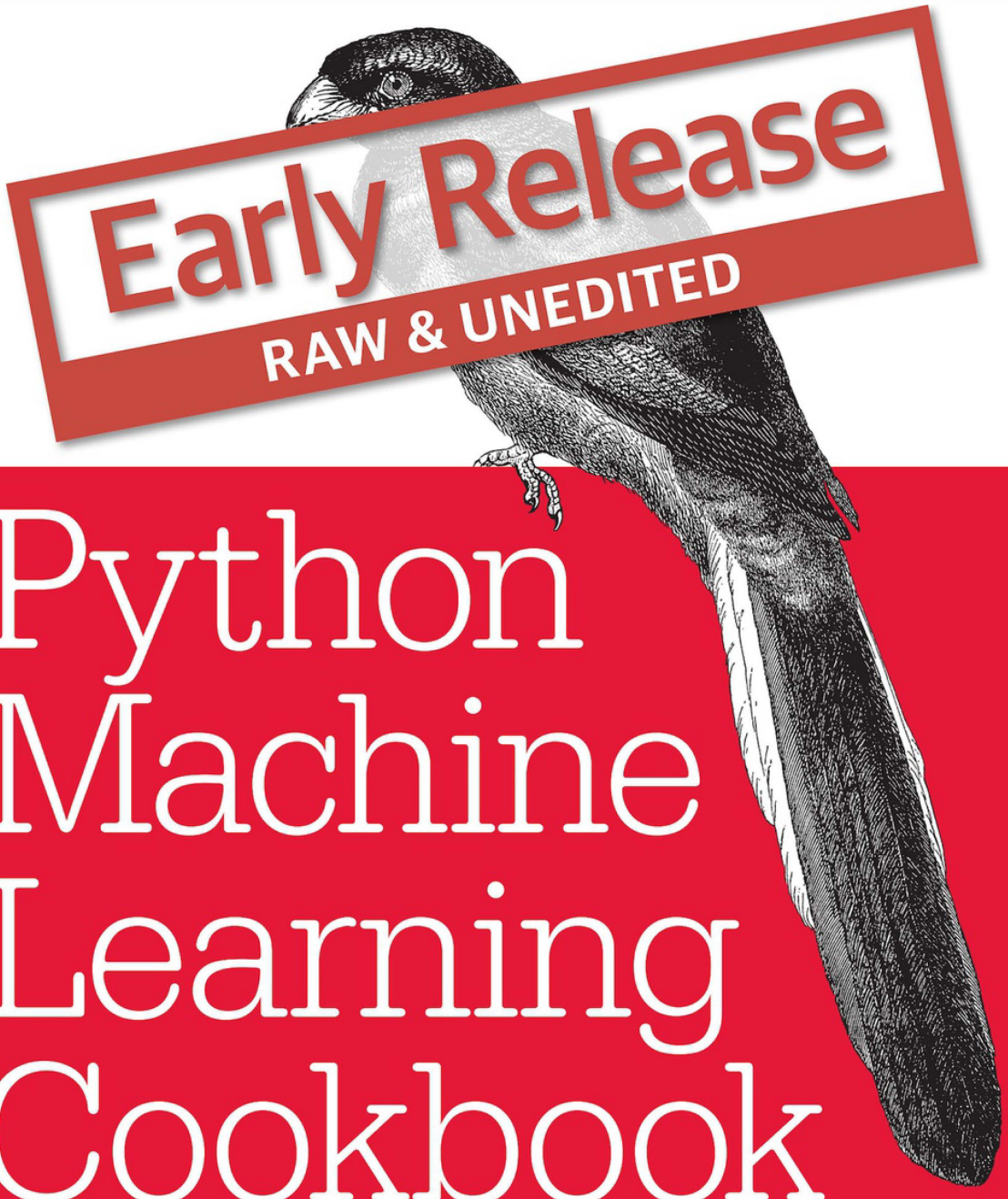
O'REILLY®



Python Machine Learning Cookbook

PRACTICAL SOLUTIONS FROM PREPROCESSING TO DEEP LEARNING

Chris Albon

1. 1. 1.0 Introduction
    1. 1.1 Loading A Sample Dataset
        1. Problem
        2. Solution
        3. Discussion
        4. See Also
    2. 1.2 Creating A Simulated Dataset
        1. Problem
        2. Solution
        3. Discussion
        4. See Also
    3. 1.3 Loading A CSV File
        1. Problem
        2. Solution
        3. Discussion
    4. 1.4 Loading An Excel File
        1. Problem
        2. Solution
        3. Discussion
    5. 1.5 Loading A JSON File
        1. Problem
        2. Solution
        3. Discussion
        4. See Also
    6. 1.6 Querying A SQL Database
        1. Problem
        2. Solution
        3. Discussion
        4. See Also

2. 2. 2.0 Introduction
    1. 2.1 Creating A DataFrame
        1. Problem
        2. Solution
        3. Discussion
    2. 2.2 Describing The Data
        1. Problem
        2. Solution
        3. Discussion
    3. 2.3 Navigating DataFrames
        1. Problem
        2. Solution
        3. Discussion
    4. 2.4 Selecting Rows Based On Conditionals
        1. Problem
        2. Solution
        3. Discussion
    5. 2.5 Replacing Values
        1. Problem
        2. Solution
        3. Discussion
    6. 2.6 Renaming Columns
        1. Problem
        2. Solution
        3. Discussion
    7. 2.7 Finding The Minimum, Maximum, Sum, Average, And Count
        1. Problem
        2. Solution
        3. Discussion

8. 2.9 Handling Missing Values
    1. Problem
    2. Solution
    3. Discussion
9. 2.8 Finding Unique Values
    1. Problem
    2. Solution
    3. Discussion
10. 2.10 Deleting A Column
    1. Problem
    2. Solution
    3. Discussion
11. 2.11 Deleting A Row
    1. Problem
    2. Solution
    3. Discussion
12. 2.12 Dropping Duplicate Rows
    1. Problem
    2. Solution
    3. Discussion
13. 2.13 Grouping Rows By Values
    1. Problem
    2. Solution
    3. Discussion
14. 2.14 Grouping Rows By Time
    1. Problem
    2. Solution
    3. Discussion
    4. See Also
15. 2.15 Looping Over A Column
    1. Problem
    2. Solution
    3. Discussion
16. 2.16 Applying A Function Over All Elements In A Column
    1. Problem
    2. Solution
    3. Discussion
17. 2.17 Applying A Function To Groups
    1. Problem
    2. Solution
    3. Discussion
18. 2.18 Concatenating DataFrames
    1. Problem
    2. Solution
    3. Discussion
19. 2.19 Merging DataFrames
    1. Problem
    2. Solution
    3. Discussion
    4. See Also

3. 3. 3.0 Introduction

1. 3.1 Rescaling A Feature
    1. Problem
    2. Solution
    3. Discussion
2. 3.2 Standardizing A Feature
    1. Problem
    2. Solution
    3. Discussion
3. 3.3 Normalizing Observations
    1. Problem
    2. Solution
    3. Discussion
4. 3.4 Generating Polynomial And Interaction Features
    1. Problem
    2. Solution
    3. Discussion
5. 3.5 Transforming Features
    1. Problem
    2. Solution
    3. Discussion
6. 3.6 Detecting Outliers
    1. Problem
    2. Solution
    3. Discussion
    4. See Also
7. 3.7 Handling Outliers
    1. Problem
    2. Solution
    3. Discussion
    4. See Also
8. 3.8 Discretizating Features
    1. Problem
    2. Solution
    3. Discussion
    4. See Also
9. 3.9 Grouping Observations Using Clustering
    1. Problem
    2. Solution
    3. Discussion
10. 3.10 Deleting Observations With Missing Values
    1. Problem
    2. Solution
    3. Discussion
    4. See Also
11. 3.11 Imputing Missing Values
    1. Problem
    2. Solution
    3. Discussion
    4. See Also

Python Machine Learning Cookbook

Practical solutions from preprocessing to deep learning

Chris Albon

Python Machine Learning Cookbook

by Chris Albon

Printed in the United States of America.

Published by O’Reilly Media, Inc. , 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com/safari). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com .

- Editor: Marie Beaugureau
- Production Editor: FILL IN PRODUCTION EDITOR
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest

- July 2017: First Edition

Revision History for the First Edition

- 2017-07-12: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491989319 for release details.

978-1-491-98931-9

[FILL IN]

Chapter 1. 1.0 Introduction

The first step in any machine learning endeavor is get to the raw data into our system. The raw data can be held in a log file, dataset file, or database. Furthermore, often we will want to get data from multiple sources. The recipes in this chapter look at methods of loading data from a variety of sources including CSV files and SQL databases. We also cover methods of generating simulated data with desirable properties for experimentation. Finally, while there are many ways to load data in the Python ecosystem, we will focus on using the pandas library's extensive set of methods for loading external data and scikit-learn -- an open source machine learning library Python -- for generating simulated data.

1.1 Loading A Sample Dataset

Problem

You need to load a pre-existing sample dataset.

Solution

scikit-learn comes with a number of popular datasets for you to use.

```
# Load scikit-learn's datasets
from sklearn import datasets

# Load the digits dataset
digits = datasets.load_digits()

# Create the features matrix
X = digits.data

# Create the target vector
y = digits.target

# View the first observation
X[0]
```

```
array([  0.,   0.,   5.,  13.,   9.,   1.,   0.,   0.,   0.,   0.,  13.,
        15.,  10.,  15.,   5.,   0.,   0.,   3.,  15.,   2.,   0.,  11.,
         8.,   0.,   0.,   4.,  12.,   0.,   0.,   8.,   8.,   0.,   0.,
         5.,   8.,   0.,   0.,   9.,   8.,   0.,   0.,   4.,  11.,   0.,
         1.,  12.,   7.,   0.,   0.,   2.,  14.,   5.,  10.,  12.,   0.,
         0.,   0.,   0.,   6.,  13.,  10.,   0.,   0.,   0.])
```

Discussion

Often we do not want to go through the work of loading, transforming, and cleaning a real world dataset before we can explore some machine learning algorithm or method. Luckily, `scikit-learn` comes with some common datasets we can quickly load. These datasets are often called "toy" datasets because they are far smaller and cleaner than a dataset we would see in the real world:

- `load_boston` contains 503 observations on Boston housing prices. It is a good dataset for exploring regression algorithms.
- `load_iris` contains 150 observations on the measurements of Iris flowers. It is a good dataset for exploring classification algorithms.
- `load_digits` contains 1797 observations from images of handwritten digits. It is good dataset for teaching image classification.

See Also

- scikit-learn toy datasets
- The Digit Dataset

1.2 Creating A Simulated Dataset

Problem

You need to generate a dataset of simulated data.

Solution

scikit-learn offers many methods for creating simulated data. Of those, three methods are particularly useful:

When we want a dataset designed to be used with linear regression, `make_regression` is a good choice:

```
# Load library
from sklearn.datasets import make_regression

# Generate features matrix, target vector, and the true coefficients
X, y, coef = make_regression(n_samples = 100,
                             n_features = 3,
                             n_informative = 3,
                             n_targets = 1,
                             noise = 0.0,
                             coef = True,
                             random_state = 1)

# View feature matrix and target vector
print('Feature Matrix\n', X[:3])
print('Target Vector\n', y[:3])
```

```
Feature Matrix
 [[ 1.29322588 -0.61736206 -0.11044703]
 [-2.793085    0.36633201  1.93752881]
 [ 0.80186103 -0.18656977  0.0465673 ]]
Target Vector
 [-10.37865986  25.5124503   19.67705609]
```

If we are interested in creating a simulated dataset for classification, we can use `make_classification`:

```
# Load library
from sklearn.datasets import make_classification

# Generate features matrix and target vector
X, y = make_classification(n_samples = 100,
                           n_features = 3,
                           n_informative = 3,
                           n_redundant = 0,
                           n_classes = 2,
                           weights = [.25, .75],
                           random_state = 1)

# View feature matrix and target vector
print('Feature Matrix\n', X[:3])
print('Target Vector\n', y[:3])
```

```
Feature Matrix
 [[ 1.06354768 -1.42632219  1.02163151]
 [ 0.23156977  1.49535261  0.33251578]
 [ 0.15972951  0.83533515 -0.40869554]]
Target Vector
 [1 0 0]
```

Finally, if we want a dataset designed to work well with clustering techniques, scikit-learn offers `make_blobs`:

```
# Load library
from sklearn.datasets import make_blobs

# Generate feature matrix and target vector
X, y = make_blobs(n_samples = 100,
                  n_features = 2,
```

```
                  centers = 3,
                  cluster_std = 0.5,
                  shuffle = True,
                  random_state = 1)

# View feature matrix and target vector
print('Feature Matrix\n', X[:3])
print('Target Vector\n', y[:3])
```

```
Feature Matrix
 [[ -1.22685609   3.25572052]
 [ -9.57463218  -4.38310652]
 [-10.71976941  -4.20558148]]
Target Vector
 [0 1 1]
```

Discussion

As might be apparent from the solutions, `make_regression` returns feature matrix of float values and a target vector of float values, while `make_classification` and `make_blobs` return a feature matrix of float values and a target vector of integers representing membership in a class.

scikit-learn's simulated datasets offer extensive options controlling the type of data generated. scikit-learn's documentation contains a full description of all the parameters, however a few are worth noting:

In `make_regression` and `make_classification`, `n_informative` determines the number of features that are used to generate the target vector. If `n_informative` is less than the total number of features (`n_features`), then the resulting dataset will have redundant features which can be identified through feature selection techniques (see Chapter 8).

In addition, `make_classification` contains a `weights` parameter allowing us to simulate datasets with imbalanced classes. For example, `weights = [.25, .75]` would return a dataset with 25% of observations belonging to one class and 75% of observations belonging to a second class.

For `make_blobs`, the `centers` parameter determines the number of clusters generated. Using the `matplotlib` visualization library we can visualize the clusters generated by `make_blobs`.

```
# Load library
import matplotlib.pyplot as plt

# View scatterplot
plt.scatter(X[:,0], X[:,1], c=y)
plt.show()
```

[Image to Come]

See Also

- `make_regression` documentation.
- `make_classification` documentation.
- `make_blobs` documentation.

1.3 Loading A CSV File

Problem

You need to import a comma-separated values (CSV) file.

Solution

Use the pandas library's `read_csv` to load a local or hosted CSV file:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/data.csv'

# Load dataset
df = pd.read_csv(url)

# View the first two rows
df.head(2)
```

	integer	datetime	category
0	5	2015-01-01 00:00:00	0
1	5	2015-01-01 00:00:01	0

Discussion

There are two things to note about loading CSV files. First, it is often useful to take a quick look at the contents of the file before loading. It can be very helpful to see how a dataset is structured beforehand and what parameters we need to set to load in the file. Second, `read_csv` has over 30 parameters and therefore the documentation can be daunting. Fortunately, those parameters are mostly there to allow it to handle a wide variety of CSV formats. For example, CSV files get their names from the fact the values are literally separated by commas (e.g. one row might be `2,"2015-01-01 00:00:00",0`), however it is common for "CSV" files to use other characters as separators, like vertical bars or tabs. pandas' `sep` parameter allows us to define the delimiter used in the file. Although it is not always the case, a common a formatting issue with CSV files is that the first line of the file is used to define column headers (e.g. `integer, datetime, category` in our solution). The `header` parameter allows us to specify if or where a header row exists. If a header row does not exists, we set `header=None`.

1.4 Loading An Excel File

Problem

You need to import an Excel spreadsheet.

Solution

Use the pandas library's `read_excel` to load an Excel spreadsheet:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/data.xlsx'

# Load data
df = pd.read_excel(url, sheetname=0, header=1)

# View the first two rows
df.head(2)
```

	5	2015-01-01 00:00:00	0
0	5	2015-01-01 00:00:01	0
1	9	2015-01-01 00:00:02	0

Discussion

This solution is similar to our solution for reading CSV files. The main difference is the additional parameter `sheetname` that specifies which sheet in the Excel file we wish to load. `sheetname` can accept both strings containing the name of the sheet and integers pointing to sheet positions (0-indexed). If we need to load multiple sheets, include them as a list. For example, `sheetname=[0,1,2, "Monthly Sales"]` will return a dictionary of pandas DataFrames containing the first, second, and third sheets and the sheet named `Monthly Sales`.

1.5 Loading A JSON File

Problem

You need to load a JSON file for data preprocessing.

Solution

The pandas library provides `read_json` to convert a JSON file into a pandas object:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/data.json'

# Load data
df = pd.read_json(url, orient='columns')

# View the first two rows
df.head(2)
```

	category	datetime	integer
0	0	2015-01-01 00:00:00	5
1	0	2015-01-01 00:00:01	5

Discussion

Importing JSON files into pandas is similar to the last few recipes we have seen. The key difference is the `orient` parameter, which indicates to pandas how the JSON file is structured. However, it might take some experimenting to figure out which argument (`split`, `records`, `index`, `columns`, and `values`) is the right one. Another helpful tool pandas offers is `json_normalize` which can help convert semi-structured JSON data into a Pandas DataFrame.

See Also

- `json_normalize` documentation

1.6 Querying A SQL Database

Problem

You need to load data from a database using the structured query language (SQL).

Solution

pandas' `read_sql_query` allows us to make a SQL query to a database and load it:

```
# Load libraries
import pandas as pd
from sqlalchemy import create_engine

# Create a connection to the database
db_connection = create_engine('sqlite:///sample.db')

# Load data
df = pd.read_sql_query('SELECT * FROM data', db_connection)

# View the first two rows
df.head(2)
```

	first_name	last_name	age	preTestScore	postTestScore
0	Jason	Miller	42	4	25
1	Molly	Jacobson	52	24	94

Discussion

Out of all of the recipes presented in this chapter, this recipe is probably the one you will use the most in the real world. SQL is the lingua franca for getting data from databases. In this recipe we first use `create_engine` to define a connection to a SQL database engine called SQLite. Next we use pandas' `read_sql_query` to query that database using SQL and put the results in a DataFrame.

SQL is a language in its own right and while beyond the scope of this book, it's certainly worth learning for anyone wanting to learn machine learning. Our SQL query, `SELECT * FROM data` asks the database to give us all columns (`*`) from the table called `data`.

See Also

- ‘SQLite’
- ‘W3Schools SQL Tutorial’

Chapter 2. 2.0 Introduction

Data wrangling is a broad term use, often informally, to describe the process of transforming raw data to a clean and organized format ready for further preprocessing, or final use. For us, data wrangling is only one step in preprocessing our data, but it is an important step.

The most common data structure used to "wrangle" data is the data frame, which can be both intuitive and incredibly versatile. Data frames are tabular, meaning that they are based on rows and columns like you would see in a spreadsheet. Here is a data frame created from data about passengers on the Titanic:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Show the first 5 rows
df.head(5)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.00	female	1	1
1	Allison, Miss Helen Loraine	1st	2.00	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.00	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.00	female	0	1
4	Allison, Master Hudson Trevor	1st	0.92	male	1	0

There are three important things to notice in this data frame:

First, in a data frame each row corresponds to one observation (e.g. a passenger) and each column corresponds to one feature (e.g. gender, age, etc.). For example, by looking at the first observation we can see that Miss Elisabeth Walton Allen stayed in first class, was 29 years old, was female, and survived the disaster.

Second, each column contains a name (e.g. `Name`, `PClass`, `Age`, etc.) and each row contains an index number (e.g. `0` for the lucky Miss Elisabeth Walton Allen). We will use these to select and manipulate observations and features.

Third, two columns, `Sex` and `SexCode`, contain the same information in different formats. In `Sex`, a women is indicated by the string `female` while in `SexCode` it is indicated using the integer `1`. We will want all our features to be unique and therefore we will need to remove one of these columns.

In this chapter, we will cover a wide variety of techniques to manipulate data frames using the pandas library with the goal of creating a clean, well structured set of observations for further preprocessing.

2.1 Creating A DataFrame

Problem

You want to create a new DataFrame.

Solution

pandas has many methods of creating a new DataFrame object. One easy method is to create an empty DataFrame using `DataFrame()` and then define each column separately:

```
# Load library
import pandas as pd

# Create DataFrame
df = pd.DataFrame()

# Add columns
df['Name'] = ['Jacky Jackson', 'Steven Stevenson']
df['Age'] = [38, 25]
df['Driver'] = [True, False]

# Show DataFrame
df
```

	Name	Age	Driver
0	Jacky Jackson	38	True
1	Steven Stevenson	25	False

Alternatively, once we have created a DataFrame object, we could also append new rows to the bottom:

```
# Create row
new_person = pd.Series(['Molly Mooney', 40, True], index=['Name','Age','Driver'])

# Append row
df.append(new_person, ignore_index=True)
```

	Name	Age	Driver
0	Jacky Jackson	38	True
1	Steven Stevenson	25	False
2	Molly Mooney	40	True

Discussion

pandas offers what can feel like an infinite number of ways of to crete a DataFrame. In the real world creating an empty DataFrame and then populating it will almost never happen. Instead, our DataFrames will be created from real data we have loading from other sources (e.g. a CSV file or database).

2.2 Describing The Data

Problem

You want to quickly view some characteristics of a DataFrame.

Solution

One of easiest things we can do after loading the data is view the first few rows using `head()`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Show two rows
df.head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

We can also take a look at the number of rows and columns:

```
# Show dimensions
df.shape
```

```
(1313, 6)
```

Additionally, we can get descriptive statistics for any numeric columns using `describe()`:

```
# Show statistics
df.describe()
```

	Age	Survived	SexCode
count	756.000000	1313.000000	1313.000000
mean	30.397989	0.342727	0.351866
std	14.259049	0.474802	0.477734
min	0.170000	0.000000	0.000000
25%	21.000000	0.000000	0.000000
50%	28.000000	0.000000	0.000000
75%	39.000000	1.000000	1.000000
max	71.000000	1.000000	1.000000

Discussion

After we load some data, it is a good idea to understand how it is structured and what kind of information it contains. Ideally, we would view the data directly in its file. But with most real world cases, the data would be too deep and varied with hundreds of thousands to millions of rows and columns, preventing us from getting a quick view. Instead, we have to rely on pulling samples to view small slices and summary statistics of the data.

In our solution above, we are using a toy dataset of the passengers of the doomed Titanic on her last voyage. Using `head()` we can take a look at the first few rows (five by default) of the data. Alternatively we can use `tail()` to view the bottom few rows. With `shape()` we can see how many rows and columns our DataFrame contains. And finally, with `describe()` we can see some basic descriptive statistics for any numerical column.

It is worth noting that summary statistics do not always tell the full story. For example, pandas treats the columns `Survived` and `SexCode` as numeric columns because they contain 1's and 0's. However, those digits indicate something beyond numbers, such as categories in this case. For example, if `Survived` equals 1, it indicates that the passenger survived the disaster. For this reason, some of the summary statistics provided don't make sense, such as the standard deviation of the `SexCode` column (an indicator of the passenger's gender).

2.3 Navigating DataFrames

Problem

You need to select individual datum or slices of a DataFrame.

Solution

Use `loc` or `iloc` to select one or more row or values:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Select first row
df.iloc[0]
```

```
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
Sex                               female
Survived                               1
SexCode                                1
Name: 0, dtype: object
```

We can use `:` to define a slice of rows we want, such as selecting the second to fourth row:

```
# Select three rows
df.iloc[1:4]
```

	Name	PClass	Age	Sex	Survived	SexCode
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0
3	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25.0	female	0	1

We can even use it to get all rows up to a point, such as all rows up to and including the third row:

```
# Select three rows
df.iloc[:3]
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

DataFrames do not need to be numerically indexed. We can set the index of a DataFrame to any value where the value is unique to each row. For example, we can set the index to be passenger names and then select rows using a name:

```
# Set index
df = df.set_index(df['Name'])

# Show row
df.loc['Allen, Miss Elisabeth Walton']
```

```
Name        Allen, Miss Elisabeth Walton
PClass                               1st
Age                                   29
```

```
Sex                                female
Survived                                1
SexCode                                 1
Name: Allen, Miss Elisabeth Walton, dtype: object
```

Discussion

All rows in a pandas DataFrames has a unique index value. By default, this index is an integer indicating its row position in the DataFrame, however it does not have to be. DataFrame indexes can be set to be unique alphanumeric strings or customer numbers. To select individual rows and slices of rows, pandas provides two methods:

- `loc` is useful when the index of the DataFrame is a label (e.g. a string).
- `iloc` works by looking for the position in the DataFrame. For example, `iloc[0]` will return the first row regardless of whether the index is an integer or a label.

It is useful to be comfortable with both `loc` and `iloc` since it will come up a lot during data cleaning.

2.4 Selecting Rows Based On Conditionals

Problem

You want to select DataFrame rows based on some condition.

Solution

This can be easily done in pandas. For example, if we wanted all the women on the Titanic:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Filter rows, show two rows
df[df['Sex'] == 'female'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Take a second and look at the format of the above solution. `df['Sex'] == 'female'` is our conditional statement, by wrapping that in `df[]` we are tell pandas to "select all the rows in the DataFrame where the value of `df['Sex']` is `'female'`.

Multiple conditions are easy as well. For example, here we select all the rows where the passenger is a female 65 or older:

```
# Filter rows
df[(df['Sex'] == 'female') & (df['Age'] >= 65)]
```

	Name	PClass	Age	Sex	Survived	SexCode
73	Crosby, Mrs Edward Gifford (Catherine Elizabet...	1st	69.0	female	1	1

Discussion

Conditionally selecting and filtering data is one of the most common tasks in data wrangling. You rarely want all the raw data you pull down from the source instead you are interested in only some subsection of it. For example, you might only be interested in stores in certain states or the records of patients over a certain age.

2.5 Replacing Values

Problem

You need to replace values in a DataFrame.

Solution

pandas `replace` is an easy way to find and replace values. For example, we replace any instance of `"female"` in `Sex` column with `"Woman"`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Replace values, show two rows
df['Sex'].replace("female", "Woman").head(2)
```

```
0    Woman
1    Woman
Name: Sex, dtype: object
```

We can also replace multiple values at the same time:

```
# Replace values, show three rows
df['Sex'].replace(["female", "male"], ["Woman", "Man"]).head(3)
```

```
0    Woman
1    Woman
2      Man
Name: Sex, dtype: object
```

We can also find and replace across the entire DataFrame by specifying `df` instead of `df['Sex']`:

```
# Replace values, show two rows
df.replace(1, "One").head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29	female	One	One
1	Allison, Miss Helen Loraine	1st	2	female	0	One

`replace` also accepts regular expressions:

```
# Replace values, show two rows
df.replace(r"1st", "First", regex=True).head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	First	29.0	female	1	1
1	Allison, Miss Helen Loraine	First	2.0	female	0	1

Discussion

`replace` is a tool we use to replace values that is simple and yet has the powerful ability to accept regular expressions.

2.6 Renaming Columns

Problem

You want to rename a column in a pandas DataFrame.

Solution

Renaming columns can be done using the `rename` method:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Rename column, show two rows
df.rename(columns={'PClass': 'Passenger Class'}).head(2)
```

	Name	Passenger Class	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Notice that the rename method can accept a dictionary as a parameter. We can use the dictionary to change multiple column names at once:

```
# Rename columns, show two rows
df.rename(columns={'PClass': 'Passenger Class', 'Sex': 'Gender'}).head(2)
```

	Name	Passenger Class	Age	Gender	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Discussion

Using `rename` with a dictionary as an argument to the `columns` parameter is my preferred way to rename columns because it works with any number of columns. If we want to rename all columns at once, this helpful snippet of code creates a dictionary with the old column names as keys and empty strings as values:

```
# Load library
import collections

# Create a dictionary
column_names = collections.defaultdict(str)

# Create keys
for name in df.columns:
    column_names[name]

# Show dictionary
column_names
```

```
defaultdict(str,
            {'Age': '',
             'Name': '',
             'PClass': '',
             'Sex': '',
             'SexCode': '',
             'Survived': ''})
```

2.7 Finding The Minimum, Maximum, Sum, Average, And Count

Problem

You want to find the min, max, sum, average, or count of a numeric column.

Solution

pandas comes with some built-in methods for commonly used descriptive statistics:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Calculate statistics
print('Maximum:', df['Age'].max())
print('Minimum:', df['Age'].min())
print('Mean:', df['Age'].mean())
print('Sum:', df['Age'].sum())
print('Count:', df['Age'].count())
```

```
Maximum: 71.0
Minimum: 0.17
Mean: 30.397989417989415
Sum: 22980.879999999997
Count: 756
```

Discussion

In addition to the statistics used in the solution, pandas offers variance (`var`), standard deviation (`std`), kurtosis (`kurt`), skewness (`skew`), standard error of the mean (`sem`), mode (`mode`), median (`median`), and a number of others.

Furthermore, we can also apply these methods to the whole DataFrame:

```
# Show counts
df.count()
```

```
Name        1313
PClass      1313
Age          756
Sex         1313
Survived    1313
SexCode     1313
dtype: int64
```

2.9 Handling Missing Values

Problem

You want to select missing values in a DataFrame:

Solution

`isnull` and `notnull` return booleans indicating if a value is missing:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

## Select missing values, show two rows
df[df['Age'].isnull()].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
12	Aubert, Mrs Leontine Pauline	1st	NaN	female	1	1
13	Barkworth, Mr Algernon H	1st	NaN	male	1	0

Discussion

Missing values are a ubiquitous problem in data wrangling yet many underestimate the difficulty working with missing data. pandas uses NumPy's `NaN` ("Not A Number") value to denote missing values, however it is important to note that `NaN` is not fully implemented natively in pandas. For example, if we wanted to replace all strings containing `male` with missing values we return an error:

```
# Attempt to replace values with NaN
df['Sex'] = df['Sex'].replace('male', NaN)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-2-129aa555dee3> in <module>()
      1 # Attempt to replace values with NaN
----> 2 df['Sex'] = df['Sex'].replace('male', NaN)


NameError: name 'NaN' is not defined
```

To have full functionality with `NaN` we need to import the NumPy library first:

```
# Load library
import numpy as np

# Replace values with NaN
df['Sex'] = df['Sex'].replace('male', np.nan)
```

Often times a dataset uses a specific value to denote a missing observation, which as `NONE`, `-999`, or .. pandas' `read_csv` includes a parameter allowing us to specify the values used to indicate missing values:

```
# Load data, set missing values
df = pd.read_csv(url, na_values=[np.nan, 'NONE', -999])
```

2.8 Finding Unique Values

Problem

You want to select all unique values in a column.

Solution

Use `unique` to view an array of all unique values in a column:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Select unique values
df['Sex'].unique()
```

```
array(['female', 'male'], dtype=object)
```

Alternatively, `value_counts` will display all unique values with the number of times each value appears:

```
# Show counts
df['Sex'].value_counts()
```

```
male      851
female    462
Name: Sex, dtype: int64
```

Discussion

Both `unique` and `value_counts` are useful for manipulating and exploring categorical columns. Very often in categorical columns there will be classes that need to be handled in the data wrangling phase. For example, in the Titanic dataset, `PClass` is a column indicating the class of a passenger's ticket. There were three classes on the Titanic, however if we use `value_counts` we can see a problem:

```
# Show counts
df['PClass'].value_counts()
```

```
3rd    711
1st    322
2nd    279
*        1
Name: PClass, dtype: int64
```

While almost all passengers belong to one of three classes as expected, a single passenger has the class `*`. There are a number of strategies for handling this type of issue which we will address in a later chapter, but for now just realize that "extra" classes are common in categorical data and should not be ignored.

2.10 Deleting A Column

Problem

You want to delete a column from your DataFrame.

Solution

The best way to delete a column is to use `drop` with the parameter `axis=1`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Delete column
df.drop('Age', axis=1).head(2)
```

	Name	PClass	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	female	1	1
1	Allison, Miss Helen Loraine	1st	female	0	1

You can also use a list of column names as the main argument to drop multiple columns at once:

```
# Drop columns
df.drop(['Age', 'Sex'], axis=1).head(2)
```

	Name	PClass	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	1	1
1	Allison, Miss Helen Loraine	1st	0	1

If a column does not have a name (which can sometimes happen), you can drop it by its column index using `df.columns`:

```
# Drop column
df.drop(df.columns[1], axis=1).head(2)
```

	Name	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	29.0	female	1	1
1	Allison, Miss Helen Loraine	2.0	female	0	1

Discussion

`drop` is the idiomatic method of deleting a column. An alternative method is `del df['Age']`, which works most of the time but is not recommended because of how it is called within pandas (the details of which are outside the scope of this book).

One habit I recommend learning is to never use pandas' `inplace=True` argument. Many pandas methods include an `inplace` parameter which when `True` edits the DataFrame directly. This can lead to problems in more complex data processing pipelines because we are treating the DataFrames as mutable objects (which it technically is). I recommend treating DataFrames as immutable objects. For example:

```
# Create a new DataFrame
df_no_name = df.drop(df.columns[0], axis=1)
```

In this example we are not mutating the DataFrame `df` but instead are making a new DataFrame that is an altered version of `df`. If you treat your DataFrames as immutable objects, you will save yourself a lot of headaches down the road.

2.11 Deleting A Row

Problem

You want to delete one or more rows from a DataFrame.

Solution

Use a boolean condition to create a new DataFrame excluding the rows you want to delete:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Delete rows, show first two rows of output
df[df['Sex'] != 'male'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Discussion

While technically you can use the `drop` method (for example `df.drop([0, 1], axis=0)` to drop the first two rows) a more practical method is simply to wrap a boolean condition inside `df[]`. The reason is because we can use the power of conditionals to delete either a single row or (far more likely) delete many rows at once.

We can use boolean conditions to easily delete single rows by matching a unique value:

```
# Delete row, show first two rows of output
df[df['Name'] != 'Allison, Miss Helen Loraine'].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

And we can even use it to delete a single row by row index:

```
# Delete row, show first two rows of output
df[df.index != 0].head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

2.12 Dropping Duplicate Rows

Problem

You want to drop duplicate rows from your DataFrame.

Solution

Use `drop_duplicates` but be mindful of the parameters:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Drop duplicates, show first two rows of output
df.drop_duplicates().head(2)
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
1	Allison, Miss Helen Loraine	1st	2.0	female	0	1

Discussion

A keen reader will notice that the solution above didn't actually drop any rows:

```
# Show number of rows
print("Number Of Rows In The Original DataFrame:", len(df))
print("Number Of Rows After Deduping:", len(df.drop_duplicates()))
```

```
Number Of Rows In The Original DataFrame: 1313
Number Of Rows After Deduping: 1313
```

The reason is because `drop_duplicates` defaults to only dropping rows that match perfect across all columns. Under this condition, every row in our DataFrame is unique. However, often we only want to consider only a subset of columns to check for duplicate rows. We can accomplish this using the `subset` parameter:

```
# Drop duplicates
df.drop_duplicates(subset=['Sex'])
```

	Name	PClass	Age	Sex	Survived	SexCode
0	Allen, Miss Elisabeth Walton	1st	29.0	female	1	1
2	Allison, Mr Hudson Joshua Creighton	1st	30.0	male	0	0

Take a close look at the output above, we told `drop_duplicates` to only consider any two rows with the same value for `Sex` to be duplicates and to drop them. Now we are left with a DataFrame of only two rows, one man and one woman. You might be asking why `drop_duplicates` decided to keep these two rows instead of two different rows. The answer is that `drop_duplicates` defaults to keeping the first occurrence of a duplicated row and dropping the rest. We can control this behavior using the `keep` parameter:

```
# Drop duplicates
df.drop_duplicates(subset=['Sex'], keep='last')
```

	Name	PClass	Age	Sex	Survived	SexCode
1307	Zabour, Miss Tamini	3rd	NaN	female	0	1
1312	Zimmerman, Leo	3rd	29.0	male	0	0

A related method is `duplicated` which returns a boolean series denoting if a row is a duplicate or now. This is a good option if you don't want to simply drop duplicates.

2.13 Grouping Rows By Values

Problem

You want to group individual rows according to some shared value.

Solution

`groupby` is one of the most powerful features in pandas:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Group rows, calculate mean
df.groupby('Sex').mean()
```

	Age	Survived	SexCode
Sex			
female	29.396424	0.666667	1.0
male	31.014338	0.166863	0.0

Discussion

`groupby` is where data wrangling really starts to take shape. It is very common to have a DataFrame where each row is a person or an event and we want to group them according to some criteria and then calculate a statistic. For example, you can imagine a DataFrame where each row is an individual sale at a national restaurant chain and we want the total sales per restaurant.

Users new to `groupby` often write a line like this and are confused by what is returned:

```
# Group rows
df.groupby('Sex')
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x10efacf28>
```

Why didn't it return something more useful? The reason is because `groupby` needs to be paired with some action we want to apply to each group, such as calculating an aggregate statistic (e.g. mean, median, total). When talking about grouping we often use shorthand and say "group by gender," but that is incomplete. For grouping to be useful we need to group by something and then apply a function to each of those groups:

```
# Group rows, count rows
df.groupby('Survived')['Name'].count()
```

```
Survived
0    863
1    450
Name: Name, dtype: int64
```

Notice `Name` added after the `groupby`? That is because particular summary statistics are only meaningful to certain types of data. For example, while calculating the average age by gender makes sense, calculating the total age by gender does not.

We can also group by a first column, then group that grouping by a second column:

```
# Group rows, calculate mean
df.groupby(['Sex','Survived'])['Age'].mean()
```

```
Sex     Survived
female  0           24.901408
        1           30.867143
male    0           32.320780
        1           25.951875
Name: Age, dtype: float64
```

2.14 Grouping Rows By Time

Problem

You need to group individual rows by time periods.

Solution

Use `resample` to group rows by chunks of time:

```
# Load libraries
import pandas as pd
import numpy as np

# Create date range
time_index = pd.date_range('06/06/2017', periods=100000, freq='30S')

# Create DataFrame
df = pd.DataFrame(index=time_index)

# Create column of random values
df['Sale_Amount'] = np.random.randint(1, 10, 100000)

# Group rows by week, calculate sum
df.resample('W').sum()
```

	Sale_Amount
2017-06-11	86375
2017-06-18	101305
2017-06-25	100680
2017-07-02	100761
2017-07-09	101035
2017-07-16	10529

Discussion

Our standard Titanic dataset does not contain a datetime column so for this recipe we have generated a simple DataFrame where each row is an individual sale. For each sale we know the date and time of that sale and the dollar amount of that sale (this data isn't realistic because every sale takes place precisely 30 second apart and is an exact dollar amount, but for the sake of simplicity let us pretend).

The raw data looks like this:

```
# Show three rows
df.head(3)
```

	Sale_Amount
2017-06-06 00:00:00	6
2017-06-06 00:00:30	8
2017-06-06 00:01:00	8

Notice that the date and time of each sale is the index of the DataFrame, this is because `resample` requires the index to be datetime-like values.

Using `resample` we can group the rows by a wide array of time periods (offsets) and then we can calculate some statistic on each time group:

```
# Group by two weeks, calculate mean
df.resample('2W').mean()
```

	Sale_Amount
2017-06-11	4.998553
2017-06-25	5.009549
2017-07-09	5.004861
2017-07-23	5.062019

```
# Group by month, count rows
df.resample('M').count()
```

	Sale_Amount
2017-06-30	72000
2017-07-31	28000

Readers might notice that in the two outputs above the datetime index is a date despite the fact we are grouping by weeks and months respectively. The reason is because by default `resample` returns the label of the right "edge" (the last label) of the time group. We can control this behavior using the `label` parameter:

```
# Group by month, count rows
df.resample('M', label='left').count()
```

	Sale_Amount

2017-05-31	72000
2017-06-30	28000

See Also

- List of pandas time offset aliases

2.15 Looping Over A Column

Problem

You want to iterate over every element in a column and apply some action.

Solution

You can treat a pandas column like any other sequence in Python:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Print first two names uppercased
for name in df['Name'][0:2]:
    print(name.upper())
```

```
ALLEN, MISS ELISABETH WALTON
ALLISON, MISS HELEN LORAINE
```

Discussion

In addition to loops (often called “for loops”), we can also use list comprehensions:

```
# Show first two names uppercased
[name.upper() for name in df['Name'][0:2]]
```

```
['ALLEN, MISS ELISABETH WALTON', 'ALLISON, MISS HELEN LORAINE']
```

Despite the temptation to fall back on for loops, a more Pythonic solution would use pandas’ `apply` method described in the next recipe.

2.16 Applying A Function Over All Elements In A Column

Problem

You want to apply some function over all elements in a column.

Solution

Use `map` to apply a built-in or custom function on every element in a column:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Create function
def uppercase(x):
    return x.upper()

# Apply function, show two rows
df['Name'].map(uppercase)[0:2]
```

```
0    ALLEN, MISS ELISABETH WALTON
1     ALLISON, MISS HELEN LORAINE
Name: Name, dtype: object
```

Discussion

`map` is a great way to do data cleaning and wrangling. It is common to write a function to perform some useful operation (e.g. separate first and last names, converting strings to floats etc.) and then map that function to every element in a column.

People are often confused by the difference between `map` and `apply` in pandas. `apply` is similar to `map`, except instead of element-wise operations, `apply` passes the entire column to the function. The confusion lies in the fact that if the function accepts sequences the two methods produce the same outputs:

```
# Load library
import numpy as np

# Map np.sum, show two values
df['Age'].map(np.sum)[0:2]
```

```
0    29.0
1     2.0
Name: Age, dtype: float64
```

```
# Apply np.sum, show two rows
df['Age'].apply(np.sum)[0:2]
```

```
0    29.0
1     2.0
Name: Age, dtype: float64
```

In general, I would suggest you default to `map` instead of `apply` because if you have any previous experience in computer science or programming the concept of mapping functions to elements of an array will be familiar.

2.17 Applying A Function To Groups

Problem

You have grouped rows using `groupby` and want to apply a function to each group.

Solution

Combine `groupby` and `apply`:

```
# Load library
import pandas as pd

# Create URL
url = 'https://raw.githubusercontent.com/chrisalbon/simulated_datasets/master/titanic.csv'

# Load data
df = pd.read_csv(url)

# Group rows, apply function to groups
df.groupby('Sex').apply(lambda x: x.count())
```

	Name	PClass	Age	Sex	Survived	SexCode
Sex						
female	462	462	288	462	462	462
male	851	851	468	851	851	851

Discussion

In the last recipe I mentioned that you should prefer `map` over `apply` in most cases. One of the exceptions is when you want to apply a function to groups. By combining `groupby` and `apply` we can calculate custom statistics or apply any function to each group separately.

2.18 Concatenating DataFrames

Problem

You want to concatenate two DataFrames.

Solution

Use `concat` with `axis=0` to concatenate along the row axis:

```
# Load library
import pandas as pd

# Create DataFrame
data_a = {'id': ['1', '2', '3'],
          'first': ['Alex', 'Amy', 'Allen'],
          'last': ['Anderson', 'Ackerman', 'Ali']}
df_a = pd.DataFrame(data_a, columns = ['id', 'first', 'last'])

# Create DataFrame
data_b = {'id': ['4', '5', '6'],
          'first': ['Billy', 'Brian', 'Bran'],
          'last': ['Bonder', 'Black', 'Balwner']}
df_b = pd.DataFrame(data_b, columns = ['id', 'first', 'last'])

# Concatenate DataFrames
pd.concat([df_a, df_b], axis=0)
```

	id	first	last
0	1	Alex	Anderson
1	2	Amy	Ackerman
2	3	Allen	Ali
0	4	Billy	Bonder
1	5	Brian	Black
2	6	Bran	Balwner

You can use `axis=1` to concatenate along the column axis:

```
# Concatenate DataFrames
pd.concat([df_a, df_b], axis=1)
```

	id	first	last	id	first	last
0	1	Alex	Anderson	4	Billy	Bonder
1	2	Amy	Ackerman	5	Brian	Black
2	3	Allen	Ali	6	Bran	Balwner

Discussion

Concatenating means to glue two objects together, and it is not a word you hear much outside of computer science and programming, so if you have not heard it before then do not worry. The informal definition of concatenate is to glue two objects together. In the solution we glued together two small DataFrames and used the `axis` parameter to indicate whether we wanted to stack the two DataFrames on top of each other or glue them together side by side.

Alternatively we can use `append` to add a new row to a DataFrame:

```
# Create row
chris = pd.Series([10, 'Chris', 'Chillon'], index=['id', 'first', 'last'])

# Append row
df_a.append(chris, ignore_index=True)
```

	id	first	last
0	1	Alex	Anderson
1	2	Amy	Ackerman
2	3	Allen	Ali
3	10	Chris	Chillon

2.19 Merging DataFrames

Problem

You want to merge two DataFrames.

Solution

To inner join, use `merge` with the `on` parameter to specify the column to merge on:

```
# Load library
import pandas as pd

# Create DataFrame
employee_data = {'employee_id': ['1', '2', '3', '4'],
                 'name': ['Amy Jones', 'Allen Keys', 'Alice Bees', 'Tim Horton']}
df_employees = pd.DataFrame(employee_data, columns = ['employee_id', 'name'])

# Create DataFrame
sales_data = {'employee_id': ['3', '4', '5', '6'],
              'total_sales': [23456, 2512, 2345, 1455]}
df_sales = pd.DataFrame(sales_data, columns = ['employee_id','total_sales'])

# Merge DataFrames
pd.merge(df_employees, df_sales, on='employee_id')
```

	employee_id	name	total_sales
0	3	Alice Bees	23456
1	4	Tim Horton	2512

`merge` defaults to inner joins. If we want to do an outer join, we can specify that with the `how` parameter:

```
# Merge DataFrames
pd.merge(df_employees, df_sales, on='employee_id', how='outer')
```

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0
4	5	NaN	2345.0
5	6	NaN	1455.0

The same parameter can be used to specify left and right joins:

```
# Merge DataFrames
pd.merge(df_employees, df_sales, on='employee_id', how='left')
```

	employee_id	name	total_sales
0	1	Amy Jones	NaN
1	2	Allen Keys	NaN
2	3	Alice Bees	23456.0
3	4	Tim Horton	2512.0

We can also specify the column name in each DataFrame to merge on:

```
# Merge DataFrames
pd.merge(df_employees, df_sales, left_on='employee_id', right_on='employee_id')
```

	employee_id	name	total_sales
0	3	Alice Bees	23456
1	4	Tim Horton	2512

Instead of merging on two columns we want to merge on the indexes of each DataFrame, we can replace the `left_on` and `right_on` parameters with `right_index=True` and `left_index=True`.

Discussion

Oftentimes, the data we need to use is complex; it doesn't always come in one piece. Instead in the real world, we're usually faced with disparate datasets, from multiple database queries or files. To get all that data into one place, we can load each data query or data file into pandas as individual DataFrames and then merge them together into a single DataFrame.

This process might be familiar to anyone who has used SQL, a popular choice for doing merging operations (called "joins"). While the exact parameters used by pandas will be different, they follow the same general patterns used by other software languages and tools.

There are three aspects to specify with any `merge` operation. First, we have to specify the two DataFrames we want to merge together. In the solution above we have named them `df_employees` and `df_sales`. Second, we have to specify the name(s) of the columns merge on -- that is, the columns whose values are shared between the two DataFrames. For example, in our solution both DataFrames have a column named `employee_id`. To merge the two DataFrames we will match up the values in each DataFrame's `employee_id` column with each other. If these two columns use the same name, we can use the `on` parameter. However, if they have different names we can use `left_on` and `right_on`.

What is the left and right DataFrame? The simple answer is that the left DataFrame is the first one we specified in `merge` and the right DataFrame is the second one. This language comes up again in the next sets of parameters we will need.

The last aspect, and most difficult for some people to grasp, is the the type of merge operation we want to conduct. This is specified by the `how` parameter. `merge` supports the four main types of joins:

1. Inner: Return only the rows that match in both DataFrames (e.g. return any row with an `employee_id` value appearing in both `df_employees` and `df_sales`)
2. Outer: Return all rows in both DataFrames. If a row exists in one DataFrame but not in the other DataFrame, fill NaN values for the missing values. (e.g. return all row in both `employee_id` and `df_sales`)
3. Left: Return all rows from the left DataFrame but only rows from the right DataFrame that matched with the left DataFrame. Fill NaN values for the missing values. (e.g. return all rows from `df_employees` but only rows from `df_sales` that have a value for `employee_id` that appears in `df_employees`)
4. Right: Return all rows from the right DataFrame but only rows from the left DataFrame that matched with the right DataFrame. Fill NaN values for the missing values. (e.g. return all rows from `df_sales` but only rows from `df_employees` that have a value for `employee_id` that appears in `df_sales`)

If you did not understand all of that right now, I encourage you to play around with the `how` parameter in your code and see how it affects what `merge` returns.

See Also

- A Visual Explanation of SQL Joins
- pandas Documentation On Merging

Chapter 3. 3.0 Introduction

Quantitative data is the measurement of something -- whether class size, monthly sales, or student scores. The natural way to represent these quantities is numerically (e.g. 29 students, $529,392 in sales, etc.). In this chapter, we will cover numerous strategies for transforming raw numerical data into features purpose-built for machine learning algorithms.

3.1 Rescaling A Feature

Problem

You need to rescale the values of a numerical feature to be between two values.

Solution

Use scikit-learn's `MinMaxScaler` to rescale a feature array:

```
# Load libraries
from sklearn import preprocessing
import numpy as np

# Create feature
x = np.array([[-500.5],
              [-100.1],
              [0],
              [100.1],
              [900.9]])

# Create scaler
minmax_scale = preprocessing.MinMaxScaler(feature_range=(0, 1))

# Scale feature
x_scale = minmax_scale.fit_transform(x)

# Show feature
x_scale

array([[ 0.        ],
       [ 0.28571429],
       [ 0.35714286],
       [ 0.42857143],
       [ 1.        ]])
```

Discussion

Rescaling is a common preprocessing task in machine learning. Many of the algorithms described later in this book will assume all features are on the same scale, typically 0 to 1 or -1 to 1. There are a number of Rescaling techniques, however one of the simplest is called **min-max scaling**. Min-max scaling uses the minimum and maximum values of a feature to rescale values to within a range. Specifically, min-max calculates:

where x is the feature vector, $x*{i}$ is an individual element of feature x, and $x'*{i}$ is the rescaled element. In our example, we can see from the outputted array that that the feature has been successful rescale to between 0 and 1:

```
array([[ 0. ], [ 0.28571429], [ 0.35714286], [ 0.42857143], [ 1. ]])
```

scikit-learn's `MinMaxScaler` offers two options to rescale a feature. One option is to use `fit` to calculate the minimum and maximum values of the feature, then use `transform` to rescale the feature. The second option is to use `fit_transform` to do both operations at once. There is no mathematical difference between the two options, however there is sometimes a practical benefit to keeping the operations separate because it allows us to apply the same transformation to difference **sets** of the data (sets are discussed more in Chapter 20: Model Evaluation).

3.2 Standardizing A Feature

Problem

You want to transform a feature to have a mean of zero and a standard deviation of one.

Solution

scikit-learn's `StandardScaler` performs both transformations:

```
# Load libraries
from sklearn import preprocessing
import numpy as np

# Create feature
x = np.array([[-1000.1],
              [-200.2],
              [500.5],
              [600.6],
              [9000.9]])

# Create scaler
scaler = preprocessing.StandardScaler()

# Transform the feature
standardized = scaler.fit_transform(x)

# Show feature
standardized

array([[-0.76058269],
       [-0.54177196],
       [-0.35009716],
       [-0.32271504],
       [ 1.97516685]])
```

Discussion

A common alternative to min-max scaling discussed in the previous recipe is rescaling of features to be approximately standard normally distributed. To achieve this, we use standardization to transform the data such that it has a mean, $\bar{x}$, of 0 and a standard deviation, σ of 1. Specifically, each element in the feature is transformed so that:

where $x'*{i}$ is our standardized form of $x*{i}$. The transformed feature represents the number of standard deviations the original value is away from the feature's mean value (also called a z-score in statistics).

Standardization is a common "go-to" scaling method for machine learning preprocessing and in my experience is used more than min-max scaling. However, it depends on the learning algorithm. For example, principal component analysis often works better using standardization while min-max scaling is often recommended for neural networks (both algorithms are discussed later in this book). As a general rule, I'd recommend defaulting to standardization unless you have a specific reason to use an alternative.

We can see the effect of standardization my looking at the mean and standard deviation of our solution's output:

```
# Print mean and standard deviation
print('Mean:', round(standardized.mean()))
print('Standard deviation:', standardized.std())
```

```
Mean: 0.0
Standard deviation: 1.0
```

If our data has significant outliers, it can negatively impact our standardization by affecting the feature's mean and variance. In this scenario, it is often helpful to instead rescale the feature using the median and quartile range. In scikit-learn, this is done using the `RobustScaler` method:

```
# Create scaler
robust_scaler = preprocessing.RobustScaler()

# Transform feature
robust_scaler.fit_transform(x)
```

```
array([[ -1.87387612],
       [ -0.875     ],
       [  0.        ],
       [  0.125     ],
       [ 10.61488511]])
```

3.3 Normalizing Observations

Problem

You want to rescale the feature values of observations to have unit norm (a total length of one).

Solution

Use `Normalizer` with a `norm` argument:

```
# Load libraries
from sklearn.preprocessing import Normalizer
import numpy as np

# Create feature matrix
X = np.array([[0.5, 0.5],
              [1.1, 3.4],
              [1.5, 20.2],
              [1.63, 34.4],
              [10.9, 3.3]])

# Create normalizer
normalizer = Normalizer(norm='l2')

# Transform feature matrix
normalizer.transform(X)

array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Discussion

Many rescaling methods (e.g. min-max scaling and standardization) operate on features, however we can also rescale across individual observations. `Normalizer` rescales the values on individual observations to have unit norm (the sum of their lengths is one). This type of rescaling is often used when we have many equivalent features (e.g. text classification when every word or n-word group is a feature).

`Normalizer` provides three norm options with Euclidean norm (often called "L2") being the default argument:

where x is an individual observation and x_{n} is that observation's value for the nth feature.

```
# Transform feature matrix
X_l2 = Normalizer(norm='l2').transform(X)

# Show feature matrix
X_l2
```

```
array([[ 0.70710678,  0.70710678],
       [ 0.30782029,  0.95144452],
       [ 0.07405353,  0.99725427],
       [ 0.04733062,  0.99887928],
       [ 0.95709822,  0.28976368]])
```

Alternatively we can specify Manhattan norm (L1):

```
# Transform feature matrix
X_l1 = Normalizer(norm='l1').transform(X)

# Show feature matrix
X_l1
```

```
array([[ 0.5       ,  0.5       ],
       [ 0.24444444,  0.75555556],
       [ 0.06912442,  0.93087558],
       [ 0.04524008,  0.95475992],
       [ 0.76760563,  0.23239437]])
```

Intuitively, L2 norm can be thought of as the distance between two points in New York for a bird (i.e. a straight line), while L1 can be thought of as the distance for a human walking on the street (e.g. walk north one block, east one block, north one block, east one block, etc.) which is why it is called "Manhattan norm" and is often called "Taxicab norm".

Practically, notice that `norm='l1'` rescales an observation's values so they sum to one which can sometimes be a desirable quality:

```
# Print sum
print('Sum of the first observation\'s values:', X_l1[0, 0] + X_l1[0, 1])
```

```
Sum of the first observation's values: 1.0
```

3.4 Generating Polynomial And Interaction Features

Problem

You want to create polynominal and interaction features.

Solution

Even though some choose to create polynomial and interaction features manually, scikit-learn offers a built-in method:

```
from sklearn.preprocessing import PolynomialFeatures
import numpy as np

X = np.array([[2, 3],
              [2, 3],
              [2, 3]])

polynomial_interaction = PolynomialFeatures(degree=2, include_bias=False)
polynomial_interaction.fit_transform(X)
```

```
array([[ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.],
       [ 2.,  3.,  4.,  6.,  9.]])
```

The `degree` parameter determines the maximum degree of the polynomial. For example, `degree=2` will create new features raised to the second power:

while `degree=3` new features raised to the second and third power:

Furthermore, by default `PolynomialFeatures` includes interaction features:

We can restrict the features created to only interaction features by setting `interaction_only` to `True`:

```
interaction = PolynomialFeatures(degree=2, interaction_only=True, include_bias=False)
interaction.fit_transform(X)
```

```
array([[ 2.,  3.,  6.],
       [ 2.,  3.,  6.],
       [ 2.,  3.,  6.]])
```

Discussion

Polynomial features are often created when we want to include the notion that there exists a nonlinear relationship between the features and the target. For example, we might suspect that the effect of age on the probability of having a major medical condition is not constant over time but increases as age increases. We can encode that non-constant effect in a feature, x, by generating that feature's higher-order forms (e.g. x^2, x^3 etc.).

Additionally, often we run into situations where the effect of one feature is dependent on another feature. A simple example would be if we were trying to predict whether or not our coffee was sweet and we had two features: 1) whether or not the coffee was stirred and 2) if we added sugar. Individually, each feature does not predict coffee sweetness, however combination of their effects does. That is, a coffee would only be sweet if the coffee had sugar and was stirred. The effects of each feature on the target (sweetness) are dependent on each other. We can encode that relationship by including an interaction feature which is the product of the individual features.

3.5 Transforming Features

Problem

You want to make a custom transformation to one or more features.

Solution

In scikit-learn, use `FunctionTransformer` to apply a function to a set of features:

```
# Load libraries
import numpy as np
from sklearn.preprocessing import FunctionTransformer

# Create feature matrix
X = np.array([[2, 3],
              [2, 3],
              [2, 3]])

# Define a simple function
def add_ten(x):
    return x + 10

# Create transformer
ten_transformer = FunctionTransformer(add_ten)

# Transform feature matrix
ten_transformer.transform(X)

array([[12, 13],
       [12, 13],
       [12, 13]])
```

We can create the same transformation in pandas using `apply`:

```
# Load library
import pandas as pd

# Create DataFrame
df = pd.DataFrame(X, columns=['feature_1', 'feature_2'])

# Apply function
df.apply(add_ten)
```

	feature_1	feature_2
0	12	13
1	12	13
2	12	13

Discussion

It is common to want to make some custom transformations to one or more features. For example, we might want to create a feature that is the natural log of the values of the different feature. We can do this my creating a feature and then mapping it to features using either scikit-learn's `FunctionTransformer` or pandas's `apply`. In the solution we created a very simple function, `add_ten`, which added ten to each input, however there is no reason we could not define a much more complex function.

3.6 Detecting Outliers

Problem

You want to identify extreme observations.

Solution

Detecting outliers is unfortunately more of an art than a science however a common method is to assume the data is normally distributed and based on that assumption "draw" an ellipse around the data, classifying any observation inside the ellipse as an inlier (labeled as `1`) and any observation outside the ellipse as an outlier (labeled as `-1`):

```
# Load libraries
import numpy as np
from sklearn.covariance import EllipticEnvelope
from sklearn.datasets import make_blobs

# Create simulated data
X, _ = make_blobs(n_samples = 10,
                  n_features = 2,
                  centers = 1,
                  random_state = 1)

# Replace the first observation's values with extreme values
X[0,0] = 10000
X[0,1] = 10000

# Create detector
outlier_detector = EllipticEnvelope(contamination=.1)

# Fit detector
outlier_detector.fit(X)

# Predict outliers
outlier_detector.predict(X)
```

```
array([-1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

A major limitation of this approach is the need to specify a `contamination` parameter which is the proportion of observations that are outliers, a value that we don't know. Think of `contamination` as our estimate of the cleanliness of our data. If we expect our data to have few outliers, we can set `contamination` to something small. However, if we believe that the data is very likely to have outliers, we can set it to a higher value.

Instead of looking at observations as a whole, we can instead took at individual features and identify extreme values in those features using interquartile range (IQR):

```
# Create one feature
feature = X[:,0]

# Create a function to return index of outliers
def indicies_of_outliers(x):
    q1, q3 = np.percentile(x, [25, 75])
    iqr = q3 - q1
    lower_bound = q1 - (iqr * 1.5)
    upper_bound = q3 + (iqr * 1.5)
    return np.where((x > upper_bound) | (x < lower_bound))

# Run function
indicies_of_outliers(feature)
```

```
(array([0]),)
```

IQR is the difference between the first and third quartile of a set of data. You can think of IQR as the spread of the bulk of the data. Outliers are commonly defined as any value 1.5 IQRs less than than the first quartile or 1.5 IQRs greater than the third quartile.

Discussion

There is no single best technique when detecting outliers. Instead we have a collection of techniques all with their own advantages and disadvantages. Our best strategy is often trying multiple techniques (e.g. both `EllipticEnvelope` and IQR-based detection) and looking at the results as a whole.

If at all possible, we should take a look at observations we detect as outliers and try to understand them. For example, if we have a dataset of houses and one feature is number of rooms: is an outlier with 100 rooms really a house or is it actually a hotel that has been misclassified?

See Also

- Three ways to detect outliers (and the source of the IQR function used in this recipe)

3.7 Handling Outliers

Problem

You have outliers.

Solution

Typically we have three strategies we can use to handle outliers. First, we can drop them:

```
# Load library
import pandas as pd

# Create DataFrame
houses = pd.DataFrame()
houses['Price'] = [534433, 392333, 293222, 4322032]
houses['Bathrooms'] = [2, 3.5, 2, 116]
houses['Square_Feet'] = [1500, 2500, 1500, 48000]

# Filter observations
houses[houses['Bathrooms'] < 20]
```

	Price	Bathrooms	Square_Feet
0	534433	2.0	1500
1	392333	3.5	2500
2	293222	2.0	1500

Second, we can mark them as outliers and include it as a feature:

```
# Load library
import numpy as np

# Create feature based on boolean condition
houses['Outlier'] = np.where(houses['Bathrooms'] < 20, 0, 1)

# Show data
houses
```

	Price	Bathrooms	Square_Feet	Outlier
0	534433	2.0	1500	0
1	392333	3.5	2500	0
2	293222	2.0	1500	0
3	4322032	116.0	48000	1

Finally, we can transform the feature to dampen the effect of the outlier:

```
# Log feature
houses['Log_Of_Square_Feet'] = [np.log(x) for x in houses['Square_Feet']]

# Show data
houses
```

	Price	Bathrooms	Square_Feet	Outlier	Log_Of_Square_Feet
0	534433	2.0	1500	0	7.313220
1	392333	3.5	2500	0	7.824046
2	293222	2.0	1500	0	7.313220
3	4322032	116.0	48000	1	10.778956

Discussion

Similar to detecting outliers, there is no hard and fast rule for handling them. How we handle them should be based on two aspects. First, we should consider what makes them an outlier. If we believe they are errors in the data such as from a broken sensor or a miscoded value, then we might drop the observation or replace outlier values with `NaN` since we can't believe those values. However, if we believe the outliers are genuine extreme values (e.g. a mansion with a lot of bedrooms) then marking them as outliers or transforming their values is more appropriate.

Second, how we handle outliers should be based on our goal for machine learning. For example, if we want to predict house prices based on features of the house, we might reasonably assume the price for mansions with over 100 rooms is driven by a different dynamic than regular family homes. Furthermore, if we are training a model to use as part of an online home loan web application, we might assume that our potential users will not include billionaires looking to buy a mansion.

So what should we do if we have outliers? Think about why are they are outliers, have an end goal in mind for the data, and most importantly remember that not making a decision to address outliers is itself a decision with implication.

One additional point, if you do have outliers standardization might not be appropriate because the mean and variance might be highly influenced by the outliers. In this case, use a rescaling method more robust against outliers like `RobustScaler`.

See Also

- `RobustScaler` documentation

3.8 Discretizating Features

Problem

You have a numerical feature and want to break it up into discrete bins.

Solution

Depending on how we want to break up the data, there are three techniques we can use. First, we can binarize the feature according to some threshold:

```
# Load libraries
from sklearn.preprocessing import Binarizer
import numpy as np

# Create feature
age = np.array([[6],
                [12],
                [20],
                [36],
                [65]])

# Create binarizer
binarizer = Binarizer(18)

# Transform feature
binarizer.fit_transform(age)

array([[0],
       [0],
       [1],
       [1],
       [1]])
```

Second, we can break up numerical features according to multiple thresholds:

```
# Bin feature
np.digitize(age, bins=[20,30,64])

array([[0],
       [0],
       [1],
       [2],
       [3]])
```

Note that the arguments for the `bins` parameter denote the left edge of each bin. For example, the `20` argument does not include the element with the value of 20, only the two values smaller than 20. We can switch this behavior by setting the parameter `right` to `True`:

```
# Bin feature
np.digitize(age, bins=[20,30,64], right=True)

array([[0],
       [0],
       [0],
       [2],
       [3]])
```

Discussion

Discretization can be a fruitful strategy when we have reason to believe that a numerical feature should behave more like a categorical feature. For example, we might believe there is very little difference in the spending habits of 19 and 20 year olds, but a significant difference between 20 and 21 years olds (the age in the United States when young adults can consume alcohol). In that example, it could useful to break up individuals in our data into those who can drink alcohol and those who cannot. Similarly in other cases it might be useful to discretize our data into three or more bins.

In the solution, while we saw two methods of discretization, scikit-learn's `Binarizer` for two bins and and NumPy's `digitize` for three or more bins, however `digitize` be used to binarize features like `Binarizer` by only specifying a single threshold:

```
# Bin feature
np.digitize(age, bins=[18])
```

```
array([[0],
       [0],
       [1],
       [1],
       [1]])
```

See Also

- `digitize` documentation

3.9 Grouping Observations Using Clustering

Problem

You want to cluster observations so that similar observations are grouped together.

Solution

If you know that you have k group, you can use k-means clustering to group similar observations and output a new feature containing each observation's group membership:

```
# Load libraries
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
import pandas as pd

# Make simulated feature matrix
X, _ = make_blobs(n_samples = 50,
                  n_features = 2,
                  centers = 3,
                  random_state = 1)

# Create DataFrame
df = pd.DataFrame(X, columns=['feature_1','feature_2'])

# Make k-means clusterer
clusterer = KMeans(3, random_state=1)

# Fit clusterer
clusterer.fit(X)

# Predict values
df['group'] = clusterer.predict(X)

# First few observations
df.head(5)
```

	feature_1	feature_2	group
0	-9.877554	-3.336145	0
1	-7.287210	-8.353986	2
2	-6.943061	-7.023744	2
3	-7.440167	-8.791959	2
4	-6.641388	-8.075888	2

Discussion

We are jumping ahead of ourselves a bit and will go much more in depth about clustering algorithms later in the book. However, I wanted to point out that we can use clustering as a preprocessing step. Specifically, we use unsupervised learning algorithms like k-means to cluster observations into groups. The end result is a categorical feature with similar observations being members of the same group.

Don’t worry if you did not understand all of that right now, just file away the idea that clustering can be used in preprocessing for now. And if you really can’t wait, feel free to flip to Chapter 18 now.

3.10 Deleting Observations With Missing Values

Problem

You need to delete observations containing missing values.

Solution

Deleting observations with missing values is easy with a clever line of NumPy:

```
# Load library
import numpy as np

# Create feature matrix
X = np.array([[1.1, 11.1],
              [2.2, 22.2],
              [3.3, 33.3],
              [4.4, 44.4],
              [np.nan, 55]])

# Remove observations with missing values
X[~np.isnan(X).any(axis=1)]
```

```
array([[  1.1,  11.1],
       [  2.2,  22.2],
       [  3.3,  33.3],
       [  4.4,  44.4]])
```

Alternatively, we can drop missing observations using pandas:

```
# Load library
import pandas as pd

# Load data
df = pd.DataFrame(X, columns=['feature_1', 'feature_2'])

# Remove observations with missing values
df.dropna()
```

	feature_1	feature_2
0	1.1	11.1
1	2.2	22.2
2	3.3	33.3
3	4.4	44.4

Discussion

Most machine learning algorithms cannot handle any missing values in the target and feature matrices. For this reason, we cannot ignore missing values in our data and must address the issue during preprocessing.

The simplest solution is to delete every observation that contains one or more missing values, a task quickly and easily accomplished using NumPy or pandas.

That said, we should be very reluctant to delete observations with missing values. Deleting them is the nuclear option, since our algorithm loses access to the information contained in the observation's non-missing values.

Just as important, depending on the cause of the missing values, deleting observations can introduce bias into our data. There are three types of missing data:

- Missing Completely At Random (MCAR). The probability a value is missing is independent of everything. For example, a survey respondent rolls a dice before answering a question, if they roll a six they skip that question.
- Missing At Random (MAR). The probability a value is missing is not completely random, but depends on the information captured in other features. For example, a survey asks about gender identity and annual salary and women are more likely to skip the salary question, however their non-response depends only on information we have captured in our gender identity feature.
- Missing Not At Random (MNAR). The probability a value is missing is not random and depends on information not captured in our features. For example, a survey asks about gender identity and women are more likely to skip the salary question, and we do not have a gender identity feature in our data.

It is sometimes acceptable to delete observations if they are MCAR or MAR. However, if the value is MNAR then the fact a value is missing is itself information. Deleting MNAR observations can inject bias into our data because we are removing observations produced by some unobserved systematic effect.

See Also

- Identifying The Three Types Of Missing Data
- Missing-Data Imputation

3.11 Imputing Missing Values

Problem

You have missing values in your data and want to fill in or predict their values.

Solution

If you have a small amount of data, predict the missing values using k-nearest neighbors (KNN):

```
# Load libraries
import numpy as np
from fancyimpute import KNN
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_blobs

# Make a simulated feature matrix
X, _ = make_blobs(n_samples = 1000,
                  n_features = 2,
                  random_state = 1)

# Standardize the features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# Replace the first feature's first value with a missing value
true_value = X[0,0]
X[0,0] = np.nan

# Predict the missing values in the feature matrix
X_knn_imputed = KNN(k=5, verbose=0).complete(X)

# Compare true and imputed values
print('True Value:', true_value)
print('Imputed Value:', X_knn_imputed[0,0])
```

```
Using TensorFlow backend.


True Value: 0.8730186114
Imputed Value: 0.97152537543
```

Alternatively, we can use scikit-learn's `Imputer` module to fill in missing values with the feature's mean, median, or most frequent value. However, we will typically get worse results than KNN:

```
# Load library
from sklearn.preprocessing import Imputer

# Create imputer
mean_imputer = Imputer(strategy='mean', axis=0)

# Impute values
X_mean_imputed = mean_imputer.fit_transform(X)

# Compare true and imputed values
print('True Value:', true_value)
print('Imputed Value:', X_mean_imputed[0,0])
```

```
True Value: 0.8730186114
Imputed Value: -0.000873892503902
```

Discussion

There are two main strategies for replacing missing data with substitute values, each of which have strengths and weaknesses. First, we can use machine learning to predict the values of the missing data. To do this we treat the feature with missing values as a target vector and use the remaining subset of features to predict missing values. While we can use a wide range of machine learning algorithms to impute values, a popular choice is KNN. KNN is addressed in depth later in Chapter 14, however the short explanation is that the algorithm uses the k nearest observations (according to some distance metric) to predict the missing value. In our solution we predicted the missing value using the five closest observations.

The downside to KNN is that in order to know which observations are the closest to the missing value, it needs to calculate the distance between the missing value and every single observation. This is reasonable in smaller datasets, but quickly becomes problematic if a dataset has millions are observations.

An alternative and more scalable strategy is to fill in all missing values with some average value. For example, in our solution we used scikit-learn to fill in missing values with feature's mean value. The imputed value is often not as close to the true value as when we used KNN, however we can scale mean-filling to data containing millions of observations easily.

If we use imputation, it is a good idea to create a binary feature indicating if the observation contains an imputed value or not.

See Also

- A Study of K-Nearest Neighbour As An Imputation Method