In this chapter, we address many practical issues that arise in forecasting, and discuss some possible solutions.

13.1 Weekly, daily and sub-daily data

Weekly, daily and sub-daily data can be challenging for forecasting, although for different reasons.

Weekly data

Weekly data is difficult to work with because the seasonal period (the number of weeks in a year) is both large and non-integer. The average number of weeks in a year is 52.18. Most of the methods we have considered require the seasonal period to be an integer. Even if we approximate it by 52, most of the methods will not handle such a large seasonal period efficiently.

The simplest approach is to use a STL decomposition along with a non-seasonal method applied to the seasonally adjusted data (as discussed in Chapter 3). Here we will use an ETS model as the `trend_forecaster` of the STL decomposition (in `StatsForecast`, `AutoETS()` and `MSTL()` with a single seasonality, respectively), setting `model='ZZN'` so that only non-seasonal ETS models are considered. Below is an example using weekly data on US finished motor gasoline products supplied (in millions of barrels per day) from February 1991 to May 2005.

```
barrels = pd.read_csv("../data/us_gasoline.csv", parse_dates=["ds"])

sf = StatsForecast(
    models=[MSTL(season_length=52, trend_forecaster=AutoETS(model="ZZN"))], freq="W"
)

fc = sf.forecast(df=barrels, h=104, level=[80, 95])
plot_series(
    barrels, fc, level=[80, 95],
    xlabel="Week", ylabel="Millions of barrels per day",
    title="Weekly US gasoline production",
    rm_legend=False,
)
```

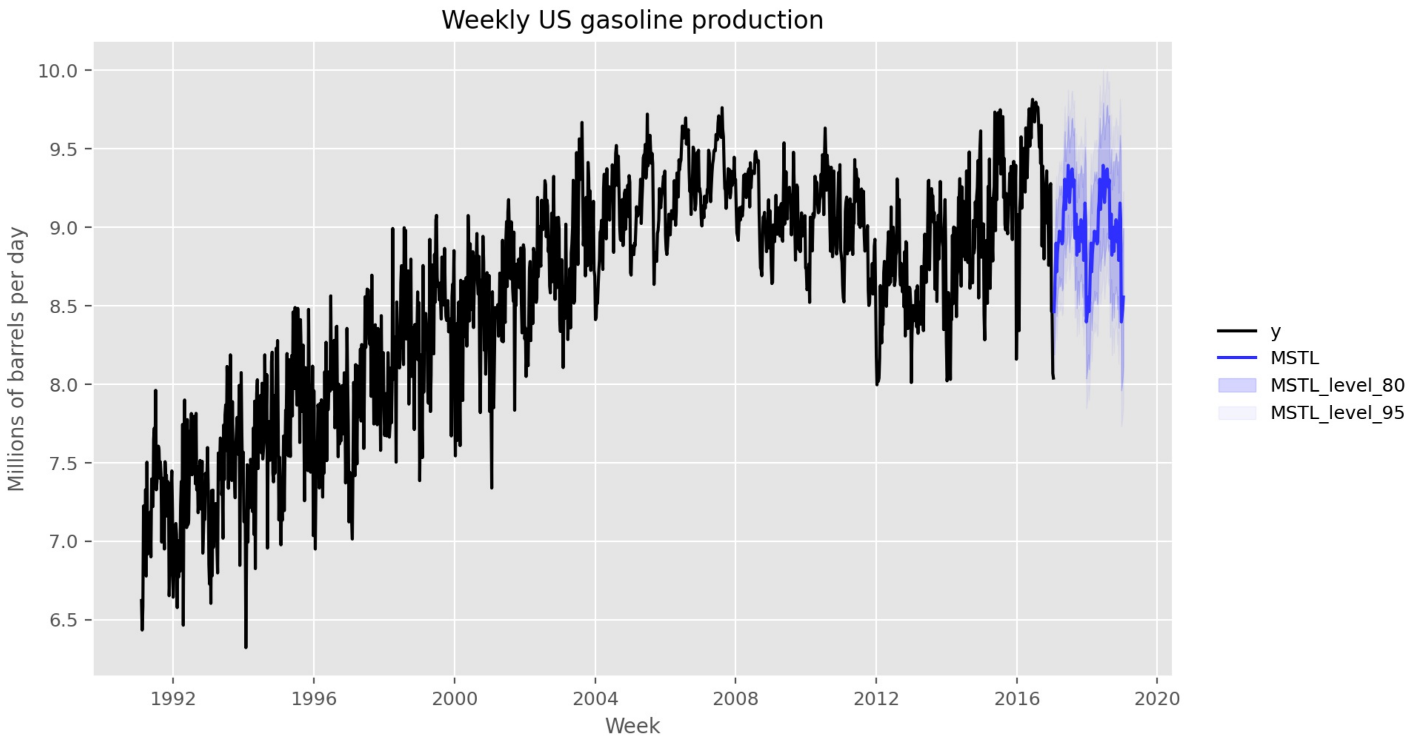

Figure 13.1: Forecasts for weekly US gasoline production using an STL decomposition with an ETS model for the seasonally adjusted data.

An alternative approach is to use a dynamic harmonic regression model, as discussed in Section 10.5. In the following example, the number of Fourier terms was selected by minimising the AICc. The order of the ARIMA model is also selected by minimising the AICc, although that is done within the `AutoARIMA()` class. We use `seasonal=False` to prevent `AutoARIMA()` trying to handle the seasonality using seasonal ARIMA components.

```
barrels_fourier, barrels_futr_fourier = pipeline(
    barrels,
    features=[partial(fourier, k=5, season_length=1)],
    freq='W',
    h=2 * 52,
)
sf = StatsForecast(
    models=[AutoARIMA(d=0, seasonal=False)],
    freq='W'
)
sf = sf.fit(barrels_fourier)
fc_fourier = sf.predict(h=2 * 52, X_df=barrels_futr_fourier, level=[80, 95])
plot_series(
    barrels_fourier, fc_fourier, level=[80, 95],
    xlabel="Week", ylabel="Millions of barrels per day",
    title="Weekly US gasoline production",
    rm_legend=False,
)
```

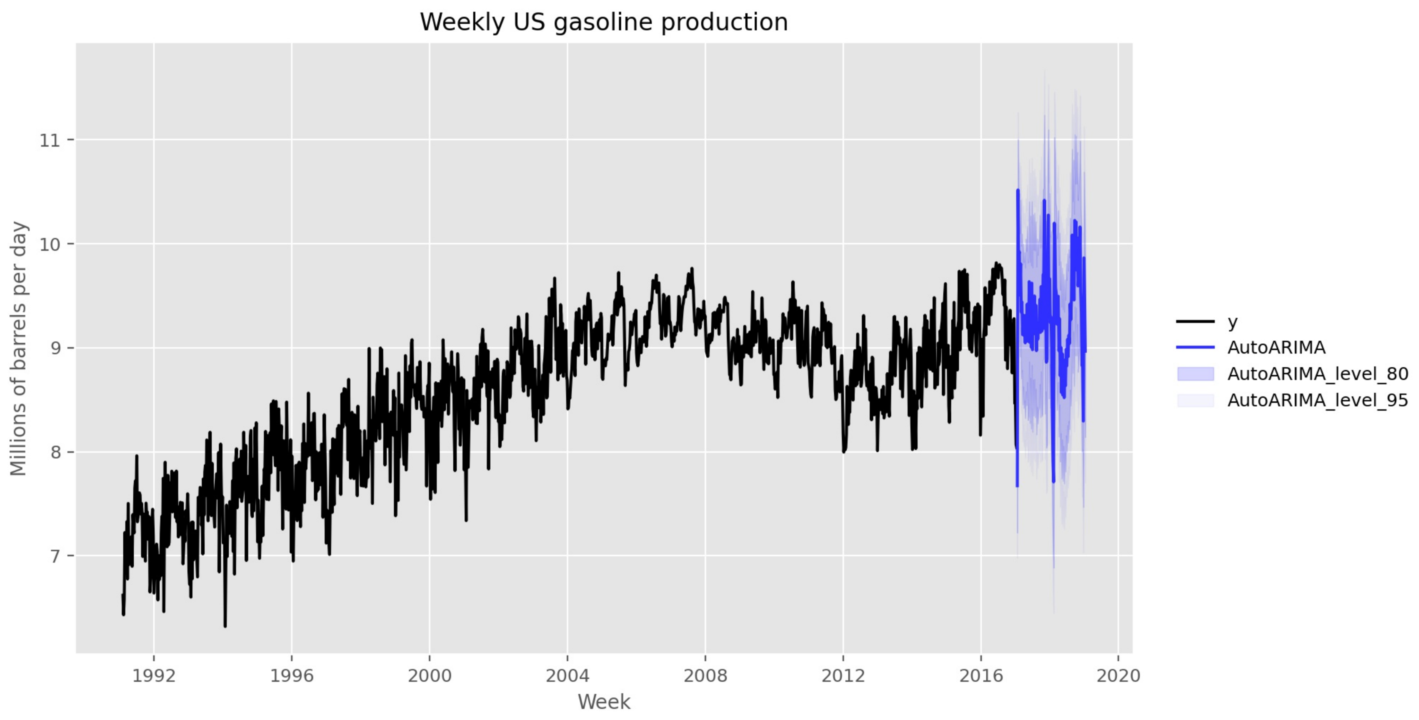

Figure 13.2: Forecasts for weekly US gasoline production using a dynamic harmonic regression model.

The fitted model has 6 pairs of Fourier terms and can be written as y_t = bt + \sum_{j=1}^{6} \left[\alpha_j\sin\left(\frac{2\pi j t}{52.18}\right) + \beta_j\cos\left(\frac{2\pi j t}{52.18}\right) \right] + \eta_t where \eta_t is an ARIMA(0, 1, 1) process. Because \eta_t is non-stationary, the model is actually estimated on the differences of the variables on both sides of this equation. There are 12 parameters to capture the seasonality, while the total number of degrees of freedom is 14 (the other two coming from the MA parameter and the drift parameter).

The STL approach is preferable when the seasonality changes over time. The dynamic harmonic regression approach is preferable if there are covariates that are useful predictors as these can be added as additional regressors.

Daily and sub-daily data

Daily and sub-daily (such as hourly) data are challenging for a different reason — they often involve multiple seasonal patterns, and so we need to use a method that handles such complex seasonality.

Of course, if the time series is relatively short so that only one type of seasonality is present, then it will be possible to use one of the single-seasonal methods we have discussed in previous chapters (e.g., AutoETS or a seasonal ARIMA model). But when the time series is long enough so that some of the longer seasonal periods become apparent, it will be necessary to use a MSTL model or Prophet, as discussed in Section 12.1.

However, these methods only allow for regular seasonality. Capturing seasonality associated with moving events such as Easter, Eid, or the Chinese New Year is more difficult. Even with monthly data, this can be tricky as the festivals can fall in either March or April (for Easter), in January or February (for the Chinese New Year), or at any time of the year (for Eid).

The best way to deal with moving holiday effects is to include dummy variables in the model. This can be done with AutoARIMA or Prophet, for example, but not with AutoETS. In fact, Prophet can easily incorporate holiday effects that can even last multiple days.

13.2 Time series of counts

All of the methods discussed in this book assume that the data have a continuous sample space. But often data comes in the form of counts. For example, we may wish to forecast the number of customers who enter a store each day. We could have 0, 1, 2, \dots, customers, but we cannot have 3.45693 customers.

In practice, this rarely matters provided our counts are sufficiently large. If the minimum number of customers is at least 100, then the difference between a continuous sample space [100,\infty) and the discrete sample space \{100,101,102,\dots\} has no perceivable effect on our forecasts. However, if our data contains small counts (0, 1, 2, \dots), then we need to use forecasting methods that are more appropriate for a sample space of non-negative integers.

Such models are beyond the scope of this book. However, there is one simple method which gets used in this context, that we would like to mention. It is "Croston's method", named after its British inventor, John Croston, and first described in Croston (1972). Actually, this method does not properly deal with the count nature of the data either, but it is used so often, that it is worth knowing about it.

With Croston’s method, we construct two new series from our original time series by noting which time periods contain zero values, and which periods contain non-zero values. Let q_i be the ith non-zero quantity, and let a_i be the time between q_{i-1} and q_i. Croston’s method involves separate simple exponential smoothing forecasts on the two new series a and q. Because the method is usually applied to time series of demand for items, q is often called the “demand” and a the “inter-arrival time”.

If \hat{q}_{i+1|i} and \hat{a}_{i+1|i} are the one-step forecasts of the (i+1)th demand and inter-arrival time respectively, based on data up to demand i, then Croston’s method gives \begin{align*} \hat{q}_{i+1|i} & = (1-\alpha_q)\hat{q}_{i|i-1} + \alpha_q q_i, \\ \hat{a}_{i+1|i} & = (1-\alpha_a)\hat{a}_{i|i-1} + \alpha_a a_i. \end{align*} The smoothing parameters \alpha_a and \alpha_q take values between 0 and 1. Let j be the time for the last observed positive observation. Then the h-step ahead forecast for the demand at time T+h, is given by the ratio \hat{y}_{T+h|T} = \hat{q}_{j+1|j}/\hat{a}_{j+1|j}. There are no algebraic results allowing us to compute prediction intervals for this method, because the method does not correspond to any statistical model (Shenstone and Hyndman 2005).

`StatsForecast` offers an implementation of Croston’s method via the `CrostonClassic()` function. It also offers an optimized implementation called `CrostonOptimized()` where the smoothing parameter is optimally selected from the range [0.1,0.3], and a variation of Croston’s method developed by Syntetos and Boylan (2005). This variation applies a deflating factor to the Croston estimates to remove the bias and is available via the `CrostonSBA()` function.

Example: Pharmaceutical sales

Figure 13.3 shows the numbers of scripts sold each month for immune sera and immunoglobulin products in Australia. The data contain small counts, with many months registering no sales at all, and only small numbers of items sold in other months.

```
j06 = (
    pd.read_csv("../data/PBS_unparsed.csv", parse_dates=["Month"])
    .query('ATC2 == "J06"')
    .groupby("Month")["Scripts"]
    .sum()
    .reset_index()
    .rename(columns={"Month": "ds", "Scripts": "y"})
    .assign(unique_id="J06")
)

plot_series(
    j06,
    xlabel="Month [1M]",
    ylabel="Number of scripts",
    title="Sales for immune sera and immunoglobulins",
)
```

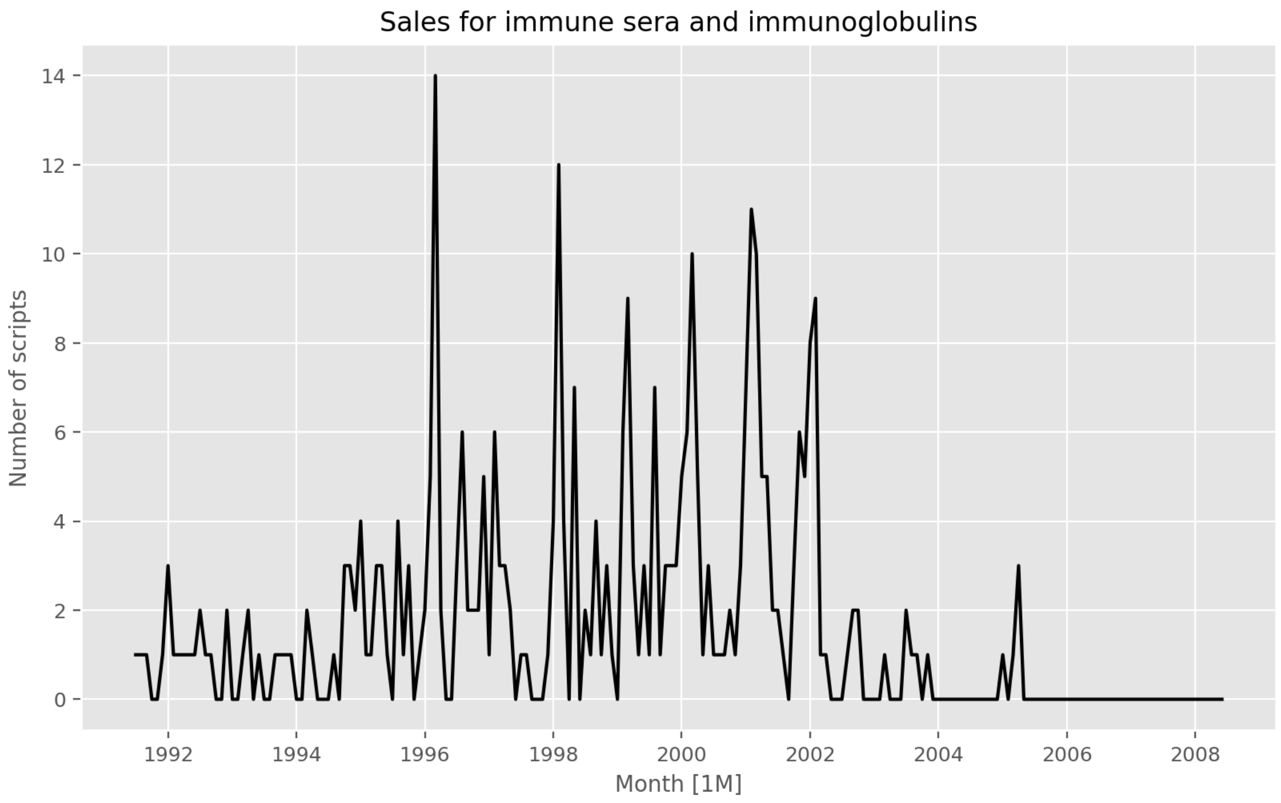

Figure 13.3: Numbers of scripts sold for Immune sera and immunoglobulins on the Australian Pharmaceutical Benefits Scheme.

Tables 13.1 and 13.2 shows the first 10 demand values, with their corresponding inter-arrival times.

Table 13.1: The first 10 non-zero demand values.

Month	Scripts
1991 Jul	1
1991 Aug	1
1991 Sep	1
1991 Oct	0
1991 Nov	0
1991 Dec	1
1992 Jan	3
1992 Feb	1
1992 Mar	1
1992 Apr	1
1992 May	1
1992 Jun	1

Table 13.2: The first 10 non-zero demand values shown as demand and inter-arrival series.

i	1	2	3	4	5	6	7	8	9	10
q_i	1	1	1	1	3	1	1	1	1	1
a_i		1	1	3	1	1	1	1	1	1

```
sf = StatsForecast(
    models=[CrostonClassic(), CrostonOptimized(), CrostonSBA()], freq="M"
)

fc = sf.forecast(df=j06, h=6)
fc
```

	unique_id	ds	CrostonClassic	CrostonOptimized	CrostonSBA
0	J06	2008-06-30	0.869	0.824	0.825
1	J06	2008-07-31	0.869	0.824	0.825
2	J06	2008-08-31	0.869	0.824	0.825
3	J06	2008-09-30	0.869	0.824	0.825
4	J06	2008-10-31	0.869	0.824	0.825
5	J06	2008-11-30	0.869	0.824	0.825

Unlike other models in `StatsForecast`, no implementation of Croston's method can provide prediction intervals because there is no underlying stochastic model.

Forecasting models that deal more directly with the count nature of the data, and allow for a forecasting distribution, are described in Christou and Fokianos (2015).

13.3 Ensuring forecasts stay within limits

It is common to want forecasts to be positive, or to require them to be within some specified range [a,b]. Both of these situations are relatively easy to handle using transformations.

As discussed in Section 5.6, we transform the data from the constrained range to allow it to take any positive or negative value, fit a model and produce forecasts, and then back-transform the forecasts and the prediction intervals to the original constrained range. If we don't do a bias adjustment, this results in point forecasts that are equal to medians of the forecast distribution. The prediction intervals are correct without adjustment.

Positive forecasts

To impose a positivity constraint, we can simply work on the log scale. For example, consider the real price of a dozen eggs (1900-1993; in cents) shown in Figure 13.4. Because of the log transformation, the forecast distributions are constrained to stay positive, and so they will become progressively more skewed as the mean decreases.

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

sf = StatsForecast(models=[AutoETS(season_length=1, model="ZAZ")], freq="Y")

fc = sf.forecast(df=log_prices, h=50, level=[80, 95])
fc.iloc[:, 2:] = np.exp(fc.iloc[:, 2:])
plot_series(
    egg_prices, fc, level=[80, 95],
    xlabel="year",
    ylabel="$US (in cents adjusted for inflation)",
    title="Annual egg prices",
    rm_legend=False,
)
```

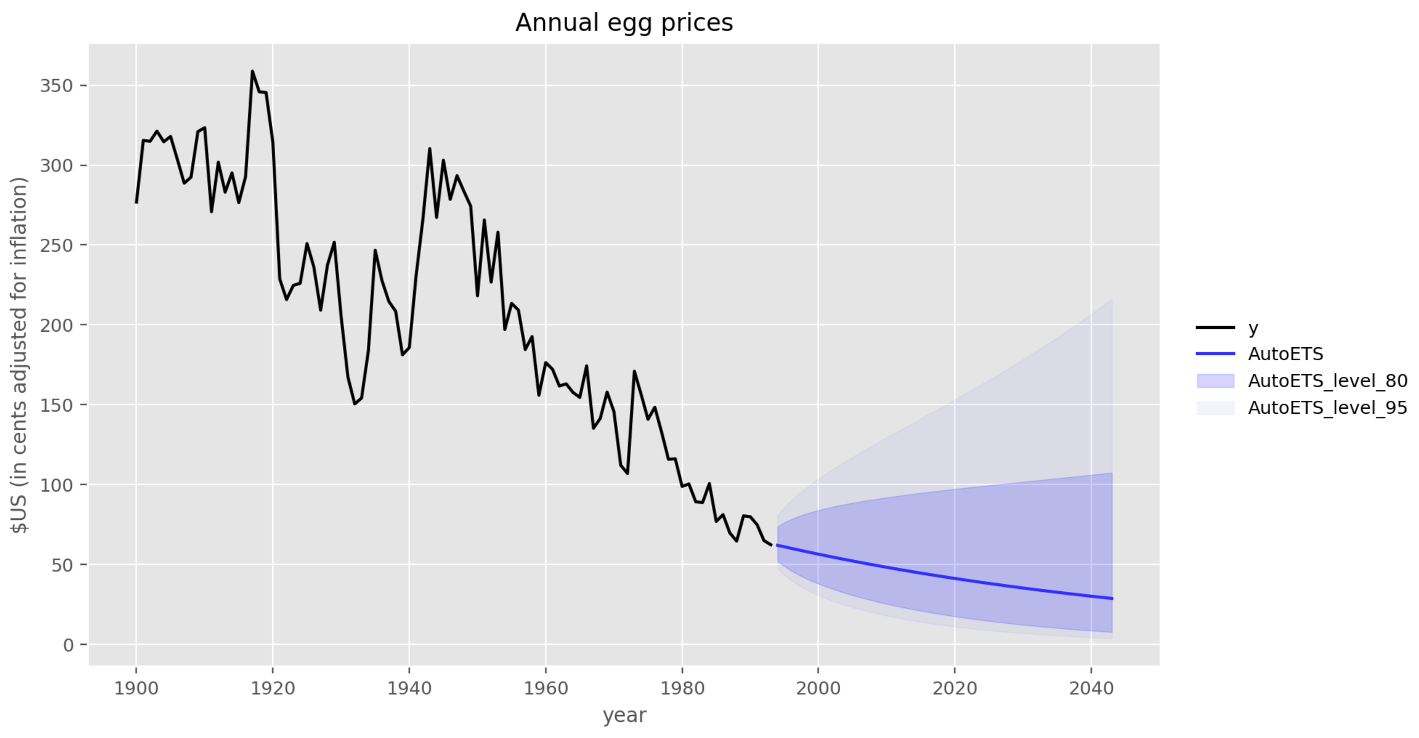

Figure 13.4: Forecasts for the price of a dozen eggs, constrained to be positive using a log transformation.

Forecasts constrained to an interval

To see how to handle data constrained to an interval, imagine that the egg prices were constrained to lie within a=50 and b=400. Then we can transform the data using a scaled logit transform which maps (a,b) to the whole real line: y = \log\left(\frac{x-a}{b-x}\right), where x is on the original scale and y is the transformed data. To reverse the transformation, we will use x = \frac{(b-a)e^y}{1+e^y} + a. This is not a built-in transformation, so we will need to first setup the transformation functions.

```python
def scaled_logit(df, target_col="y", lower=0, upper=1):
    Y_df = df.copy()
    Y_df[target_col] = np.log((Y_df[target_col] - lower) / (upper - Y_df[target_col]))
    return Y_df


def inv_scaled_logit(fc, lower=0, upper=1):
    Y_df = fc.copy()
    Y_df.iloc[:, 2:] = (upper - lower) * np.exp(Y_df.iloc[:, 2:]) / (
        1 + np.exp(Y_df.iloc[:, 2:])
    ) + lower
    return Y_df


scaled_logit_prices = scaled_logit(egg_prices, target_col="y", lower=50, upper=400)

sf = StatsForecast(models=[AutoETS(season_length=1, model="ZAZ")], freq="Y")

fc = sf.forecast(df=scaled_logit_prices, h=50, level=[80, 95])
fc = inv_scaled_logit(fc, lower=50, upper=400)
plot_series(
    egg_prices, fc, level=[80, 95],
    xlabel="year",
    ylabel="$US (in cents adjusted for inflation)",
    title="Annual egg prices",
    rm_legend=False,
)
```

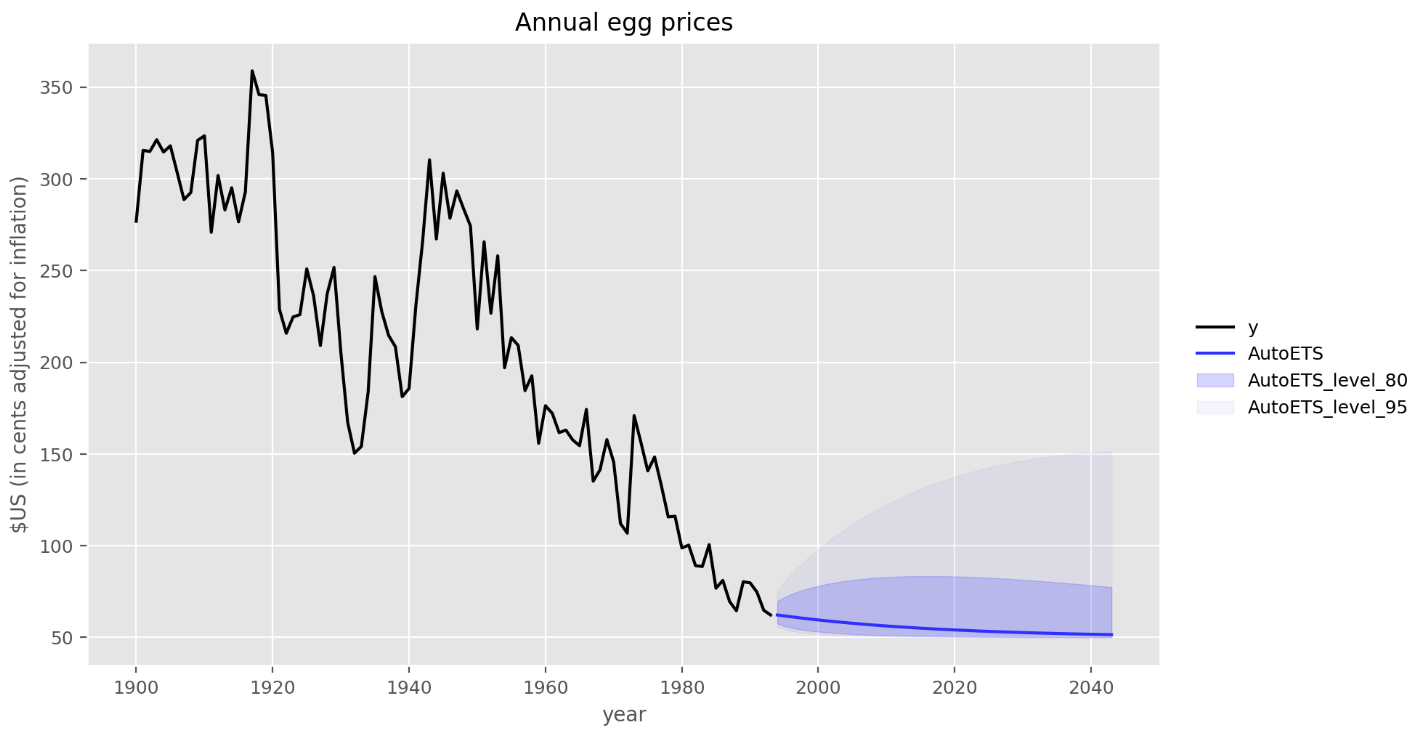

Figure 13.5: Forecasts for the price of a dozen eggs, constrained to be between 50 and 400 using a logit transformation.

The prediction intervals from these transformations maintain the same coverage probability as on the transformed scale because quantiles are preserved under monotonically increasing transformations. They also lie above 50 due to the transformation. As a result of this artificial (and unrealistic) constraint, the forecast distributions have become extremely skewed.

13.4 Forecast combinations

An easy way to improve forecast accuracy is to use several different methods on the same time series, and to average the resulting forecasts. Over 50 years ago, John Bates and Clive Granger wrote a famous paper (Bates and Granger 1969), showing that combining forecasts often leads to better forecast accuracy. Twenty years later, Clemen (1989) wrote

> The results have been virtually unanimous: combining multiple forecasts leads to increased forecast accuracy. In many cases one can make dramatic performance improvements by simply averaging the forecasts.

While there has been considerable research on using weighted averages, or some other more complicated combination approach, using a simple average has proven hard to beat (Wang et al. (2023)).

Here is an example using monthly revenue from take-away food in Australia, from April 1982 to December 2018. We use forecasts from the following models: `AutoETS()`, `MSTL()`, and `AutoARIMA()`; and we compare the results using the last 3 years (36 months) of observations. Note that, as in the preceding section, we will use a log transformation. The validation set will be defined here but used later on.

```python
auscafe = (
    pd.read_csv("../data/aus_retail.csv", parse_dates=["Month"])
    .query('Industry.str.contains("Takeaway")')
    .groupby("Month")["Turnover"]
    .sum()
    .reset_index()
    .rename(columns={"Turnover": "y", "Month": "ds"})
    .sort_values("ds")
)
auscafe.insert(0, "unique_id", "auscafe")

train = auscafe.query('ds <= "2013-12-01"')
validation = auscafe.query('ds > "2013-12-01" & ds <= "2015-12-01"')
test = auscafe.query('ds > "2015-12-01"')

log_train = train.copy()
log_train["y"] = np.log(train["y"])

sf = StatsForecast(
    models=[
        AutoETS(season_length=12),
        MSTL(season_length=12),
        AutoARIMA(season_length=12),
    ],
    freq="MS",
)

fc = sf.forecast(df=log_train, h=60, level=[80, 95], fitted=True)
fc.iloc[:, 2:] = np.exp(fc.iloc[:, 2:])
```

Notice that we form a combination by simply taking a linear function of the estimated models.

```python
fc["Combination"] = fc[["AutoETS", "MSTL", "AutoARIMA"]].mean(axis=1)
plot_series(
    auscafe.query('ds > "2008-12-01"'), fc,
    xlabel="Month",
    ylabel="$ billion",
    title="Australian monthly expenditure on eating out",
    rm_legend=False,
)
```

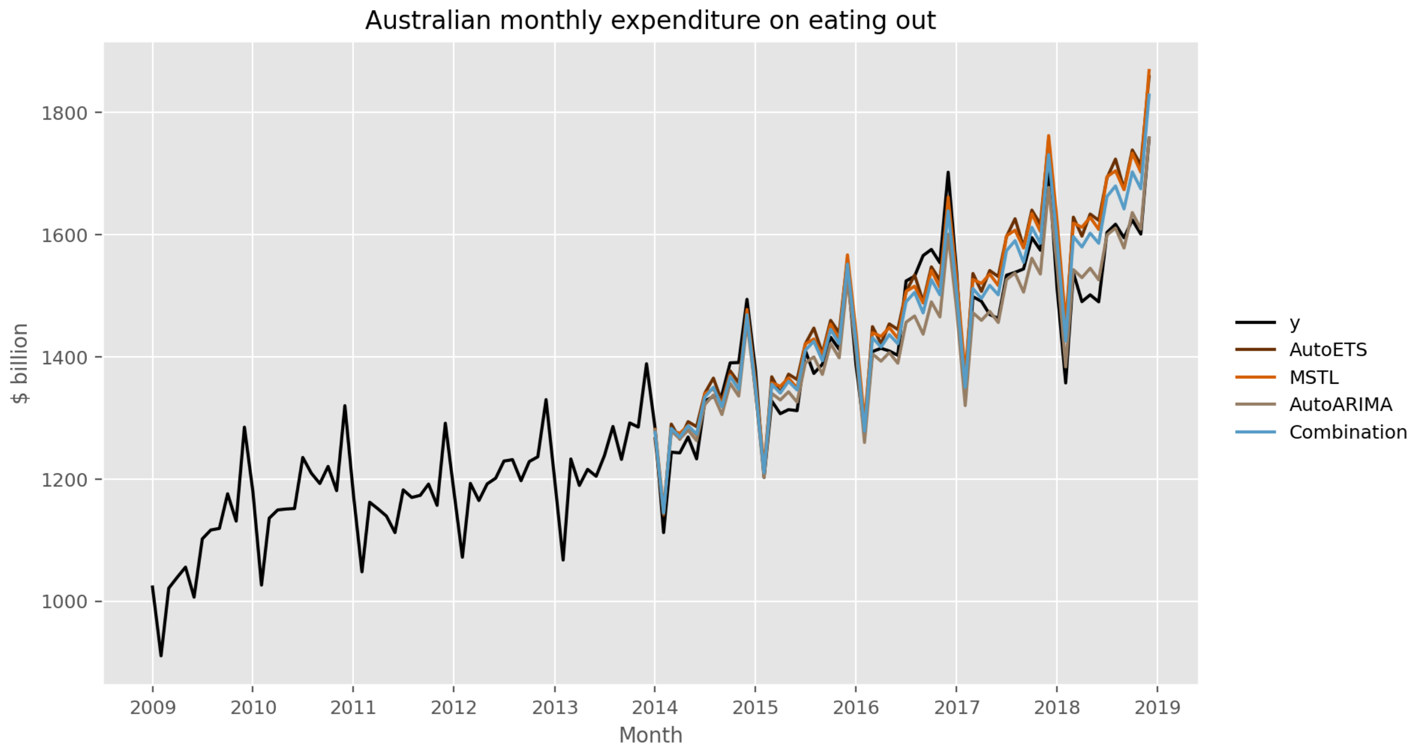

Figure 13.6: Point forecasts from various methods applied to Australian monthly expenditure on eating out.

```
res = fc.merge(test, on=["unique_id", "ds"])
evaluate(res, metrics=[rmse, mae, mape, partial(mase, seasonality=1)], train_df=train)
```

	unique_id	metric	AutoETS	MSTL	AutoARIMA	Combination
0	auscafe	rmse	70.725	69.462	44.209	53.135
1	auscafe	mae	60.636	60.164	31.766	45.058
2	auscafe	mape	0.040	0.039	0.021	0.029
3	auscafe	mase	2.102	2.086	1.101	1.562

AutoARIMA does particularly well with this series, while the combination approach does better than the AutoETS and MSTL models. For other data, AutoARIMA may be quite poor, while the combination approach is usually not far off, or better than, the best component method.

Forecast combination distributions

We will use conformal prediction (discussed in Section 5.5) to compute 80% and 90% prediction intervals for the combination of forecasts. Since this combination is not itself a model from `StatsForecast`, we need to compute the intervals directly. First, we need to obtain the conformal scores, which are calculated from the residuals of the validation set defined earlier. Then, we will generate a new forecast for the test set using the combination of base models and compute its prediction intervals using the conformal scores.

```
fc_val = sf.forecast(df=train, h=24)
fc_val["Combination"] = fc_val[["AutoETS", "MSTL", "AutoARIMA"]].mean(axis=1)
fc_val = fc_val.merge(validation, on=["unique_id", "ds"])
residuals = fc_val["y"] - fc_val["Combination"]

fc_cp = sf.forecast(df=pd.concat([train, validation]), h=36, level=[80, 90])
fc_cp["Combination"] = fc_cp[["AutoETS", "MSTL", "AutoARIMA"]].mean(axis=1)
fc_cp["Combination-lo-90"] = fc_cp["Combination"] + np.quantile(residuals, 0.1 / 2)
fc_cp["Combination-hi-90"] = fc_cp["Combination"] + np.quantile(residuals, 1 - 0.1 / 2)
fc_cp["Combination-lo-80"] = fc_cp["Combination"] + np.quantile(residuals, 0.2 / 2)
fc_cp["Combination-hi-80"] = fc_cp["Combination"] + np.quantile(residuals, 1 - 0.2 / 2)

plot_series(
    auscafe.query('ds > "2008-12-01"'), fc_cp, models=["Combination"],
    level=[80, 90],
    xlabel="Month",
    ylabel="$ billion",
    title="Australian monthly expenditure on eating out",
    rm_legend=False,
)
```

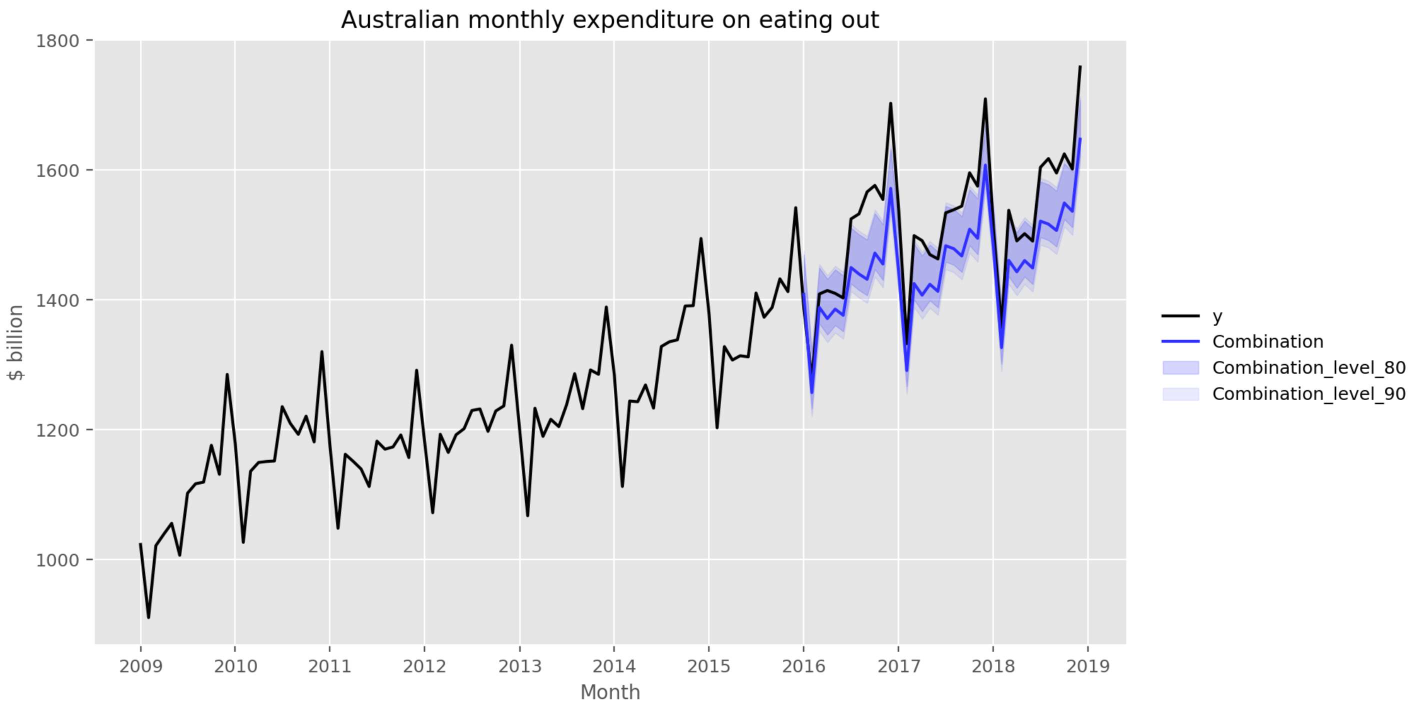

Figure 13.7: Conformal prediction intervals for the combination forecast of Australian monthly expenditure on eating out.

To check the accuracy of the 80% and 95% prediction intervals, we can use a Winkler score (defined in Section 5.9).

```python
def winkler_score(y_true, lower, upper, alpha=0.05):
    interval_width = upper - lower
    penalties = np.zeros_like(y_true)
    for i in range(len(y_true)):
        if y_true[i] < lower[i]:
            penalties[i] = (2 / alpha) * (lower[i] - y_true[i])
        elif y_true[i] > upper[i]:
            penalties[i] = (2 / alpha) * (y_true[i] - upper[i])

    return np.mean(interval_width + penalties)


res = fc_cp.merge(test, on=["unique_id", "ds"])
alpha = {"90": 0.1, "80": 0.2}
models = ["Combination", "AutoETS", "MSTL", "AutoARIMA"]
results = []

for alpha_key, alpha_value in alpha.items():
    for model in models:
        score = winkler_score(
            res["y"],
            res[f"{model}-lo-{alpha_key}"],
            res[f"{model}-hi-{alpha_key}"],
            alpha=alpha_value,
        )
        results.append(
            {"Model": model, "Prediction Interval": alpha_key, "Winkler Score": score}
        )

winkler_score_df = pd.DataFrame(results)
winkler_score_df.pivot(
    index="Model", columns="Prediction Interval", values="Winkler Score"
)
```

Prediction Interval	80	90
Model		
AutoARIMA	295.025	393.507
AutoETS	410.812	496.483
Combination	257.326	385.064
MSTL	258.167	317.461

For the 80% prediction intervals, the Winkler score of the combination of models is the lowest, while for the 90% it is the second lowest. This shows that the combination of models can also provide reasonable prediction intervals.

13.5 Prediction intervals for aggregates

A common problem is to forecast the aggregate of several time periods of data, using a model fitted to the disaggregated data. For example, we may have monthly data but wish to forecast the total for the next year. Or we may have weekly data, and want to forecast the total for the next four weeks.

If the point forecasts are means, then adding them up will give a good estimate of the total. But prediction intervals are more tricky due to the correlations between forecast errors.

A general solution is to use simulations. Here is an example using the AutoETS, AutoARIMA, and MSTL models applied to Australian take-away food sales, assuming we wish to forecast the aggregate revenue in the next 12 months.

```
# Filter data from 2018 onwards
res_agg = res.query('ds >= "2018-01-01"')
models = ["AutoETS", "AutoARIMA", "MSTL", "Combination"]
columns = ["unique_id", "ds"] + [col for model in models for col in [
    model,
    f"{model}-lo-80",
    f"{model}-hi-80",
    f"{model}-lo-90",
    f"{model}-hi-90"
]]
sum_df = res_agg[columns].drop(columns=["unique_id", "ds"]).sum().reset_index()
sum_df.columns = ["Column", "Total"]
reshaped_df = pd.DataFrame([{
    'Model': model,
    'Mean': sum_df[sum_df['Column'] == model]['Total'].iloc[0],
    'lo-80': sum_df[sum_df['Column'] == f"{model}-lo-80"]['Total'].iloc[0],
    'hi-80': sum_df[sum_df['Column'] == f"{model}-hi-80"]['Total'].iloc[0],
    'lo-90': sum_df[sum_df['Column'] == f"{model}-lo-90"]['Total'].iloc[0],
    'hi-90': sum_df[sum_df['Column'] == f"{model}-hi-90"]['Total'].iloc[0]
} for model in models])
reshaped_df
```

	Model	Mean	lo-80	hi-80	lo-90	hi-90
0	AutoETS	17556.983	16793.603	18320.363	16577.195	18536.771
1	AutoARIMA	17846.192	16380.539	19311.844	15965.047	19727.336
2	MSTL	18275.345	16276.119	20274.571	15709.367	20841.324
3	Combination	17892.840	17595.302	18623.629	17457.333	18690.296

As expected, the sum of the mean of the simulated data is close to the sum of the individual forecasts.

13.6 Backcasting

Sometimes it is useful to “backcast” a time series — that is, forecast in reverse time. Although there are no in-built Python functions to do this, it is easy to implement by creating a new time index.

Suppose we want to extend our Australian takeaway to the start of 1981 (the actual data starts in April 1982).

```
backcasts = auscafe.copy()
backcasts["y"] = auscafe["y"].iloc[::-1].values

sf = StatsForecast(models=[AutoETS(season_length=12)], freq="MS")

fc = sf.forecast(df=backcasts, h=15, level=[80, 95])
fc_reversed = fc.copy()[::-1]
fc_reversed["ds"] = pd.date_range(end="1982-03-01", periods=15, freq="MS")

plot_series(
    auscafe.query('ds < "1990-01-01"'), fc_reversed, level=[80, 95],
    xlabel="Month",
    ylabel="$ (billions)",
    title="Backcasts of Australian food expenditure",
    rm_legend=False,
)
```

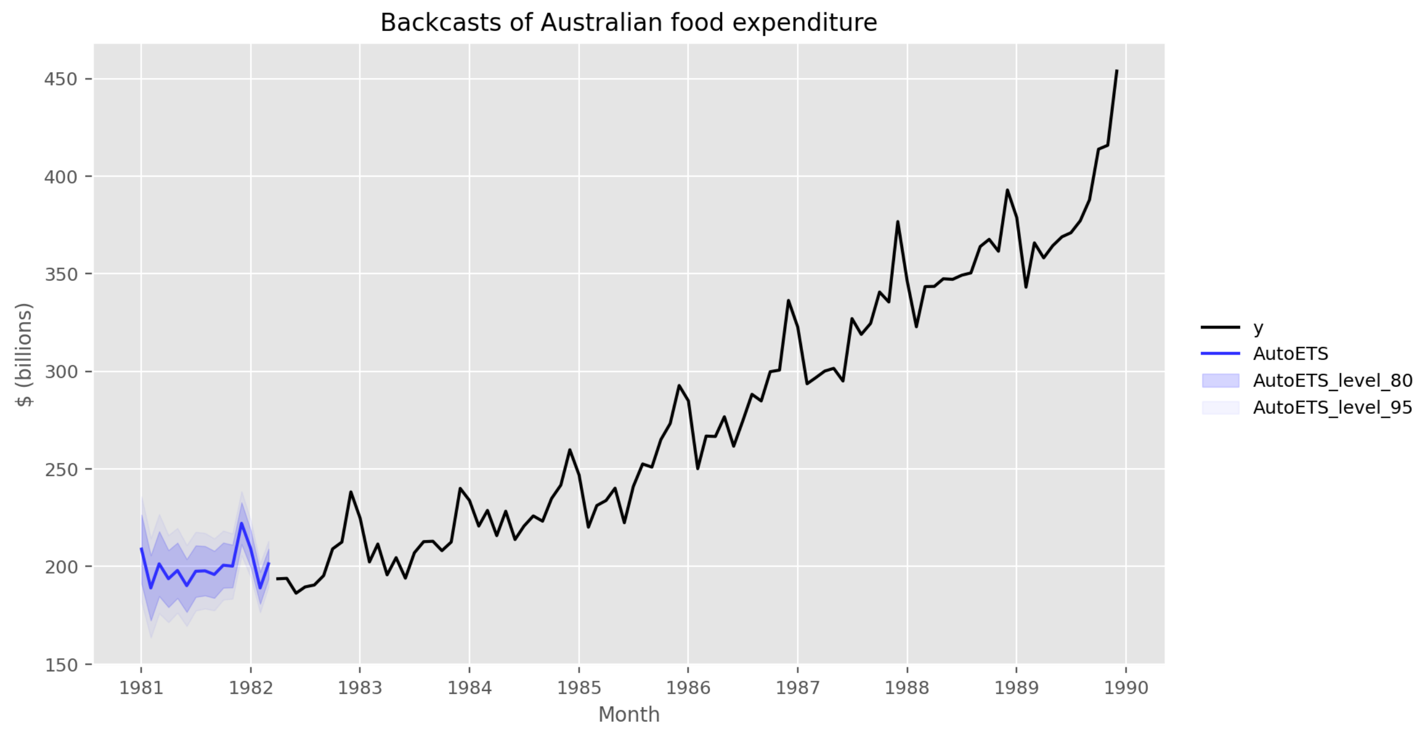

Figure 13.8: Backcasts for Australian monthly expenditure on cafés, restaurants and takeaway food services using an AutoETS model.

Most of the work here is in re-indexing the input data and then re-indexing the forecast DataFrame.

13.7 Very long and very short time series

Forecasting very short time series

We often get asked how *few*data points can be used to fit a time series model. As with almost all sample size questions, there is no easy answer. It depends on the *number of model parameters to be estimated and the amount of randomness in the data*. The sample size required increases with the number of parameters to be estimated, and the amount of noise in the data.

Some textbooks provide rules-of-thumb giving minimum sample sizes for various time series models. These are misleading and unsubstantiated in theory or practice. Further, they ignore the underlying variability of the data and often overlook the number of parameters to be estimated as well. There is, for example, no justification for the magic number of 30 often given as a minimum for ARIMA modelling. The only theoretical limit is that we need more observations than there are parameters in our forecasting model. However, in practice, we usually need substantially more observations than that.

Ideally, we would test if our chosen model performs well out-of-sample compared to some simpler approaches. However, with short series, there is not enough data to allow some observations to be withheld for testing purposes, and even time series cross validation can be difficult to apply. The AICc is particularly useful here, because it is a proxy for the one-step forecast out-of-sample MSE. Choosing the model with the minimum AICc value allows both the number of parameters and the amount of noise to be taken into account.

What tends to happen with short series is that the AICc suggests simple models because anything with more than one or two parameters will produce poor forecasts due to the estimation error.

Forecasting very long time series

Most time series models do not work well for very long time series. The problem is that real data do not come from the models we use. When the number of observations is not large (say up to about 200) the models often work well as an approximation to whatever process generated the data. But eventually we will have enough data that the difference between the true process and the model starts to become more obvious. An additional problem is that the optimisation of the parameters becomes more time consuming because of the number of observations involved.

What to do about these issues depends on the purpose of the model. A more flexible and complicated model could be used, but this still assumes that the model structure will work over the whole period of the data. A better approach is usually to allow the model itself to change over time. ETS models are designed to handle this situation by allowing the trend and seasonal terms to evolve over time. ARIMA models with differencing have a similar property. But dynamic regression models do not allow any evolution of model components.

If we are only interested in forecasting the next few observations, one simple approach is to throw away the earliest observations and only fit a model to the most recent observations. Then an inflexible model can work well because there is not enough time for the relationships to change substantially.

For example, we fitted a dynamic harmonic regression model to 26 years of weekly gasoline production in Section 13.1. It is, perhaps, unrealistic to assume that the seasonal pattern remains the same over nearly three decades. So we could simply fit a model to the most recent years instead.

13.8 Dealing with outliers and missing values

Real data often contains missing values, outlying observations, and other messy features. Dealing with them can sometimes be troublesome.

Outliers

Outliers are observations that are very different from the majority of the observations in the time series. They may be errors, or they may simply be unusual. (See Section 7.3 for a discussion of outliers in a regression context.) None of the methods we have considered in this book will work well if there are extreme outliers in the data. In this case, we may wish to replace them with an estimate that is more consistent with the majority of the data.

Simply replacing outliers without thinking about why they have occurred is a dangerous practice. They may provide useful information about the process that produced the data, which should be taken into account when forecasting. However, if we are willing to assume that the outliers are genuinely errors, or that they won’t occur in the forecasting period, then replacing them can make the forecasting task easier.

Figure 13.9 shows the number of visitors to the Adelaide Hills region of South Australia. There appears to be an unusual observation in 2002 Q4.

```python
tourism = pd.read_csv("../data/tourism.csv", parse_dates=["ds"])
tourism = tourism.query('Region == "Adelaide Hills" & Purpose == "Visiting"')
tourism.insert(0, "unique_id", "trips")
tourism = tourism[["unique_id", "ds", "y"]]

plot_series(
    tourism,
    xlabel="Quarter [1Q]",
    ylabel="Number of trips",
    title="Quarterly overnight trips to Adelaide Hills",
)
```

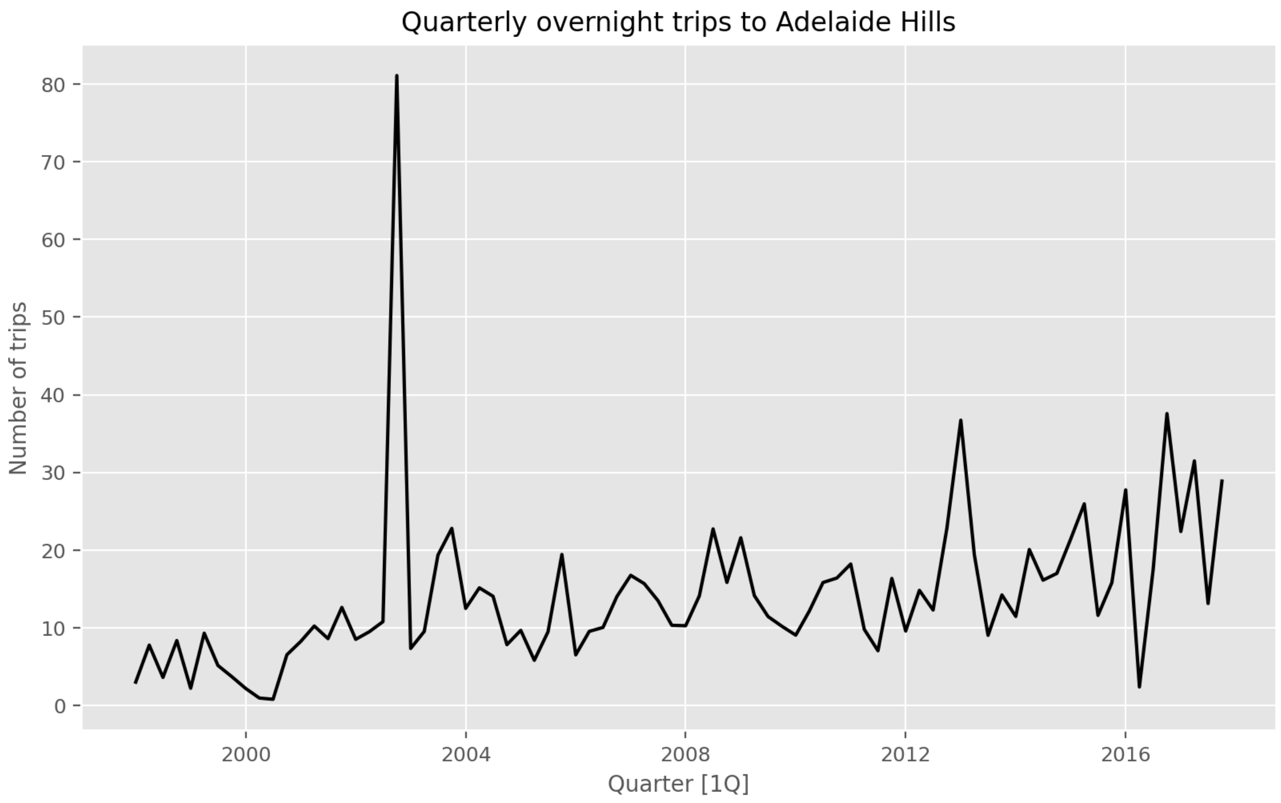

Figure 13.9: Number of overnight trips to the Adelaide Hills region of South Australia.

One useful way to find outliers is to apply an STL decomposition (using `MSTL()` with a single seasonality). Then any outliers should show up in the remainder series.

```
sf = StatsForecast(models=[MSTL(season_length=4, stl_kwargs={"robust": True})], freq="QS")

sf = sf.fit(tourism)
dcmp = sf.fitted_[0, 0].model_

fig, axes = plt.subplots(nrows=3, ncols=1, sharex=True)
sns.lineplot(data=dcmp, x=dcmp.index, y="data", ax=axes[0], color="black")
sns.lineplot(data=dcmp, x=dcmp.index, y="trend", ax=axes[1], color="black")
sns.lineplot(data=dcmp, x=dcmp.index, y="remainder", ax=axes[2], color="black")
fig.suptitle("STL decomposition", ha='center')
fig.text(0.5, 0.935, "Trips = trend + remainder", ha='center')
#fig.subplots_adjust(top=0.93)
plt.xlabel("")
plt.show()
```

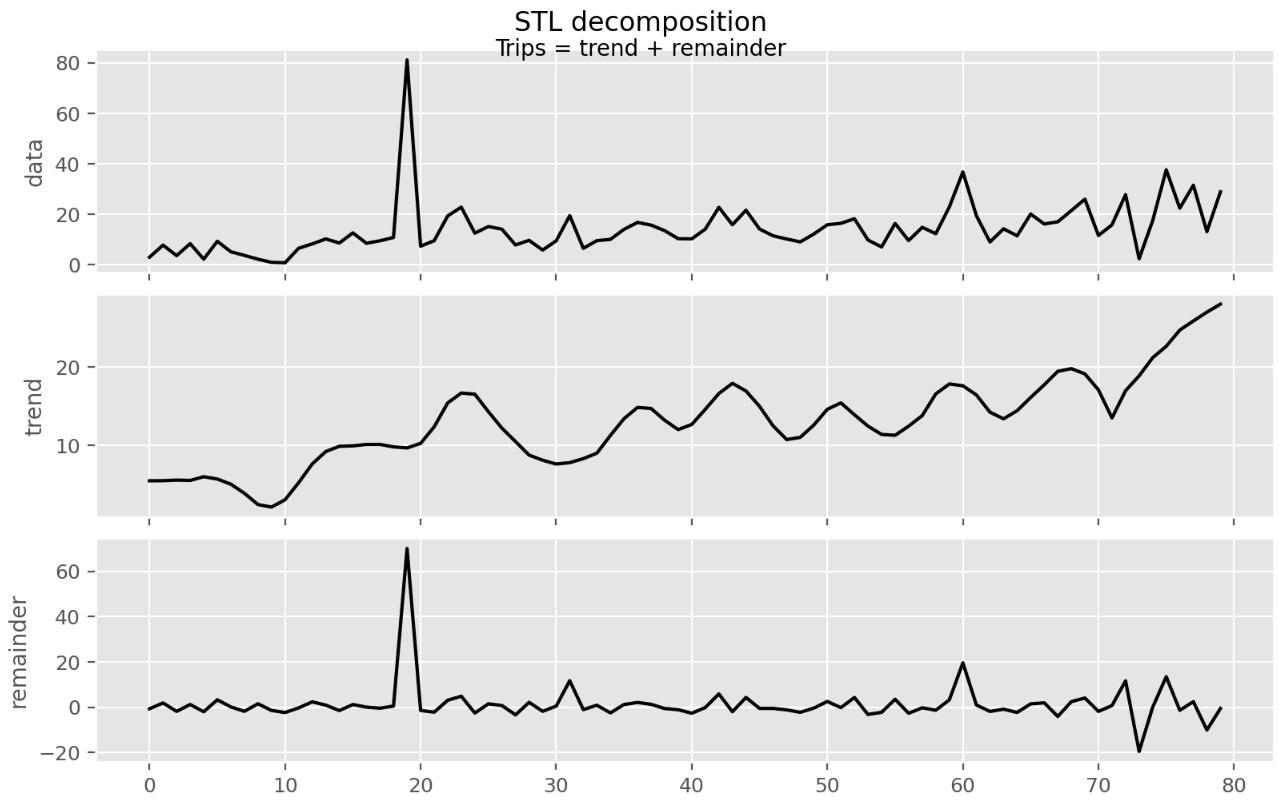

Figure 13.10: Decomposition of the number of overnight trips to the Adelaide Hills region of South Australia.

In the above example the outlier was easy to identify. In more challenging cases, using a boxplot of the remainder series would be useful. We can identify as outliers those that are greater than 1.5 interquartile ranges (IQRs) from the central 50% of the data. If the remainder was normally distributed, this would show 7 in every 1000 observations as "outliers". A stricter rule is to define outliers as those that are greater than 3 interquartile ranges (IQRs) from the central 50% of the data, which would make only 1 in 500,000 normally distributed observations to be outliers. This is the rule we prefer to use.

```
q1 = dcmp["remainder"].quantile(0.25)
q3 = dcmp["remainder"].quantile(0.75)
iqr = q3 - q1

outliers = dcmp.loc[(dcmp["remainder"] < (q1 - 3 * iqr)) | (dcmp["remainder"] > (q3 + 3 * iqr))]
outliers
```

	data	trend	seasonal	remainder
19	81.102	9.674	1.393	70.035
60	36.718	17.579	-0.425	19.564
73	2.372	18.843	3.014	-19.485
75	37.566	22.604	1.515	13.447

This finds the one outlier that we suspected from Figure 13.9, along with three additional ones. Something similar could be applied to the full data set to identify unusual observations in other series.

Missing values

Missing data can arise for many reasons, and it is worth considering whether the missingness will induce bias in the forecasting model. For example, suppose we are studying sales data for a store, and missing values occur on public holidays when the store is closed. The following day may have increased sales as a result. If we fail to allow for this in our forecasting model, we will most likely under-estimate sales on the first day after the public holiday, but over-estimate sales on the days after that. One way to deal with this kind of situation is to set the missing values to zero and use the `AutoARIMA()` function with dummy variables indicating whether the day is a public holiday or the day after a public holiday. No automated method can handle such effects on its own as they depend on the specific forecasting context.

In other situations, the missingness may be essentially random. For example, someone may have forgotten to record the sales figures, or the data recording device may have malfunctioned. In these cases, how to replace the missing values depends on the context of the data.

Keep in mind that `StatsForecast` does not allow missing values in the time series data. If you have missing values, there are at least two ways to handle the problem. First, we could just take the section of data after the last missing value, assuming there is a long enough series of observations to produce meaningful forecasts. Alternatively, we could replace the missing values with estimates.

We will replace the outlier identified in Figure 13.10 by an estimate using the `interpolate` function from `pandas`. The resulting series is shown in Figure 13.11.

```
tourism.loc[tourism["y"].isin(outliers["data"]), "y"] = np.nan
tourism["y"] = tourism["y"].interpolate()
plot_series(
    tourism,
    xlabel="Quarter [1Q]",
    ylabel="Number of trips",
    title="Quarterly overnight trips to Adelaide Hills",
)
```

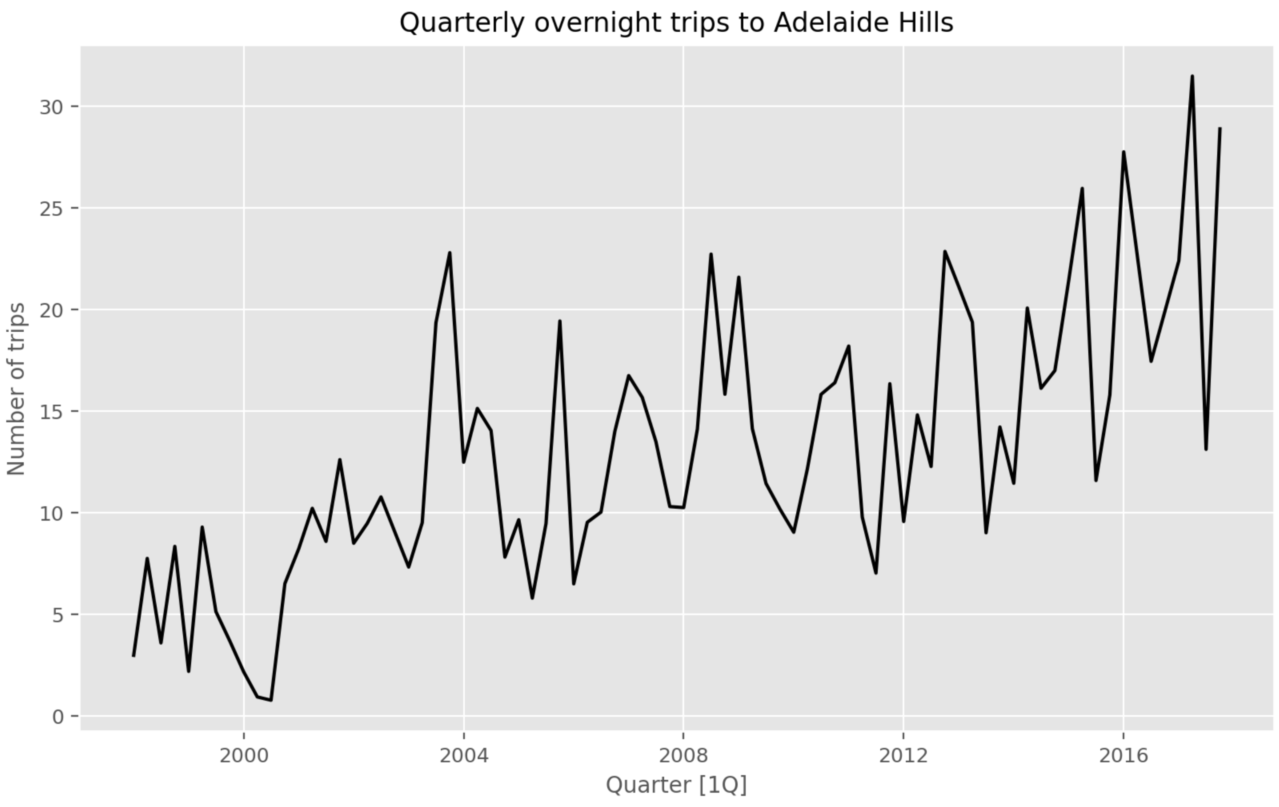

Figure 13.11: Number of overnight trips to the Adelaide Hills region of South Australia with the 2002Q4 outlier being replaced using interpolation.

13.9 Further reading

So many diverse topics are discussed in this chapter, that it is not possible to point to specific references on all of them. The last chapter in Ord, Fildes, and Kourentzes (2017) also covers "Forecasting in practice" and discusses other issues that might be of interest to readers.

13.10 Used modules and classes

StatsForecast

- `StatsForecast` class - Core forecasting engine
- `AutoETS` model - For automatic exponential smoothing
- `MSTL` model - For multiple seasonal decomposition
- `AutoARIMA` model - For automatic ARIMA modeling
- `CrostonClassic`, `CrostonOptimized`, `CrostonSBA` models - For intermittent demand forecasting

UtilsForecast

- `plot_series` utility - For time series visualization
- `evaluate` utility - For forecast accuracy evaluation
- `rmse`, `mae`, `mape`, `mase` metrics - For forecast error calculation

← Chapter 12 Advanced forecasting methods

Chapter 14 Neural networks →