CHAPTER 2

Finding and Wrangling Time Series Data

In this chapter we discuss problems that might arise while you are preprocessing time series data. Some of these problems will be familiar to experienced data analysts, but there are specific difficulties posed by timestamps. As with any data analysis task, cleaning and properly processing data is often the most important step of a timestamp pipeline. Fancy techniques can't fix messy data.

Most data analysts will need to find, align, scrub, and smooth their own data either to learn time series analysis or to do meaningful work in their organizations. As you prepare data, you'll need to do a variety of tasks, from joining disparate columns to resampling irregular or missing data to aligning time series with different time axes. This chapter helps you along the path to an interesting and properly prepared time series data set.

We discuss the following skills useful for finding and cleaning up time series data:

- Finding time series data from online repositories
- Discovering and preparing time series data from sources not originally intended for time series
- Addressing common conundrums you will encounter with time series data, especially the difficulties that arise from timestamps

After reading this chapter, you will have the skills needed to identify and prepare interesting sources of time series data for downstream analysis.

Where to Find Time Series Data

If you are interested in where to find time series data and how to clean it, the best resource for you in this chapter depends on which of these is your main goal:

- Finding an appropriate data set for learning or experimentation purposes
- Creating a time series data set out of existing data that is not stored in an explicitly time-oriented form

In the first case, you should find existing data sets with known benchmarks so you can see whether you are doing your analysis correctly. These are most often found as contest data sets (such as Kaggle) or repository data sets. In these cases, you will likely need to clean your data for a specific purpose even if some preliminary work has been done for you.

In the second case, you should think about effective ways to identify interesting timestamped data, turn it into a series, clean it, and align it with other timestamped data to make interesting time series data. I will refer to these found in the wild data sets as *found time series* (this is my own term and not technical language).

We discuss both prepared data sets and found time series next.

Prepared Data Sets

The best way to learn an analytical or modeling technique is to run through it on a variety of data sets and see both how to apply it and whether it helps you reach a concrete goal. In such cases it is helpful to have prepared options.

While time series data is everywhere, it is not always easy to find the kind of data you want when you want it. If you often find yourself in this position, you will want to get familiar with some commonly used time series data repositories, so we'll discuss a few options you should consider.

The UCI Machine Learning Repository

The UCI Machine Learning Repository (see Figure 2-1) contains around 80 time series data sets, ranging from hourly air quality samples in an Italian city to Amazon file access logs to diabetes patients' records of activity, food, and blood glucose information. These are very different kinds of data, and a look at the files shows they reflect distinct ways of tracking information across time, yet each is a time series.

Figure 2-1. The UCI Machine Learning Repository includes an annotated list of time series data sets.

Consider the very first data set listed under the Time Series section in the UCI repository, which is a data set about absenteeism at work (see Figure 2-2).

A quick look at the data reveals that the time columns are limited to "Month of absence," "Day of the week," and "Seasons," with no column for year. There are duplicate time indices, but there is also a column indicating employee identity so that we can differentiate between these duplicate time points. Finally, there are various employee attribute columns.

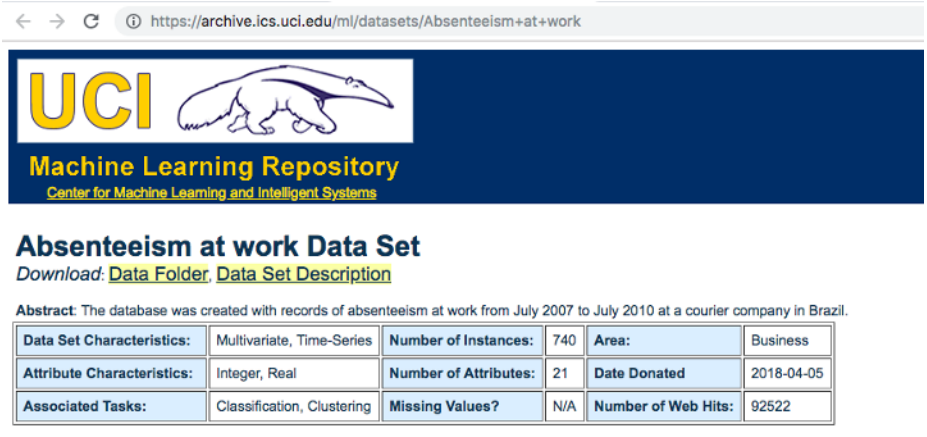

Figure 2-2. The absenteeism at work data set is the first in the list of time series data sets in the UCI Machine Learning Repository.

This data set could be quite challenging to process, because you would first need to determine whether the data was all from one year or whether the cycling of months from 1 to 12 several times through the row progressions indicates that the year is changing. You would also need to decide whether to look at the problem in the aggregate, from a total absenteeism per unit time perspective, or to look at absenteeism per ID for those reported in the data set (see Figure 2-3). In the first case you would have a single time series, whereas in the latter case you would have multiple time series with overlapping timestamps. How you looked at the data would depend on what question you wanted to answer.

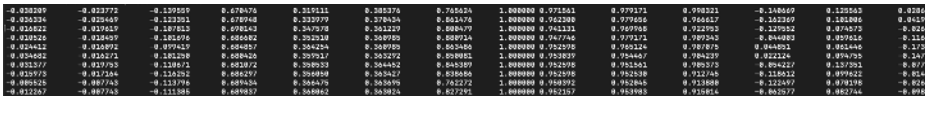

Figure 2-3. The first few lines of one file in the Australian sign language data set. As you can see, this is a wide file format.

Contrast the absenteeism data set with another data set early on in the list, the Australian Sign Language signs data set, which includes recordings from a Nintendo PowerGlove as subjects signed in Australian Sign Language. The data is formatted as wide CSV files, each within a folder indicating the individual measured and with a filename indicating the sign.

In this data set, the columns are unlabeled and do not have a timestamp. This is nonetheless time series data; the time axis counts the time steps forward, regardless of when the actual events happened. Notice that for the purpose of thinking about signs

as time series, it doesn't matter what the unit of time is; the point is the sequencing rather than the exact time. In that case, all you would care about is the ordering of the event, and whether you could assume or confirm from reading the data description that the measurements were taken at regular intervals.

As we can see from inspecting these two data sets, you will run into all sorts of data munging challenges. Some of the problems we've already noticed are:

- Incomplete timestamps
- Time axes can be horizontal or vertical in your data
- Varying notions of time

The UEA and UCR Time Series Classification Repository

The UEA and UCR Time Series Classification Repository is a newer effort providing a common set of time series data available for experimentation and research in time series classification tasks. It also shows a very diverse set of data. We can see this by looking at two data sets.

One data set is a yoga movement classification task. The classification task involves distinguishing between two individual actors who performed a series of transitions between yoga poses while images were recorded. The images were converted to a one-dimensional series. The data is stored in a CSV file, with the label as the leftmost column and the remaining columns representing time steps. Time passes from left to right across the columns, rather than from top to bottom across the rows.

Plots of two sample time series per gender are shown in Figure 2-4.

Univariate Versus Multivariate Time Series

The data sets we have looked at so far are *univariate* time series; that is, they have just one variable measured against time.

Multivariate time series are series with multiple variables measured at each timestamp. They are particularly rich for analysis because often the measured variables are interrelated and show temporal dependencies between one another. We will encounter multivariate time series data later.

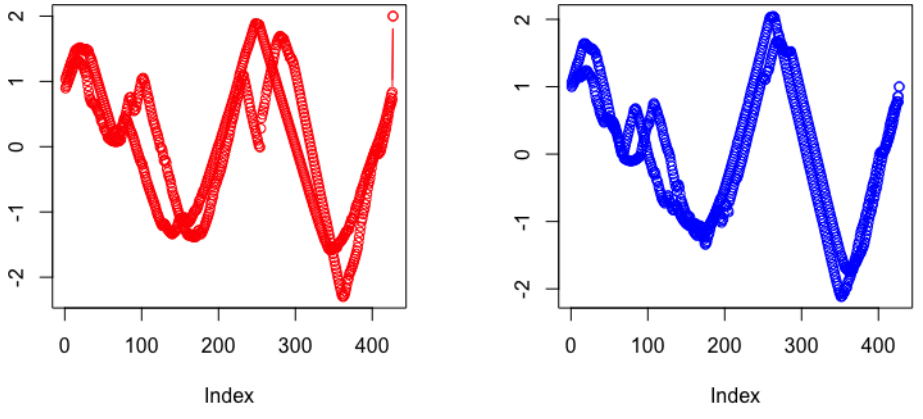

Figure 2-4. Plots of a male and a female actor performing a yoga move repeatedly. We plot two sample time series per actor. There are no explicit time labels on the x-axis. Rather than the unit of time, what is important is whether the x-axis data points are evenly spaced, as they are presented here.

Also consider the wine data set, in which wines were classified by region according to the shapes of their spectra. So what makes this relevant to time series analysis? A *spectrum* is a plot of light wavelength versus intensity. Here we see a time series classification task where there is no passage of time at all. Time series analysis applies, however, because there is a unique and meaningful ordering of the x-axis, with a concrete meaning of distance along that axis. Time series analysis distinguishes itself from cross-sectional analysis by using the additional information provided by ordering on the x-axis, whether it is time or wavelength or something else. We can see a plot of such a "time" series in Figure 2-5. There is no temporal element, but we are nonetheless looking at an ordered series of data, so the usual ideas of time series apply.[1]

1 To learn more about this kind of data analysis, also see references on

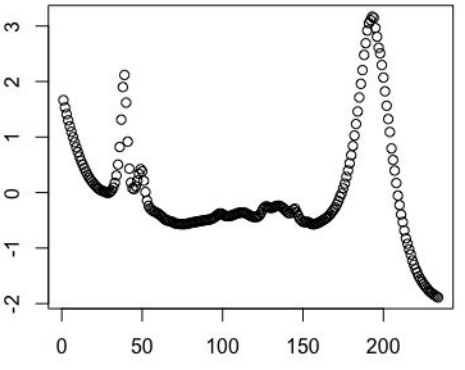

Figure 2-5. A sample spectrum of the wine sampled in the UCI wine data set. Peaks in the curve indicate wavelength regions that have particularly high rates of absorption. The wavelengths are uniformly spaced along the x-axis, whereas the y-axis indicates the rate of absorption, also on a linear scale. We can use time series analysis to compare curves, such as the one above, to one another.

Government time series data sets

The US government has been a reliable provider of time series data for decades, or even centuries. For example, the NOAA National Centers for Environmental Information publishes a variety of time series data relating to temperatures and precipitation at granularities as fine as every 15 minutes for all weather stations across the country. The Bureau of Labor Statistics publishes a monthly index of the national unemployment rate. The Centers for Disease Control and Prevention publishes weekly flu case counts during the flu season. The Federal Reserve Bank of St. Louis offers a particularly generous and helpful set of economic time series data.

For initial forays into time series analysis, I recommend that you access these real-world government data sets only for exploratory analysis and visualization. It can be difficult to learn on these data sets because they present extremely complicated problems. For example, many economists spend their entire careers trying to predict the unemployment rate in advance of its official publication, with only limited success.

For the important but intractable problems faced by governments, predicting the future would not only be socially beneficial but also highly remunerative. Many smart and well-trained people are attacking these problems even as the state of the art remains somewhat disappointing. It is great to work on difficult problems, but it is not a good idea to learn on such problems.

Found Time Series in Government Data

It's not difficult to find a variety of timely and promising data sets from government websites, such as those just mentioned. However, there is also a lot of potential for found time series in government data. For example, imagine laying out parallel time series for economics and climate, or correlating different crimes with government spending patterns. That data would come from a variety of sources, and you would need to integrate them.

You should be wary of working with found time series in government data sets for a number of reasons. Recording conventions, column names, or even column definitions shift over time without accompanying documentation. Projects start and stop depending on politics, budgets, and other exogenous considerations. Also data formats from government websites can be very messy, and are usually messier and less continuous than comparable data sets in the private sector. This makes it particularly challenging to construct found time series.

Additional helpful sources

While we cannot extensively cover all good sources of time series data, there are several other repositories that you should explore:

CompEngine
: This "self organizing database of time-series data" has more than 25,000 time series databases that total almost 140 million individual data points. The emphasis of this repository, and the associated software it offers on its web interface, is to facilitate and promote *highly comparative time-series analysis* (hctsa). The goal of such analysis is to generate high-level insights and understanding of how many kinds of temporal behavior can be understood without discipline-specific data.

Mcomp and M4comp2018 R packages
: These R packages provide the competition data from the 1982 M-competition (1,001 time series), the M3 competition in 2000 (3,003 time series), and the M4 competition in 2018 (100,000 time series). These time series forecasting competitions were previously discussed in the Chapter 1 mention of Professor Rob Hyndman's history of time series forecasting. Additional time series forecasting competition data is included in R's *tscompdata* package. Finally, more specialized time series data sets can also be found in a variety of packages described in the CRAN repository listing of time series packages under the "Time Series Data" header.

Found Time Series

Earlier in the chapter, we discussed the concept of a found time series, which is time series data we put together ourselves from data sources in the wild. More specifically, such time series would be put together from individual data points recorded without any special allowances for time series analysis but with enough information to construct a time series. Putting together a time series of transactions for a particular customer from a SQL database that stores a company's transactions is a clean example. In such a case, the time series could be constructed so long as a timestamp, or some proxy for a timestamp, was saved in the database.[2] We can also imagine other time series constructed from the same data, such as a time series of total transaction volume per day for the company or total dollar volume for female customers per week. We can even imagine generating multivariate time series data, such as a time series that would separately indicate total volume of all customers under 18 per week, total dollars spent by women over 65 per week, and total ad spend by the company per week. This would give us three indicators at each time step, a multivariate time series.

Finding time series data in structured data not explicitly stored as a time series can be easy in the sense that timestamping is ubiquitous. Here are a few examples of where you'll see timestamps in your database:

Timestamped recordings of events
: If there is a timestamp on your data, you have the potential to construct a time series. Even if all you do is record the time a file was accessed with no other information, you have a time series. For example, in that case, you could model the delta time between timestamps with each marked according to its later timestamp, so that your time series would consist of time on your temporal axis and delta time on your value axis. You could go further, aggregating these delta times as means or totals over larger periods, or you could keep them individually recorded.

"Timeless" measurements where another measurement substitutes for time
: In some cases, time is not explicit in data but is accounted for in the underlying logic of the data set. For example, you may think of your data as "distance versus value" when the distance is being caused by a known experimental parameter, such as retracting a sensor from a position at a known rate. If you can map one of your variables to time, you have a time series. Alternately, if one of your axes has a known distance and ordering relationship (such as wavelength), you are also looking at time series data such as the case of the wine spectra mentioned earlier.

2 SQL databases are traditional databases using a table-based mechanism of storing data. For example, if you wanted to store customer transactions in a SQL database, you might have a table with customer information, including a unique identifier, and another table with transactions, each one including one of those unique customer identifiers.

Physical traces
: Many scientific disciplines record physical traces, be it for medicine, audiology, or weather. These used to be physically generated traces collected via analog processes, but nowadays they are stored in a digital format. These are also time series, although they may stored in a format that doesn't make this obvious, such as in an image file or in a single vector within a single field of a database.

Retrofitting a Time Series Data Collection from a Collection of Tables

The quintessential example of a found time series is one extracted from state-type and event-type data stored in a SQL database. This is also the most relevant example because so much data continues to be stored in traditional structured SQL databases.

Imagine working for a large nonprofit organization. You have been tracking a variety of factors that could lend themselves to time series analysis:

- An email recipient's reaction to emails over time: Did they open the emails or not?
- A membership history: Were there periods when a member let their membership lapse?
- Transaction history: When does an individual buy and can we predict this?

You could look at the data with several time series techniques:

- You can generate a 2D histogram of member responses to emails over time with a member-specific time line to get an idea of whether members develop fatigue from emails. (We'll illustrate the use of 2D histograms for understanding time series in Chapter 3.)
- You can turn donation predictions into a time series forecasting problem. (We'll discuss classic statistical forecasting in Chapter 4.)
- You could examine whether there are typical patterns of trajectories for member behavior in important situations. For example, is there a typical pattern of events that indicates when a member is about to leave your organization (perhaps three email deletes in a row)? In time series analysis, you could frame this as detecting a member's underlying state based on external actions. (We'll cover this when we discuss state space methods of time series analysis in Chapter 7.)

As we can see, there are many time series questions and answers in a simple SQL database. In many cases, organizations don't plan for time series analysis when they are designing their database schema. In such examples, we need to collect and assemble time series from disparate tables and sources.

A Worked Example: Assembling a Time Series Data Collection

If you are lucky enough to have several related data sources available, you will need to line them up together, possibly dealing with disparate timestamping conventions or different levels of granularity in the data. Let's create some numbers for the nonprofit example we were using. Suppose you have data shown in Table 2-1 through Table 2-3:

Table 2-1. The year each member joined and current status of membership

MemberId	YearJoined	MemberStatus
1	2017	gold
2	2018	silver
3	2016	inactive

Table 2-2. Number of emails you sent out in a given week that were opened by the member

MemberId	Week	EmailsOpened
2	2017-01-08	3
2	2017-01-15	2
1	2017-01-15	1

Table 2-3. Time a member donated to your organization

MemberId	Timestamp	DonationAmount
2	2017-05-22 11:27:49	1,000
2	2017-04-13 09:19:02	350
1	2018-01-01 00:15:45	25

You have likely worked with data in this tabular form. With such data, you can answer many questions, such as how the overall number of emails opened by a member correlates with the overall donations.

You can also answer time-oriented questions, such as whether a member donates soon after joining or long after. Without putting this data into a more time-series-friendly format, however, you cannot get at more granular behaviors that might help you predict when someone is likely to make a donation (say, based on whether they have recently been opening emails).

You need to put this data into a sensible format for time series analysis. There are a number of challenges you will need to address.

You should start by considering the temporal axes of the data we have. In the preceding tables we have three levels of temporal resolution:

- A yearly member status

- A weekly tally of emails opened
- Instantaneous timestamps of donations

You will also need to examine whether the data means what you think it means. For example, you would want to determine whether the member status is a yearly status or just the most recent status. One way to answer this is to check whether any member has more than one entry:

```
## python
>>> YearJoined.groupby('memberId').count().
                  groupby('memberStats').count()

1000
```

Here we can see that all 1,000 members have only one status, so that the year they joined is indeed likely to be the `YearJoined`, accompanied by a status that may be the member's current status or status when they joined. This distinction affects how you would use the status variable, so if you were going to analyze this data further you'd want to clarify with someone who knows the data pipeline. If you were applying a member's current status to an analysis of past data, that would be a *lookahead* because you would be inputting something into a time series model that could not be known at the time. This is why you would not want to use a status variable, such as `YearJoined`, without knowing when it was assigned.

What Is a Lookahead?

The term *lookahead* is used in time series analysis to denote any knowledge of the future. You shouldn't have such knowledge when designing, training, or evaluating a model. A lookahead is a way, through data, to find out something about the future earlier than you ought to know it.

A lookahead is any way that information about what will happen in the future might propagate back in time in your modeling and affect how your model behaves earlier in time. For example, when choosing hyperparameters for a model, you might test the model at various times in your data set, then choose the best model and start at the beginning of your data to test this model. This is problematic because you chose the model for one time knowing things that would happen at a subsequent time—a lookahead.

Unfortunately, there is no automated code or statistical test for a lookahead, so it is something you must be vigilant and thoughtful about.

Looking at the emails table, both the column name `week` and its contents suggest that the data is a weekly timestamp or time period. This must be an aggregate over the

week, so we should think of these timestamps as weekly periods rather than timestamps occurring one week apart.

You should assess some important characteristics. For example, you could start by asking how the weeks are reported in time. While we may not have information to restructure the table, if the week is divided in a somewhat strange way relative to our industry, we might want to know about this too. For analyzing human activities, it generally makes sense to look at the calendar week of Sunday through Saturday or Monday through Sunday rather than weeks that are less in line with the cycle of human activity. So, for example, don't arbitrarily start your week with January 1st.

You could also ask whether null weeks are reported? That is, do the weeks in which the member opened 0 emails have a place in the table? This matters when we want to do time-oriented modeling. In such cases we need to always have the null weeks present in the data because a 0 week is still a data point.

```
## python
>>> emails[emails.EmailsOpened < 1]

Empty DataFrame
Columns: [EmailsOpened, member, week]
Index: []
```

There are two possibilities: either nulls are not reported or members always have at least one email event. Anyone who has worked with email data knows that it's difficult to get people to open emails, so the hypothesis that members always open at least one email per week is quite unlikely. In this case, we can resolve this by looking at the history of just one user:

```
## python
>>> emails[emails.member == 998]
      EmailsOpened member  week
25464  1           998   2017-12-04
25465  3           998 2017-12-11
25466  3           998 2017-12-18
25467  3           998 2018-01-01
25468  3           998 2018-01-08
25469  2           998 2018-01-15
25470  3           998 2018-01-22
25471  2           998 2018-01-29
25472  3           998 2018-02-05
25473  3           998 2018-02-12
25474  3           998 2018-02-19
25475  2           998 2018-02-26
25476  2           998 2018-03-05
```

We can see that some weeks are missing. There aren't any December 2017 email events after December 18, 2017.

We can check this more mathematically by calculating how many weekly observations we should have between the first and last event for that member. First we calculate the length of the member's tenure, in weeks:

```
## python
>>> (max(emails[emails.member == 998].week) -
                    min(emails[emails.member == 998].week)).days/7
25.0
```

Then we see how many weeks of data we have for that member:

```
## python
>>> emails[emails.member == 998].shape
(24, 3)
```

We have 24 rows here, but we should have 26. This shows some weeks of data are missing for this member. Incidentally, we could also run this calculation on all members simultaneously with group-by operations, but it's more approachable to think about just one member for example purposes.

Why 26 Rows?

You may be surprised that we need 26 instead of 25 given the subtraction we just performed, but that was an incomplete calculation. When you work with time series data, one thing you should always ask yourself after doing this kind of subtraction is whether you should add 1 to account for the offset at the end. In other words, did you subtract the positions you wanted to count?

Consider this example. Let's say I have information for April 7th, 14th, 21st, and 28th. I want to know how many data points I should have in total. Subtracting 7 from 28 and dividing by 7 yields 21/7 or 3. However, I should obviously have four data points. I subtracted out April 7th and need to put it back in, so the proper calculation is the difference between the first and last days divided by 7, *plus 1* to account for the subtracted start date.

We'll move on to filling in the blanks so that we have a complete data set now that we have confirmed we do indeed have missing weeks. We can't be sure of identifying all missing weeks, since some may have occurred before our earliest recorded date or after our last recorded date. What we can do, however, is fill in the missing values between the first and last time a member had a non-null event.

It's a lot easier to fill in all missing weeks for all members by exploiting Pandas' indexing functionality, rather than writing our own solution. We can generate a `MultiIndex` for a Pandas data frame, which will create all combinations of weeks and members—that is, a Cartesian product:

```
## python
>>> complete_idx = pd.MultiIndex.from_product((set(emails.week),
                                               set(emails.member)))
```

We use this index to reindex the original table and fill in the missing values—in this case with 0 on the assumption that nothing recorded means there was nothing to record. We also reset the index to make the member and week information available as columns, and then name those columns:

```
## python
>>> all_email = emails.set_index(['week', 'member']).
                               reindex(complete_idx, fill_value = 0).
                               reset_index()
>>> all_email.columns = ['week', 'member', 'EmailsOpened']
```

Let's take a look at member 998 again:

```
## python
>>> all_email[all_email.member == 998].sort_values('week')
  week     member EmailsOpened
2015-02-09 998    0
2015-02-16 998    0
2015-02-23 998    0
2015-03-02 998    0
2015-03-09 998    0
```

Python's Pandas

Pandas is a data frame analysis package in Python that is used widely in the data science community. Its very name indicates its suitability for time series analysis: "Pandas" refers to "panel data," which is what social scientists call time series data.

Pandas is based on tables of data with row and column indices. It has SQL-like operations built in, such as group by, row selection, and key indexing. It also has time series–specific functionality, such as indexing by time period, downsampling, and time-based grouping operations.

If you are unfamiliar with Pandas, I strongly recommend looking at a brief overview, such as that provided in the official documentation.

Notice that we have a large number of zeros at the start. These are likely before the member joined the organization, so they would not have been on an email list. There are not too many kinds of analyses where we'd want to keep the member's truly null weeks around—specifically those weeks before the member ever indicated opening an email. If we had the precise date a member started receiving emails, we would have an objective cutoff. As it is, we will let the data guide us. For each member we determine the `start_date` and `end_date` cutoffs by grouping the email `DataFrame` per member and selecting the maximum and minimum week values:

```
## python
>>> cutoff_dates = emails.groupby('member').week.
                           agg(['min', 'max']).reset_index)
>>> cutoff_dates = cutoff_dates.reset_index()
```

We drop rows from the `DataFrame` that don't contribute sensibly to the chronology, specifically 0 rows before each member's first nonzero count:

```
## python
>>> for _, row in cutoff_dates.iterrows():
>>>   member     = row['member']
>>>   start_date = row['min']
>>>   end_date   = row['max']
>>>   all_email.drop(
            all_email[all_email.member == member]
           [all_email.week < start_date].index, inplace=True)
>>>   all_email.drop(all_email[all_email.member == member]
           [all_email.week > end_date].index, inplace=True)
```

< or <= ?

We use the < and > operations, without equality, because the `start_date` and `end_date` are inclusive of the meaningful data points and because we are dropping data, not retaining data, as our code is written. In this case we want to include those weeks in our analysis because they were the first and last meaningful data points.

You will do well to work with your data engineers and database administrators to convince them to store the data in a time aware manner, especially with respect to how timestamps are created and what they mean. The more you can solve problems upstream, the less work you have to do downstream in the data pipeline.

Now that we have cleaned up our email data, we can consider new questions. For example, if we want to think about the relationship of member email behavior to donations, we can do a few things:

- Aggregate the `DonationAmount` to weekly so that the time periods are comparable. Then it becomes reasonable to ask whether donations correlate in some way to member responses to emails.
- Treat the prior week's `EmailsOpened` as a predictor for a given week's `DonationAmount`. Note that we have to use the prior week because `EmailsOpened` is a summary statistic for the week. If we want to predict a Wednesday donation, and our `EmailsOpened` summarizes email opening behavior from Monday to Sunday, then using the same week's information will potentially give us information about what the member did subsequent to when we could have known it (for example, whether they opened an email on the Friday after the donation).

Constructing a Found Time Series

Consider how to relate the email and donations data to one another. We can downsample the donation data to turn it into a weekly time series, comparable to the email data. As an organization, we are interested in the total weekly amounts, so we aggregate the timestamps into weekly periods by summing. More than one donation in a week is unlikely, so the weekly donation amounts will reflect the individual donation amounts for most donors.

```
## python
>>> donations.timestamp = pd.to_datetime(donations.timestamp)
>>> donations.set_index('timestamp', inplace = True)
>>> agg_don = donations.groupby('member').apply(
              lambda df: df.amount.resample("W-MON").sum().dropna())
```

In this code we first convert a string character to a proper timestamped data class so as to benefit from Pandas' built-in date-related indexing. We set the timestamp as an index, as is necessary for resampling a data frame. Finally for the data frame that we obtain from subsetting down to each member, we group and sum donations by week, drop weeks that do not have donations, and then collect these together.

Note that we resampled with an anchored week so that we will match the same weekly dates we already have in our email table. Note also that a week anchored to "Monday" makes sense from a human perspective.

We now have donation information and email information sampled at the same frequency, and we can join them. Pandas makes this simple so long as we anchor the weeks to the same day of the week, as we've done already. We can iterate through each member and merge the data frames per member:

```
## python
>>> for member, member_email in all_email.groupby('member'):
>>>     member_donations = agg_donations[agg_donations.member
                                         == member]

>>>      member_donations.set_index('timestamp', inplace = True)
>>>      member_email.set_index    ('week', inplace = True)

>>>      member_email = all_email[all_email.member == member]
>>>      member_email.sort_values('week').set_index('week')

>>>      df = pd.merge(member_email, member_donations, how = 'left',
                       left_index = True,
                       right_index = True)
>>>      df.fillna(0)

>>>      df['member'] = df.member_x
>>>      merged_df = merged_df.append(df.reset_index()
                        [['member', 'week', 'emailsOpened', 'amount']])
```

We now have our email and donations data lined up per member. For each member we include only meaningful weeks, and not weeks that appear to be before or after their membership period.

We might treat the email behavior as a "state" variable relative to the donation behavior, but we probably want to carry forward the state from the previous week so as to avoid a lookahead. Suppose, for example, we were building a model that uses email behavior to predict a member's next donation. In this case we might consider looking at the pattern of email opening over time as a possible indicator. We would need to line up a given week's donation with the previous week's email behavior. We can easily take our data, processed to align week-to-week, and then shift by the appropriate number of weeks. If, say, we want to shift the donation a week forward, we can easily do so with the shift operator, although we would need to be sure to do this on a per-member basis:

```
## python
>>> df = merged_df[merged_df.member == 998]
>>> df['target'] = df.amount.shift(1)
>>> df = df.fillna(0)
>>> df
```

It's good practice to store this target in a new column rather than overwrite the old, particularly if you are not also bringing forward the timestamp for the donation amount separately. We have shifted the donation amount one week into the future using Pandas' built-in shift functionality. You can also shift back in time with negative numbers. Generally you will have more predictors than targets, so it makes sense to shift your targets. We can see the outcome of the code here:

```
amount   emailsOpened   member    week         target
 0        1             998   2017-12-04      0
 0        3             998   2017-12-11      0
 0        3             998   2017-12-18      0
 0        0             998   2017-12-25      0
 0        3             998   2018-01-01      0
50        3             998   2018-01-08      0
 0        2             998   2018-01-15     50
```

Now that we have filled in the missing rows, we have the desired 26 rows for member 998. Our data is now cleaner and more complete.

To recap, these are the time-series-specific techniques we used to restructure the data:

1. *Recalibrate the resolution of our data* to suit our question. Often data comes with more specific time information than we need.
2. Understand how we can *avoid lookahead* by not using data for timestamps that produce the data's availability.

3. Record *all relevant time periods* even if "nothing happened." A zero count is just as informative as any other count.
4. *Avoid lookahead* by not using data for timestamps that produce information we shouldn't yet know about.

So far, we have created raw found time series by aligning our donation and email time series to be sampled for the same points in time and at the same frequency. However, we haven't done a thorough job of cleaning up this data or fully exploring before analysis. That's something we'll do in Chapter 3.

Timestamping Troubles

Timestamps are quite helpful for time series analysis. From timestamps, we can extrapolate a number of interesting features, such as time of day or day of the week. Such features can be important to understanding your data, especially for data concerning human behavior. However, timestamps are tricky. Here we discuss some of the difficulties of timestamped data.

Whose Timestamp?

The first question you should ask when seeing a timestamp is what process generated the timestamp, how, and when. Often an event happening is not coincident with an event being recorded. For example, a researcher might write something down in their notebook and later transcribe it to a CSV file used as a log. Does the timestamp indicate when they wrote it down or when they transcribed it to the CSV file? Or a mobile app user may take actions that trigger logging when their phone is offline, so that the data is only later uploaded to your server with some combination of timestamps. Such timestamps could reflect when the behavior took place, when the action was recorded by the app, when the metadata was uploaded to the server, or when the data was last accessed for download from the server back to the app (or any number of other events along the data pipeline).

Timestamps may appear at first to offer clarity but this quickly falls away if proper documentation is lacking. When you're looking at novel timestamps, your first step should always be to find out as best you can what established the time of an event.

A concrete example will illustrate these difficulties. Suppose you are looking at data taken from a mobile app for weight loss and see a meal diary with entries such as those in Table 2-4.

Table 2-4. Sample meal diary from a weight loss app

Time	Intake
Mon, April 7, 11:14:32	pancakes
Mon, April 7, 11:14:32	sandwich
Mon, April 7, 11:14:32	pizza

It's possible this user had pancakes, a sandwich, and a pizza all at once, but there are more likely scenarios. Did the user specify this time or was it automatically created? Does the interface perhaps offer an automatic time that the user can adjust or choose to ignore? Some answers to these questions would explain the identical timestamps better than the possibility that a user trying to lose weight would have had pancakes with a side of pizza and a sandwich appetizer.

Even if the user did have all these foods at 11:14, where in the world was it 11:14? Was this the user's local time or a global clock? Even in the unlikely case that the user had all this food at one meal, we still don't know very much about the temporal aspect of the meal from these rows alone. We don't know if this was breakfast, lunch, dinner, or a snack. To say something interesting to the user, we'd want to be able to talk in concrete terms about local hour of the day, which we can't do without time zone information.

The best way to address these questions is to see all the code that collects and stores the data or talk to the people who wrote that code. After going through all available human and technical data specifications on the system, you should also try the full system out yourself to ensure that the data behaves as you have been told it does. The better you understand your data pipeline, the less likely you are to ask the wrong questions because your timestamps don't really mean what you think they do.

You bear the ultimate responsibility for understanding the data. People who work upstream in the pipeline don't know what you have in mind for analysis. Try to be as hands-on as possible in assessing how timestamps are generated. So, if you are analyzing data from a mobile app pipeline, download the app, trigger an event in a variety of scenarios, and see what your own data looks like. You're likely to be surprised about how your actions were recorded after speaking to those who manage the data pipeline. It's hard to track multiple clocks and contingencies, so most data sets will flatten the temporal realities. You need to know exactly how they do so.

Guesstimating Timestamps to Make Sense of Data

If you are dealing with legacy data pipelines or undocumented data, you may not have the option of exploring a working pipeline and talking to those who maintain it. You will need to do some empirical investigation to understand whether you can make inferences about what the timestamps mean:

- Reading the data as we did in the previous example, you can generate initial hypotheses about what the timestamps mean. In the preceding case, look at data for multiple users to see whether the same pattern (multiple rows with identical timestamps and improbable single meal contents) held or whether this was an anomaly.
- Using aggregate-level analyses, you can test hypotheses about what timestamps mean or probably mean. For the preceding data, there are a couple of open questions:
  - Is the timestamp local or universal time?
  - Does the time reflect a user action or some external constraint, such as connectivity?

Local or Universal Time?

Most timestamps are stored in universal (UTC) time or in a single time zone, depending on the server's location but independent of the user's location. It is quite unusual to store data according to local time. However, we should consider both possibilities, because both are found in "the wild."

We form the hypothesis that if the time is a local timestamp (local to each user), we should see daily trends in the data reflecting daytime and nighttime behavior. More specifically, we should expect not to see much activity during the night when our users are sleeping. For our mobile app example, if we create a histogram of meal time counts per hour, there should be hours with significantly fewer meals logged, because in most cultures people don't eat in the middle of the night.

If we did not see a day-oriented pattern in the hours as displayed, we could conclude that the data was most likely timestamped by some universal clock and that the user base must be spread out across many time zones. In such a case, it would be quite challenging to extrapolate individual users' local hours (assuming time zone information wasn't available). We might consider individual explorations per user to see if we could code heuristics to label users approximately by time zone, but such work is both computationally taxing and also not always accurate.

Even if you cannot pinpoint an exact user time zone, it's still useful to have a global timestamp available. For starters, you can determine likely usage patterns for your app's servers, knowing when meals are most frequently logged at a given time of day and day of the week. You can also calculate the time differential between each meal the user recorded, knowing that since the timestamps are in absolute time, you don't need to worry about whether the user changed time zones. In addition to sleuthing, this would be interesting as a form of feature generation:

```
## python
>>> df['dt'] = df.time - df.time.shift(-1)
```

The `dt` column would be a feature you could pass forward in your analysis. Using this time differential could also give you an opportunity to estimate each user's time zone. You could look at the time of day when the user generally had a long `dt`, which could point to when nighttime occurs for that user. From there you could begin to figure out each individual's "nighttime" without having to do peak-to-peak-style analysis.

User behavior or network behavior?

To return to another question posed by our short data sample, we asked whether our user had a strange meal or whether our timestamps relate to upload activity.

The same analyses used to pinpoint a user's time zone are applicable to determining whether the timestamps are a function of user or network behavior. Once you had a `dt` column (as previously calculated), you could look for clusters of 0s and qualitatively determine whether they were likely a single behavioral event or a single network event. You could also look at whether the `dts` appeared to be periodic across different days. If they were a function of user behavior, they would be more likely to be periodic than if they were a function of network connectivity or other software-related behaviors.

To summarize, here are some questions you could likely address with the available data set, even with little or no information about how timestamps are generated:

- Use differences in timestamps per user to get a sense of spacing between meals or spacing between data entries (depending on your working hypothesis of what the times indicate, a user behavior or a network behavior).
- Describe aggregate user behavior to determine when your servers are most likely to be active in the 24-hour cycle.

Time Zones

In the best-case scenario, your data is timestamped in UTC time. Most databases and other storage systems will default to this. However, there are some situations in which you are likely to run into data that is not UTC timestamped:

- Timestamps created in ways that did not use date-specific data objects, such as API calls. Such API calls use strings rather than time-specific objects.
- Data created "by hand" in small, local organizations where time zones are not relevant, such as spreadsheets generated by business analysts or field biologists.

What's a Meaningful Time Scale?

You should take the temporal resolution of the timestamping you receive with a grain of salt, based on domain knowledge about the behavior you are studying and also based on the details you can determine relating to how the data was collected.

For example, imagine you are looking at daily sales data, but you know that in many cases managers wait until the end of the week to report figures, estimating rough daily numbers rather than recording them each day. The measurement error is likely to be substantial due to recall problems and innate cognitive biases. You might consider changing the resolution of your sales data from daily to weekly to reduce or average out this systematic error. Otherwise, you should build a model that factors in the possibility of biased errors across different days of the week. For example, it may be that managers systematically overestimate their Monday performance when they report the numbers on a Friday.

Psychological Time Discounting

Time discounting is a manifestation of a phenomenon known as *psychological distance*, which names our tendency to be more optimistic (and less realistic) when making estimates or assessments that are more "distant" from us. Time discounting predicts that data reported from further in the past will be biased systematically compared to data reported from more recent memory. This is distinct from the more general problem of forgetting and implies a nonrandom error. You should keep this in mind whenever you are looking at human-generated data that was entered manually but not contemporaneously with the event recorded.

Another situation involves physical knowledge of the system. For example, there is a limit to how quickly a person's blood glucose can change, so if you are looking at a series of blood glucose measurements within seconds of one another, you should probably average them rather than treating them as distinct data points. Any physician will tell you that you are studying the error of the device rather than the rate of change of blood glucose if you are looking at many measurements within a few seconds of one another.

Humans Know Time Is Passing

Whenever you are measuring humans, keep in mind that people respond in more than one way to the passage of time. For example, recent research shows how manipulating the speed of a clock that is in a person's view influences how quickly that person's blood glucose level changes.

Cleaning Your Data

In this section we will tackle the following common problems in time series data sets:

- Missing data
- Changing the frequency of a time series (that is, upsampling and downsampling)
- Smoothing data
- Addressing seasonality in data
- Preventing unintentional lookaheads

Handling Missing Data

Missing data is surprisingly common. For example, in healthcare, missing data in medical time series can have a number of causes:

- The patient didn't comply with a desired measurement.
- The patient's health stats were in good shape, so there was no need to take a particular measurement.
- The patient was forgotten or undertreated.
- A medical device had a random technical malfunction.
- There was a data entry error.

To generalize at great peril, missing data is even more common in time series analysis than it is in cross-sectional data analysis because the burden of longitudinal sampling is particularly heavy: incomplete time series are quite common and so methods have been developed to deal with holes in what is recorded.

The most common methods to address missing data in time series are:

Imputation
: When we fill in missing data based on observations about the entire data set.

Interpolation
: When we use neighboring data points to estimate the missing value. Interpolation can also be a form of imputation.

Deletion of affected time periods
: When we choose not to use time periods that have missing data at all.

We will discuss imputations and interpolations as well as illustrate the mechanics of these methods soon. We focus on preserving data, whereas a method such as deleting time periods with missing data will result in less data for your model. Whether to

preserve the data or throw out problematic time periods will depend on your use case and whether you can afford to sacrifice the time periods in question given the data needs of your model.

Preparing a data set to test missing data imputation methodologies

We will work with monthly unemployment data that has been released by the US government since 1948 (this is freely available for download). We will then generate two data sets from this baseline data: one in which data is truly missing at random, and one in which are the highest unemployment months in the history of the time series. This will give us two test cases to see how imputation behaves in the presence of both random and systematic missing data.

We move to R for the next example. We will freely switch between R and Python throughout the book. I assume you have some background in working with data frames, as well as matrices, in both R and Python.

```
## R
> require(zoo)        ## zoo provides time series functionality
> require(data.table) ## data.table is a high performance data frame

> unemp <- fread("UNRATE.csv")
> unemp[, DATE := as.Date(DATE)]
> setkey(unemp, DATE)

> ## generate a data set where data is randomly missing
> rand.unemp.idx <- sample(1:nrow(unemp), .1*nrow(unemp))
> rand.unemp     <- unemp[-rand.unemp.idx]

> ## generate a data set where data is more likely
> ## to be missing when unemployment is high
> high.unemp.idx <- which(unemp$UNRATE > 8)
> num.to.select  <- .2 * length(high.unemp.idx)
> high.unemp.idx <- sample(high.unemp.idx,)
> bias.unemp     <- unemp[-high.unemp.idx]
```

Because we have deleted rows from our data tables to create a data set with missing data, we will need to read the missing dates and NA values, and for this the `data.table` package's *rolling join* functionality is quite useful.

R's data.table Package

R's `data.table` package is a highly performant but woefully underutilized alternative to R's mainstream `data.frame` functionality. While many users are familiar with the `tidyverse` selection of packages, `data.table` is a standalone package built only on top of `data.frame`. It has many helpful functions for time series analysis. This is likely due to its origins as a package developed by software engineers who worked in finance and who needed highly performant R operations to work with financial data, most often time series data.

While there is a steeper learning curve for `data.table` than for `data.frame`, I have found this package invaluable in working with time series data in R, and so I use it heavily throughout the book. I've provided a few quick examples of how `data.table` works and may differ from `data.frame` here:

```
dt[, new.col := old.col + 7]
```

In this example, we assume that the original `data.table` has an old column called `old.col`. We make a new column, in place, which means that we add a column to the `data.table` without copying the entire `data.table` over to a new object, thereby reducing memory use. You will see this operator `:=` used frequently throughout the book.

To access a subset of columns in a `data.table`, we will usually simply put them in a list in the column indexing argument, which is the second argument to a data.table:

```
dt[, .(col1, col2, col3)]
```

This will return a subsetted `data.table` including only the `col1`, `col2`, and `col3` values. Notice the `.()` operator surrounding the column names. This is shorthand for writing out `list()`.

If we wish to do row selection rather than column selection, we use the first argument to the `data.table` to subset on rows like so:

```
dt[col1 < 3 & col2 > 5]
```

This will return all columns of the `data.table` for which the two logic conditions were met. And of course we can combine row and column selection, as follows:

```
dt[col1 < 3 & col2 > 5, .(col1, col2, col3)]
```

Finally, group-by operations are quite performant and fast. For example, to count the number of rows in each group among the groups designated in `col1`, we could use the following code.

```
dt[, .N, col1]
```

And if we wanted to do multiple operations on our grouped objects, we would again make a list and could even name the columns:

```
dt[, .(total = .N, mean = mean(col2), col1]
```

The above would, by groups as designated by `col1`, return the total number of rows in the group and the mean of `col2` for each group.

The `data.table` package is under active development, and it's an invaluable tool for time series analysis, particularly for large data sets. I strongly recommend reading the official introduction to `data.table`.

```
## R
> all.dates <- seq(from = unemp$DATE[1], to = tail(unemp$DATE, 1),
                                                    by = "months")
> rand.unemp = rand.unemp[J(all.dates), roll=0]
> bias.unemp = bias.unemp[J(all.dates), roll=0]
> rand.unemp[, rpt := is.na(UNRATE)]
## here we label the missing data for easy plotting
```

With a rolling join, we generate the sequence of all dates that should be available between the start and end date of the data set. This gives us rows in the data set to fill in as `NA`.

Now that we have data sets with missing values, we will take a look at a few specific ways to fill in numbers for those missing values:

- Forward fill
- Moving average
- Interpolation

We will compare the performance of these methods in both the randomly missing and systematically missing data sets. Since we have generated these data sets from a complete data set, we can in fact determine how they did instead of speculating. In the real world, of course, we will never have the missing data to check our data imputation.

Rolling Joins

In `data.table`'s rolling join, we enjoy the benefit of a special spatially aware SQL-like join designed for timestamps. Although tables will not always match exactly on their timestamps, rolling joins can deal with timestamps intelligently.

```
## R
> ## we have a small data.table of donation dates
> donations <- data.table(
>   amt = c(99, 100, 5, 15, 11, 1200),
>   dt = as.Date(c("2019-2-27", "2019-3-2", "2019-6-13",
>                  "2019-8-1", "2019-8-31", "2019-9-15"))
> )
```

```
> ## we also have information on the dates of each
> ## publicity campaign
> publicity <- data.table(
>                 identifier = c("q4q42", "4299hj", "bbg2"),
>                 dt         = as.Date(c("2019-1-1",
>                                        "2019-4-1",
>                                        "2019-7-1")))


> ## we set the primary key on each data.table
> setkey(publicity, "dt")
> setkey(donations, "dt")

> ## we wish to label each donation according to
> ## what publicity campaign most recently preceded it
> ## we can easily see this with roll = TRUE
> publicity[donations, roll = TRUE]
```

This yields the following delightful output:

```
## R
> publicity[donations, roll = TRUE]
   identifier         dt  amt
1:      q4q42 2019-02-27   99
2:      q4q42 2019-03-02  100
3:     4299hj 2019-06-13    5
4:       bbg2 2019-08-01   15
5:       bbg2 2019-08-31   11
6:       bbg2 2019-09-15 1200
```

Each donation now is paired with the publicity identifier that indicates the campaign that most immediately preceded the donation.

Forward fill

One of the simplest ways to fill in missing values is to carry forward the last known value prior to the missing one, an approach known as *forward fill*. No mathematics or complicated logic is required. Simply consider the experience of moving forward in time with the data that was available, and you can see that at a missing point in time, all you can be confident of is what data has already been recorded. In such a case, it makes sense to use the most recent known measurement.

Forward fill can be accomplished easily using `na.locf` from the `zoo` package:

```
## R
> rand.unemp[, impute.ff := na.locf(UNRATE, na.rm = FALSE)]
> bias.unemp[, impute.ff := na.locf(UNRATE, na.rm = FALSE)]
>
> ## to plot a sample graph showing the flat portions
> unemp[350:400, plot (DATE, UNRATE,
```

```
                          col = 1, lwd = 2, type = 'b')]
> rand.unemp[350:400, lines(DATE, impute.ff,
                          col = 2, lwd = 2, lty = 2)]
> rand.unemp[350:400][rpt == TRUE, points(DATE, impute.ff,
                          col = 2, pch = 6, cex = 2)]
```

This will result in a plot that looks natural except where you see repeated values to account for missing data, as in Figure 2-6. As you will notice in the plot, the forward-filled values usually do not deviate far from the true values.

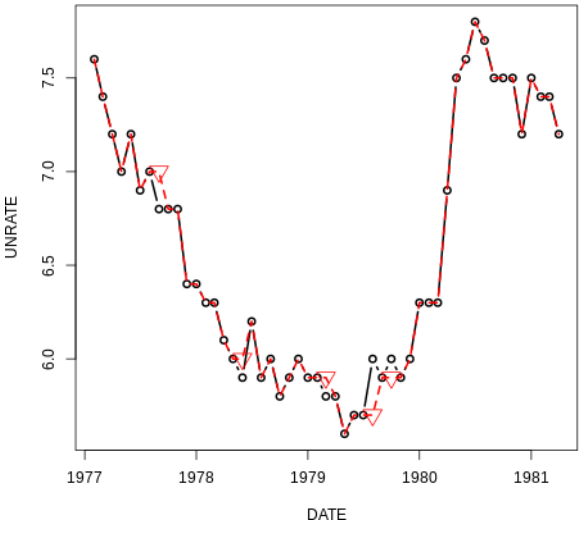

Figure 2-6. The original time series plotted with a solid line and the time series with forward filled values for randomly missing points plotted with a dashed line. The forward filled values are marked with downward pointing triangles.

We can also compare the values in the series by plotting the values of the series against one another. That is, for each time step, we plot the true known value against the value at the same time from the series with imputed values. Most values should match exactly since most data is present. We see that manifested in the 1:1 line in Figure 2-7. We also see scattered points off this line, but they do not appear to be systematically off.

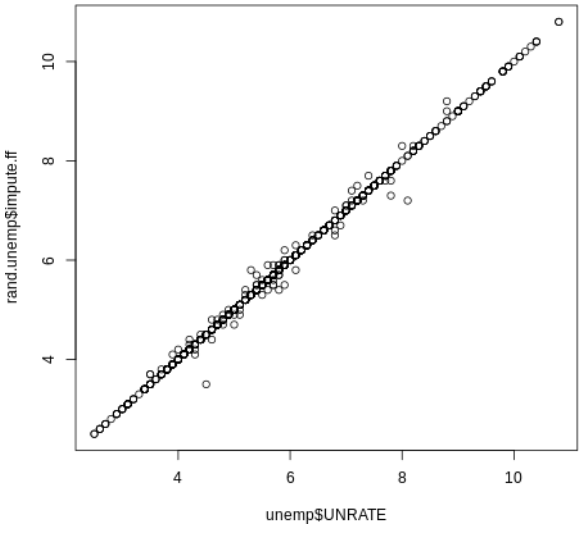

Figure 2-7. Plotting the true unemployment rate versus the forward-filled series. This plot shows that forward fill did not systematically distort the data.

Backward Fill

Just as you can bring values from the past forward to fill in missing data, you can also choose to propagate values backward. However, this is a case of a lookahead, so you should only do this when you are not looking to predict the future from the data and when, from domain knowledge, it makes more sense to fill in data backward rather than forward in time.

In some settings, forward fill makes sense as the best way to complete missing data, even if "fancier methods" are possible. For example, in medical settings, a missing value often indicates that a medical worker did not think it necessary to remeasure a value, likely because the patient's measurement was expected to be normal. In many medical cases, this means we could apply forward fill to missing values with the last known value since this was the presumption motivating the medical worker not to retake a measurement.

There are many advantages to forward fill: it is not computationally demanding, it can be easily applied to live-streamed data, and it does a respectable job with imputation. We will see an example soon.

Moving average

We can also impute data with either a rolling mean or median. Known as a *moving average*, this is similar to a forward fill in that you are using past values to "predict"

missing future values (imputation can be a form of prediction). With a moving average, however, you are using input from *multiple* recent times in the past.

There are many situations where a moving average data imputation is a better fit for the task than a forward fill. For example, if the data is noisy, and you have reason to doubt the value of any individual data point relative to an overall mean, you should use a moving average rather than a forward fill. Forward filling could include random noise more than the "true" metric that interests you, whereas averaging can remove some of this noise.

To prevent a lookahead, use only the data that occurred before the missing data point. So, your implementation would like something like this:

```
## R
> ## rolling mean without a lookahead
> rand.unemp[, impute.rm.nolookahead := rollapply(c(NA, NA, UNRATE), 3,
>              function(x) {
>                          if (!is.na(x[3])) x[3] else mean(x, na.rm = TRUE)
>                          })]
> bias.unemp[, impute.rm.nolookahead := rollapply(c(NA, NA, UNRATE), 3,
>              function(x) {
>                          if (!is.na(x[3])) x[3] else mean(x, na.rm = TRUE)
>                          })]
```

We set the value of the missing data to the mean of the values that come before it (because we index off the final value and use this value to determine whether it is missing and how to replace it).

A moving average doesn't have to be an arithmetic average. For example, exponentially weighted moving averages would give more weight to recent data than to past data. Alternately, a geometric mean can be helpful for time series that exhibit strong serial correlation and in cases where values compound over time.

When imputing missing data with a moving average, consider whether you need to know the value of the moving average with only forward-looking data, or whether you are comfortable building in a lookahead. If you are unconcerned about a lookahead, your best estimate will include points both before and after the missing data because this will maximize the information that goes into your estimates. In that case, you can implement a rolling window, as illustrated here with the `zoo` package's `rollapply()` functionality:

```
## R
> ## rolling mean with a lookahead
> rand.unemp[, complete.rm := rollapply(c(NA, UNRATE, NA), 3,
>             function(x) {
>                         if (!is.na(x[2]))
>                             x[2]
```

```
>                         else
>                             mean(x, na.rm = TRUE)
>                         })]
```

Using both past and future information is useful for visualizations and recordkeeping applications, but, as mentioned before, it is not appropriate if you are preparing your data to be fed into a predictive model.

The results for both a forward-looking moving average and a moving average computed with future and past data are shown in Figure 2-8.

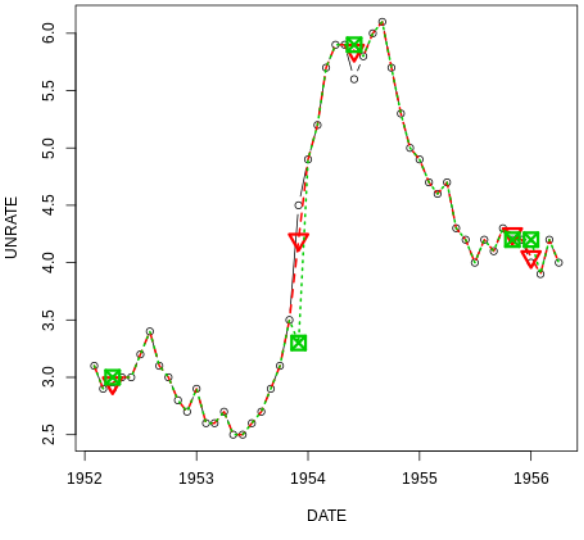

Figure 2-8. The dotted line shows the moving average imputation without a lookahead, while the dashed line shows the moving average imputation with a lookahead. Likewise, the squares show the nonlookahead imputed points while the upside down triangles show the moving average with a lookahead.

A rolling mean data imputation reduces variance in the data set. This is something you need to keep in mind when calculating accuracy, R^2 statistics, or other error metrics. Your calculation may overestimate your model's performance, a frequent problem when building time series models.

Using a Data Set's Mean to Impute Missing Data

In a cross-sectional context, it is common to impute missing data by filling in the mean or median for that variable where it is missing. While this can be done with time series data, it is not appropriate for most cases. Knowing the mean for the data set involves looking into the future…and that's a lookahead!

Interpolation

Interpolation is a method of determining the values of missing data points based on geometric constraints regarding how we want the overall data to behave. For example, a linear interpolation constrains the missing data to a linear fit consistent with known neighboring points.

Linear interpolation is particularly useful and interesting because it allows you to use your knowledge of how your system behaves over time. For example, if you know a system behaves in a linear fashion, you can build that knowledge in so that only linear trends will be used to impute missing data. In Bayesian speak, it allows you to inject a *prior* into your imputation.

As with a moving average, interpolation can be done such that it is looking at both past and future data or looking in only one direction. The usual caveats apply: allow your interpolation to have access to future data only if you accept that this creates a lookahead and you are sure this is not a problem for your task.

Here we apply interpolation using both past and future data points (see Figure 2-9):

```
## R
> ## linear interpolation
> rand.unemp[, impute.li := na.approx(UNRATE)]
> bias.unemp[, impute.li := na.approx(UNRATE)]
>
> ## polynomial interpolation
> rand.unemp[, impute.sp := na.spline(UNRATE)]
> bias.unemp[, impute.sp := na.spline(UNRATE)]
>
> use.idx = 90:120
> unemp[use.idx, plot(DATE, UNRATE, col = 1, type = 'b')]
> rand.unemp[use.idx, lines(DATE, impute.li, col = 2, lwd = 2, lty = 2)]
> rand.unemp[use.idx, lines(DATE, impute.sp, col = 3, lwd = 2, lty = 3)]
```

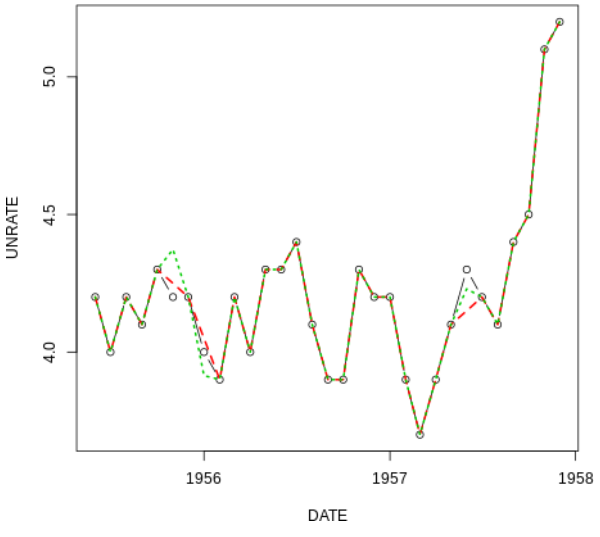

Figure 2-9. The dashed line shows the linear interpolation while the dotted line shows the spline interpolation.

There are many situations where a linear (or spline) interpolation is appropriate. Consider mean average weekly temperature where there is a known trend of rising or falling temperatures depending on the time of year. Or consider yearly sales data for a growing business. If the trend has been for business volume to increase linearly from year to year, it is a reasonable data imputation to fill in missing data based on that trend. In other words, we could use a linear interpolation, which would account for the trend, rather than a moving average, which would not. If there were a trend of increasing values, the moving average would systematically underestimate the missing values.

There are also plenty of situations for which linear (or spline) interpolation is not appropriate. For example, if you are missing precipitation data in a weather data set, you should not extrapolate linearly between the known days; as we all know, that's not how precipitation works. Similarly, if we are looking at someone's hours of sleep per day but are missing a few days of data, we should not linearly extrapolate the hours of sleep between the last known days. As an example, one of the known endpoints might include an all-nighter of studying followed by a 30-minute catnap. This is unlikely to estimate the missing data.

Overall comparison

Now that we have performed a few different kinds of imputations, we can take a look at the results to compare how the different data imputations behaved on this data set.

We generated two data sets with missing data, one in which data was missing at random and another in which unfavorable data points (high unemployment) were missing. When we compare the methods we employed to see which yields the best results, we can see that the mean square error can differ by a high percentage:

```
## R
> sort(rand.unemp[ , lapply(.SD, function(x) mean((x - unemp$UNRATE)^2,
>              na.rm = TRUE)),
>          .SDcols = c("impute.ff", "impute.rm.lookahead",
>                      "impute.rm.nolookahead", "impute.li",
>                      "impute.sp")])
impute.li   impute.rm.lookahead   impute.sp   impute.ff   impute.rm.nolookahead
0.0017      0.0019           0.0021      0.0056      0.0080

> sort(bias.unemp[ , lapply(.SD, function(x) mean((x - unemp$UNRATE)^2,
>               na.rm = TRUE)),
>          .SDcols = c("impute.ff", "impute.rm.lookahead",
>                      "impute.rm.nolookahead", "impute.li",
>                      "impute.sp")])
impute.sp    impute.li   impute.rm.lookahead  impute.rm.nolookahead   impute.ff
0.0012       0.0013      0.0017               0.0030                  0.0052
```

Remember that many of the preceding methods include a lookahead. The only methods that do not are the forward fill and the moving average without a lookahead (there is also a moving average with a lookahead). For this reason, it's not surprising that there is a range of difference in errors and that the methods without a lookahead do not perform as well as the others.

Final notes

Here we covered the simplest and most often-used methods of missing data imputation for time series applications. Data imputation remains an important area of data science research. The more significant the decisions you are making, the more important it is to carefully consider to the potential reasons for data to be missing and the potential ramifications of your corrections. Here are a few cautionary tips you should keep in mind:

- It is impossible to prove that data is truly missing at random, and it is unlikely that missingness is truly random in most real-world occurrences.
- Sometimes the probability that a measurement is missing is explainable by the variables you have measured, but sometimes not. Wide data sets with many features are the best way to investigate possible explanations for patterns of missing data, but these are not the norm for time series analysis.
- When you need to understand the uncertainty introduced by imputing values to missing data, you should run through a variety of scenarios and also speak to as many people involved in the data collection process as possible.

- How you handle missing data should account for your downstream use of that data. You must guard carefully against lookaheads or decide how seriously a lookahead will affect the validity of your subsequent work.

Upsampling and Downsampling

Often, related time series data from different sources will not have the same sampling frequency. This is one reason, among many, that you might wish to change the sampling frequency of your data. Of course you cannot change the actual rate at which information was measured, but you can change the frequency of the timestamps in your data collection. This is called *upsampling* and *downsampling*, for increasing or decreasing the timestamp frequency, respectively.

We downsampled temporal data in "Retrofitting a Time Series Data Collection from a Collection of Tables" on page 26. Here, we address the topic more generally, learning the how's and why's of both downsampling and upsampling.

Downsampling is subsetting data such that the timestamps occur at a lower frequency than in the original time series. Upsampling is representing data as if it were collected more frequently than was actually the case.

Downsampling

Anytime you reduce frequency of your data, you are downsampling. This is most often done in the following cases.

The original resolution of the data isn't sensible. There can be many reasons that the original granularity of the data isn't sensible. For example, you may be measuring something too often. Suppose you have a data set where someone had measured the outside air temperature every second. Common experience dictates that this measurement is unduly frequent and likely offers very little new information relative to the additional data storage and processing burden. In fact, it's likely the measurement error could be as large as the second-to-second air temperature variation. So, you likely don't want to store such excessive and uninformative data. In this case—that is, for regularly sampled data—downsampling is as simple as selecting out every *n*th element.

Focus on a particular portion of a seasonal cycle. Instead of worrying about seasonal data in a time series, you might choose to create a subseries focusing on only one season. For example, we can apply downsampling to create a subseries, as in this case, where we generate a time series of January measurements out of what was originally a

monthly time series. In the process, we have downsampled the data to a yearly frequency.

```
## R
> unemp[seq.int(from = 1, to = nrow(unemp), by = 12)]
DATE         UNRATE
1948-01-01    3.4
1949-01-01    4.3
1950-01-01    6.5
1951-01-01    3.7
1952-01-01    3.2
1953-01-01    2.9
1954-01-01    4.9
1955-01-01    4.9
1956-01-01    4.0
1957-01-01    4.2
```

Match against data at a lower frequency. You may want to downsample data so that you can match it with other low-frequency data. In such cases you likely want to aggregate the data or downsample rather than simply dropping points. This can be something simple like a mean or a sum, or something more complicated like a weighted mean, with later values given more weight. We saw earlier in the donation data the idea of summing all donations over a single week, since it was the total amount donated that was likely to be most interesting.

In contrast, for our economic data, what is most likely to be interesting is a yearly average. We use a mean instead of a rolling mean because we want to summarize the year rather than get the latest value of that year to emphasize recency. (Note the difference from data imputation.) We group by formatting the date into a string, representing its year as an example of how you can creatively exploit SQL-like operations for time series functionality:

```
## R
> unemp[, mean(UNRATE), by = format(DATE, "%Y")]
format   V1
1948    3.75
1949    6.05
1950    5.21
1951    3.28
1952    3.03
1953    2.93
1954    5.59
1955    4.37
1956    4.13
1957    4.30
```

Upsampling

Upsampling is not simply the inverse of downsampling. Downsampling makes inherent sense as something that can be done in the real world; it's simple to decide to

measure less often. In contrast, upsampling can be like trying to get something for free—that is, not taking a measurement but still somehow thinking you can get high-resolution data from infrequent measurements. To quote the author of the popular R time series package XTS:

> It is not possible to convert a series from a lower periodicity to a higher periodicity - e.g., weekly to daily or daily to 5 minute bars, as that would require magic.

However, there are legitimate reasons to want to label data at a higher frequency than its default frequency. You simply need to keep in mind the data's limitations as you do so. Remember that you are adding more time labels but not more information.

Let's discuss a few situations where upsampling can make sense.

Irregular time series. A very common reason to upsample is that you have an irregularly sampled time series and you want to convert it to a regularly timed one. This is a form of upsampling because you are converting all the data to a frequency that is likely higher than indicated by the lags between your data. If you are upsampling for this reason, you already know how to do it with a rolling join, as we did this in the case of filling in missing economic data, via R:

```
## R
> all.dates <- seq(from = unemp$DATE[1], to = tail(unemp$DATE, 1),
>                  by = "months")
> rand.unemp = rand.unemp[J(all.dates), roll=0]
```

Inputs sampled at different frequencies. Sometimes you need to upsample low-frequency information simply to carry it forward with your higher-frequency information in a model that requires your inputs to be aligned and sampled contemporaneously. You must be vigilant with respect to lookahead, but if we assume that known states are true until a new known state comes into the picture, we can safely upsample and carry our data forward. For example, suppose we know it's (relatively) true that most new jobs start on the first of the month. We might decide we feel comfortable using the unemployment rate for a given month indicated by the jobs report for the *entire* month (not considering it a lookahead because we make the assumption that the unemployment rate stays steady for the month).

```
## R
> daily.unemployment = unemp[J(all.dates), roll = 31]
> daily.unemployment
   DATE      UNRATE
1948-01-01    3.4
1948-01-02    3.4
1948-01-03    3.4
1948-01-04    3.4
1948-01-05    3.4
```

Knowledge of time series dynamics. If you have underlying knowledge of the usual temporal behavior of a variable, you may also be able to treat an upsampling problem as a missing data problem. In that case, all the techniques we've discussed already still apply. An interpolation is the most likely way to produce new data points, but you would need to be sure the dynamics of your system could justify your interpolation decision.

As discussed earlier, upsampling and downsampling will routinely happen even in the cleanest data set because you will almost always want to compare variables of different timescales. It should also be noted that Pandas has particularly handy upsampling and downsampling functionality with the `resample` method.

Smoothing Data

Smoothing data can be done for a variety of reasons, and often real-world time series data is smoothed before analysis, especially for visualizations that aim to tell an understandable story about the data. In this section we discuss further why smoothing is done, as well as the most common time series smoothing technique: exponential smoothing.

Purposes of smoothing

While outlier detection is a topic in and of itself, if you have reason to believe your data should be smoothed, you can do so with a moving average to eliminate measurement spikes, errors of measurement, or both. Even if the spikes are accurate, they may not reflect the underlying process and may be more a matter of instrumentation problems; this is why it's quite common to smooth data.

Smoothing data is strongly related to imputing missing data, and so some of those techniques are relevant here as well. For example, you can smooth data by applying a rolling mean, with or without a lookahead, as that is simply a matter of the point's position relative to the window used to calculate its smoothed value.

When you are smoothing, you want to think about a number of questions:

- Why are you smoothing? Smoothing can serve a number of purposes:

  Data preparation
  : Is your raw data unsuitable? For example, you may know very high values are unlikely or unphysical, but you need a principled way to deal with them. Smoothing is the most straightforward solution.

  Feature generation
  : The practice of taking a sample of data, be it many characteristics about a person, image, or anything else, and summarizing it with a few metrics. In

this way a fuller sample is collapsed along a few dimensions or down to a few traits. Feature generation is especially important for machine learning.

Prediction
: The simplest form of prediction for some kinds of processes is mean reversion, which you get by making predictions from a smoothed feature.

Visualization
: Do you want to add some signal to what seems like a noisy scatter plot? If so, what is your intention in doing so?

- How will your outcomes be affected by smoothing or not smoothing?
  - Does your model assume noisy and uncorrelated data, whereby your smoothing could compromise this assumption?
  - Will you need to smooth in a live production model? If so, you need to choose a smoothing method that does not employ a lookahead.
  - Do you have a principled way to smooth, or will you simply do a hyperparameter grid search? If the latter, how will you make sure that you use a time-aware form of cross-validation such that future data does not leak backward in time?

What Are Hyperparameters?

The term *hyperparameters* can sound awfully scary, but they are simply the numbers you use to tune, or adjust, the performance of a statistical or machine learning model. To optimize a model, you try several variations of the hyperparameters of that model, often performing a *grid search* to identify these parameters. A grid search is exactly that: you construct a "grid" of all possible combinations of hyperparameters in a search space and try them all.

For example, imagine an algorithm takes hyperparameters A and B, where we think reasonable values for A are 0, 1, or 2, and we think reasonable values for B are 0.7, 0.75, or 0.8. Then we would have nine possible hyperparameter options for our grid: every possible combination of potential A values (of which there are 3) with every possible B value (of which there are 3) for a total of $3 \times 3 = 9$ possibilities. As you can see, hyperparameter tuning can quickly get quite labor-intensive.

Exponential smoothing

You often won't want to treat all time points equally when smoothing. In particular, you may want to treat more recent data as more informative data, in which case exponential smoothing is a good option. In contrast to the moving average we looked at before—where each point where data was missing could be imputed to the mean of

its surrounding points—exponential smoothing is geared to be more temporally aware, weighting more recent points higher than less recent points. So, for a given window, the nearest point in time is weighted most heavily and each point earlier in time is weighted exponentially less (hence the name).

The mechanics of exponential smoothing work as follows. For a given time period t, you find the smoothed value of a series by computing:

$$\text{Smoothed value at time } t = S_t = d \times ; S_{t-1} + (1 - d) \times x_t$$

Think about how this propagates over time. The smoothed value at time $(t - 1)$ is itself a product of the same thing:

$$S_{t-1} = d \times S_{t-2} + (1 - d) \times x_{t-1}$$

So we can see a more complex expression for the smoothed value at time t:

$$d \times (d \times S_{t-2} + (1 - d) \times x_{t-1}) + (1 - d) \times x_t$$

Mathematically inclined readers will notice that we have a series of the form:

$$d^3 \times x_{t-3} + d^2 \times x_{t-2} + d \times x_{t-1}$$

In fact, it is thanks to this form that exponential moving averages are quite tractable. More details are widely available online and in textbooks; see my favorite summaries in "More Resources" on page 69.

I will illustrate smoothing in Python, as Pandas includes a variety of smoothing options. Smoothing options are also widely available in R, including base R, as well as many time series packages.

While we have been looking at US unemployment rate data, we will switch to another commonly used data set: the airline passenger data set (which dates back to Box and Jenkins's famous time series book and is widely available). The original data set is an account of the thousands of monthly airline passengers broken down by month:

```
## python
>>> air
        Date  Passengers
0    1949-01         112
1    1949-02         118
2    1949-03         132
3    1949-04         129
4    1949-05         121
5    1949-06         135
```

```
6    1949-07       148
7    1949-08       148
8    1949-09       136
9    1949-10       119
10   1949-11       104
```

We can easily smooth the values of passengers using a variety of decay factors and applying the Pandas `ewma()` function, like so:

```
## python
>>> air['Smooth.5'] = pd.ewma(air, alpha = .5).Passengers
>>> air['Smooth.9'] = pd.ewma(air, alpha = .9).Passengers
```

As we can see, the level of the `alpha` parameter, also called the *smoothing factor*, affects how much the value is updated to its current value versus retaining information from the existing average. The higher the `alpha` value, the more quickly the value is updated closer to its current price. Pandas accepts a number of parameters, all of which plug into the same equation, but they offer multiple ways to think about specifying an exponential moving average.[3]

```
## python
>>> air
       Date  Passengers    Smooth.5  Smooth.9
0    1949-01         112  112.000000  112.000000
1    1949-02         118  116.000000  117.454545
2    1949-03         132  125.142857  130.558559
3    1949-04         129  127.200000  129.155716
4    1949-05         121  124.000000  121.815498
5    1949-06         135  129.587302  133.681562
6    1949-07         148  138.866142  146.568157
7    1949-08         148  143.450980  147.856816
8    1949-09         136  139.718200  137.185682
9    1949-10         119  129.348974  120.818568
10   1949-11         104  116.668295  105.681857
```

However, simple exponential smoothing does not perform well (for prediction) in the case of data with a long-term trend. Holt's Method and Holt–Winters smoothing , are two exponential smoothing methods applied to data with a trend, or with a trend and seasonality.

There are many other widely used smoothing techniques. For example, Kalman filters smooth the data by modeling a time series process as a combination of known dynamics and measurement error. LOESS (short for "locally estimated scatter plot smoothing") is a nonparametric method of locally smoothing data. These methods, and others, gradually offer more complex ways of understanding smoothing but at increased computational cost. Of note, Kalman and LOESS incorporate data both

3 You can specify the alpha smoothing factor, the halflife, the span, or the center of mass according to which you feel comfortable with. Details are available in the documentation.

earlier and later in time, so if you use these methods keep in mind the leak of information backward in time, as well as the fact that they are usually not appropriate for preparing data to be used in forecasting applications.

Smoothing is a commonly used form of forecasting, and you can use a smoothed time series (without lookahead) as one easy null model when you are testing whether a fancier method is actually producing a successful forecast.

How to Start a Smoothing Calculation

One important reason to use a packaged exponential smoothing function is that it is difficult to get the start of your smoothed series correct. Imagine that the first point in your series is equal to 3 and the second point is equal to 6. Also imagine that your discounting factor (which you set according to how quickly you want the average to adapt to the most recent information) is equal to 0.7, so that you would compute the second point in your series as:

$$3 \times 0.7 + 6 \times (1 - 0.7) = 3.9$$

In fact, you would be wrong. The reason is that multiplying your discounting factor of 0.7 by 3 implicitly assumes that 3 is the sum of your knowledge going back in time to infinity, rather than a mere single data point you just measured. Therefore, it unduly weighs 3 relative to 6 as though 3 has a significantly more longstanding set of knowledge than it actually does.

The algebra to derive the correct expression is too complicated for a short sidebar, but check the additional resources at the end of this book for more details. In the meantime, I will leave you with the general guidance that 6 should be weighed a little more heavily than it is here, and 3 a little less. As you get further along in a time series, your older values should be weighted closer to your true discounting factor of 0.7. How soon this happens can be seen in the algebra, so take a look at the additional resources if you are interested.

Seasonal Data

Seasonality in data is any kind of recurring behavior in which the frequency of the behavior is stable. It can occur at many different frequencies at the same time. For example, human behavior tends to have a daily seasonality (lunch at the same time every day), a weekly seasonality (Mondays are similar to other Mondays), and a yearly seasonality (New Year's Day has low traffic). Physical systems also demonstrate seasonality, such as the period the Earth takes to revolve around the sun.

Identifying and dealing with seasonality is part of the modeling process. On the other hand, it's also a form of data cleaning, such as the economically important US Jobs Report. Indeed, many government statistics, particularly economic ones, are deseasonalized in their released form.

To see what seasonal data smoothing can do, we return to the canonical data set on airline passenger counts. A plot quickly reveals that this is highly seasonal data, but only if you make the right plot.

Notice the difference between using R's default plot (which uses points; see Figure 2-10) versus adding the argument to indicate that you want a line (Figure 2-11).

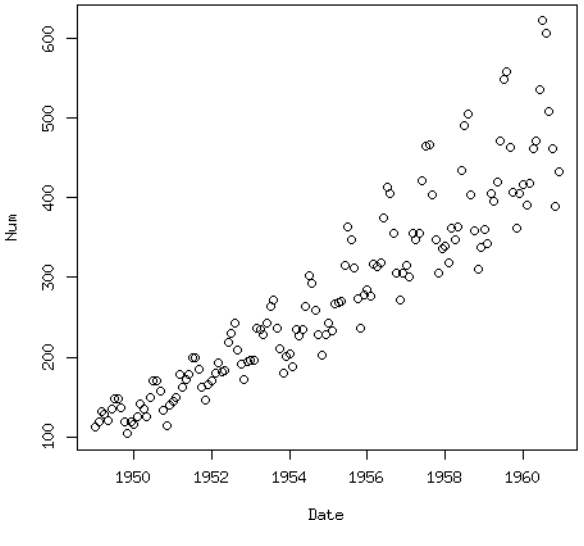

Figure 2-10. The increasing mean and variance of the data are apparent from a scatter plot, but we do not see an obvious seasonal trend.

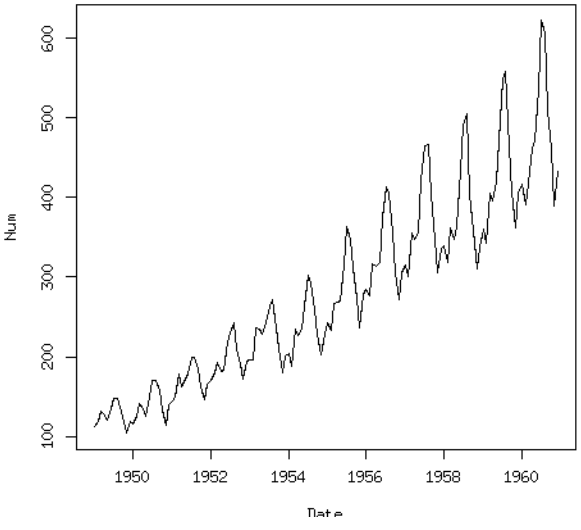

Figure 2-11. A line plot makes the seasonality abundantly clear.

If you were looking only at the default R plot, the seasonal nature of the data might elude you. Hopefully that wouldn't remain the case for long, as no doubt you would also do things to explore your data more fully, perhaps with an autocorrelation plot (discussed in Chapter 3) or other diagnostics.

Humans Are Creatures of Habit

With human behavior data there is almost always some form of seasonality, even with several cycles (an hourly pattern, a weekly pattern, a summer-winter pattern, etc.).

The scatter plot does show some information more clearly than the line plot. The variance of our data is increasing, as is the mean, which is most obvious when we see a cloud of data points beaming outward in a conical shape, tilted upward. This data clearly has a trend, and so we would likely transform it with a log transform or differencing, depending on the demands of our model. This data also clearly has a trend of increasing variance. We'll talk more about model-specific data transformations on seasonal data in the modeling chapters, so we won't elaborate here.

We also get useful information apart from the evidence for seasonality from the line plot. We get information about what *kind* of seasonality. That is, we see that the data is not only seasonal, but seasonal in a multiplicative way. As the overall values get larger, so do the season swings (think of this as sizing from peak to trough).

We can decompose the data into its seasonal, trend, and remainder components very easily as shown here with just one line of R:

```
## R
> plot(stl(AirPassengers, "periodic"))
```

The resulting plot seems very sensible based on the original data (see Figure 2-12). We can imagine adding the seasonal, trend, and remainder data back together to get the original series. We can also see that this particular decomposition did not take into account the fact that this series shows a multiplicative seasonality rather than an additive one because the residuals are greatest at the beginning and end of the time series. It seems like this decomposition settled on the average seasonal variance as the variance for the seasonal component.

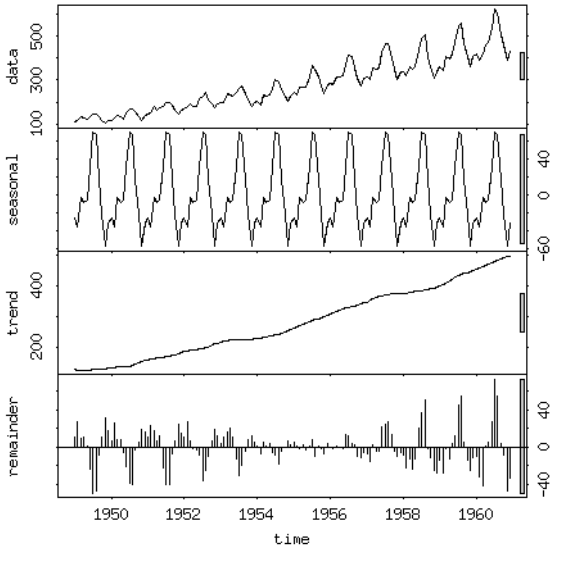

Figure 2-12. A decomposition of the original time series into a seasonal component, a trend, and the residuals. Pay attention to each plot's y-axis, as they are quite different. Note that this is the reason for the gray bars on the right side of each plot. These gray bars are all the same absolute size (in units of the y-axis), so that their relatively different display is a visual reminder of the different y-axis scales across different components.

To get an initial understanding of how this works, we can look at the official R documentation:

> The seasonal component is found by LOESS smoothing the seasonal subseries (the series of all January values, ...). If s.window = "periodic" smoothing is effectively replaced by taking the mean. The seasonal values are removed, the remainder smoothed to find the trend. The overall level is removed from the seasonal component and added to the

trend component. This process is iterated a few times. The remainder component is the residuals from the seasonal plus trend fit.

LOESS, introduced earlier, is a computationally taxing method for smoothing data points that involves a moving window to estimate the smoothed value of each point based on its neighbors (I hope your lookahead alarm is sounding!).

Seasonal Data and Cyclical Data

Seasonal time series are time series in which behaviors recur over a fixed period. There can be multiple periodicities reflecting different tempos of seasonality, such as the seasonality of the 24-hour day versus the 12-month calendar season, both of which exhibit strong features in most time series relating to human behavior.

Cyclical time series also exhibit recurring behaviors, but they have a variable period. A common example is a business cycle, such as the stock market's boom and bust cycles, which have an uncertain duration. Likewise, volcanoes show cyclic but not seasonal behaviors. We know the approximate periods of eruption, but these are not precise and vary over time.

Time Zones

Time zones are intrinsically tedious, painful, and difficult to get right even with a lot of effort. That is why you should never use your own solution. From their very invention, time zones have been complicated, and they have only gotten more so with the advent of personal computers. There are many reasons for this:

- Time zones are shaped by political and social decisions.
- There is no standard way to transport time zone information between languages or via an HTTP protocol.
- There is no single protocol for naming time zones or for determining start and end dates of daylight savings offsets.
- Because of daylight savings, some times occur twice a year in their time zones!

Most languages rely on the underlying operating system to get time zone information. Unfortunately, the built-in automatic time retrieval function in Python, `datetime.datetime.now()`, does not return a time zone–aware timestamp. Part of this is by design. Some decisions made in the standard library include forbidding time zone information in the `datetime` module (because this information changes so often) and allowing `datetime` objects both with and without time zone information. However, comparing an object with a time zone to one without a time zone will cause a `TypeError`.

Some bloggers claim that a majority of libraries are written with the assumption that `tzinfo==None`. Although this is a difficult claim to substantiate, it is consistent with much experience. People also report difficulty pickling time zone–stamped objects, so this is also something to check on early if you plan to use pickling.[4]

So let's have a look at working with time zones in Python. The main libraries you are likely to use are `datetime`, `pytz`, and `dateutil`. Additionally Pandas offers convenient time zone–related functionality built on top of the latter two libraries.

We'll cover the most important time zone functionality next.

First, notice that when you retrieve a "now" from the `datetime` module, it does not come with time zone information, even though it will give you the appropriate time for your time zone. Note, for example, the difference in the responses for `now()` and `utcnow()`:

```
## python
>>> datetime.datetime.utcnow()
datetime.datetime(2018, 5, 31, 14, 49, 43, 187680)

>>> datetime.datetime.now()
datetime.datetime(2018, 5, 31, 10, 49, 59, 984947)
>>> # as we can see, my computer does not return UTC
>>> # even though there is no time zone attached

>>> datetime.datetime.now(datetime.timezone.utc)
datetime.datetime(2018, 5, 31, 14, 51, 35, 601355,
                  tzinfo=datetime.timezone.utc)
```

Notice that if we do pass in a time zone, we will get the correct information, but this is not the default behavior. To work with time zones in Python, we create a time zone object, such `western` as for the US Pacific time zone:

```
## python
>>> western = pytz.timezone('US/Pacific')
>>> western.zone
'US/Pacific'
```

We can then use these objects to `localize` a time zone as follows:

```
## python
>>> ## the API supports two ways of building a time zone-aware time,
>>> ## either via 'localize' or to convert a time zone from one locale
>>> ## to another
>>> # here we localize
>>> loc_dt = western.localize(datetime.datetime(2018, 5, 15, 12, 34, 0))
```

4 Pickling is the process of storing Python objects in byte format. This is done via the popular built-in `pickle` module. Most of the time pickling will work well, but sometimes time related objects can be difficult to pickle.

```
datetime.datetime(2018, 5, 15, 12, 34,
            tzinfo=<DstTzInfo 'US/Pacific' PDT-1 day, 17:00:00 DST>)
```

Note, however, that passing the time zone directly into the `datetime` constructor will often not produce the result we were expecting:

```
## python
>>> london_tz = pytz.timezone('Europe/London')
>>> london_dt = loc_dt.astimezone(london_tz)

>>> london_dt
datetime.datetime(2018, 5, 15, 20, 34,
              tzinfo=<DstTzInfo 'Europe/London' BST+1:00:00 DST>)

>>> f = '%Y-%m-%d %H:%M:%S %Z%z'
>>> datetime.datetime(2018, 5, 12, 12, 15, 0,
                      tzinfo = london_tz).strftime(f)
'2018-05-12 12:15:00 LMT-0001'

>>> ## as highlighted in the pytz documentation using the tzinfo of
>>> ## the datetime.datetime initializer does not always lead to the
>>> ## desired outcome, such as with the London example

>>> ## according to the pytz documentation, this method does lead to
>>> ## the desired results in time zones without daylight savings
```

This is important, such as when you are calculating time deltas. The first example of the following three is the gotcha:

```
## python
>>> # generally you want to store data in UTC and convert only when
>>> # generating human-readable output
>>> # you can also do date arithmetic with time zones
>>> event1 = datetime.datetime(2018, 5, 12, 12, 15, 0,
                               tzinfo = london_tz)
>>> event2 = datetime.datetime(2018, 5, 13, 9, 15, 0,
                               tzinfo = western)
>>> event2 - event1
>>> ## this will yield the wrong time delta because the time zones
>>> ## haven't been labelled properly

>>> event1 = london_tz.localize(
                datetime.datetime(2018, 5, 12, 12, 15, 0))
>>> event2 = western.localize(
                datetime.datetime(2018, 5, 13, 9, 15, 0))
>>> event2 - event1

>>> event1 = london_tz.localize(
              (datetime.datetime(2018, 5, 12, 12, 15, 0))).
                 astimezone(datetime.timezone.utc)
>>> event2 = western.localize(
              datetime.datetime(2018, 5, 13, 9, 15, 0)).
```

```
                    astimezone(datetime.timezone.utc)
>>> event2 - event1
```

pytz provides a list of common time zones and time zones by country, both of which can be handy references:

```
## python
## have a look at pytz.common_timezones
>>> pytz.common_timezones
(long output...)

## or country specific
>>> pytz.country_timezones('RU')
['Europe/Kaliningrad', 'Europe/Moscow', 'Europe/Simferopol',
'Europe/Volgograd', 'Europe/Kirov', 'Europe/Astrakhan',
'Europe/Saratov', 'Europe/Ulyanovsk', 'Europe/Samara',
'Asia/Yekaterinburg', 'Asia/Omsk', 'Asia/Novosibirsk',
'Asia/Barnaul', 'Asia/Tomsk', 'Asia/Novokuznetsk',
'Asia/Krasnoyarsk', 'Asia/Irkutsk', 'Asia/Chita',
'Asia/Yakutsk', 'Asia/Khandyga', 'Asia/Vladivostok',
'Asia/Ust-Nera', 'Asia/Magadan', 'Asia/Sakhalin',
'Asia/Srednekolymsk', 'Asia/Kamchatka', 'Asia/Anadyr']
>>>
>>> pytz.country_timezones('fr')
>>> ['Europe/Paris']
```

A particularly hairy issue is the matter of daylight savings. Certain human-readable times exist twice (falling behind in the autumn), while others do not exist at all (skip ahead in the spring):

```
## python
>>> ## time zones
>>> ambig_time = western.localize(
                    datetime.datetime(2002, 10, 27, 1, 30, 00)).
                       astimezone(datetime.timezone.utc)
>>> ambig_time_earlier = ambig_time - datetime.timedelta(hours=1)
>>> ambig_time_later = ambig_time + datetime.timedelta(hours=1)
>>> ambig_time_earlier.astimezone(western)
>>> ambig_time.astimezone(western)
>>> ambig_time_later.astimezone(western)

>>> #results in this output
datetime.datetime(2002, 10, 27, 1, 30,
            tzinfo=<DstTzInfo 'US/Pacific' PDT-1 day, 17:00:00 DST>)
datetime.datetime(2002, 10, 27, 1, 30,
            tzinfo=<DstTzInfo 'US/Pacific' PST-1 day, 16:00:00 STD>)
datetime.datetime(2002, 10, 27, 2, 30,
            tzinfo=<DstTzInfo 'US/Pacific' PST-1 day, 16:00:00 STD>)
>>> # notice that the last two timestamps are identical, no good!


>>> ## in this case you need to use is_dst to indicate whether daylight
>>> ## savings is in effect
```

```
>>> ambig_time = western.localize(
        datetime.datetime(2002, 10, 27, 1, 30, 00), is_dst = True).
          astimezone(datetime.timezone.utc)
>>> ambig_time_earlier = ambig_time - datetime.timedelta(hours=1)
>>> ambig_time_later = ambig_time + datetime.timedelta(hours=1)
>>> ambig_time_earlier.astimezone(western)
>>> ambig_time.astimezone(western)
>>> ambig_time_later.astimezone(western)

datetime.datetime(2002, 10, 27, 0, 30,
           tzinfo=<DstTzInfo 'US/Pacific' PDT-1 day, 17:00:00 DST>)
datetime.datetime(2002, 10, 27, 1, 30,
           tzinfo=<DstTzInfo 'US/Pacific' PDT-1 day, 17:00:00 DST>)
datetime.datetime(2002, 10, 27, 1, 30,
           tzinfo=<DstTzInfo 'US/Pacific' PST-1 day, 16:00:00 STD>)
## notice that now we don't have the same time happening twice.
## it may appear that way until you check the offset from UTC
```

Time zone concerns may not be important to your work, so the utility of this knowledge will depend on the nature of your data. There are certainly situations where getting this wrong could be catastrophic (say, generating weather forecasts for commercial airliners flying during the time change and suddenly finding their positions drastically altered).

Preventing Lookahead

Lookahead is dangerously easy to introduce into a modeling pipeline, especially with vectorized functional data manipulation interfaces, such as those offered by R and Python. It is easy to shift a variable in the wrong direction, shift it more or less than you intended, or otherwise find yourself with data that is not entirely "honest" in that you have data available before it would have been known in your system.

Unfortunately, there isn't a definitive statistical diagnosis for lookahead—after all, the whole endeavor of time series analysis is modeling the unknown. Unless a system is somewhat deterministic with known dynamical laws, it can be difficult to distinguish a very good model from a model with lookahead—that is, until you put a model into production and realize either that you are missing data when you planned to have it, or simply that your results in production do not reflect what you see during training.

The best way to prevent this embarrassment is constant vigilance. Whenever you are time-shifting data, smoothing data, imputing data, or upsampling data, ask yourself whether you could know something at a given time. Remember that doesn't just include calendar time. It also includes realistic time lags to reflect how long a delay there is between something happening and your organization having that data available. For example, if your organization scrapes Twitter only weekly to gather its sentiment analysis data, you need to include this weekly periodicity in your training and validation data segmentation. Similarly, if you can retrain your model only once a

month, you need to figure out what model would apply to what data over time. You can't, for example, train a model for July and then apply it to July for testing, because in a real situation, you wouldn't have that model trained up in time if training takes you a long time.

Here are some other ideas to use as a general checklist. Keep them in mind both when planning to build a model and when auditing your process after the fact:

- If you are smoothing data or imputing missing data, think carefully about whether it might impact your results by introducing a lookahead. And don't just think about it—experiment as we did earlier and see how the imputations and smoothing work. Do they seem to be forward looking? If so, can you justify using them? (Probably not.)
- Build your entire process with a very small data set (only a few rows in a `data.table` or a few row time steps in whatever data format). Then, do random spot checks at each step in the process and see whether you accidentally shift any information temporally to an inappropriate place.
- For each kind of data, find out what the lag is for it relative to its own timestamp. For example, if the timestamp is when the data "happened" but not when it was uploaded to your servers, you need to know that. Different columns of a data frame may have different lags. To address this, you can either customize your lag per data frame or (better and more realistic) pick the biggest lag and apply that to everything. While you won't want to unduly pessimize your model, it's a good starting point after which you can relax these overly constrained rules one at a time, carefully!
- Use time-aware error (rolling) testing or cross-validation. This will be discussed in Chapter 11, but remember that randomizing your training versus testing data sets does not work with time series data. You do not want information from the future to leak into models for the past.
- Intentionally introduce a lookahead and see how your model behaves. Try various degrees of lookahead, so you have an idea how it shifts accuracy. If you have some idea of the accuracy with lookahead, you have an idea of what the ceiling on a real model without unfair knowledge of the future will do. Remember that many time series problems are extremely difficult, so a model with a lookahead may seem great until you realize you are dealing with a high-noise/low-signal data set.
- Add features slowly, particularly features you might be processing, so that you can look for jumps. One sign of a lookahead is when a particular feature is unexpectedly good, and there isn't a very good explanation. At the top of your explanation list should always be "lookahead."

Processing and cleaning time-related data can be a tedious and detail-oriented process.

There is tremendous danger in data cleaning and processing of introducing a *lookahead*! You should have lookaheads only if they are intentional, and this is rarely appropriate.

More Resources

- On missing data:

  Steffen Moritz et al., "Comparison of Different Methods for Univariate Time Series Imputation in R," unpublished research paper, October 13, 2015, https://perma.cc/M4LJ-2DFB.

  This thorough summary from 2015 outlines available methods for imputing time series data in the case of univariate time series data. Univariate time series data is a particular challenge because many advanced missing data imputation methods rely on looking at distributions among covariates, an option that is not available in the case of a univariate time series. This paper summarizes both the usability and performance of various R packages as well as empirical results of the methods available on a variety of data sets.

  James Honaker and Gary King, "What to Do About Missing Values in Time-Series Cross-Section Data," American Journal of Political Science 54, no. 2 (2010): 561–81, https://perma.cc/8ZLG-SMSX.

  This article explores best practices for missing data in time series with wide panels of covariates.

  Léo Belzile, "Notes on Irregular Time Series and Missing Values," n.d. https://perma.cc/8LHP-92FP.

  The author provides examples of working with irregular data as a missing data problem and an overview of some commonly used R packages.

- On time zones:

  Tom Scott, The Problem with Time & Timezones, Computerphile video, December 30, 2013, https://oreil.ly/iKHkp.

  This 10-minute YouTube video with over 1.5 million views outlies the perils and challenges of dealing with time zones, particularly in the context of a web application.

  Wikipedia, "Time Zone," https://perma.cc/J6PB-232C.

  This is a fascinating history in short form that gives some insight into how timekeeping worked before the last century and how technology advancements (beginning with the railway) led to the increasing need for people in

different locations to coordinate their time. There are also some fun maps of time zones.

Declan Butler, "GPS Glitch Threatens Thousands of Scientific Instruments," Nature, April 3, 2019, https://perma.cc/RPT6-AQBC.

This article is not directly related to time zones but to the problem of timestamping more generally. It describes a recent problem in which a bug in the US Global Positioning System could cause a problem with timestamped data because it transmits a binary 10-digit "week number" that began counting from January 6, 1980. This system can cover only 1,024 weeks total (2 raised to the power of 10). This count was reached for the second time in April 2019. Devices that were not designed to accommodate this limitation reset to zero and incorrectly timestamped scientific and industrial data. This article describes some of the difficulties with a fairly limited representation of time combined with scientific devices that were not future-proofed to account for this problem.

- On smoothing and seasonality:

Rob J. Hyndman and George Athanasopoulos,"Exponential Smoothing," in Forecasting: Principles and Practices, 2nd ed. (Melbourne: OTexts, 2018), https://perma.cc/UX4K-2V5N.

This chapter of Hyndman and Athanasopoulos's introductory academic textbook covers exponential smoothing methods for time series data, including a helpful taxonomy of exponential smoothing and extensions of exponential smoothing to forecasting applications.

David Owen, "The Correct Way to Start an Exponential Moving Average," Forward Motion blog, January 31, 2017, https://perma.cc/ZPJ4-DJJK.

As highlighted in a side note earlier, exponential moving averages are conceptually simple and easy to calculate, with the complication that how to "start" off the calculation is a little complicated. We want to make sure our moving average adapts to new information in a way that recognizes how long it has been recording information. If we start a moving average without giving consideration to this, even a new moving average will behave as though it has infinite lookback and unduly discount new information relative to what it should do. More details and a computational solution are presented in this blog post.

Avner Abrami, Aleksandr Arovkin, and Younghun Kim, "Time Series Using Exponential Smoothing Cells," unpublished research paper, last revised September 29, 2017, https://perma.cc/2JRX-K2JZ.

This is an extremely accessible time series research paper elaborating on the idea of simple exponential smoothing to develop "exponential smoothing cells."

- On functional data analysis:

Jane-Ling Wang, Jeng-Min Chiou, and Hans-Georg Müller, "Review of Functional Data Analysis," Annual Reviews of Statistics and its Application, 2015, https://perma.cc/3DNT-J9EZ.

This statistics article offers an approachable mathematical overview of important functional data analysis techniques. There are also helpful visualization techniques illustrated throughout the article.

Shahid Ullah and Caroline F. Finch, "Applications of Functional Data Analysis: A Systematic Review," BMC Medical Research Methodologyv, 13, no. 43 (2013), https://perma.cc/VGK5-ZEUX.

This review takes a multidisciplinary approach to surveying recently published analyses using functional data analysis. The authors argue that the techniques of functional data analysis have much wider applicability than is currently appreciated and make the case that the biological and health sciences could benefit from more use of these techniques to look at medical time series data.