IN THIS CHAPTER

- Defining data error types
- Obtaining data reliably
- Performing data validation
- Trimming data in various ways

Chapter 1
Locating Errors in Your Data

Your data likely contains errors, which seems like a sweeping statement when you consider that only you really understand your data. However, most data available today contains various kinds of errors that can derail your analysis. If you don't catch these errors, you may make a prediction that has no chance whatsoever of being accurate — even if your algorithms and logic are both bulletproof. The problem is in figuring out where the errors lie because they can be quite difficult to see. Consequently, this chapter begins by helping you understand the types of data errors so that you have a better chance of finding them.

The source of your data often determines the kind of errors you find, how deep you have to go into the code to locate them, and how difficult they are to find. Consider the simple act of scraping data from a website online. Even if the data is in the right form, doesn't have any missing elements, and appears reasonably correct, you have no way of knowing that the data is accurate unless you research it yourself. Of course, if you take all the time required to perform in depth research, you may as well generate the data yourself to ensure accuracy.

One means of ensuring that the data is less error prone is to perform various kinds of automated data validation. The validation process can tell you a lot about the data and even indicate, to some degree, the data accuracy. The process isn't

perfect, but anything you can do to reduce errors will make your analysis more accurate.

After validating your data, you can use various methods of trimming the data to include only those elements you can be sure will contain accurate information. The act of trimming the data, and performing other sorts of data maintenance, will greatly improve the results you receive from your analysis. Many data scientists spend a majority of their time performing the process outlined in this chapter. The act of ensuring that data is as accurate as possible consumes considerable time, but it's a necessary part of any data analysis.


REMEMBER

You don't have to type the source code for this chapter manually. In fact, using the downloadable source is a lot easier. The source code for this chapter appears in the `DSPD_0601_Data_Errors.ipynb` source code file for Python and the `DSPD_R_0601_Data_Errors.ipynb` source code file for R. See the Introduction for details on how to find these source files.

Considering the Types of Data Errors

For many people, *error* equates to *wrong.* However, in many cases, data is correct, yet also erroneous. You can consider data errant when it meets any of these criteria:

- Incorrect
- Missing
- Wrong type
- Malformatted (perhaps using an outdated standard)
- Wrong format for the task
- Incomplete
- Imprecise
- Misaligned (shifted in position within a field)
- Outdated
- Consists of opinion rather than fact
- Misclassified

In fact, this list could extend further, but it presents the kinds of things you should consider when looking for data errors. Trying to find just these types of data errors would be difficult, to say the least, but you can also classify data errors in another way:

- Automatic code detection, such as missingness
- Deterrence through form design, such as incompleteness
- External cleaning, such as misalignment
- Time stamping and other currency techniques, such as outdated data

Classifying the data error types by the techniques used to avoid or fix them is also helpful when considering how to manage your database. Part of your initial assessment of a data source must include a thorough examination of the kinds of data errors that the data contains, along with measures you can use to fix these errors well enough to perform analysis text.

IT'S ALL IN THE PREPARATION

This minibook may seem to spend a lot of time massaging data and little time in actually analyzing it. However, the majority of a data scientist's time is actually spent preparing data because the data is seldom in any order to actually perform analysis. To prepare data for use, a data scientist must:

- Get the data
- Aggregate the data
- Create data subsets
- Clean the data
- Develop a single dataset by merging various datasets together

Fortunately, you don't need to die of boredom while wading your way through these various tasks. Using Python and the various libraries it provides makes the task a lot simpler, faster, and more efficient, which is the point of spending all the time on seemingly mundane topics in these early chapters. The better you know how to use Python to speed your way through these repetitive tasks, the sooner you begin having fun performing various sorts of analysis on the data.

WARNING

After spending hours fixing a data source, it's always easy to think that the data is now somehow clean and pure. The problem is that it isn't clean or pure because many measures of correctness are subjective and biased toward a specific need. You must always assume that your data contains some number of errant entries, even when those entries might normally be considered correct, because they aren't correct for your particular need.

Obtaining the Required Data

Having plentiful data available isn't enough to perform analysis tasks successfully. Presently, an algorithm can't extract information directly from raw data. You can't simply tell an algorithm to analyze data from a number of unrelated sites and expect anything but gibberish as a result — assuming that the analysis even completes. Most algorithms rely on external collection and manipulation prior to analysis. When an algorithm collects useful information, it may not represent the right information. The following sections help you understand how to collect, manipulate, and automate data collection from an overview perspective.

Considering the data sources

The data you use comes from a number of sources. The most common data source is from information entered by humans at some point. Even when a system collects shopping-site data automatically, humans initially enter the information. A human clicks various items, adds them to a shopping cart, specifies characteristics (such as size) and quantity, and then checks out. Later, after the sale, the human gives the shopping experience, product, and delivery method a rating and makes comments. In short, every shopping experience becomes a data collection exercise as well.

WARNING

Consider that most data sources are incomplete or provide a biased perspective. For example, the shopping-site data may include purchases, but not returns or failed deliveries. As a consequence, the view the algorithm receives of sales is both incomplete and biased because it doesn't reflect the reality of the actual sales.

Many data sources today rely on input gathered from human sources. Humans also provide manual input. You call or go into an office somewhere to make an appointment with a professional. A receptionist then gathers information from you that's needed for the appointment. This manually collected data eventually ends up in a dataset somewhere for analysis purposes. When the receptionist makes a mistake, the mistake also appears in the dataset.

Data is also collected from sensors, and these sensors can take almost any form. For example, many organizations base physical data collection, such as the number of people viewing an object in a window, on cellphone detection. Facial recognition software could potentially detect repeat customers.

Sensors can create datasets from almost anything; however, their recordings are not always completely reliable. Depending on the application, you need a certain amount of data adjustment and cleaning. The weather service relies on datasets created by sensors that monitor environmental conditions such as rain, temperature, humidity, cloud cover, and so on. Robotic monitoring systems help correct small flaws in robotic operation by constantly analyzing data collected by monitoring sensors. A sensor, combined with a small AI application, could tell you when your dinner is cooked to perfection tonight. The sensor collects data, but the AI application uses rules to help define when the food is properly cooked.

Obtaining reliable data

The word *reliable* seems so easy to define, yet so hard to implement. Something is reliable when the results it produces are both expected and consistent. A reliable data source produces mundane data that contains no surprises; no one is shocked in the least by the outcome. Depending on your perspective, it could actually be a good thing that most people aren't yawning and then falling asleep when reviewing data. The surprises make the data worth analyzing and reviewing. Consequently, data has an aspect of duality. People want reliable, mundane, fully anticipated data that simply confirms what they already know, but the unexpected is what makes collecting the data useful in the first place.

Still, you don't want data that is so far out of the ordinary that it becomes almost frightening to review. You need to maintain balance when obtaining data. The data must fit within certain limits (as described in the "Manicuring the Data" section, later in this chapter). It must also meet specific criteria as to truth value (as described in the "Considering the Five Mistruths in Data" section of Chapter 2 of this minibook). The data must also come at expected intervals, and all the fields of the incoming data record must be complete.

REMEMBER

To some extent, data security also affects data reliability. Data consistency comes in several forms. When the data arrives, you can ensure that it falls within expected ranges and appears in a particular form. However, after you store the data, the reliability can decrease unless you ensure that the data remains in the expected form. An entity fiddling with the data affects reliability, making the data suspect and potentially unusable for analysis later. Ensuring data reliability means that after the data arrives, no one tampers with it to make it fit within an expected domain (making it mundane as a result).

Making human input more reliable

Humans make mistakes — it's part of being human. In fact, expecting that humans won't make mistakes is unreasonable. Yet, many application designs assume that humans somehow won't make mistakes of any sort. The design expects that everyone will simply follow the rules. Unfortunately, the vast majority of users are guaranteed to not even read the rules because most humans are also lazy or too pressed for time when it comes to doing things that don't really help them directly.

Consider the entry of a state into a form. If you provide just a text field, some users might input the entire state name, such as Kansas. Of course, some users will make a typo or capitalization error and come up with Kansus or kANSAS. People and organizations have various approaches to performing the task of entering a state name that makes these sorts of errors more prevalent:

- Someone in the publishing industry might use the Associated Press (AP) style guide and input Kan.
- Someone who is older and used to the Government Printing Office (GPO) guidelines might input Kans.
- The U.S. Post Office (USPS) uses KS.
- The U.S. Coast Guard uses KA.
- The International Standards Organization (ISO) form goes with US-KS.

Mind you, this is just a state entry, which is reasonably straightforward — or so you thought before reading this section. Clearly, because the state isn't going to change names anytime soon, you could simply provide a drop-down list box on the form for choosing the state in the required format, thereby eliminating differences in abbreviation use, typos, and capitalization errors in one fell swoop.

REMEMBER

Drop-down list boxes work well for an amazing array of data inputs, and using them ensures that human input into those fields becomes extremely reliable because the human has no choice but to use one of the default entries. Of course, the human can always choose the incorrect entry, which is where double-checks come into play. Some newer applications compare the ZIP code to the city and state entries to see whether they match. When they don't match, the user is asked again to provide the correct input. This double-check verges on being annoying (see the "More annoying than useful input aids" sidebar for details), but the user is unlikely to see it very often, so it shouldn't become too annoying.

MORE ANNOYING THAN USEFUL INPUT AIDS

Input validation is almost more of an art than a science because using rote rules seldom produces a useful and pleasurable input experience. Using a drop-down list for states works as long as the list is complete and the person is where you expect them to be. However, if you have an international customer who has an address outside the country for which you wrote the application, you suddenly find that the bulletproof input field almost makes it impossible for the user to complete the form. The input field has gone from being useful and convenient to being frustrating and annoying because of a limit on the kind of correct data the user can enter.

Input fields can also control the form of input, to an extent. The problem is that what the developer thinks is helpful really isn't helpful at all. You have probably encountered shopping sites that disallow dashes between the numbers of a credit card or require a telephone number to appear in a specific form. The form beeps unhelpfully in many cases until you discover just what format the developer wants. In some cases, the customer will finally give up and go anywhere else that has a better form and the desired product. A helpful input is one that accepts the odd formatting that the user may want to provide and automatically reformats the data as needed. When the user enters (555)123-4567, the form might automatically reformat it as 1-555-123-4567. The user is happy, the database is less error prone, and the developer gets to go home for the weekend.

A user may also not want to share certain personal details, such as age, gender, or sexual orientation. If your form requires such entries, the user might go somewhere else rather than provide the personal information. In fact, given the manner in which society has progressed, assuming anything about a person based on predefined biases will almost certainly cause problems for the business. For example, some people don't identify as either male or female, so assuming that they do is a bad choice.

Forms that require more information than the user is willing to provide are also unhelpful. A user may be willing to provide a numeric evaluation of a product, but may not want to provide a written comment. Many forms require both, which means that a survey or other means of obtaining information goes unanswered. A form should have a default non-answer value.

The ultimate of unhelpful aids, however, is the input field that simply assumes that the user knows what to fill in. Address fields are especially bad in this area. All you might see is a series of inputs that look sort of like a mailing label on an envelope, which

(continued)

(continued)

assumes that the user is aware of that particular label format. It also assumes that the user can see the form, although a user with special visual needs may use a screen reader that will have no idea of what to do with the form. Every input should have an associated label that's short, yet expresses precisely what information to provide. It should also have a help screen to provide more details and examples of what information to provide. The error message associated with the input should also say precisely what is wrong with the input, rather than make the user guess.

Even with cross-checks and static entries, humans still have plenty of room for making mistakes. For example, entering numbers can be problematic. When a user needs to enter 2.00, you might see 2, or 2.0, 2., or any of a variety of other entries. Fortunately, parsing the entry and reformatting it will fix the problem, and you can perform this task automatically, without the user's aid.

Unfortunately, reformatting won't correct an errant numeric input. You can partially mitigate such errors by including range checks. Consider, for example, how to process a customer's return of some purchased merchandise. A customer can't buy –5 bars of soap. The legitimate way to show the customer returning the bars of soap is to process a return, not a sale. However, the user might have simply made an error, and you can provide a message stating the proper input range for the value.

Using automated data collection

Some people think that automated data collection solves all the human input issues associated with datasets. In fact, automated data collection does provide a number of benefits:

- Better consistency
- Improved reliability
- Lower probability of missing data
- Enhanced accuracy
- Reduced variance for things like timed inputs

Unfortunately, to say that automated data collection solves every issue is simply incorrect. Automated data collection still relies on sensors, applications, and computer hardware designed by humans that provide access only to the data that humans decide to allow. Because of the limits that humans place on the characteristics of automated data collection, the outcome often provides less helpful information than hoped for by the designers. Consequently, automated data collection is in a constant state of flux as designers try to solve the input issues.

Automated data collection also suffers from both software and hardware errors present in any computing system, but with a higher potential for *soft issues* (which arise when the system is apparently working but isn't providing the desired result) than other kinds of computer-based setups. When the system works, the reliability of the input far exceeds human abilities. However, when soft issues occur, the system often fails to recognize that a problem exists, as a human might, and therefore the dataset could end up containing more mediocre or even bad data.

Validating Your Data

When it comes to data, no one really knows what a large database contains. Yes, everyone has seen bits and pieces of it, but when you consider the size of some databases, viewing it all would be physically impossible. Because you don't know what's in there, you can't be sure that your analysis will actually work as desired and provide valid results. In short, you must validate your data before you use it to ensure that the data is at least close to what you expect it to be. This means performing tasks such as removing duplicate records before you use the data for any sort of analysis (duplicates would unfairly weight the results).

REMEMBER

However, you do need to consider what validation actually does for you. It doesn't tell you that the data is correct or that there won't be values outside the expected range. In fact, later chapters help you understand the techniques for handling these sorts of issues. What validation does is ensure that you can perform an analysis of the data and reasonably expect that analysis to succeed. Later, you need to perform additional massaging of the data to obtain the sort of results that you need in order to perform the task you set out to perform in the first place.

Figuring out what's in your data

Figuring out what your data contains is important because checking data by hand is sometimes simply impossible because of the number of observations and variables. In addition, hand verifying the content is time consuming, error prone, and, most important, really boring. Finding duplicates is important because you end up

- Spending more computational time to process duplicates, which slows your algorithms down.
- Obtaining false results because duplicates implicitly overweight the results. Because some entries appear more than once, the algorithm considers these entries more important.

As a data scientist, you want your data to enthrall you, so it's time to get it to talk to you — not figuratively, of course, but through the wonders of pandas, as shown in the following example:

```
from lxml import objectify
import pandas as pd

xml = objectify.parse(open('XMLData2.xml'))
root = xml.getroot()
df = pd.DataFrame(columns=('Number', 'String', 'Boolean'))

for i in range(0,4):
    obj = root.getchildren()[i].getchildren()
    row = dict(zip(['Number', 'String', 'Boolean'],
                   [obj[0].text, obj[1].text,
                    obj[2].text]))
    row_s = pd.Series(row)
    row_s.name = i
    df = df.append(row_s)

search = pd.DataFrame.duplicated(df)
print(df)
print()
print(search[search == True])
```

This example shows how to find duplicate rows. It relies on a modified version of the `XMLData.xml` file found in Book 2, Chapter 4, `XMLData2.xml`, which contains a simple repeated row in it. A real data file contains thousands (or more) of records and possibly hundreds of repeats, but this simple example does the job. The example begins by reading the data file into memory using the same technique as that explored in the "Working with a simple XML file" section of Book 2, Chapter 4. It then places the data into a `DataFrame`.

At this point, your data is corrupted because it contains a duplicate row. However, you can get rid of the duplicated row by searching for it. The first task is to create a search object containing a list of duplicated rows by calling `pd.DataFrame.duplicated()`. The duplicated rows contain a `True` next to their row number.

Of course, now you have an unordered list of rows that are and aren't duplicated. The easiest way to determine which rows are duplicated is to create an index in which you use `search == True` as the expression. Following is the output you see from this example. Notice that row 3 is duplicated in the `DataFrame` output and that row 3 is also called out in the `search` results:

```
  Number  String Boolean
0      1   First    True
1      2  Second   False
2      3   Third    True
3      3   Third    True

3    True
dtype: bool
```

Removing duplicates

To get a clean dataset, you want to remove the duplicates from it. Fortunately, you don't have to write any weird code to get the job done; pandas does it for you, as shown in the following example:

```
from lxml import objectify
import pandas as pd

xml = objectify.parse(open('XMLData2.xml'))
root = xml.getroot()
df = pd.DataFrame(columns=('Number', 'String', 'Boolean'))
for i in range(0,4):
    obj = root.getchildren()[i].getchildren()
    row = dict(zip(['Number', 'String', 'Boolean'],
                   [obj[0].text, obj[1].text,
                    obj[2].text]))
    row_s = pd.Series(row)
    row_s.name = i
    df = df.append(row_s)

print(df.drop_duplicates())
```

As with the previous example, you begin by creating a DataFrame that contains the duplicate record. To remove the errant record, all you need to do is call `drop_duplicates()`. Here's the result you get:

```
  Number  String Boolean
0      1   First    True
1      2  Second   False
2      3   Third    True
```

Creating a data map and a data plan

You need to know about your dataset — that is, how it looks statically. A *data map* is an overview of the dataset. You use it to spot potential problems in your data, such as

- Redundant variables
- Possible errors
- Missing values
- Variable transformations

Checking for these problems goes into a *data plan*, which is a list of tasks you have to perform to ensure the integrity of your data. The following example shows a data map, A, with two datasets, B and C:

```
import pandas as pd
pd.set_option('display.width', 55)

df = pd.DataFrame({'A': [0,0,0,0,0,1,1],
                   'B': [1,2,3,5,4,2,5],
                   'C': [5,3,4,1,1,2,3]})

a_group_desc = df.groupby('A').describe()
print(a_group_desc)
```

In this case, the data map uses 0s for the first series and 1s for the second series. The `groupby()` function places the datasets, B and C, into groups. To determine whether the data map is viable, you obtain statistics using `describe()`. What you end up with is a dataset B, series 0 and 1, and dataset C, series 0 and 1, as shown in the following output:

```
      B                                                \
  count mean       std  min   25%  50%   75%  max
A
0   5.0  3.0  1.581139  1.0  2.00  3.0  4.00  5.0
1   2.0  3.5  2.121320  2.0  2.75  3.5  4.25  5.0

      C
  count mean       std  min   25%  50%   75%  max
A
0   5.0  2.8  1.788854  1.0  1.00  3.0  4.00  5.0
1   2.0  2.5  0.707107  2.0  2.25  2.5  2.75  3.0
```

These statistics tell you about the two dataset series. The breakup of the two datasets using specific cases is the *data plan.* As you can see, the statistics tell you that this data plan may not be viable because some statistics are relatively far apart.

TIP

The default output from `describe()` shows the data unstacked. Unfortunately, the unstacked data can print with an unfortunate break, making it very hard to read. To keep this break from happening, you set the width you want to use for the data by calling `pd.set_option('display.width', 55)`. You can set a number of pandas options this way by using the information found at `https://pandas.pydata.org/pandas-docs/stable/generated/pandas.set_option.html`.

Although the unstacked data is relatively easy to read and compare, you may prefer a more compact presentation. In this case, you can stack the data using the following code:

```
stacked = a_group_desc.stack()
print(stacked)
```

Using `stack()` creates a new presentation. Here's the output shown in a compact form:

```
                B         C
A
0 count  5.000000  5.000000
  mean   3.000000  2.800000
  std    1.581139  1.788854
  min    1.000000  1.000000
  25%    2.000000  1.000000
  50%    3.000000  3.000000
  75%    4.000000  4.000000
  max    5.000000  5.000000
1 count  2.000000  2.000000
  mean   3.500000  2.500000
  std    2.121320  0.707107
  min    2.000000  2.000000
  25%    2.750000  2.250000
  50%    3.500000  2.500000
  75%    4.250000  2.750000
  max    5.000000  3.000000
```

Of course, you may not want all the data that `describe()` provides. Perhaps you really just want to see the number of items in each series and their mean. Here's how you reduce the size of the information output:

```
print(a_group_desc.loc[:,(slice(None),['count','mean']),])
```

Using `loc` lets you obtain specific columns. Here's the final output from the example showing just the information you absolutely need to make a decision:

```
      B          C
  count mean count mean
A
0   5.0  3.0   5.0  2.8
1   2.0  3.5   2.0  2.5
```

Manicuring the Data

Some people use the term *manipulation* when speaking about data, giving the impression that the data is somehow changed in an unscrupulous or devious manner. Perhaps a better term would be *manicuring,* which makes the data well shaped and lovely. No matter what term you use, however, raw data seldom meets the requirements for processing and analysis. To get something out of the data, you must manicure it to meet specific needs. The following sections discuss data manicuring needs.

Dealing with missing data

To answer a given question correctly, you must have all the facts. You can guess the answer to a question without all the facts, but then the answer is just as likely to be wrong as correct. Often, someone who makes a decision, essentially answering a question, without all the facts is said to jump to a conclusion. The following sections discuss the issue of missing data and what to do about it.

Understanding how missing data affects a dataset

When analyzing data, you have probably jumped to more conclusions than you think because of missing data. A *data record,* which is one entry in a *dataset* (which in turn is all the data), consists of *fields* that contain facts used to answer a question. Each field contains a single kind of data that addresses a single fact. If that field is empty, you don't have the data you need to answer the question using that particular data record.


REMEMBER

As part of the process of dealing with missing data, you must know that the data is missing. Identifying that your dataset is missing information can actually be quite hard because it requires you to look at the data at a low level — something that most people aren't prepared to do and is time consuming even if you do have the required skills. Often, your first clue that data is missing is the preposterous

answers that your questions get from the algorithm and associated dataset. When the algorithm is the right one to use, the dataset must be at fault.

A problem can occur when the data collection process doesn't include all the data needed to answer a particular question. Sometimes you're better off to actually drop a fact rather than use a considerably damaged fact.

Less damaged fields can have data missing in one of two ways. Randomly missing data is often the result of human or sensor error. It occurs when data records throughout the dataset have missing entries. Sometimes a simple glitch causes the damage. Sequentially missing data occurs during some type of generalized failure. An entire segment of the data records in the dataset lack the required information, which means that the resulting analysis can become quite skewed.

Fixing randomly missing data is easiest. You can use a simple median or average value as a replacement. No, the dataset isn't completely accurate, but it will likely work well enough to obtain a reasonable answer. In some cases, data scientists used a special algorithm to compute the missing value, which can make the dataset more accurate at the expense of computational time.

Sequentially missing data is significantly harder, if not impossible, to fix because you lack any surrounding data on which to base any sort of guess. If you can find the cause of the missing data, you can sometimes reconstruct it. However, when reconstruction becomes impossible, you can choose to ignore the field. Unfortunately, some answers will require that field, which means that you might need to ignore that particular sequence of data records — potentially causing incorrect output.

Finding the missing data

TIP

As a first step, count the number of missing cases in each variable. When a variable has too many missing cases, you may need to drop it from the training and test dataset. A good rule of thumb is to drop a variable if more than 90 percent of its instances are missing.

Some learning algorithms do not know how to deal with missing values and report errors in both training and test phases, whereas other models treat them as zero values, causing an underestimation of the predicted value or probability (it's just as if part of the formula isn't working properly). Consequently, you need to replace all the missing values in your data matrix with some suitable value for your analysis to succeed.

Many reasons exist for missing data, but the essential point is whether the data is missing randomly or in a specific order. Random missing data is ideal because you can guess its value using a simple average, a median, or another algorithm

without too many concerns. Some cases contain a strong bias toward certain kinds of examples. For instance, think of the case of studying the income of a population. Wealthy people (for taxation reasons, presumably) tend to hide their true income by reporting to you that they don't know. Poor people, on the other hand, may say that they don't want to report their income for fear of negative judgement. If you miss information from certain strata of the population, repairing the missing data can be difficult and misleading because you may think that such cases are just like the others. Instead, they are quite different. Therefore, you can't simply use average values to replace the missing values — you must use complex approaches and tune them carefully. Moreover, identifying cases that aren't missing data at random is difficult because it requires a closer inspection of how missing values are associated with other variables in the dataset.


REMEMBER

When data is missing at random, you can easily repair the empty values because you obtain hints to their true value from other variables. When data isn't missing at random, you can't get good hints from other available information unless you understand the data association with the missing case. Therefore, if you have to figure out missing income in your data, and it is missing because the person is wealthy, you can't replace the missing value with a simple average because you'll replace it with a medium income. Instead, you should use an average of the income of wealthy people as a replacement.


TIP

When data isn't missing at random, the fact that the value is missing is informative because it helps track down the missing group. You can leave the chore of looking for the reason that it's missing to your algorithm by building a new binary feature that reports when the value of a variable is missing. Consequently, the algorithm will determine the best value to use as a replacement by itself.

Encoding missingness

You have a few possible strategies to handle missing data effectively. Your strategy may change if you have to handle missing values in *quantitative* (values expressed as numbers) or qualitative features. *Qualitative* features, although also expressed by numbers, are in reality referring to concepts, so their values are somewhat arbitrary and you cannot meaningfully compute an average or perform other computations on them.


TIP

When working with qualitative features, your value guessing should always produce integer numbers, based on the numbers used as codes. Common strategies for missing data handling are as follows:

- **Replace missing values with a computed constant such as the mean or the median value.** If your feature is a category, you must provide a specific value because the numbering is arbitrary, and using mean or median doesn't make sense. Use this strategy when the missing values are random.

- **Replace missing values with a value outside the normal value range of the feature.** For instance, if the feature is positive, replace missing values with negative values. This approach works fine with decision tree–based algorithms and qualitative variables.
- **Replace missing values with 0, which works well with regression models and standardized variables.** This approach is also applicable to qualitative variables when they contain binary values.
- **Interpolate the missing values when they are part of a series of values tied to time.** This approach works only for quantitative values. For instance, if your feature is daily sales, you could use a moving average of the last seven days or pick the value at the same time the previous week.
- **Impute their value using the information from other predictor features (but never use the response variable).** Particularly in R, specialized libraries like missForest (`https://cran.r-project.org/web/packages/missForest/index.html`), MICE (`https://cran.r-project.org/web/packages/mice/index.html`), and Amelia II (`https://gking.harvard.edu/amelia`) can do everything for you. Scikit-learn recently introduced an experimental missing values imputer (which you can find at `https://scikit-learn.org/stable/modules/generated/sklearn.impute.IterativeImputer.html`) that allows imputing data in Python using Multivariate Imputation by Chained Equations (MICE), missForest, or even Amelia methodologies.

TIP

Another good practice is to create a new binary feature for each variable whose values you repaired. The binary variable will track variations due to replacement or imputing with a positive value, and your machine learning algorithm can figure out when it must make additional adjustments to the values you actually used.

Inputting missing data

In Python, notification of missing values is made possible using the `ndarray` data structure from the NumPy package. Python marks missing values with a special value that appears printed on the screen as NaN (Not a Number). The `DataFrame` data structure from the pandas package offers methods for both replacing missing values and dropping variables.

The following Python example demonstrates how to perform replacement tasks. It begins by creating a dataset of 5 observations and 3 features, named "A," "B," "C":

```
import pandas as pd
import numpy as np
data = pd.DataFrame([[1,2,np.nan],[np.nan,2,np.nan],
                     [3,np.nan,np.nan],[np.nan,3,8],
                     [5,3,np.nan]],columns=['A','B','C'])
```

```
print(data,'\n') # prints the data
# counts NaN values for each feature
print(data.isnull().sum(axis=0))
```

Notice the use of `np.nan` to mark missing values in the code. When you run this example, you see the following output:

```
    A   B   C
0   1   2 NaN
1 NaN   2 NaN
2   3 NaN NaN
3 NaN   3   8
4   5   3 NaN

A    2
B    1
C    4
dtype: int64
```

Because feature C has just one value, you can drop it from the dataset. The code then replaces the missing values in feature B with a medium value and interpolates the value in feature A because it displays a progressive order:

```
# Drops definitely C from the dataset
data.drop('C', axis=1, inplace=True)
# Creates a placeholder for B's missing values
data['missing_B'] = data['B'].isnull().astype(int)
# Fills missings in B using B's average
data['B'].fillna(data['B'].mean(), inplace=True)
# Interpolates A
data['A'].interpolate(method='linear', inplace=True)
print(data)
```

Here is the output you see when you run the code:

```
   A    B  missing_B
0  1  2.0          0
1  2  2.0          0
2  3  2.5          1
3  4  3.0          0
4  5  3.0          0
```

The printed output is the final dataset. Be sure to note that the mean of B isn't an integer value, so the code converted all B values to floating numbers. This approach makes sense if B is numeric. If it were a category, and the numbering

were marking a class, the code should have filled the feature using the command `data['B'].fillna(data['B'].mode().iloc[0], inplace=True)`, which uses the mode, that is, the first most frequent value in the series.

TIP

As shown in the example, sometimes you can't do much with examples that have a lot of missing values in their features. In such cases:

- When the example is for training, remove it from the set (a procedure called *listwise deletion*) so that the incomplete cases won't affect learning.
- When the example is part of your test, you shouldn't remove it, but rather use it to evaluate how well your algorithm handles such situations.

Considering data misalignments

Data might exist for each of the data records in a dataset, but it might not align with other data in other datasets you own. For example, the numeric data in a field in one dataset might be a floating-point type (with decimal point), but an integer type in another dataset. Before you can combine the two datasets, the fields must contain the same type of data.

All sorts of other kinds of misalignment can occur. For example, date fields are notorious for being formatted in various ways. To compare dates, the data formats must be the same. However, dates are also insidious in their propensity for looking the same but not being the same. For example, dates in one dataset might use Greenwich Mean Time (GMT) as a basis, while the dates in another dataset might use some other time zone. Before you can compare the times, you must align them to the same time zone. It can become even weirder when dates in one dataset come from a location that uses Daylight Saving Time (DST), but dates from another location don't.

Even when the data types and format are the same, other data misalignments can occur. For example, the fields in one dataset may not match the fields in the other dataset. In some cases, these differences are easy to correct. One dataset may treat first and last name as a single field, while another dataset might use separate fields for first and last name. The answer is to change all datasets to use a single field or to change them all to use separate fields for first and last name. Unfortunately, many misalignments in data content are harder to figure out. In fact, it's entirely possible that you might not be able to figure them out at all. However, before you give up, consider these potential solutions to the problem:

- Calculate the missing data from other data that you can access.
- Locate the missing data in another dataset.

- Combine datasets to create a whole that provides consistent fields.
- Collect additional data from various sources to fill in the missing data.
- Redefine your question so that you no longer need the missing data.

Separating out useful data

Some organizations are of the opinion that they can never have too much data, but an excess of data becomes as much of a problem as not enough. To solve problems efficiently, an algorithm-based task, such as AI, requires just enough data. Defining the question that you want to answer concisely and clearly helps, as does using the correct algorithm (or algorithm ensemble). Of course, the major problems with having too much data are that finding the solution (after wading through all that extra data) takes longer, and sometimes you get confusing results because you can't see the forest for the trees.

WARNING

As part of creating the dataset you need for analysis, you make a copy of the original data rather than modify it. Always keep the original, raw data pure so that you can use it for other analyses later. In addition, creating the right data output for analysis can require a number of tries because you may find that the output doesn't meet your needs. The point is to create a dataset that contains only the data needed for analysis, but keep in mind that the data may need specific kinds of pruning to ensure the desired output.

Dealing with Dates in Your Data

Dates can present problems in data. For one thing, dates are stored as numeric values. However, the precise value of the number depends on the representation for the particular platform and could even depend on the users' preferences. For example, Excel users can choose to start dates in 1900 or 1904 (`https://support.microsoft.com/en-us/help/214330/differences-between-the-1900-and-the-1904-date-system-in-excel`). The numeric encoding for each is different, so the same date can have two numeric values depending on the starting date.

In addition to problems of representation, you also need to consider how to work with time values. Creating a time value format that represents a value that the user can understand is hard. For example, you might need to use Greenwich Mean Time (GMT) in some situations but a local time zone in others. Transforming between

various times is also problematic, such as differentiating between 12-hour time and 24-hour time. With these kinds of time differences in mind, the following sections provide you with details on dealing with time issues.

Formatting date and time values

Obtaining the correct date and time representation can make performing analysis a lot easier. For example, you often have to change the representation to obtain a correct sorting of values. Python provides two common methods of formatting date and time. The first technique is to call `str()`, which simply turns a `datetime` value into a string without any formatting. The `strftime()` function requires more work because you must define how you want the `datetime` value to appear after conversion. When using `strftime()`, you must provide a string containing special directives that define the formatting. You can find a listing of these directives at `http://strftime.org/`.

Now that you have some idea of how time and date conversions work, it's time to see an example. The following example creates a `datetime` object and then converts it into a string using two different approaches:

```
import datetime as dt

now = dt.datetime.now()

print(str(now))
print(now.strftime('%a, %d %B %Y'))
```

In this case, you can see that using `str()` is the easiest approach. However, as shown by the following output, it may not provide the output you need. Using `strftime()` is infinitely more flexible:

```
2018-09-21 11:39:49.698891
Fri, 21 September 2018
```

Using the right time transformation

Time zones and differences in local time can cause all sorts of problems when you're performing analysis. In addition, some types of calculations simply require a time shift in order to get the right results. No matter what the reason, you may

need to transform one time into another time at some point. The following examples show some techniques you can employ to perform the task:

```
import datetime as dt

now = dt.datetime.now()
timevalue = now + dt.timedelta(hours=2)

print(now.strftime('%H:%M:%S'))
print(timevalue.strftime('%H:%M:%S'))
print(timevalue - now)
```

The `timedelta()` function makes the time transformation straightforward. You can use any of these parameter names with `timedelta()` to change a time and date value:

- `days`
- `seconds`
- `microseconds`
- `milliseconds`
- `minutes`
- `hours`
- `weeks`

You can also manipulate time by performing addition or subtraction on time values. You can even subtract two time values to determine the difference between them. Here's the output from this example:

```
11:42:22
13:42:22
2:00:00
```

Note that `now` is the local time, `timevalue` is two time zones different from this one, and there is a two-hour difference between the two times. You can perform all sorts of transformations using these techniques to ensure that your analysis always shows precisely the time-oriented values you need.