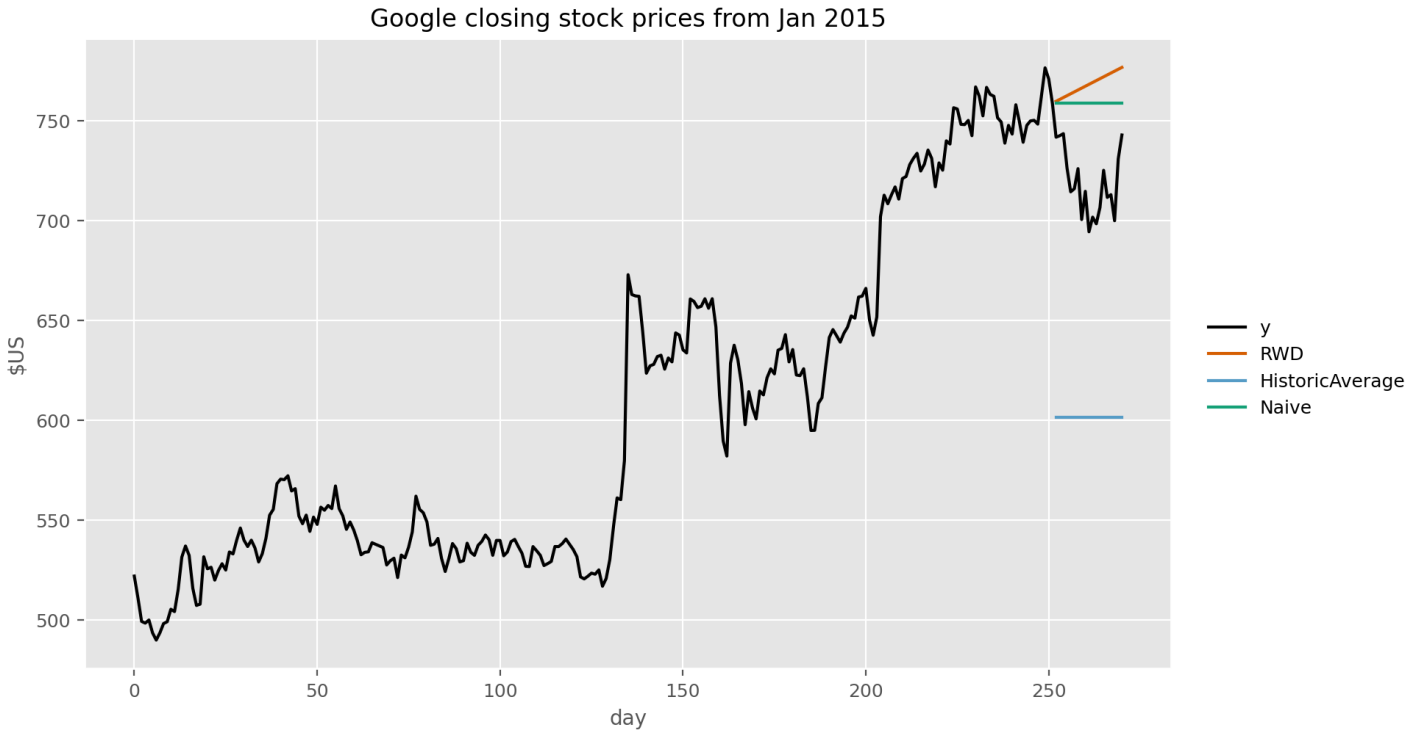

Figure 5.24: Forecasts of the Google stock price for Jan 2016.

```
evaluation = evaluate(preds, metrics=[rmse, mae, mape,
  partial(mase, seasonality=1)], train_df=train_df)
```

	Method	RMSE	MAE	MAPE	MASE
0	RWD	53.070	49.824	6.992	6.990
1	HistoricAverage	118.032	116.945	16.235	16.406
2	Naive	43.432	40.384	5.673	5.666

Here, the best method is the naïve method (regardless of which accuracy measure is used).

5.9 Evaluating distributional forecast accuracy

The preceding measures all measure point forecast accuracy. When evaluating distributional forecasts, we need to use some other measures.

Quantile scores

Consider the Google stock price example from the previous section. Figure 5.25 shows an 80% prediction interval for the forecasts from the naïve method.

```
train_df = goog_df.query("ds.dt.year == 2015")
test_df = goog_df.query("ds.dt.year == 2016")
train_df["ds"] = np.arange(len(train_df["ds"]))
test_df["ds"] = np.arange(len(test_df["ds"]))

naive_model = Naive()

sf = StatsForecast(models=[naive_model], freq=1)
preds = sf.forecast(h=19, level=[80], df=train_df)
preds["y"] = test_df["y"].values

plot_series(train_df, preds, level=[80],
            xlabel="day",
            ylabel="$US",
            title="Google closing stock prices",
            rm_legend=False,)
```

Figure 5.25: Naïve forecasts of the Google stock price for Jan 2016, along with 80% prediction intervals.

The lower limit of this prediction interval gives the 10th percentile (or 0.1 quantile) of the forecast distribution, so we would expect the actual value to lie below the lower limit about 10% of the time, and to lie above the lower limit about 90% of the time. When we compare the actual value to this percentile, we need to allow for the fact that it is more likely to be above than below.

More generally, suppose we are interested in the quantile forecast with probability p at future time t, and let this be denoted by f_{p,t}. That is, we expect the observation y_t to be less than f_{p,t} with probability p. For example, the 10th percentile would be f_{0.10,t}. If y_{t} denotes the observation at time t, then the **Quantile Score** is Q_{p,t} = \begin{cases} 2(1 - p) \big(f_{p,t} - y_{t}\big), & \text{if $y_{t} < f_{p,t}$}\\ 2p \big(y_{t} - f_{p,t}\big), & \text{if $y_{t} \ge f_{p,t}$} \end{cases} This is sometimes called the "pinball loss function" because a graph of it resembles the trajectory of a ball on a pinball table. The multiplier of 2 is often omitted, but including it makes the interpretation a little easier. A low value of Q_{p,t} indicates a better estimate of the quantile.

The quantile score can be interpreted like an absolute error. In fact, when p=0.5, the quantile score Q_{0.5,t} is the same as the absolute error. For other values of p, the "error" (y_t - f_{p,t}) is weighted to take account of how likely it is to be positive or negative. If p>0.5, Q_{p,t} gives a heavier penalty when the observation is greater than the estimated quantile than when the observation is less than the estimated quantile. The reverse is true for p<0.5.

In Figure 5.25, the one-step-ahead 10% quantile forecast (for 4 January 2016) is f_{0.1,t} = 744.54 and the observed value is y_t = 741.84. Then Q_{0.1,t} = 2(1-0.1) * (744.54 - 741.84) = 4.86 This is easily computed for all forecasts using `evaluation = evaluate(preds, metrics=[quantile_loss], level=[80])`.:

```
evaluation = evaluate(preds, metrics=[quantile_loss], level=[80])

methods = ['Naive']
evaluation_transformed = pd.DataFrame({
    'Method': methods,
    'Quantile loss (q=0.1)': evaluation[evaluation['metric'] ==
        'quantile_loss_q0.1'][methods].iloc[0].values,
    'Quantile loss (q=0.9)': evaluation[evaluation['metric'] ==
        'quantile_loss_q0.9'][methods].iloc[0].values,
})
evaluation_transformed
```

	Method	Quantile loss (q=0.1)	Quantile loss (q=0.9)
0	Naive	4.775	8.355

Winkler Score

It is often of interest to evaluate a prediction interval, rather than a few quantiles, and the Winkler score proposed by Winkler (1972)

is designed for this purpose. If the 100(1-\alpha)% prediction interval at time t is given by [\ell_{\alpha,t}, u_{\alpha,t}], then the Winkler score is defined as the length of the interval plus a penalty if the observation is outside the interval: W_{\alpha,t} = \begin{cases} (u_{\alpha,t} - \ell_{\alpha,t}) + \frac{2}{\alpha} (\ell_{\alpha,t} - y_t) & \text{if } y_t < \ell_{\alpha,t} \\ (u_{\alpha,t} - \ell_{\alpha,t}) & \text{if } \ell_{\alpha,t} \le y_t \le u_{\alpha,t} \\ (u_{\alpha,t} - \ell_{\alpha,t}) + \frac{2}{\alpha} (y_t - u_{\alpha,t}) & \text{if } y_t > u_{\alpha,t}. \end{cases} For observations that fall within the interval, the Winkler score is simply the length of the interval. Thus, low scores are associated with narrow intervals. However, if the observation falls outside the interval, the penalty applies, with the penalty proportional to how far the observation is outside the interval.

Prediction intervals are usually constructed from quantiles by setting \ell_{\alpha,t} = f_{\alpha/2,t} and u_{\alpha,t} = f_{1-\alpha/2,t}. If we add the corresponding quantile scores and divide by \alpha, we get the Winkler score: W_{\alpha,t} = (Q_{\alpha/2,t} + Q_{1-\alpha/2,t})/\alpha.

The one-step-ahead 80% interval shown in Figure 5.25 for 4 January 2016 is [744.54, 773.22], and the actual value was 741.84, so the Winkler score is

W_{\alpha,t} = (773.22 - 744.54) + \frac{2}{0.2} (744.54 - 741.84) = 55.68

This is easily computed using `compute_winkler_score()`:

```
def compute_winkler_score(df, alpha=0.2):
    def winkler_score(row, alpha):
        L_t = row["Naive-lo-80"]
        U_t = row["Naive-hi-80"]
        y_t = row["y"]

        if L_t <= y_t <= U_t:
            return U_t - L_t
        elif y_t < L_t:
            return U_t - L_t + (2 / alpha) * (L_t - y_t)
        else:  # y_t > U_t
            return U_t - L_t + (2 / alpha) * (y_t - U_t)

    df["Winkler"] = df.apply(winkler_score, axis=1, alpha=alpha)
    winkler_score = df["Winkler"].iloc[0]
    data = [{"Method": "Naive", "Winkler score": winkler_score}]

    return pd.DataFrame(data=data, index=None)

winkler_score_df = compute_winkler_score(preds)
winkler_score_df
```

	Method	Winkler score
0	Naive	55.68

Continuous Ranked Probability Score

Often we are interested in the whole forecast distribution, rather than particular quantiles or prediction intervals. In that case, we can average the quantile scores over all values of p to obtain the **Continuous Ranked Probability Score** or CRPS (Gneiting and Katzfuss 2014).

In the Google stock price example, we can compute the average CRPS value for all days in the test set. A CRPS value is a little like a weighted absolute error computed from the entire forecast distribution, where the weighting takes account of the probabilities.

```
train_df = goog_df.query("ds.dt.year == 2015")
test_df = goog_df.query("ds.dt.year == 2016")

mean_method = HistoricAverage()
naive_method = Naive()
drift_method = RandomWalkWithDrift()

sf = StatsForecast(models=[drift_method, naive_method, mean_method], freq="B")
levels = list(np.arange(0, 100, 1 / 10))
preds = sf.forecast(df=train_df, h=len(test_df), level=levels)
preds["y"] = test_df["y"].values

models = ['RWD', 'HistoricAverage', 'Naive']
crps_df = evaluate(
    df=preds,
    models=models,
    metrics = [mqloss],
    level=levels,
)
crps_df = crps_df[models].T.reset_index()
crps_df.columns = ["method", "CRPS"]
crps_df["CRPS"] *= 2
crps_df
```

	method	CRPS
0	RWD	33.557
1	HistoricAverage	76.789
2	Naive	26.500

Here, the naïve method is giving better distributional forecasts than the drift or mean methods.

Scale-free comparisons using skill scores

As with point forecasts, it is useful to be able to compare the distributional forecast accuracy of several methods across series on different scales. For point forecasts, we used scaled errors for that purpose. Another approach is to use skill scores. These can be used for both point forecast accuracy and distributional forecast accuracy.

With skill scores, we compute a forecast accuracy measure relative to some benchmark method. For example, if we use the naïve method as a benchmark, and also compute forecasts using the drift method, we can compute the CRPS skill score of the drift method relative to the naïve method as \frac{\text{CRPS}_{\text{Naïve}} - \text{CRPS}_{\text{Drift}}}{\text{CRPS}_{\text{Naïve}}}. This gives the proportion that the drift method improves over the naïve method based on CRPS. It is easy to compute using the `calculate_skill_score()` function.

```
naive_crps = crps_df.query("method == 'Naive'")["CRPS"].iloc[0]
crps_df['skill_score'] = crps_df.apply(
    lambda row: (naive_crps - row['CRPS']) / \
      naive_crps if row['method'] != 'Naive' else 0,
    axis=1
)
crps_df
```

	method	CRPS	skill_score
0	RWD	33.557	-0.266
1	HistoricAverage	76.789	-1.898
2	Naive	26.500	0.000

Of course, the skill score for the naïve method is 0 because it can't improve on itself. The other two methods have larger CRPS values than naïve, so the skills scores are negative; the drift method is 26.6% worse than the naïve method.

The `calculate_skill_score()` function will always compute the CRPS for the appropriate benchmark forecasts, even if these are not included in the dataframe. When the data are seasonal, the benchmark used is the seasonal naïve method rather than the naïve method. To ensure that the same training data are used for the benchmark forecasts, it is important that the data provided to the `evaluate()` function starts at the same time as the training data.

5.10 Time series cross-validation

A more sophisticated version of training/test sets is time series cross-validation. In this procedure, there are a series of test sets, each consisting of a single observation. The corresponding training set consists only of observations that occurred *prior* to the observation that forms the test set. Thus, no future observations can be used in constructing the forecast. Since it is not possible to obtain a reliable forecast based on a small training set, the earliest observations are not considered as test sets.

The following diagram illustrates the series of training and test sets, where the blue observations form the training sets, and the orange observations form the test sets.

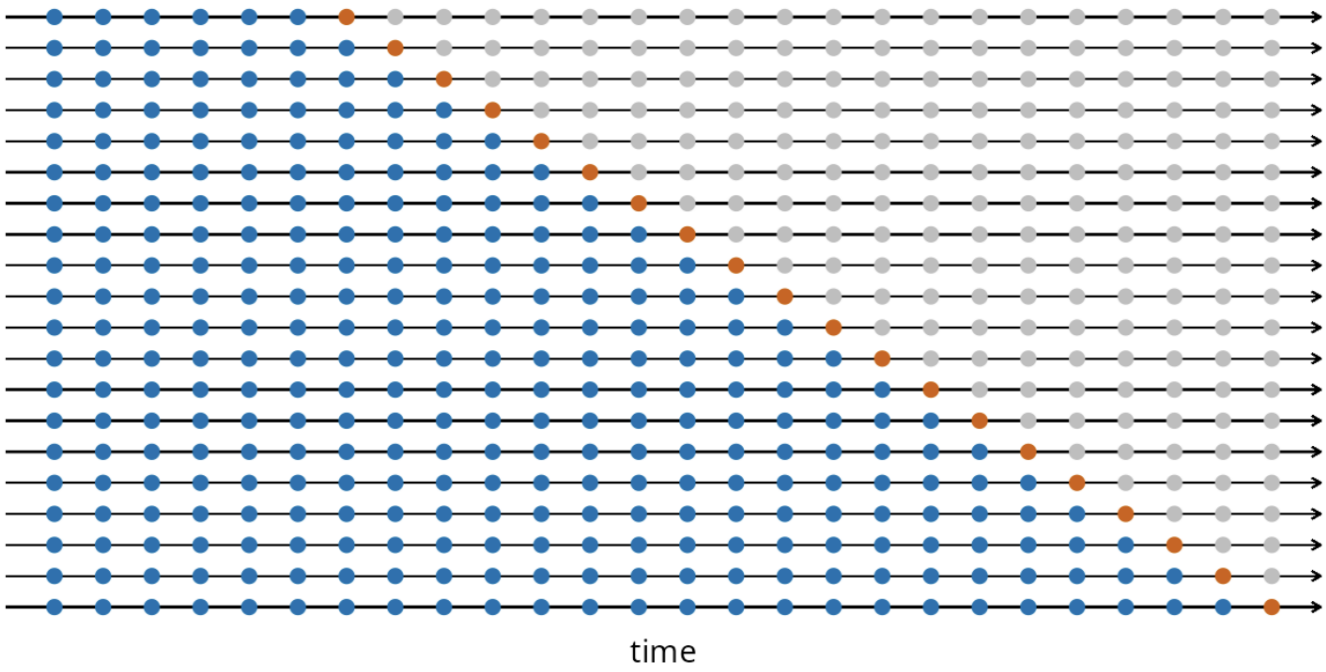

Figure 5.26: Time series cross-validation

The forecast accuracy is computed by averaging over the test sets. This procedure is sometimes known as "evaluation on a rolling forecasting origin" because the "origin" at which the forecast is based rolls forward in time.

With time series forecasting, one-step forecasts may not be as relevant as multi-step forecasts. In this case, the cross-validation procedure based on a rolling forecasting origin can be modified to allow multi-step errors to be used. Suppose that we are interested in models that produce good 4-step-ahead forecasts. Then the corresponding diagram is shown below.

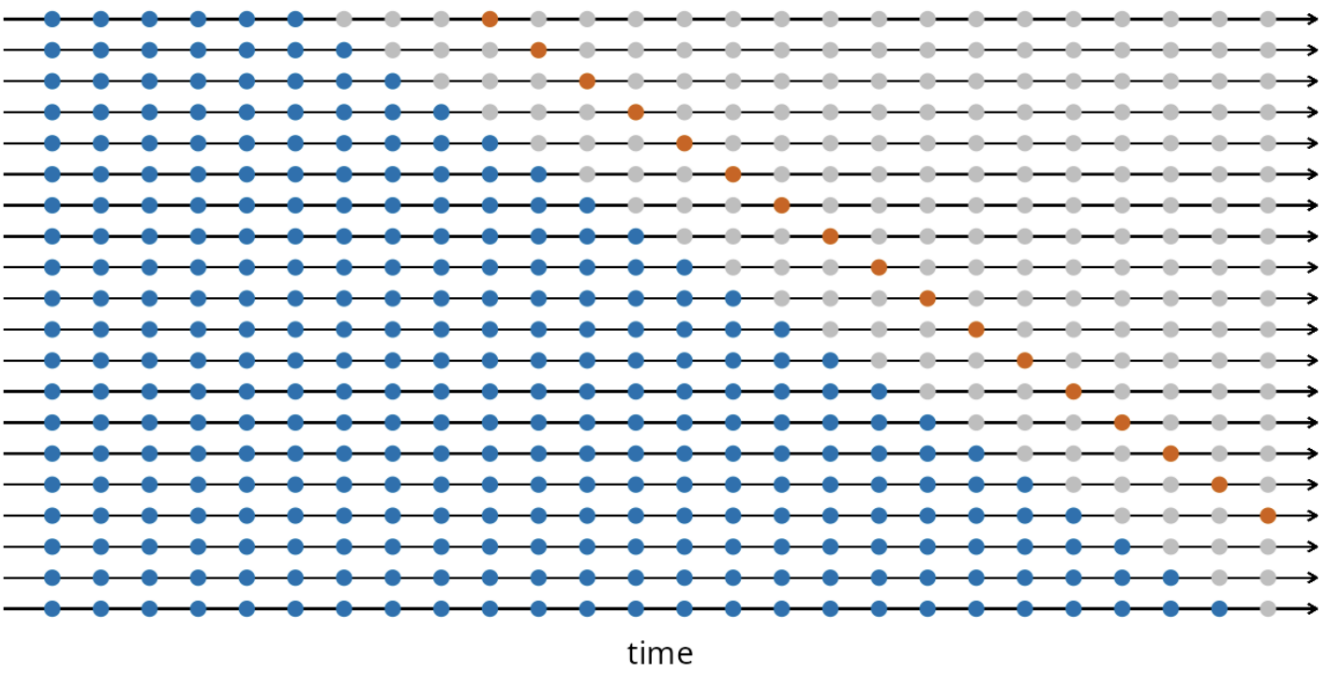

Figure 5.27: Time series cross-validation with multi-step models

In the following example, we compare the accuracy obtained via time series cross-validation with the residual accuracy. The `cross_validation()` function is used to create many training sets. Here, we create 249 validation windows of 1 time step.

```
goog_2015 = goog_df.query("ds.dt.year == 2015")
drift_method = RandomWalkWithDrift()

sf = StatsForecast(models=[drift_method], freq="B")
cv_df = sf.cross_validation(h=1, df=goog_2015, step_size=1, test_size=249)

cv_df.head()
```

	unique_id	ds	cutoff	y	RWD
0	GOOG_Close	2015-01-07	2015-01-06	498.358	487.850
1	GOOG_Close	2015-01-08	2015-01-07	499.929	490.497
2	GOOG_Close	2015-01-09	2015-01-08	493.454	494.427
3	GOOG_Close	2015-01-12	2015-01-09	489.854	487.758
4	GOOG_Close	2015-01-13	2015-01-12	493.464	484.507

The `evaluate()` function can be used to evaluate the forecast accuracy across the training sets.

	Method	RMSE	MAE	MAPE	MASE
0	Cross-validation - ['RWD']	11.27	7.26	119.40	1.02
0	Training - ['RWD']	11.15	7.16	117.74	1.00

As expected, the accuracy measures from the residuals are smaller, as the corresponding “forecasts” are based on a model fitted to the entire data set, rather than being true forecasts.

A good way to choose the best forecasting model is to find the model with the smallest RMSE computed using time series cross-validation.

Example: Forecast horizon accuracy with cross-validation

The `google_2015` subset of the `gafa_stock` data, plotted in Figure 5.10, includes daily closing stock price of Google Inc from the NASDAQ exchange for all trading days in 2015.

The code below evaluates the forecasting performance of 1- to 8-step-ahead drift forecasts. The plot shows that the forecast error increases as the forecast horizon increases, as we would expect.

```
rmses = []
drift_model = RandomWalkWithDrift()

horizons = np.arange(1, 9)

for horizon in horizons:
    sf = StatsForecast(models=[drift_model], freq="B")
    cv_df = sf.cross_validation(h=horizon, df=goog_2015, step_size=1,
      test_size=249)
    cv_evaluation = evaluate(cv_df, metrics=[rmse], models=["RWD"])
    rmses.append(cv_evaluation["RWD"].iloc[0])

fig, ax = plt.subplots()
ax.scatter(horizons, rmses)
ax.set_xlabel("h")
ax.set_ylabel("RMSE")
ax.set_title(
    "RMSE as a function of forecast horizon for the drift method " +
    "applied to Google closing stock prices."
)
plt.show()
```

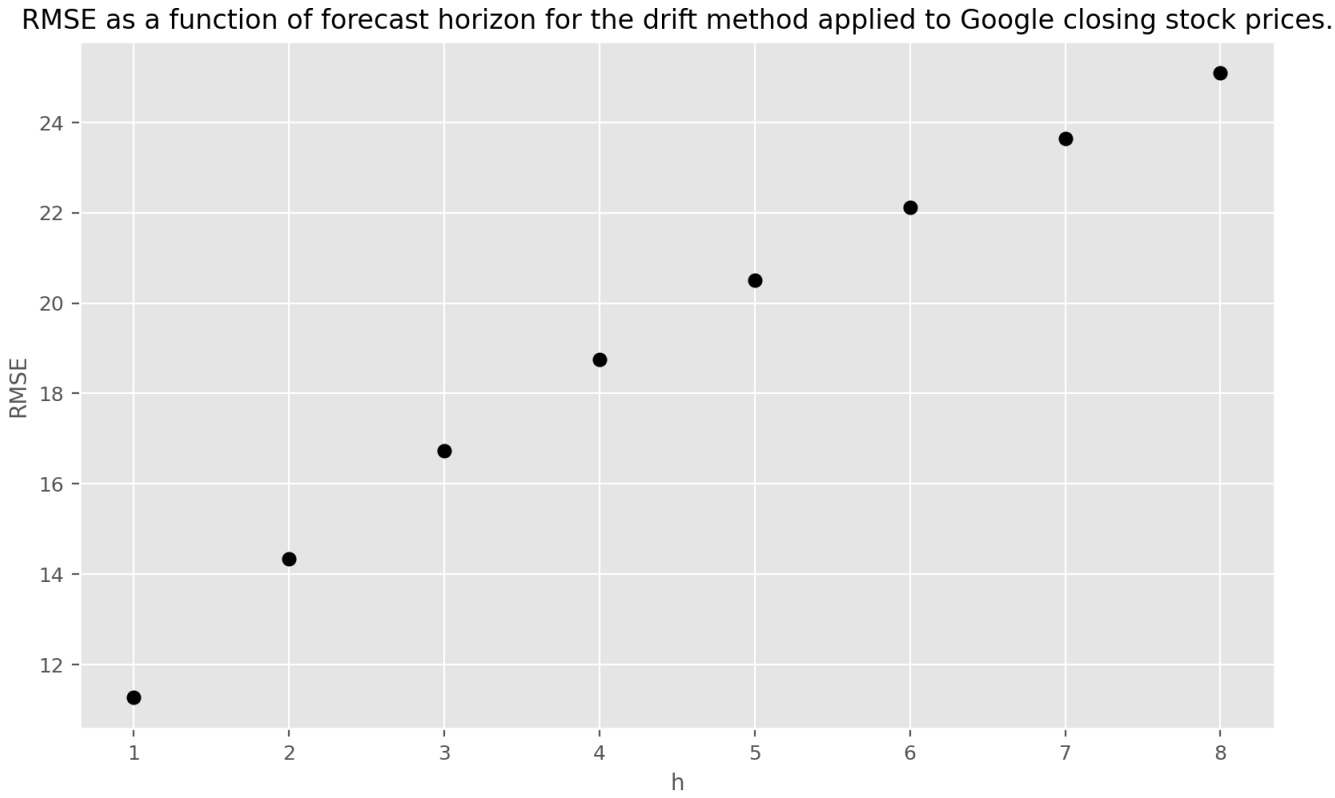

Figure 5.28: RMSE as a function of forecast horizon for the drift method applied to Google closing stock prices.

5.11 Exercises

1. Produce forecasts for the following series using whichever of `Naive()`, `SeasonalNaive()` or `RandomWalkWithDrift()` is more appropriate in each case:
    - Australian Population (`global_economy`)
    - Bricks (`aus_production`)

- NSW Lambs (`aus_livestock`)
- Household wealth (`hh_budget`).
- Australian takeaway food turnover (`aus_retail`).

2. Use the Facebook stock price (data set `gafa_stock`) to do the following:

   a. Produce a time plot of the series.
   b. Produce forecasts using the drift method and plot them.
   c. Show that the forecasts are identical to extending the line drawn between the first and last observations.
   d. Try using some of the other benchmark functions to forecast the same data set. Which do you think is best? Why?

3. Apply a seasonal naïve method to the quarterly Australian beer production data from 1992. Check if the residuals look like white noise, and plot the forecasts. The following code will help.

```
# Extract data of interest
recent_production = aus_production.query(
  "ds.dt.year >= '1992' and unique_id == 'Beer'"
)
# Define and estimate a model
seasonal_naive = SeasonalNaive(4)
sf = Statsforecast(models=[seasonal_naive], freq='Q')
sf.fit(recent_production)
preds = sf.forecast(h=10, fitted=True)
insample_preds = sf.forecast_fitted_values()
insample_preds['resid'] = \
  insample_preds['y'] - insample_preds['SeasonalNaive']
# Look at the residuals
plot_diagnostics(insample_preds)
# Look a some forecasts
plot_series(recent_production, preds)
```

   What do you conclude?

4. Repeat the previous exercise using the Australian Exports series from `global_economy` and the Bricks series from `aus_production`. Use whichever of `Naive()` or `SeasonlNaive()` is more appropriate in each case.

5. Produce forecasts for the 7 Victorian series in `aus_livestock` using `SeasonalNaive()`. Plot the resulting forecasts including the historical data. Is this a reasonable benchmark for these series?

6. Are the following statements true or false? Explain your answer.

   a. Good forecast methods should have normally distributed residuals.
   b. A model with small residuals will give good forecasts.
   c. The best measure of forecast accuracy is MAPE.
   d. If your model doesn't forecast well, you should make it more complicated.
   e. Always choose the model with the best forecast accuracy as measured on the test set.

7. For your retail time series (from Exercise 7 in Section 2.10):

   a. Create a training dataset consisting of observations before 2011 using

```
train = data.query("ds.dt.year < 2011")
test = data.query("ds.dt.year >= 2011")
```

   b. Check that your data have been split appropriately by producing the following plot.

```
plot_series(train, test)
```

   c. Fit a seasonal naïve model using `SeasonalNaive()` applied to your training data.

```
# Replace the following variables by their appropriate value
m = seasonal_period
freq = frequency_of_data

seasonal_naive = SeasonalNaive(m)
sf = Statsforecast(models=[seasonal_naive], freq=freq)
sf.fit(train)
```

d. Check the residuals.

```
preds = sf.forecast(h=len(test), df=train, fitted=True)
fitted_values = sf.forecast_fitted_values()
train['fitted'] = fitted_values['HistoricAverage']
train['resid'] = train['y'] - train['fitted']

plot_diagnostics(train)
```

Do the residuals appear to be uncorrelated and normally distributed?

e. Produce forecasts for the test data

```
preds = sf.forecast(h=len(test), df=train, fitted=True)
```

f. Compare the accuracy of your forecasts against the actual values.

```
preds['y'] = test['y'].values
evaluation = evaluate(preds, metrics=[rmse, mae, mape,
  partial(mase, seasonality=4)], train_df=train_df)
```

g. How sensitive are the accuracy measures to the amount of training data used?

8. Consider the number of pigs slaughtered in New South Wales (data set `aus_livestock`).

   a. Produce some plots of the data in order to become familiar with it.
   b. Create a training set of 486 observations, withholding a test set of 72 observations (6 years).
   c. Try using various benchmark methods to forecast the training set and compare the results on the test set. Which method did best?
   d. Check the residuals of your preferred method. Do they resemble white noise?

9. a. Create a training set for household wealth (`hh_budget`) by withholding the last four years as a test set.
   b. Fit all the appropriate benchmark methods to the training set and forecast the periods covered by the test set.
   c. Compute the accuracy of your forecasts. Which method does best?
   d. Do the residuals from the best method resemble white noise?

10. a. Create a training set for Australian takeaway food turnover (`aus_retail`) by withholding the last four years as a test set.
    b. Fit all the appropriate benchmark methods to the training set and forecast the periods covered by the test set.
    c. Compute the accuracy of your forecasts. Which method does best?
    d. Do the residuals from the best method resemble white noise?

11. We will use the Bricks data from `aus_production` (Australian quarterly clay brick production 1956–2005) for this exercise.

    a. Use a STL decomposition to calculate the trend-cycle and seasonal components.
    b. Compute and plot the seasonally adjusted data.
    c. Do the residuals look uncorrelated?
    d. Repeat with a robust STL decomposition by setting `robust = True` in the `STL` method from `statsmodels`. Does it make much difference?

12. `tourism` contains quarterly visitor nights (in thousands) from 1998 to 2017 for 76 regions of Australia.

    a. Extract data from the Gold Coast region using `query()` and aggregate total overnight trips (sum over `Purpose`) using `agg()`. Call this new dataset `gc_tourism`.

    b. Using slicing or `query()`, create three training sets for this data excluding the last 1, 2 and 3 years. For example, `gc_train_1 = aus_tourism.query("ds.dt.year < 2017")`.

    c. Compute one year of forecasts for each training set using the seasonal naïve (`SeasonalNaive()`) method. Call these

`gc_fc_1`, `gc_fc_2` and `gc_fc_3`, respectively.

d. Use `evaluate()` to compare the test set forecast accuracy using MAPE. Comment on these.

5.12 Further reading

- Ord, Fildes, and Kourentzes (2017) provides further discussion of simple benchmark forecasting methods.
- A review of forecast evaluation methods is given in Hyndman and Koehler (2006), looking at the strengths and weaknesses of different approaches. This is the paper that introduced the MASE as a general-purpose forecast accuracy measure.
- For a discussion of forecasting using STL, see Theodosiou (2011).
- An excellent discussion of evaluating distributional forecast accuracy is provided by Gneiting and Katzfuss (2014).

5.13 Used modules and classes

StatsForecast

- `StatsForecast` class - Core forecasting engine for fitting and generating predictions
- `Naive`, `SeasonalNaive`, `RandomWalkWithDrift`, `WindowAverage` models - Basic benchmark forecasting models
- `cross_validation` method - For performing time series cross-validation

UtilsForecast

- `evaluate` function - For evaluating forecast accuracy
- `plot_series`, `plot_diagnostics` functions - For visualizing time series data, forecasts and residual diagnostics
- `rmse`, `mae`, `mape`, `mase`, `scaled_crps` metrics - For calculating forecast accuracy metrics

---

1. For the ARIMA models discussed in Chapters 9 , the degrees of freedom is adjusted to give better results. □□

2. That is, a percentage is valid on a ratio scale, but not on an interval scale. Only ratio scale variables have meaningful zeros. □□

← Chapter 4 Time series features

Chapter 6 Judgmental forecasts →