Data Scientist Path - Python

Dataquest

By

Daniel Ruiz

December 2019

Prelude

This set of notes tracks my journey through DataQuests' *Data Scientist Path* module with Python. These notes are one of many that I have decided to polish and make available for anyone interested. One of the benefits of this series are having a compendium of definitions, examples and personal thoughts that I can always refer back to if I need a reminder on a particular topic. In addition, I find that this medium reduces the search time for specific definitions, theorems, examples etc and thus aids in reinforcing my own knowledge when frequented. The act of writing also works as a ritual in helping to encode information within my brain.

In these notes, I have provided many definitions and examples that I have encountered through the *Data Scientist* module on Dataquest. We will now outline their brief summaries:

- §1: **Data Analysis and Visualization** goes over the fundamentals of the Python Pandas and NumPy modules. I also provide details on Data Exploration and Visualization and how to do this in Python.

Contents

1 Data Analysis and Visualization

1.1 Pandas and NumPy Fundamentals

1.1.1 Introduction to NumPy

- CSV: Open, Read etc
- NdArray.shape
- Selecting rows/columns from an array
- NdArray Arithmetic: Addition, Subtraction, Multiplication, Division
- NdArray Methods: Mean, min, max, sum
- Overlap with Numpy Functions and Methods

Definition 1.1: NumPy Library

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- *A powerful N-dimensional array object.*
- *Sophisticated (broadcasting) functions.*
- *Tools for integrating C/C++ and Fortran code.*
- *Useful linear algebra, Fourier transform, and random number capabilities*

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Definition 1.2: NumPy NdArray

class numpy.ndarray(shape, dtype=float, buffer=None, offset=0, strides=None, order=None)

An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

Arrays should be constructed using array, zeros or empty (refer to the See Also section below). The parameters given here refer to a low-level method (ndarray(...)) for instantiating an array.

PARAMETERS

- ***shape*** *: tuple of ints*
  Shape of created array.
- ***dtype*** *: data-type, optional*
  Any object that can be interpreted as a numpy data type.
- ***buffer****: object exposing buffer interface, optional*
  Used to fill the array with data.
- ***offset****: int, optional*
  Offset of array data in buffer.
- ***strides*** *: tuple of ints, optional*
  Strides of data in memory.

- ***order***: {'C', 'F'}, *optional*
  Row-major (C-style) or column-major (Fortran-style) order.

Definition 1.3: SIMD

Single instruction, multiple data *(SIMD) is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Such machines exploit data level parallelism, but not concurrency: there are simultaneous (parallel) computations, but only a single process (instruction) at a given moment.*

Definition 1.4: Slice Object

class slice(start, stop, step)

Return a slice object representing the set of indices specified by range(start, stop, step). The start and step arguments default to None. Slice objects have read-only data attributes start, stop and step which merely return the argument values (or their default).

Ndarray Indexing

Ndarrays can be indexed using the standard Python x[obj] syntax, where x is the array and obj the selection. There are three kinds of indexing available: field access, basic slicing, advanced indexing. Which one occurs depends on obj.

Basic slicing extends Python's basic concept of slicing to N dimensions. Basic slicing occurs when obj is a slice object (constructed by start:stop:step notation inside of brackets), an integer, or a tuple of slice objects and integers. Ellipsis and newaxis objects can be interspersed with these as well.

1.1.2 Boolean Indexing with NumPy

- genfromtxt function
- NaN (Not a number)
- Boolean Arrays
- Boolean Indexing
- Assigning values to NdArrays with Indexing

Definition 1.5: NumPy genfromtxt()

numpy.genfromtxt(fname, dtype=¡class 'float'¿, comments='#', delimiter=None, skip_header=0)

Load data from a text file, with missing values handled as specified.
Each line past the first skip_header lines is split at the delimiter character, and characters following the comments character are discarded. For further parameters and documentation, see [1].

PARAMETERS

- ***fname***: *file, str, pathlib.Path, list of str, generator.*
  File, filename, list, or generator to read. If the filename extension is gz or bz2, the file is first decompressed. Note that generators must return byte strings in Python 3k. The strings in a list or produced by a generator are treated as lines.

- ***dtype***: *dtype, optional.*
  Data type of the resulting array. If None, the dtypes will be determined by the contents of each column, individually.
- ***comments***: *str, optional.*
  The character used to indicate the start of a comment. All the characters occurring on a line after a comment are discarded.
- ***delimiter***: *str, int, or sequence, optional.*
  The string used to separate values. By default, any consecutive whitespaces act as delimiter. An integer or sequence of integers can also be provided as width(s) of each field.
- ***skip_header*** *: int, optional.*
  The number of lines to skip at the beginning of the file.

Definition 1.6: NumPy nan

numpy.nan

Python constant. IEEE 754 floating point representation of Not a Number (NaN).
NaN and NAN are equivalent definitions of nan. Please use nan instead of NAN.

1.1.3 Introduction to Pandas

- Dataframe Methods: Head, Tail, Info, Loc
- Series object
- Selecting Columns/Rows in a Dataframe

Definition 1.7: Pandas Library

Pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Some key features of the library include:

- *A fast and efficient DataFrame object for data manipulation with integrated indexing.*
- *Tools for reading and writing data between in-memory data structures and different formats: CSV and text files, Microsoft Excel, SQL databases, and the fast HDF5 format.*
- *Intelligent data alignment and integrated handling of missing data: gain automatic label-based alignment in computations and easily manipulate messy data into an orderly form.*

Definition 1.8: Dataframe

class pandas.DataFrame(data=None, index=None, columns=None, dtype=None, copy=False

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure. For a list of methods and further documentation, see [2].

PARAMETERS

- ***data***: *ndarray (structured or homogeneous), Iterable, dict, or DataFrame.*

Dict can contain Series, arrays, constants, or list-like objects.

- ***index***: *Index or array-like.*
  Index to use for resulting frame. Will default to RangeIndex if no indexing information part of input data and no index provided.
- ***columns***: *Index or array-like.*
  Column labels to use for resulting frame. Will default to RangeIndex (0, 1, 2,.., n) if no column labels are provided.
- ***dtype***: *dtype, default None.*
  Data type to force. Only a single dtype is allowed. If None, infer.
- ***copy***: *boolean, default False.*
  Copy data from inputs. Only affects DataFrame / 2d ndarray input.

Example 1.1: Pandas Dataframe

In vanilla numpy, our arrays must house elements of the same data type. Pandas extends this by allowing for multiple data types.

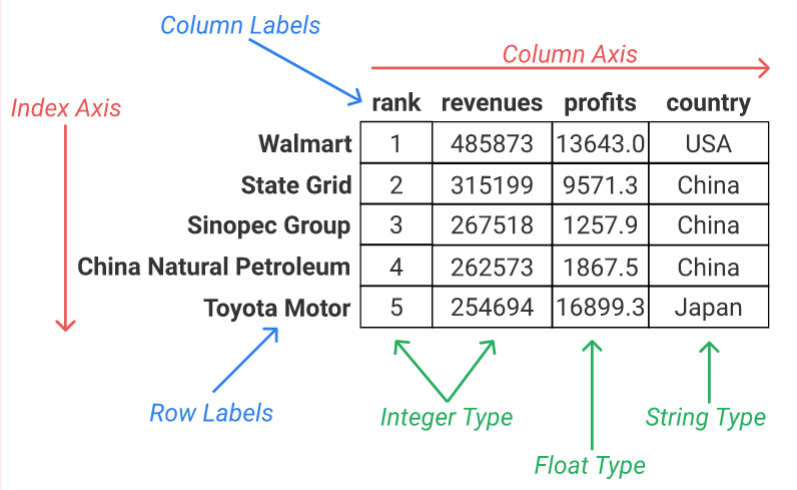

Figure 1.1: Pandas Dataframe

Definition 1.9: File-like Object (Pandas)

By file-like object, we refer to objects with a read() method, such as a file handler (e.g. via builtin open function) or StringIO.

Definition 1.10: Pandas read_csv()

pandas.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None)

Read a comma-separated values (csv) file into DataFrame. Also supports optionally iterating or breaking of the file into chunks. There are much more parameters available for this function, see documentation [3].

PARAMETERS

- ***filepath_or_buffer*** *: str, path object or file-like object.*
  Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv. If you want to pass in a path object, pandas accepts any os.PathLike.
- ***sep****: str, default ‘,’.*
  Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python’s builtin sniffer tool, csv.Sniffer.
- ***delimiter****: str, default None.*
  Alias for sep.
- ***header****: int, list of int, default ‘infer’.*
  Row number(s) to use as the column names, and the start of the data.
- ***names****: array-like, optional.*
  List of column names to use. If file contains no header row, then you should explicitly pass header=None. Duplicates in this list are not allowed.
- ***index_col****: int, str, sequence of int / str, or False, default None.*
  Column(s) to use as the row labels of the DataFrame, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

Definition 1.11: Series

class *pandas.Series(data=None, index=None, dtype=None, name=None, copy=False, fastpath=False)}*

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index. Statistical methods from ndarray have been overridden to automatically exclude missing data (currently represented as NaN). For further methods and documentation, see [4].

*Operations between Series (+, -, /, *) align values based on their associated index values– they need not be the same length. The result index will be the sorted union of the two indexes.*

PARAMETERS

- ***data:*** *array-like, Iterable, dict, or scalar value.*
  Contains data stored in Series.
  Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.
- ***index****: array-like or Index (1d). Values must be hashable and have the same length as data. Non-unique index values are allowed. Will default to RangeIndex (0, 1, 2, .., n) if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.*
- ***dtype****: str, numpy.dtype, or ExtensionDtype, optional.*
  Data type for the output Series. If not specified, this will be inferred from data. See the user guide for more usages.

- ***copy***: *bool, default False.*
  Copy input data.

Example 1.2: Dataframe Indexing

Let df denote a Dataframe. Then, we can select some subset of the dataframe's columns via the following syntax:

Select by Label	Explicit Syntax	Common Shorthand
Single column	`df.loc[:,"col1"]`	`df["col1"]`
List of columns	`df.loc[:,["col1", "col7"]]`	`df[["col1", "col7"]]`
Slice of columns	`df.loc[:,"col1":"col4"]`	

Figure 1.2: Indexing a Dataframe's Columns

Example 1.3: Series Indexing

Let s denote a Series. Then, we can select some subset of the series rows via the following syntax:

Select by Label	Explicit Syntax	Shorthand Convention
Single item from series	`s.loc["item8"]`	`s["item8"]`
List of items from series	`s.loc[["item1","item7"]]`	`s[["item1","item7"]]`
Slice of items from series	`s.loc["item2":"item4"]`	`s["item2":"item4"]`

Figure 1.3: Series Item(s) Selection

Select by Label	Explicit Syntax	Shorthand Convention
Single column from dataframe	`df.loc[:,"col1"]`	`df["col1"]`
List of columns from dataframe	`df.loc[:,["col1","col7"]]`	`df[["col1","col7"]]`
Slice of columns from dataframe	`df.loc[:,"col1":"col4"]`	
Single row from dataframe	`df.loc["row4"]`	
List of rows from dataframe	`df.loc[["row1", "row8"]]`	
Slice of rows from dataframe	`df.loc["row3":"row5"]`	`df["row3":"row5"]`
Single item from series	`s.loc["item8"]`	`s["item8"]`
List of items from series	`s.loc[["item1","item7"]]`	`s[["item1","item7"]]`
Slice of items from series	`s.loc["item2":"item4"]`	`s["item2":"item4"]`

Figure 1.4: Summary of Dataframe / Series Item Selection

1.1.4 Exploring Data with Pandas: Fundamentals

- Dataframe Assignment
- Boolean Indexing with Dataframes
- New Columns in Dataframes

Pandas Arithmetic with Series

Because pandas is designed to operate like NumPy, a lot of concepts and methods from Numpy are supported. Recall that one of the ways NumPy makes working with data easier is with vectorized operations, or operations applied to multiple data points at once.

Just like with NumPy, we can use any of the standard Python numeric operators with series, including:

- *series_a + series_b: Addition*
- *series_a - series_b: Subtraction*
- *series_a * series_b: Multiplication (Element-wise)*
- *series_a / series_b: Division (Element-wise)*

Definition 1.12: Method Chaining

Method chaining, also known as named parameter idiom, is a common syntax for invoking multiple method calls in object-oriented programming languages. Each method returns an object, allowing the calls to be chained together in a single statement without requiring variables to store the intermediate results.

Useful Dataframe Methods

- ***df.describe()**: Generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.*
- ***df.mean()**: Return the mean of the values for the requested axis.*
- ***df.rename()**: Alter axes labels. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.*

1.1.5 Exploring Data with Pandas: Intermediate

- Dataframe iloc indexing
- Boolean Operators; Combining Boolean Arrays
- Dataframe sort_values method

Select by integer position	Explicit Syntax	Shorthand Convention
Single column from dataframe	`df.iloc[:,3]`	
List of columns from dataframe	`df.iloc[:,[3,5,6]]`	
Slice of columns from dataframe	`df.iloc[:,3:7]`	
Single row from dataframe	`df.iloc[20]`	
List of rows from dataframe	`df.iloc[[0,3,8]]`	
Slice of rows from dataframe	`df.iloc[3:5]`	`df[3:5]`
Single items from series	`s.iloc[8]`	`s[8]`
List of item from series	`s.iloc[[2,8,1]]`	`s[[2,8,1]]`
Slice of items from series	`s.iloc[5:10]`	`s[5:10]`

Figure 1.5: Dataframe Integer Indexing

pandas	Python equivalent	Meaning
`a & b`	`a and b`	`True` if both `a` and `b` are `True`, else `False`
`a | b`	`a or b`	`True` if either `a` or `b` is `True`
`~a`	`not a`	`True` if `a` is `False`, else `False`

Figure 1.6: Boolean Operators in Pandas

1.1.6 Data Cleaning Basics

- Data Encoding
- Dataframe Column Cleaning
- Dataframe Text to Numeric Cleaning
- Series.astype() method

Example 1.4: Cleaning Column Names

The column labels have a variety of upper and lowercase letters, as well as parentheses, which will make them harder to work with and read. We aim to clean our column labels by using several python string methods that will:

- *Replace spaces with underscores.*
- *Remove special characters.*
- *Make all labels lowercase.*
- *Shorten any long column names. (Specifically changing 'Operating System' to 'os')*

```
# CB 1.1.1 #

import pandas as pd
laptops = pd.read_csv('laptops.csv', encoding='Latin-1')
```

```
def mr_clean(s):
    s = s.strip()
    s = s.replace("Operating System", "os")
    s = s.replace(" ", "_")
    s = s.replace(")", "")
    s = s.replace("(","")
    sfinal = s.lower()

    return sfinal

new_columns = []
for col in laptops.columns:
    new_columns.append(mr_clean(col))

laptops.columns = new_columns
```

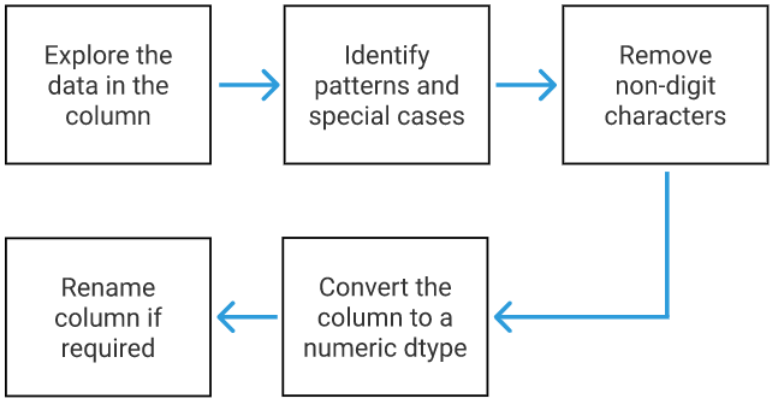

Figure 1.7: Text to Numeric Data Workflow

Example 1.5: Cleaning Text into Numerical Data

We now consider the column for ram data. Exploring this column, we find a clear pattern that all stored values are strings of integers with the character 'GB' at the end of the string:

['8GB', '16GB', '4GB', '8GB', '32GB', '4GB', '8GB', '2GB']

Naturally, we want to modify the data type stored in this column so that we can easily manipulate it numerically (such as computing statistics). To do this, we'll remove the GB from the string and then recast the remaining integer-string to an integer.

```
# CB 1.1.2 #

laptops["ram"] = laptops["ram"].str.replace('GB','').astype(int)
laptops.rename({"ram":"ram_gb"},axis=1, inplace = True)
ram_gb_desc = laptops['ram_gb'].describe()
```

Example 1.6: Extracting Data from Text

```
print(laptops["cpu"].head())
```

```
0          Intel Core i5 2.3GHz
1          Intel Core i5 1.8GHz
2    Intel Core i5 7200U 2.5GHz
3          Intel Core i7 2.7GHz
4          Intel Core i5 3.1GHz
Name: cpu, dtype: object
```

Figure 1.8: First 5 entries of the Laptops['cpu'] column

We can observe that the CPU column in Figure 1.8 contains several pieces of information. They all appear to express the manufacturer and then model. Naturally, we want to break this apart so that we can organize it according to manufacturer and model. We use the fact spaces separate key pieces of information, allowing us to use the split() method to construct a list of strings. Selecting the first element out of this list would give us the manufacturer. Constructing a new cpu_manufacturer column is therefore achieved with the following code:

```
# CS 1.1.3 #

laptops["cpu_manufacturer"] = (laptops["cpu"]
                               .str.split()
                               .str[0]
                              )
```

The parenthesis are included so that one line of code can be expressed on multiple lines. This is done for readability purposes.

Definition 1.13: Pandas Series.map()

Series.map(self, arg, na_action=None)

Map values of Series according to input correspondence. Used for substituting each value in a Series with another value, that may be derived from a function, a dict or a Series. When arg is a dictionary, values in Series that are not in the dictionary (as keys) are converted to NaN. However, if the dictionary is a dict subclass that defines __missing__ (i.e. provides a method for default values), then this default is used rather than NaN. For further documentation, see [5].

PARAMETERS

- ***arg**: function, dict, or Series.*
  Mapping correspondence.
- ***na_action**: None, 'ignore', default None.*
  If 'ignore', propagate NaN values, without passing them to the mapping correspondence.

Example 1.7: Who Needs A Map?

Let s be a series containing fruit names. We note that s currently contains incorrectly spelt fruit names. Printing the series gives us a display of that incorrect spelling.

```
# CS 1.1.4 #

print(s)

Output:
0        pair
1      oranje
2     bannana
3      oranje
4      oranje
5      oranje
dtype: object
```

Hence, to fix the problem seen in CS 1.1.4, we define a dictionary called corrections that will be fed into the argument of the map() method. For each string that appears in the dictionary argument, it will be mapped to the corresponding value of that dictionary argument. For instance, pair $\mapsto$ pear.

```
# CS 1.1.5 #

corrections = {
"pair": "pear",
"oranje": "orange",
"bananna": "banana"
}
s = s.map(corrections)
print(s)

Output:
0        pear
1      orange
2      banana
3      orange
4      orange
5      orange
dtype: object
```

Handling Null Values

In pandas, null values will be indicated by either ***NaN*** *or* ***None****. There are a few main options for handling missing values:*

- *Remove any rows that have missing values.*
- *Remove any columns that have missing values.*
- *Fill the missing values with some other values.*
- *Leave the missing values as is.*

The first two options are often used to prepare data for machine learning algorithms, which are unable to

be used with data that includes null values. We can use the DataFrame.dropna() method to remove or drop rows and columns with null values.

Example 1.8: Cleaning the Weight Column

In this example, the weights column is filled with strings of numeric values followed by a kg or kgs string that needs to be removed. After both these strings are removed, we recast the entries as floats.

```
# CS 1.1.6 #

laptops["weight"] = laptops["weight"].str.replace("kg","")
laptops["weight"] = laptops["weight"].str.replace("s","").astype(float)

laptops.rename({"weight":"weight_kg"}, axis=1, inplace = True)

laptops.to_csv('laptops_cleaned.csv', index=False)
```

A subtlety in the above approach is that once kg is removed, all entries that contained 'Xkgs' now contain 'Xs' (X denotes a numeric value), hence why I use the replace("s","") argument. In addition, I have to recast as float with the astype() method on the second line due to the weight column still containing 'Xs' entries after the first replace() call.

1.2 Exploratory Data Visualization

1.2.1 Line Charts

Definition 1.14: Data Visualization

Data visualization is the graphical representation of information and data. By using visual elements like charts, graphs, and maps, data visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data.

Definition 1.15: Matplotlib Library

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits. The library allows us to:

- *Quickly create common plots using high-level functions.*
- *Extensively tweak plots.*
- *Create new kinds of plots from the ground up.*

Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, error charts, scatter plots, etc., with just a few lines of code.

When working with commonly used plots in matplotlib, the general workflow is:

- Create a plot using data.
- Customize the appearance of the plot.

- Display the plot.
- Edit and repeat until satisfied.

Definition 1.16: Pyplot Module

matplotlib.pyplot

Provides a MATLAB-like plotting framework.

Definition 1.17: Plot() Function

*matplotlib.pyplot.plot(*args, scalex=True, scaley=True, data=None, **kwargs)*

Plot y versus x as lines and/or markers.

PARAMETERS:

- ***x, y**: array-like or scalar.*
  The horizontal / vertical coordinates of the data points. x values are optional and default to range(len(y)).
  Commonly, these parameters are 1D arrays. They can also be scalars, or two-dimensional (in that case, the columns represent separate data sets).
- ***fmt:** str, optional.*
  A format string, e.g. 'ro' for red circles. See the Notes section for a full description of the format strings. Format strings are just an abbreviation for quickly setting basic line properties. All of these and more can also be controlled by keyword arguments.
- ***data**: indexable object, optional.*
  An object with labelled data. If given, provide the label names to plot in x and y.

Returns:
lines - A list of Line2D objects representing the plotted data.

Example 1.9: Plotting Monthly Unemployment Trends in 1948

```
# CB 1.2.1 #

import matplotlib.pylot as plt
import pandas as pd

unrate = pd.read_csv('unrate.csv')

plt.plot(unrate['DATE'].head(12), unrate['VALUE'].head(12))
plt.xticks(rotation = 90)
plt.xlabel("Month")
plt.ylabel("Unemployment Rate")
plt.title("Monthly Unemployment Trends, 1948")
plt.show()
```

Output:

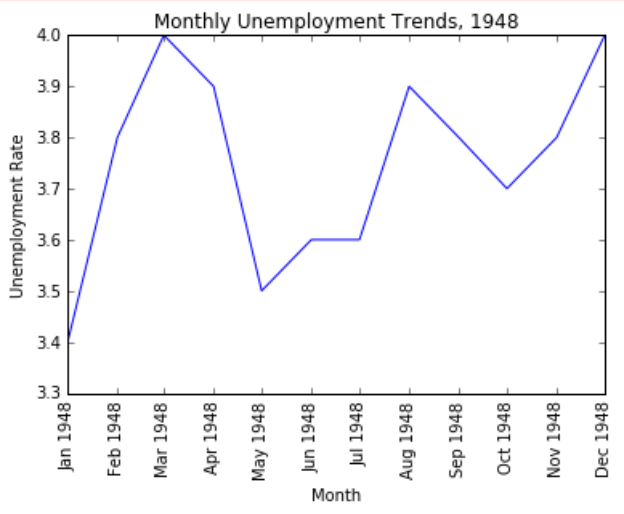

Figure 1.9: Monthly Unemployment Trends, 1948

One can notice that we didn't assign the plot in Example 1.9 to a variable and then call a method on the variable to display it. We instead called the functions plot(), show() on the pyplot module directly.

This is because every time we call a pyplot function, the module maintains and updates the plot internally (also known as state). When we call show(), the plot is displayed and the internal state is destroyed. While this workflow isn't ideal when we're writing functions that create plots on a repeated basis as part of a larger application, it's useful when exploring data.

1.2.2 Multiple Plots

When we want to work with multiple plots, however, we need to be more explicit about which plot we're making changes to. This means we need to understand the matplotlib classes that pyplot uses internally to maintain state so we can interact with them directly. Let's first start by understanding what pyplot was automatically storing under the hood when we create a single plot:

- A container for all plots was created (returned as a Figure object)
- A container for the plot was positioned on a grid (the plot returned as an Axes object)
- Visual symbols were added to the plot (using the Axes methods)

Definition 1.18: Pyplot Figure Function

*matplotlib.pyplot.figure(num=None, figsize=None, dpi=None, facecolor=None, frameon=True, clear=False, **kwargs)*

Create a new figure. For further documentation, see [6].

PARAMETERS:

- ***num**: integer or string, optional, default: None. If not provided, a new figure will be created, and the figure number will be incremented. The figure objects holds this number in a number attribute. If num is provided, and a figure with this id already exists, make it active, and returns a reference to it. If this figure does not exists, create it and returns it. If num is a string, the window title will be set to*

this figure's num.

- ***figsize***: *(float, float), optional, default: None. Width, height in inches. If not provided, defaults to rcParams["figure.figsize"] = [6.4, 4.8] = [6.4, 4.8].*
- ***dpi***: *integer, optional, default: None. Resolution of the figure. If not provided, defaults to rcParams["figure.dpi"] = 100.0 = 100.*
- ***facecolor***: *color spec. The background color. If not provided, defaults to rcParams["figure.facecolor"] = 'white' = 'w'.*
- ***frameon***: *bool, optional, default: True. If False, suppress drawing the figure frame.*
- ***clear***: *bool, optional, default: False If True and the figure already exists, then it is cleared.*

Definition 1.19: Matplotlib Axes Class

*class matplotlib.axes.Axes(fig, rect, facecolor=None, frameon=True, sharex=None, sharey=None, label='', xscale=None, yscale=None, **kwargs)*

The Axes contains most of the figure elements: Axis, Tick, Line2D, Text, Polygon, etc., and sets the coordinate system.
Build an axes in a figure. For further documentation, see [7].

PARAMETERS:

- ***fig***: *Figure. The axes is build in the Figure fig.*
- ***rect***: *[left, bottom, width, height] The axes is build in the rectangle rect. rect is in Figure coordinates.*
- ***sharex, sharey***: *Axes, optional. The x or y axis is shared with the x or y axis in the input Axes.*
- ***frameon***: *bool, optional. True means that the axes frame is visible.*

Useful Matplotlib Methods

- *fig = plt.figure(figsize=(width, height))*
- *axes_obj = fig.add_subplot(nrows, ncols, plot_number)*

Example 1.10: Plotting Monthly Unemployment Trends from 1948-1952 on Subplots

```
# CB 1.2.2 #

import matplotlib.pyplot as plt
import pandas as pd

unrate = pd.read_csv('unrate.csv')

fig = plt.figure(figsize=(12,12))

axes = []
```

```
height = 5

for axnum in range(height):
    axes.append(fig.add_subplot(height,1,axnum+1))
    axes[axnum].plot(unrate['DATE'][axnum*12:(axnum+1)*12],
    unrate['VALUE'][axnum*12:(axnum+1)*12])

plt.show()
```

Output:

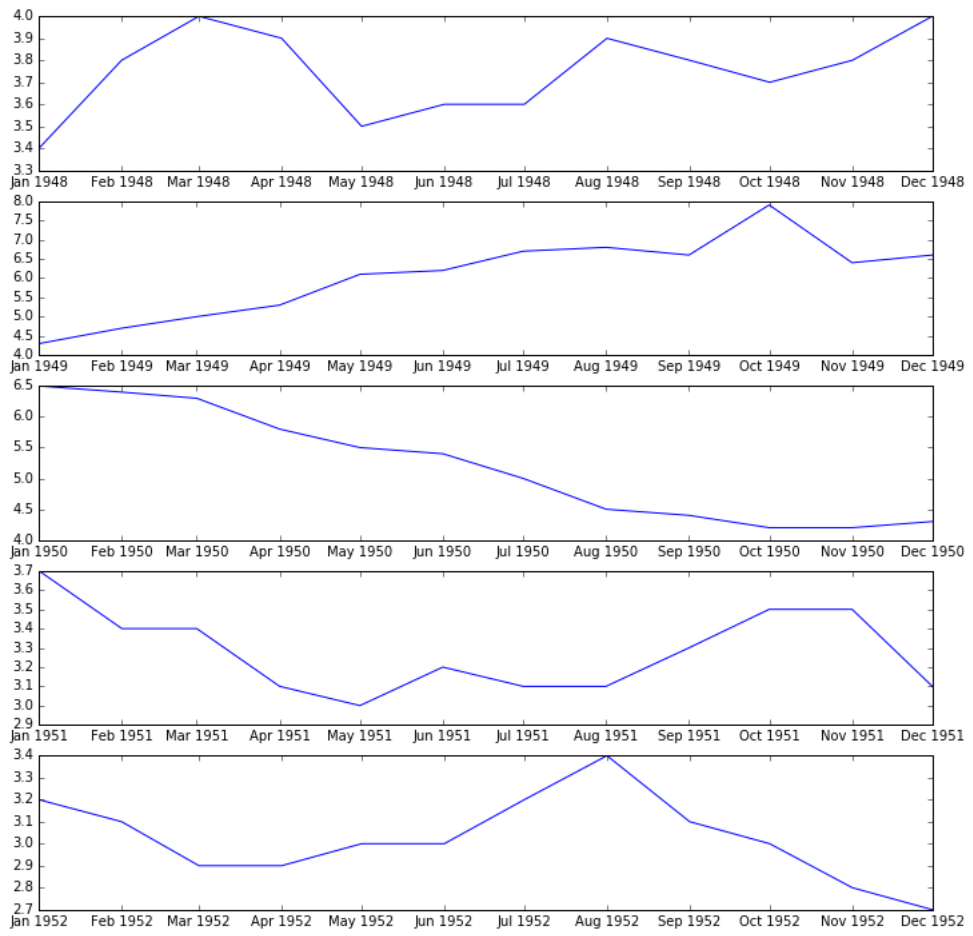

Figure 1.10: Monthly Unemployment Trends, 1948-1952 Subplots

Example 1.11: Plotting Monthly Unemployment Trends from 1948-1952 on Base Plot

```
# CB 1.2.3 #

import matplotlib.pyplot as plt
import pandas as pd
```

```
unrate = pd.read_csv('unrate.csv')

fig = plt.figure(figsize=(10,6))
colors = ['red', 'blue', 'green', 'orange', 'black']
for i in range(5):
    start_index = i*12
    end_index = (i+1)*12
    subset = unrate[start_index:end_index]
    label = str(1948 + i)
    plt.plot(subset['MONTH'], subset['VALUE'], c=colors[i], label=label)

plt.title("Monthly Unemployment Trends, 1948-1952")
plt.xlabel("Month, Integer")
plt.ylabel("Unemployment Rate, Percent")
plt.legend(loc='upper left')

plt.show()
```

Output:

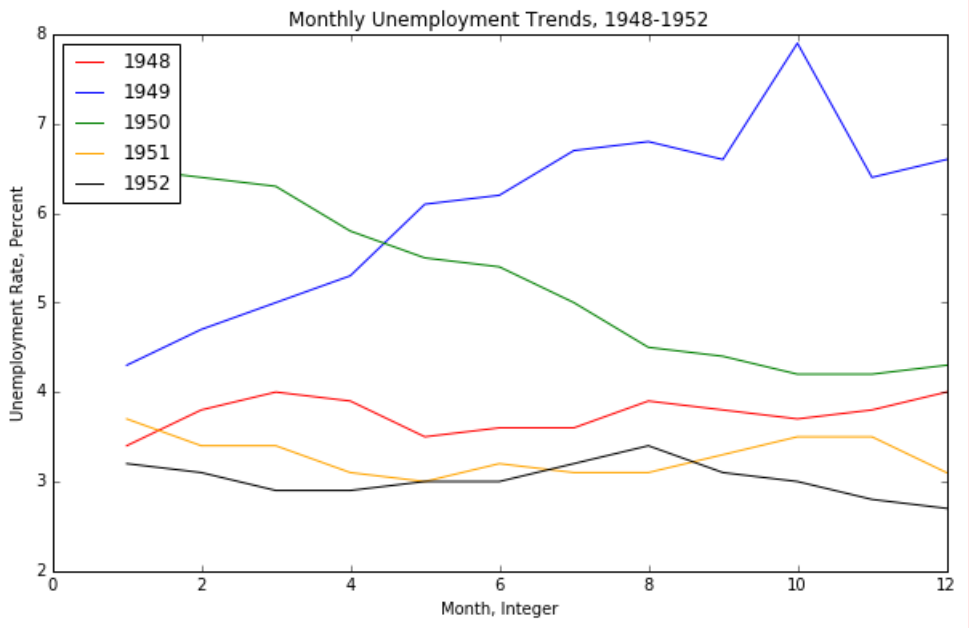

Figure 1.11: Monthly Unemployment Trends, 1948-1952

1.2.3 Bar and Scatter Plots

Example 1.12: Vertical Bar Plot for Age of Ultron Ratings

```
# CB 1.2.4 #

import matplotlib.pyplot as plt
```

```
num_cols = ['RT_user_norm', 'Metacritic_user_nom', 'IMDB_norm',
'Fandango_Ratingvalue', 'Fandango_Stars']

bar_heights = norm_reviews[num_cols].iloc[0].values
bar_positions = arange(5) + 0.75
tick_positions = range(1,6)

fig, ax = plt.subplots()
ax.bar(bar_positions, bar_heights, 0.5)
ax.set_xticks(tick_positions)
ax.set_xticklabels(num_cols, rotation=90)
ax.set_xlabel("Rating Source")
ax.set_ylabel("Average Rating")
ax.set_title("Average User Rating For Avengers: Age of Ultron (2015)")
plt.show()
```

Output:

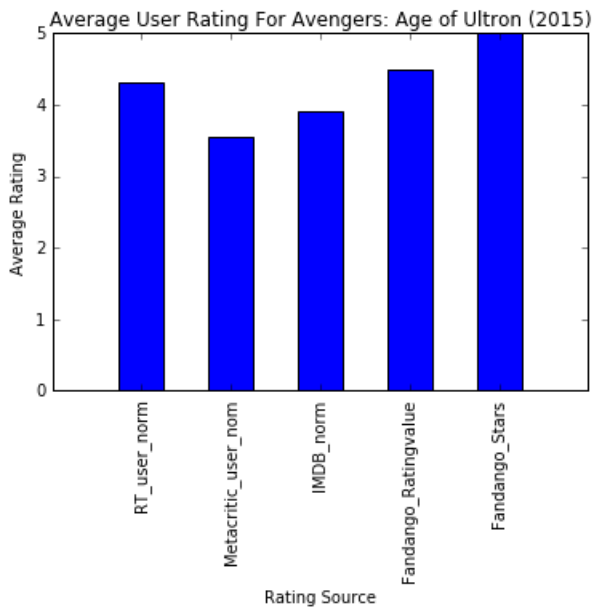

Figure 1.12: Vertical Bar Plot for Age of Ultron Ratings

Example 1.13: Horizontal Bar Plot for Age of Ultron Ratings

```
# CB 1.2.5 #

import matplotlib.pyplot as plt
from numpy import arange

num_cols = ['RT_user_norm', 'Metacritic_user_nom', 'IMDB_norm',
'Fandango_Ratingvalue', 'Fandango_Stars']
```

```
bar_widths = norm_reviews[num_cols].iloc[0].values
bar_positions = arange(5) + 0.75
tick_positions = range(1,6)

fig, ax = plt.subplots()
ax.barh(bar_positions, bar_widths, 0.5)
ax.set_yticks(tick_positions)
ax.set_yticklabels(num_cols)
ax.set_ylabel("Rating Source")
ax.set_xlabel("Average Rating")
ax.set_title("Average User Rating For Avengers: Age of Ultron (2015)")
plt.show()
```

Output:

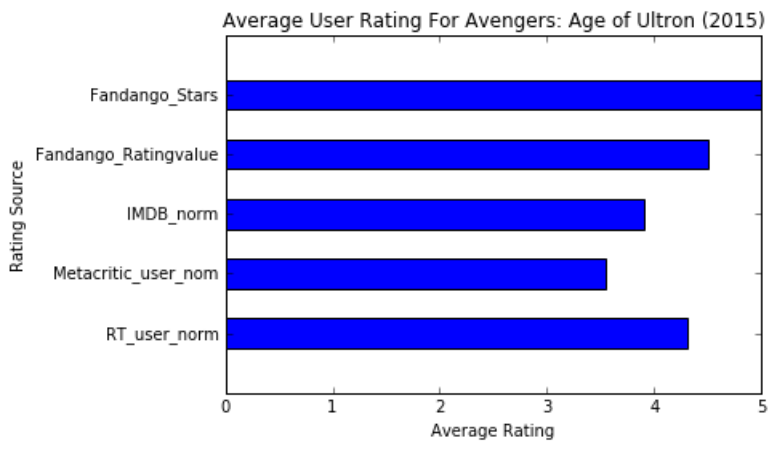

Figure 1.13: Horizontal Bar Plot for Age of Ultron Ratings

Example 1.14: Scatter Plots for Movie Rating Sites vs Fandango

```
# CB 1.2.6 #

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(5,10))
ax1 = fig.add_subplot(3,1,1)
ax2 = fig.add_subplot(3,1,2)
ax3 = fig.add_subplot(3,1,3)

axes = [ax1,ax2,ax3]

review_cols = ["RT_user_norm", "Metacritic_user_norm", "IMDB_norm"]
labels = ["Rotten Tomatoes", "Metacritic", "IMDB"]

for i in range(3):
    axes[i].scatter(norm_reviews["Fandango_Ratingvalue"],
```

```
    norm_reviews[review_cols[i]])
    axes[i].set_xlabel("Fandango")
    axes[i].set_ylabel(labels[i])
    axes[i].set_xlim(0,5)
    axes[i].set_ylim(0,5)

plt.show()
```

Output:

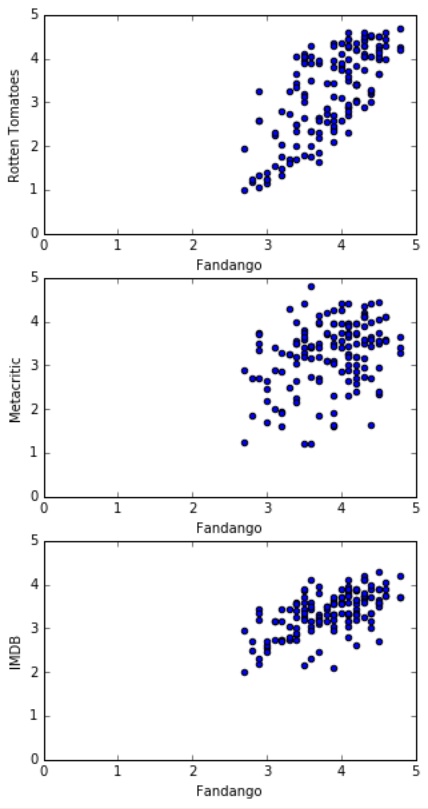

Figure 1.14: Scatter Plots among Movie Rating Sites

1.2.4 Histograms and Box Plots

Example 1.15: Movie Rating Distribution Histograms

```
# CB 1.2.7 #

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(5,20))
ax1 = fig.add_subplot(4,1,1)
ax2 = fig.add_subplot(4,1,2)
ax3 = fig.add_subplot(4,1,3)
```

```
ax4 = fig.add_subplot(4,1,4)
axes = [ax1, ax2, ax3, ax4]

col_titles = ["Fandango", "Rotten Tomatoes", "Metacritic", "IMDB"]
col_names = ['Fandango_Ratingvalue', 'RT_user_norm', 'Metacritic_user_nom',
'IMDB_norm']

for i in range(4):
    axes[i].hist(norm_reviews[col_names[i]], bins =20, range = (0,5))
    axes[i].set_title("Distribution of" + col_titles[i] + "Ratings")
    axes[i].set_ylim(0,50)

plt.show()
```

Output:

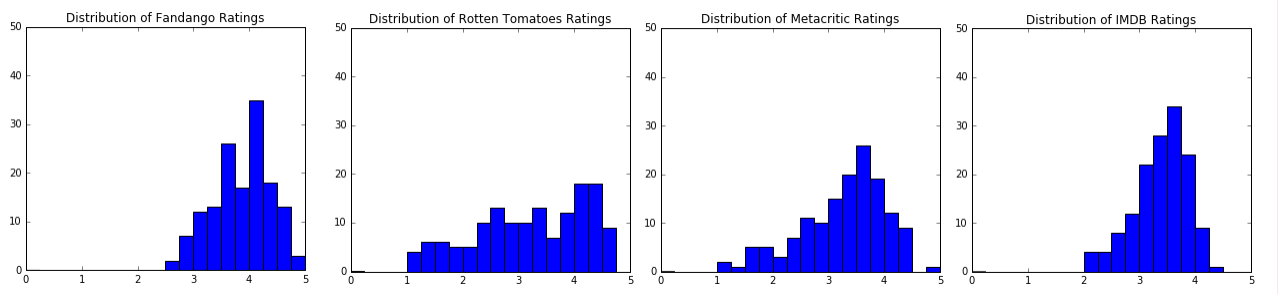

Figure 1.15: Movie Distribution Histogram Ratings

! For the sake of visual aesthetic, I've placed the plots horizontally. They are normally outputted vertically[a] in the same order presented above (Fandango on top and IMDB on bottom).

[a] In the same way as Figure 1.14.

Box Plot Quartiles

In descriptive statistics, a box plot or boxplot is a method for graphically depicting groups of numerical data through their quartiles. Box plots may also have lines extending vertically from the boxes (whiskers) indicating variability outside the upper and lower quartiles, hence the terms box-and-whisker plot and box-and-whisker diagram.

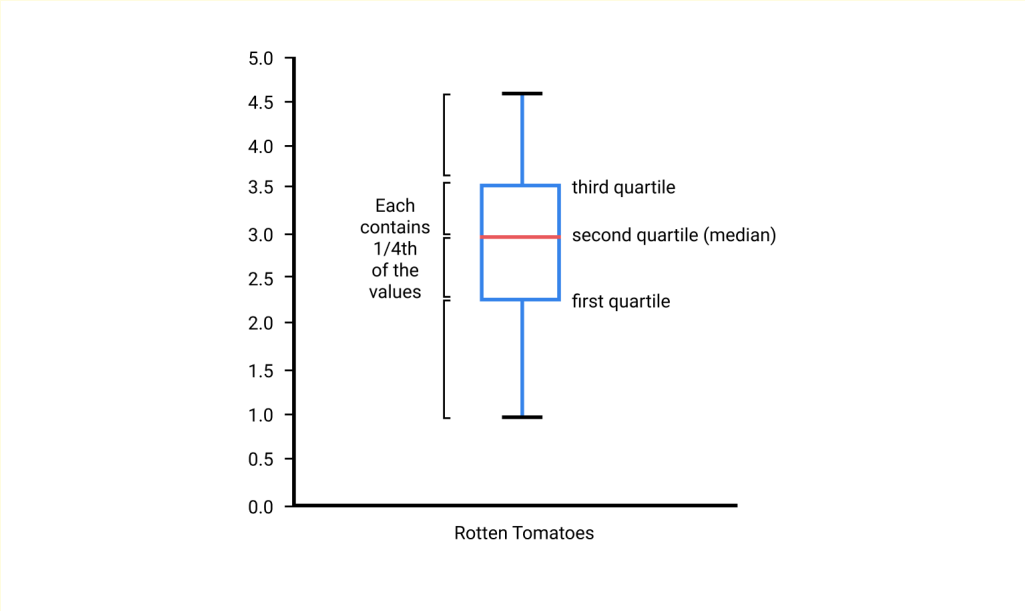

Figure 1.16: Box Plot Quartiles

We have graphically depicted an example of such a box plot in Figure 1.16 for the movie rating website Rotten Tomatoes.

Example 1.16: Movie Rating Box Plots

```
# CB 1.2.8 #

num_cols = ['RT_user_norm', 'Metacritic_user_nom', 'IMDB_norm',
'Fandango_Ratingvalue']

fix, ax = plt.subplots()

ax.boxplot(norm_reviews[num_cols].values)
ax.set_xticklabels(num_cols, rotation = 90)
ax.set_ylim(0,5)

plt.show()
```

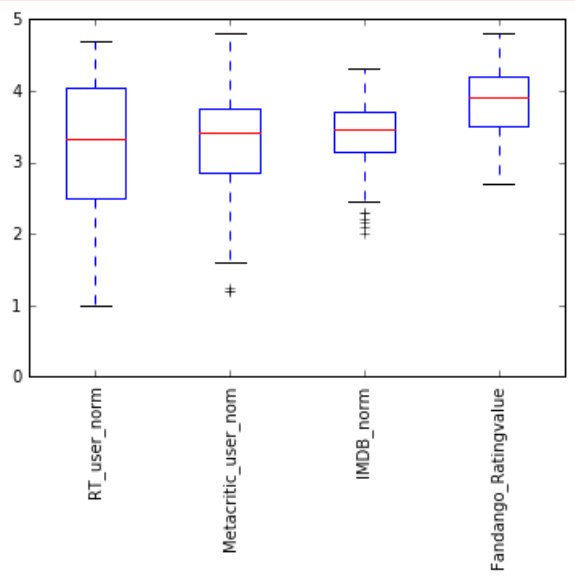

Figure 1.17: Movie Distribution Ratings Box Plot

1.2.5 Guided Project: Visualizing Majors Based on College Majors

Pandas has many methods for quickly generating common plots from data in DataFrames. Like pyplot, the plotting functionality in pandas is a wrapper for matplotlib. This means we can customize the plots when necessary by accessing the underlying Figure, Axes, and other matplotlib objects. For further documentation on visualization in Pandas, see [8].

Example 1.17: Pandas Plot on Male Dominated Undergrad Majors

```
# CB 1.2.9 #

import pandas as pd
import matplotlib.pyplot as plt

recent_grads = pd.read_csv('recent_grads.csv')
male_dominant = recent_grads[recent_grads['ShareWomen'] < 0.25]
male_dominant.plot(x = 'Major', y = 'ShareWomen',
title = 'Male-Dominated_Majors', kind = 'bar', legend = None)
```

Output:

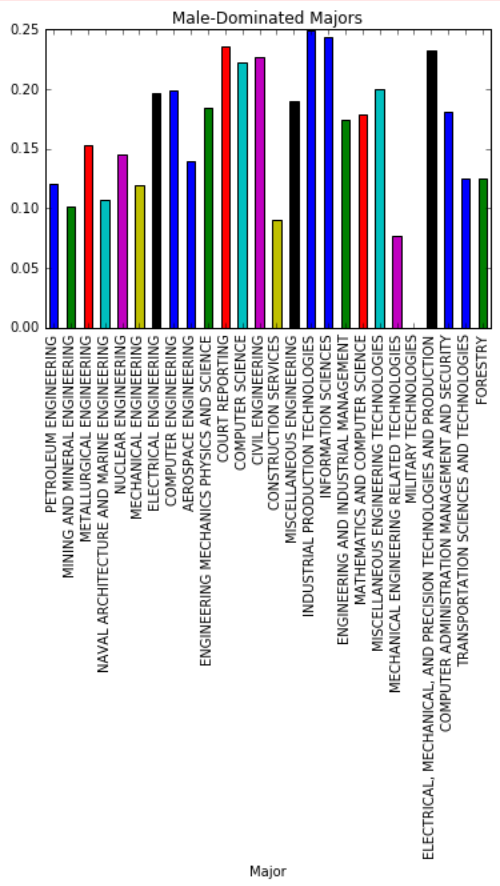

Figure 1.18: Male Dominated Majors

Example 1.18: Scatter Matrix Plots

```
# CB 1.2.10 #

import pandas as pd
import matplotlib.pyplot as plt

recent_grads = pd.read_csv('recent_grads.csv')
scatter_matrix(recent_grads[['Sample_size','Median']], figsize=(10,10))
```

Output:

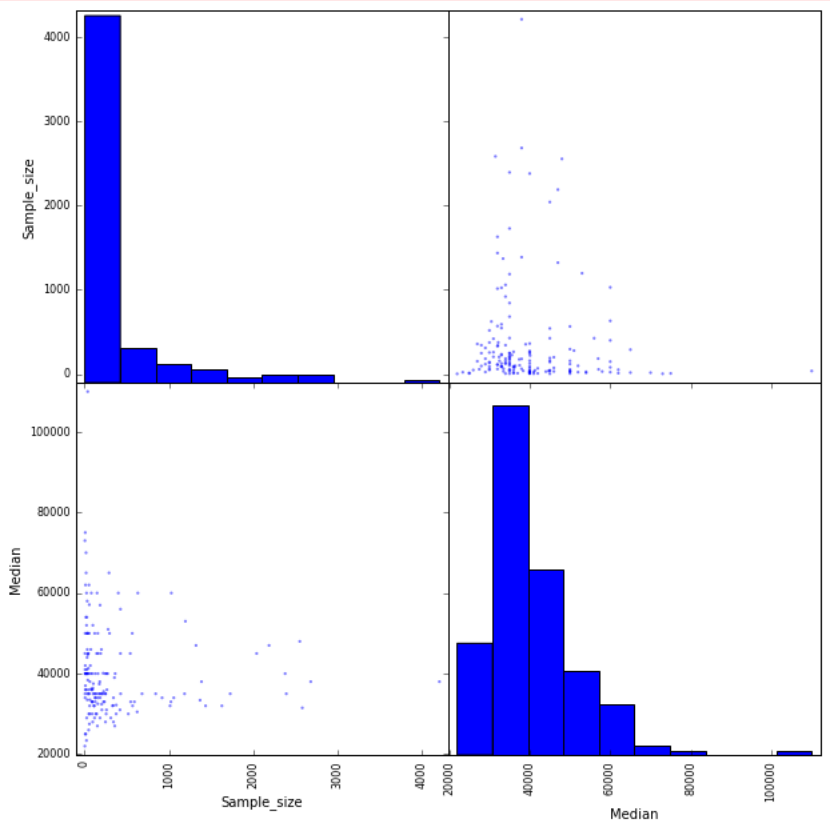

Figure 1.19: Scatter Matrix with 'Sample Size', 'Median' Columns

1.3 Storytelling Through Data Visualization

1.3.1 Improving Plot Aesthetics

Definition 1.20: Chartjunk

Chartjunk refers to all visual elements in charts and graphs that are not necessary to comprehend the information represented on the graph, or that distract the viewer from this information.

Markings and visual elements can be called chartjunk if they are not part of the minimum set of visuals necessary to communicate the information understandably. Examples of unnecessary elements that might be called chartjunk include heavy or dark grid lines, unnecessary text, inappropriately complex or gimmicky font faces, ornamented chart axes, and display frames, pictures, backgrounds or icons within data graphs, ornamental shading and unnecessary dimensions.

Example 1.19: Spine and Tick Removal

With the axis tick marks gone, the data-ink ratio is improved and the chart looks much cleaner. In addition, the spines in the chart now are no longer necessary. When we're exploring data, the spines and the ticks complement each other to help us refer back to specific data points or ranges. When a viewer is viewing our chart and trying to understand the insight we're presenting, the ticks and spines can get in the way.

```
# CB 1.3.1 #

import pandas as pd
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(women_degrees['Year'], women_degrees['Biology'], label='Women')
ax.plot(women_degrees['Year'], 100-women_degrees['Biology'], label='Men')
ax.tick_params(bottom="off", top="off", left="off", right="off")
ax.spines["right"].set_visible(False)
ax.spines["left"].set_visible(False)
ax.spines["top"].set_visible(False)
ax.spines["bottom"].set_visible(False)

ax.legend(loc='upper right')
ax.set_title('Percentage of Biology Degrees Awarded By Gender')
plt.show()
```

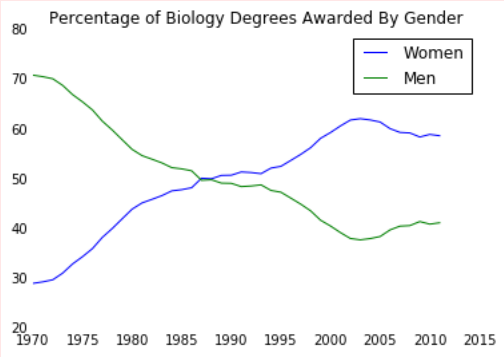

Figure 1.20: Percentage of Women Biology Degree Holders over Time

1.3.2 Conditional Plots

Definition 1.21: Seaborn Module

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn works similarly to the pyplot module from matplotlib. We primarily use seaborn interactively, by calling functions in its top level namespace. Like the pyplot module from matplotlib, seaborn creates a matplotlib figure or adds to the current, existing figure each time we generate a plot.

Example 1.20: Seaborn

Under the hood, seaborn creates a histogram using matplotlib, scales the axes values, and styles it. In addition, seaborn uses a technique called kernel density estimation [9], or KDE for short, to create a smoothed line chart over the histogram.

```
# CB 1.3.2 #
```

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.distplot(titanic["Age"])
plt.show()
```

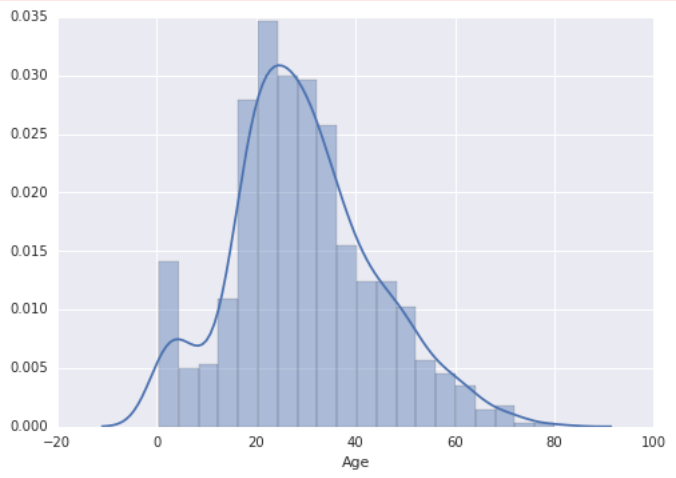

Figure 1.21: Seaborn Plot

Example 1.21: Seaborn KDE

If we wish to only view the KDE plot in seaborn, one can use the kdeplot() function. In addition, shading the area underneath the KDE function can be accomplished by setting the shade parameter to True.

```
sns.kdeplot(titanic['Age'], shade = True)
plt.xlabel("Age")
```

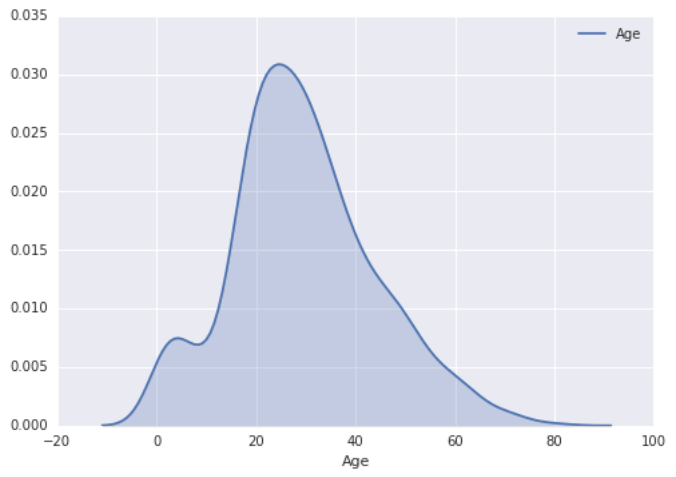

Figure 1.22: KDEplot with Shade

Example 1.22: Seaborn Despining and FacetGrid

The Seaborn despine function allows us to remove the spines from the graph. By default it will remove the top and right spines, but to remove left and bottom, we have to set these parameters to True.

The FacetGrid function allows us to display multiple graphs at the same time through conditional relationships. Along the row and column axes of the graph position in this grid, we can specify how to subset the titanic dataframe. In this example, we subset unique values for 'Pclass' along the column axis and unique values for 'Survived' along the row axis. In addition, we can also plot multiple figures on the same subplot with the hue parameter, further allowing us to subset unique values of 'Sex'.

```
# CS 1.3.3 #

g = sns.FacetGrid(titanic, col="Pclass", row="Survived", hue = 'Sex', size = 3)
g.map(sns.kdeplot, "Age", shade=True).add_legend()
sns.despine(left=True, bottom=True)
plt.show()
```

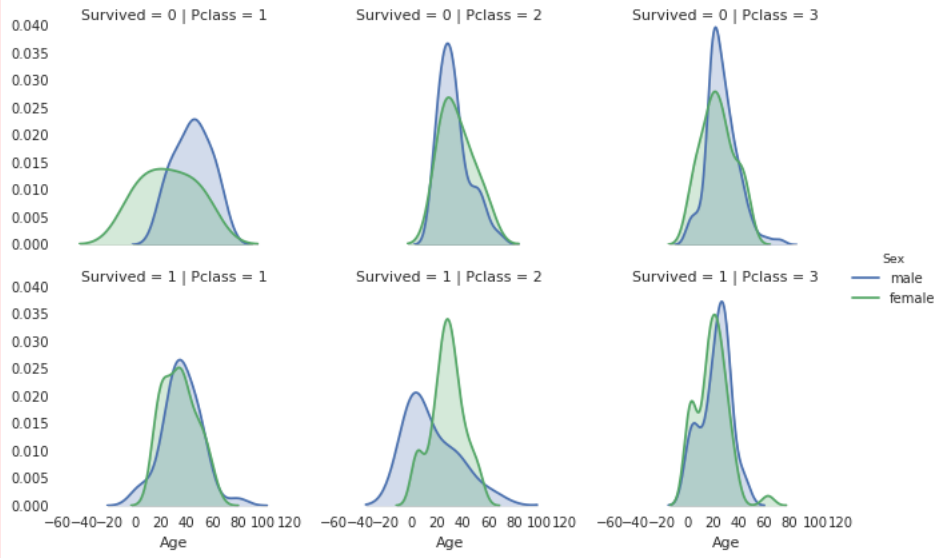

Figure 1.23: Seaborn FacetGrid with Despining

1.3.3 Visualizing Geographic Data

Definition 1.22: Basemap

class *mpl_toolkits.basemap.Basemap*

Sets up a basemap[10] with specified map projection. and creates the coastline data structures in map projection coordinates.

Calling a Basemap class instance with the arguments lon, lat will convert lon/lat (in degrees) to x/y map projection coordinates (in meters). The inverse transformation is done if the optional keyword inverse is set to True.

Example 1.23: Basemap Visualization with Airline Routes Data

```
# CB 1.3.4 #

import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

fig, ax = plt.subplots(figsize = (15, 20))
ax.set_title('Scaled Up Earth With Coastlines')
m = Basemap(projection='merc', llcrnrlat=-80, urcrnrlat=80, llcrnrlon=-180,
urcrnrlon=180)
longitudes = airports["longitude"].tolist()
latitudes = airports["latitude"].tolist()
x, y = m(longitudes, latitudes)
m.scatter(x, y, s=1)
m.drawcoastlines()
plt.show()
```

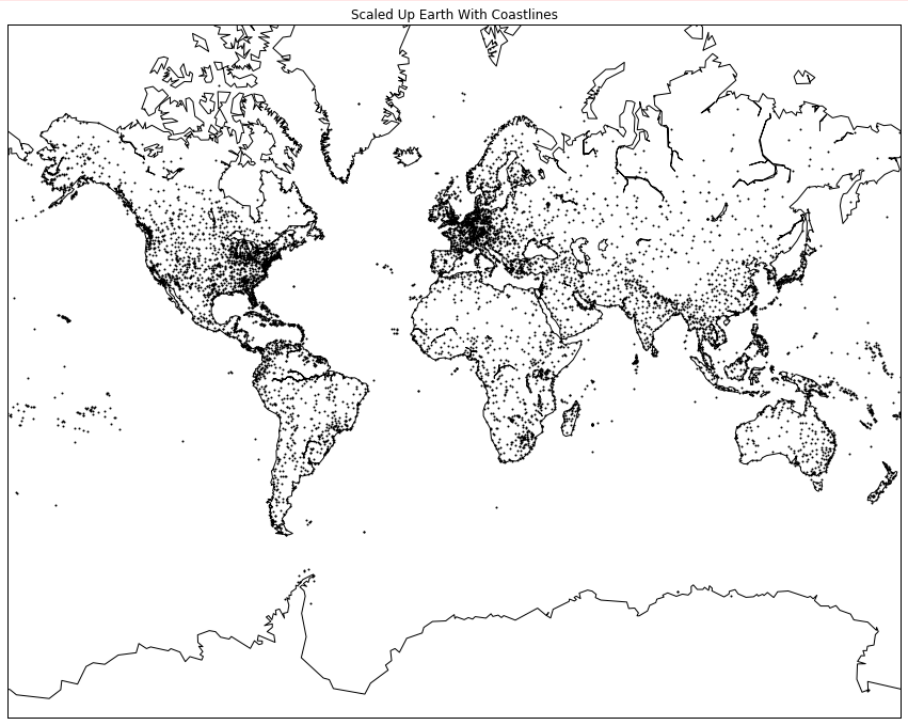

Figure 1.24: Geographic Data With Basemap

Example 1.24: Great Circles for Airline Flights

```
# CB 1.3.5 #

fig, ax = plt.subplots(figsize=(15,20))
m = Basemap(projection='merc', llcrnrlat=-80, urcrnrlat=80, llcrnrlon=-180,
```

```
urcrnrlon=180)
m.drawcoastlines()

# Start writing your solution below this line

def create_great_circles(dataframe):

    for index, row in dataframe.iterrows():
        if abs(row['end_lat'] - row['start_lat']) < 180 and
        abs(row['end_lon'] - row['start_lon']) < 180:
            m.drawgreatcircle(row['start_lon'], row['start_lat'],
            row['end_lon'], row['end_lat'])
        else:
            continue

dfw = geo_routes[geo_routes['source']=='DFW']

create_great_circles(dfw)
plt.show()
```

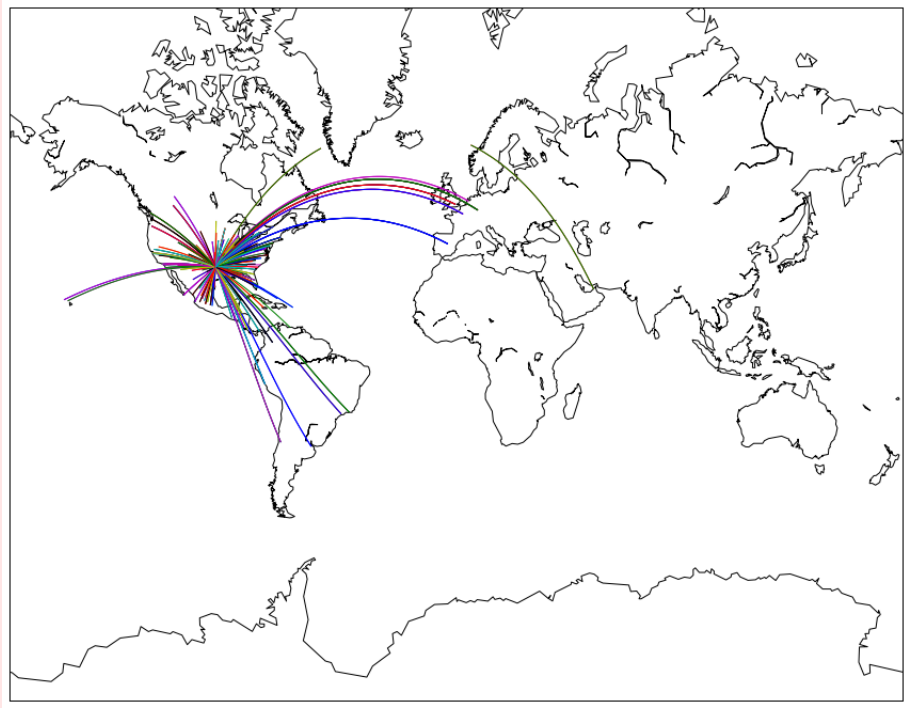

Figure 1.25: Great Circles for Airline Flights

2 The Command Line

2.1 Elements of the Command Line

2.1.1 Introduction to the Command Line

Definition 2.1: Command Line Interface

A Command Line Interface (CLI) is a text only interface through which users interact with computers by typing text instructions in a console (or terminal), using specific syntax.

As technology evolved and computers became ubiquitous, terminals were emulated within GUIs, giving rise to terminal emulators (also terminal window or just terminal).

Definition 2.2: Commands

Instructions sent to the CLI are called commands which, once input, are interpreted by a type of program called a shell or command language interpreter, and then run by your machine. Some of the most popular shells are Bash, Z shell, KornShell, Command Prompt and Windows PowerShell.

Definition 2.3: Command Parameters

The general syntax for commands are:

```
utility_name parameter1 parameter2 ... parameterN
```

By definition, a parameter is either an option or an argument. An ***option*** *is a string of symbols that modifies the behavior of the command and it always starts with a dash (-). Other possible names for this parameter are flag and switch, but depending on who you ask they might not always be interchangeable. An* ***argument*** *— or operand — is an object upon which the command acts. The* ***utility*** *(also command or program) is the first item in the instruction.*

3 Working with Data Sources

3.1 SQL Fundamentals

3.1.1 Introduction to SQL

Definition 3.1: SQL

SQL (Structured Query Language) is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS). It is particularly useful in handling structured data, i.e. data incorporating relations among entities and variables.

Definition 3.2: RDBMS

Connolly and Begg define Database Management System (DBMS) as a "software system that enables users to define, create, maintain and control access to the database". RDBMS is an extension of that acronym that is sometimes used when the underlying database is relational.

SQL Comparison Operators

We can use the following comparison operators in SQL:

- *<: Less than*
- *<=: Less than or equal to*
- *>: Greater than*
- *>=: Greater than or equal to*
- *=: Equal to*
- *! =: Not equal to*

SQL Logical Operators

We can use the following logical operators in SQL [11]:

Operator	*Result*
ALL	*TRUE if all of the subquery values meet the condition.*
AND	*TRUE if all the conditions separated by AND is TRUE.*
ANY	*TRUE if any of the subquery values meet the condition.*
BETWEEN	*TRUE if any of the subquery values meet the condition.*
EXISTS	*TRUE if the subquery returns one or more records.*
IN	*TRUE if the operand is equal to one of a list of expressions.*
LIKE	*TRUE if the operand matches a pattern.*
NOT	*Displays a record if the condition(s) is NOT TRUE.*
OR	*TRUE if any of the conditions separated by OR is TRUE.*
SOME	*TRUE if any of the subquery values meet the condition.*

Definition 3.3: SELECT Operation

The SELECT statement is used to select data from a database. The data returned is stored in a result table, called the result-set.

If you wish to select columns column1, column2,..., the syntax is given by:

```
SELECT column1, column2, ...
FROM table_name;
```

If you wish to select all columns from table_name, you can use ∗:

```
SELECT * FROM table_name
```

Definition 3.4: WHERE Operation

The WHERE clause is used to filter records. The WHERE clause is used to extract only those records that fulfill a specified condition.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Definition 3.5: ORDER BY Operation

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

Example 3.1: Select, Where and Order

In this example, we have a table named "recent_grads", from which we want to display particular rows and columns matching our criteria. We only want to display the Major, ShareWomen and Unemployment_rate column so we use the SELECT operation to do so. We use FROM to indicate the table and WHERE to establish the two desired criterions. We want to display rows where women held at least 30% of student population and had an unemployment rate less than 10%. Finally, when we display our information, we want to order it according to the ShareWomen values, descending.

```
# CB 1.4.1 #

SELECT Major, ShareWomen, Unemployment_rate
FROM recent_grads
WHERE ShareWomen > 0.3 AND Unemployment_rate < 0.1
ORDER BY ShareWomen DESC
```

Output

[121 rows x 3 columns]

Major	ShareWomen	Unemployment_rate
EARLY CHILDHOOD EDUCATION	0.9679981190000001	0.040104981
MATHEMATICS AND COMPUTER SCIENCE	0.927807246	0
ELEMENTARY EDUCATION	0.923745479	0.046585715
ANIMAL SCIENCES	0.91093257	0.050862499000000005
PHYSIOLOGY	0.9066773370000001	0.06916280000000001
MISCELLANEOUS PSYCHOLOGY	0.90558993	0.05190783
HUMAN SERVICES AND COMMUNITY ORGANIZA...	0.9040745440000001	0.037819026
NURSING	0.896018988	0.04486272400000001
GEOSCIENCES	0.881293889	0.024373731000000003
MASS MEDIA	0.8772275279999999	0.089836827
COGNITIVE SCIENCE AND BIOPSYCHOLOGY	0.854523227	0.075236167
ART HISTORY AND CRITICISM	0.845934379	0.060298284
EDUCATIONAL PSYCHOLOGY	0.8170988090000001	0.065112187
GENERAL EDUCATION	0.8128766059999999	0.057359929000000004

Figure 3.1: Displaying the result of Query CB 1.4.1

3.1.2 Summary Statistics

Definition 3.6: Aggregate Function

Aggregate functions are applied over columns of values and return a single value. MIN() and MAX(), for example, calculate and return the minimum and maximum values in a column.

Aggregate Functions [12]

Aggregate Function	*Description*
AVG	*The AVG() aggregate function calculates the average of non-NULL values in a set.*
CHECKSUM_AGG	*The CHECKSUM_AGG() function calculates a checksum value based on a group of rows.*
COUNT	*The COUNT() aggregate function returns the number of rows in a group, including rows with NULL values.*
COUNT_BIG	*The COUNT_BIG() aggregate function returns the number of rows (with BIGINT data type) in a group, including rows with NULL values.*
MAX	*The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.*
MIN	*The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL values.*
STDEV	*The STDEV() function returns the statistical standard deviation of all values provided in the expression based on a sample of the data population.*
STDEVP	*The STDEVP() function also returns the standard deviation for all values in the provided expression, but does so based on the entire data population.*
SUM	*The SUM() aggregate function returns the summation of all non-NULL values a set.*
VAR	*The VAR() function returns the statistical variance of values in an expression based on a sample of the specified population.*
VARP	*The VARP() function returns the statistical variance of values in an expression but does so based on the entire data population.*

Definition 3.7: SQL DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values. Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

Definition 3.8: SQL Alias

SQL aliases are used to give a table, or a column in a table, a temporary name. Aliases are often used to make column names more readable. An alias only exists for the duration of the query.

```
SELECT column_name AS alias_name
FROM table_name;
```

You may also drop the AS keyword so that the syntax can be expressed by

```
SELECT column_name alias_name
FROM table_name;
```

Example 3.2: Counting Distinct Values among Columns

```
# CB 1.4.2 #

SELECT COUNT(DISTINCT(Major)) "unique_majors",
COUNT(DISTINCT(Major_category)) "unique_major_categories",
COUNT(DISTINCT(Major_code)) "unique_major_codes"
FROM recent_grads
```

Output

[1 rows x 3 columns]

unique_majors	unique_major_categories	unique_major_codes
173	16	173

Figure 3.2: The Number of Unique Values in Major, Major Category and Major Code Columns

Example 3.3: Displaying Ordered Quartile Spread among Columns

```
# CB 1.4.3 #

SELECT Major, Major_category, P75th - P25th quartile_spread FROM recent_grads
ORDER BY quartile_spread LIMIT 10
```

Output

[10 rows x 3 columns]

Major	Major_category	quartile_spread
MILITARY TECHNOLOGIES	Industrial Arts & Consumer Services	0
SCHOOL STUDENT COUNSELING	Education	2000
LIBRARY SCIENCE	Education	2000
COURT REPORTING	Law & Public Policy	4000
PHARMACOLOGY	Biology & Life Science	5000
EDUCATIONAL ADMINISTRATION AND SUPERV...	Education	6000
COUNSELING PSYCHOLOGY	Psychology & Social Work	6800
SPECIAL NEEDS EDUCATION	Education	10000
MATHEMATICS TEACHER EDUCATION	Education	10000
SOCIAL WORK	Psychology & Social Work	10000

Figure 3.3: Quartile Spread for Majors

3.1.3 Group Summary Statistics

Definition 3.9: PRAGMA TABLE_INFO

The table_info pragma is used to query information about a specific table. The result set will contain one row for each column in the table. It will display information such as the type of objects stored in the columns, Null values and meanings.

Definition 3.10: SQL GROUP BY Statement

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country". The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

Example 3.4: GROUP BY in Action

The GROUP BY statement works by partitioning the relevant column into its unique entries, then performing the desired operation on each group.

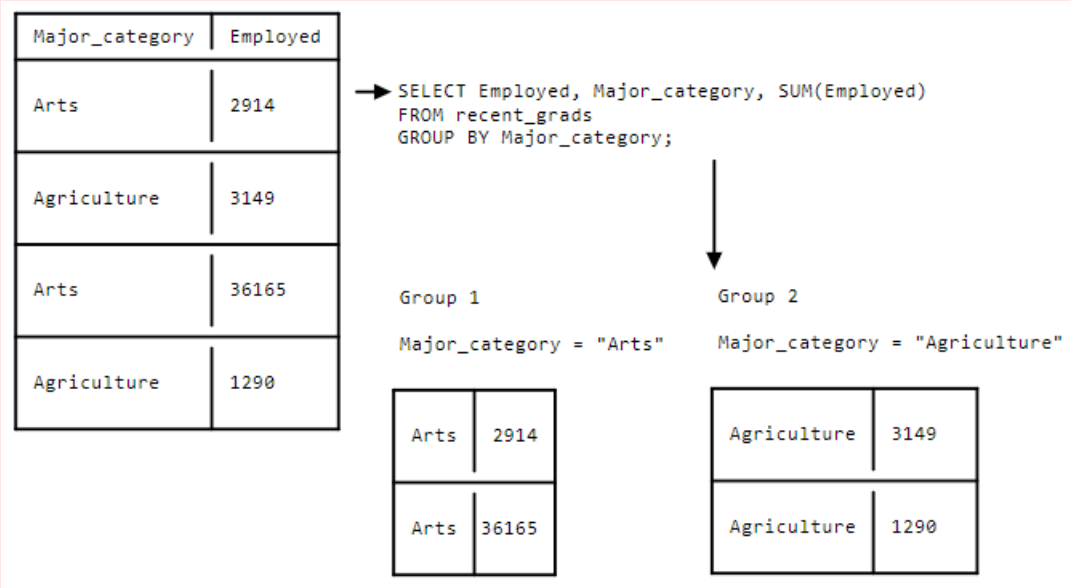

Figure 3.4: Grouping Operation

In this example, we aim to compute the average of women population percentages among each major category. Once done, we display it below in Figure 3.5.

 # CB 1.4.4 #

SELECT *Major_category* , ***AVG***(*ShareWomen*) ***FROM*** *recent_grads*
GROUP BY *Major_category*

Output

[16 rows x 2 columns]

Major_category	AVG(ShareWomen)
Agriculture & Natural Resources	0.6179384232
Arts	0.56185119575
Biology & Life Science	0.584518475857143
Business	0.4050631853076923
Communications & Journalism	0.64383484025
Computers & Mathematics	0.5127519954545455
Education	0.6749855163125
Engineering	0.2571578951034483
Health	0.6168565694166667
Humanities & Liberal Arts	0.6761934042
Industrial Arts & Consumer Services	0.4493512688571429
Interdisciplinary	0.495397153
Law & Public Policy	0.3359896912
Physical Sciences	0.5087494197

Figure 3.5: Group By ShareWomen Averages in Major Categories

Definition 3.11: SQL HAVING Statement

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

Definition 3.12: ROUND Function

The ROUND() function rounds a number to a specified number of decimal places.

```
SELECT ROUND(numeric_column_name, dec_places)
```

Definition 3.13: CAST Function

The CAST() function converts a value (of any type) into the specified datatype.

Example 3.5: Casting

In this example, we want to display the ratio of women to the total, grouping by each Major_category, summing and dividing by the total.

```
# CB 1.4.5 #

SELECT Major_category, CAST(Sum(Women) as float) / CAST(Sum(total) as float) SW
FROM recent_grads
GROUP BY Major_category
ORDER BY SW
```

Output

[16 rows x 2 columns]

Major_category	SW
Law & Public Policy	0.030585069260274586
Business	0.08474280852841269
Industrial Arts & Consumer Services	0.16024926890405236
Computers & Mathematics	0.20935560252568494
Engineering	0.2195958577559186
Communications & Journalism	0.25032539397505355
Arts	0.39332735978495226
Humanities & Liberal Arts	0.490051410855147
Health	0.6735876346523325
Interdisciplinary	0.8009108653220559
Social Science	0.8748032892676134
Psychology & Social Work	1.0491780784895022
Education	1.0962729531109994
Biology & Life Science	1.273805694241862

Figure 3.6: The Output of Query 1.4.5

3.1.4 Subqueries

Definition 3.14: Subquery

A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause.A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <= IN, BETWEEN, etc.

Definition 3.15: IN Operator

The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

Example 3.6: Subqueries with IN Keyword

In this example, we want to write a query that returns the Major, Major_category columns for the rows where Major_category is one of the 5 highest group level sums for the Total column.

```
# CB 1.4.6 #

SELECT Major, Major_category FROM recent_grads
WHERE Major_category IN (SELECT Major_category FROM recent_grads
GROUP BY Major_category
ORDER BY SUM(Total) DESC
LIMIT 5)
```

Output

[82 rows x 2 columns]

Major	Major_category
PETROLEUM ENGINEERING	Engineering
MINING AND MINERAL ENGINEERING	Engineering
METALLURGICAL ENGINEERING	Engineering
NAVAL ARCHITECTURE AND MARINE ENGINEE...	Engineering
CHEMICAL ENGINEERING	Engineering
NUCLEAR ENGINEERING	Engineering
ACTUARIAL SCIENCE	Business
MECHANICAL ENGINEERING	Engineering
ELECTRICAL ENGINEERING	Engineering
COMPUTER ENGINEERING	Engineering
AEROSPACE ENGINEERING	Engineering
BIOMEDICAL ENGINEERING	Engineering
MATERIALS SCIENCE	Engineering
ENGINEERING MECHANICS PHYSICS AND SCI...	Engineering

Figure 3.7: The Output of Query CB 1.4.6

3.1.5 Guided Project: Analyzing CIA Factbook Data Using SQL

Definition 3.16: SQLite

SQLite is a relational database management system (RDBMS) contained in a C library. In contrast to many other database management systems, SQLite is not a client–server database engine. Rather, it is embedded into the end program.

Example 3.7: Analyzing CIA Factbook Data

To enable the ability to use SQL and load up the factbook.db [13], we enter the following code into the Jupyter Notebook:

```
%%capture
%load_ext sql
%sql sqlite:///factbook.db
```

```
In [36]: %%sql
SELECT name, birth_rate, death_rate, ROUND(death_rate - birth_rate, 5) AS "Population Decline"
FROM facts
ORDER BY death_rate DESC

Done.
```

Out[36]:

name	birth_rate	death_rate	Population Decline
Lesotho	25.47	14.89	-10.58
Ukraine	10.72	14.46	3.74
Bulgaria	8.92	14.44	5.52
Guinea-Bissau	33.38	14.33	-19.05
Latvia	10.0	14.31	4.31
Chad	36.6	14.28	-22.32
Lithuania	10.1	14.27	4.17

Figure 3.8: Birth / Death Rates ordered by Death Rate per Country

3.2 SQL Intermediate: Table Relations and Joins

3.2.1 Joining Data in SQL

Definition 3.17: JOIN Operation

A JOIN clause is used to combine rows from two or more tables, based on a related column between them. The syntax for a join clause is given below.

SELECT *[column_names]* ***FROM*** *[table_name_one]*
INNER JOIN *[table_name_two]* ***ON*** *[join_constraint];*

ON*, which tells the SQL engine what columns to use to join the two tables. The syntax for specifying table column names is given by "table_name.column_name".*

Definition 3.18: Inner and Outer Join

(INNER) JOIN: *Returns records that have matching values in both tables.*
LEFT (OUTER) JOIN: *Returns all records from the left table, and the matched records from the right table.*
RIGHT (OUTER) JOIN: *Returns all records from the right table, and the matched records from the left table.*
FULL (OUTER) JOIN*: Returns all records when there is a match in either left or right table.*

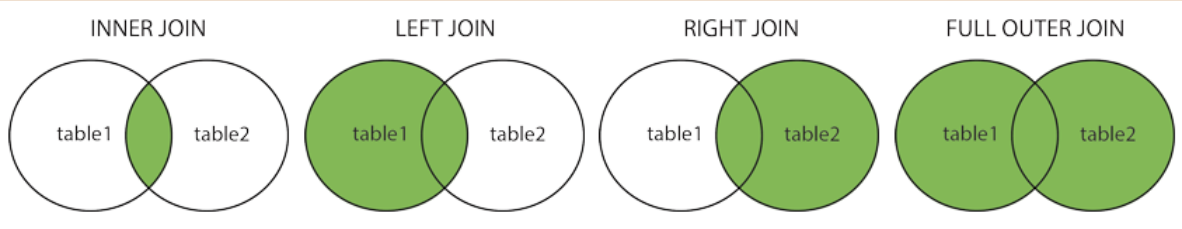

Figure 3.9: Inner and Outer Joins

Example 3.8: Population among Capital Cities

We want to write a query that returns the 10 capital cities with the highest population ranked from biggest to smallest population. It will include columns in the following order:
- *capital_city, the name of the city.*
- *country, the name of the country the city is from.*
- *population, the population of the city.*

```
# CB 1.4.7 #

SELECT c.name capital_city, f.name country, c.population population
FROM cities c
INNER JOIN facts f ON f.id = c.facts_id
WHERE c.capital = 1
ORDER BY population DESC
LIMIT 10
```

Output
[10 rows x 3 columns]

capital_city	country	population
Tokyo	Japan	37217000
New Delhi	India	22654000
Mexico City	Mexico	20446000
Beijing	China	15594000
Dhaka	Bangladesh	15391000
Buenos Aires	Argentina	13528000
Manila	Philippines	11862000
Moscow	Russia	11621000
Cairo	Egypt	11169000
Jakarta	Indonesia	9769000

Figure 3.10: The Output of Query CB 1.4.7

Example 3.9: Population Density in Cities Across a Country

We we will write a query that generates columns in the following order:

- *country, the name of the country.*
- *urban_pop, the sum of the population in major urban areas belonging to that country.*
- *total_pop, the total population of the country.*
- *urban_pct, the percentage of the population within urban areas, calculated by dividing urban_pop by total_pop.*

Lastly, we subject these rows to the criteria that we only want countries that have an urban_pct greater than 0.5 and that the rows should be sorted by urban_pct in ascending order.

```
# CB 1.4.8 #

SELECT f.name country, urb.urban_pop urban_pop, f.population total_pop,
CAST(urban_pop as float) / CAST(f.population as float) urban_pct
FROM facts f
INNER JOIN (SELECT SUM(c.population) urban_pop, facts_id FROM cities c
GROUP BY facts_id) urb ON urb.facts_id = f.id
WHERE urban_pct > 0.5
ORDER BY urban_pct
```

Output

[18 rows x 4 columns]

country	urban_pop	total_pop	urban_pct
Uruguay	1672000	3341893	0.5003152404939356
Congo, Republic of the	2445000	4755097	0.5141850944365594
Brunei	241000	429646	0.5609269026128487
New Caledonia	157000	271615	0.5780240413821034
Virgin Islands	60000	103574	0.5792959623071428
Falkland Islands (Islas Malvinas)	2000	3361	0.5950609937518596
Djibouti	496000	828324	0.5987995035758954
Australia	13789000	22751014	0.6060828761302683
Iceland	206000	331918	0.6206352171319423
Israel	5226000	8049314	0.6492478737939655
United Arab Emirates	3903000	5779760	0.6752875551926032
Puerto Rico	2475000	3598357	0.6878139106264332
Bahamas, The	254000	324597	0.7825087724162577
Kuwait	2406000	2788534	0.8628189579183901

Figure 3.11: The Output of Query CB 1.4.8

3.2.2 Intermediate Joins in SQL

Joining more than Two Tables

The syntax for joining more than two tables in SQL is given below:

```
SELECT [column_names] FROM [table_name_one]
[join_type] JOIN [table_name_two] ON [join_constraint]
[join_type] JOIN [table_name_three] ON [join_constraint];
```

Chinook Database

In this mission, we use a database named chinook. It's schema diagram is presented below:

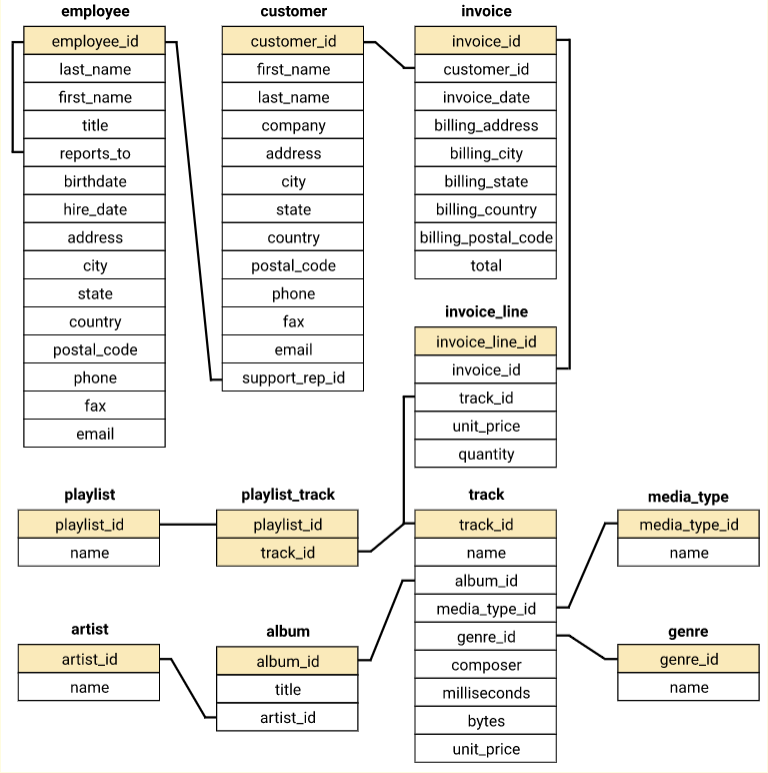

Figure 3.12: Chinook Database Schema

Example 3.10: Popular Albums

We want to write a query that returns the top 5 albums, as calculated by the number of times a track from that album has been purchased. The query will be sorted from most tracks purchased to least tracks purchased and return the following columns, in order:

- *album, the title of the album.*
- *artist, the artist who produced the album.*
- *tracks_purchased the total number of tracks purchased from that album.*

```
# CB 1.4.9 #

SELECT
    trkart.album album,
    trkart.artist_name artist,
    SUM(il.quantity) tracks_purchased
FROM invoice_line il
INNER JOIN
    (SELECT
        t.track_id,
        art.name artist_name,
```

```
        alb.title  album
    FROM artist  art
    INNER JOIN album  alb ON alb.artist_id = art.artist_id
    INNER JOIN track  t ON t.album_id = alb.album_id
    ) trkart ON trkart.track_id = il.track_id
GROUP BY album
ORDER BY tracks_purchased DESC
LIMIT 5
```

Output

[5 rows x 3 columns]

album	artist	tracks_purchased
Are You Experienced?	Jimi Hendrix	187
Faceless	Godsmack	96
Mezmerize	System Of A Down	93
Get Born	JET	90
The Doors	The Doors	83

Figure 3.13: The Output of Query CB 1.4.9

Definition 3.19: Self / Recursive Join

A self JOIN is a regular join, but the table is joined with itself.

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

T1 and T2 are different table aliases for the same table.

Definition 3.20: Concatenation Operator

|| or concatenation operator is use to link columns or character strings. We can also use a literal. A literal is a character, number or date that is included in the SELECT statement.

Definition 3.21: SQL CONCAT Function

The CONCAT() function adds two or more strings together.

```
CONCAT(string1, string2, ..., string_n)
```

Example 3.11: Employees and Supervisors

We will write a query that returns information about each employee and their supervisor. The report will include employees even if they do not report to another employee. The report will be sorted alphabetically by the employee_name column. The query will return the following columns, in order:

- ***employee_name*** *- containing the first_name and last_name columns separated by a space (eg Luke Skywalker).*
- ***employee_title*** *- the title of that employee.*
- ***supervisor_name*** *- the first and last name of the person the employee reports to, in the same format as*

employee_name.
• ***supervisor_title*** *- the title of the person the employee reports to.*

```
# CB 1.4.10 #

SELECT
    e1.first_name || " " || e1.last_name employee_name,
    e1.title employee_title,
    e2.first_name || " " || e2.last_name supervisor_name,
    e2.title supervisor_title
FROM employee e1
LEFT JOIN employee e2 ON e1.reports_to = e2.employee_id
ORDER BY employee_name
```

Output
[8 rows x 4 columns]

employee_name	employee_title	supervisor_name	supervisor_title
Andrew Adams	General Manager		
Jane Peacock	Sales Support Agent	Nancy Edwards	Sales Manager
Laura Callahan	IT Staff	Michael Mitchell	IT Manager
Margaret Park	Sales Support Agent	Nancy Edwards	Sales Manager
Michael Mitchell	IT Manager	Andrew Adams	General Manager
Nancy Edwards	Sales Manager	Andrew Adams	General Manager
Robert King	IT Staff	Michael Mitchell	IT Manager
Steve Johnson	Sales Support Agent	Nancy Edwards	Sales Manager

Figure 3.14: The Output of Query CB 1.4.10

Definition 3.22: SQL LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
There are two wildcards often used in conjunction with the LIKE operator:
% - The percent sign represents zero, one, or multiple characters.
_ - The underscore represents a single character.

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

Definition 3.23: SQL CASE Statement

The CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL.

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END
```

```
AS [new_column_name]
```

Example 3.12: Customer Spending Habits

We will write a query that summarizes the purchases of each customer. For the purposes of this exercise, we do not have any two customers with the same name. The query will include the following columns, in order:

- ***customer_name*** *- containing the first_name and last_name columns separated by a space (eg Luke Skywalker).*
- ***number_of_purchases*** *- counts the number of purchases made by each customer.*
- ***total_spent*** *- the total sum of money spent by each customer.*
- ***customer_category*** *- a column that categorizes the customer based on their total purchases. The column should contain the following values:*
  - *small spender - If the customer's total purchases are less than $40.*
  - *big spender - If the customer's total purchases are greater than $100.*
  - *regular - If the customer's total purchases are between $40 and $100 (inclusive).*

Results will be ordered by the customer_name column.

```
# CB 1.4.11 #

SELECT
    c.first_name || "␣" || c.last_name customer_name,
    COUNT(i.invoice_id) number_of_purchases,
    SUM(i.total) total_spent,
    CASE
        WHEN sum(i.total) < 40 THEN "small␣spender"
        WHEN sum(i.total) > 100 THEN "big␣spender"
        ELSE "regular"
        END
        AS customer_category
FROM customer c
LEFT JOIN invoice i ON c.customer_id = i.customer_id
GROUP BY customer_name
ORDER BY customer_name
```

Output
[59 rows x 4 columns]

customer_name	number_of_purchases	total_spent	customer_category
Aaron Mitchell	8	70.28999999999999	regular
Alexandre Rocha	10	69.3	regular
Astrid Gruber	9	69.3	regular
Bjørn Hansen	9	72.27000000000001	regular
Camille Bernard	9	79.2	regular
Daan Peeters	7	60.38999999999999	regular
Dan Miller	12	95.03999999999999	regular
Diego Gutiérrez	5	39.6	small spender
Dominique Lefebvre	9	72.27	regular
Eduardo Martins	12	60.39	regular
Edward Francis	13	91.08	regular
Ellie Sullivan	12	75.24000000000001	regular
Emma Jones	8	68.31	regular
Enrique Muñoz	11	98.01	regular

Figure 3.15: The Output of Query CB 1.4.11

3.2.3 Building and Organizing Complex Queries

Query Readability

A little time put into whitespace and capitalization pays off. A few tips to help make your queries more readable:

- *If a select statement has more than one column, put each on a new line, indented from the select statement.*
- *Always capitalize SQL function names and keywords*
- *Put each clause of your query on a new line.*
- *Use indenting to make subqueries appear logically separate.*

Another important consideration when writing readable queries is the use of alias names and shortcuts. Name aliases should be clear– a common convention is using the first letter of the table name, however if you feel that a query is complex you should consider using more explicit aliases. Similarly, at times lines like GROUP BY 1 can be confusing, and explicitly naming the column will make your query more readable. For further information, you can also consult [14].

Definition 3.24: SQL WITH Statement

WITH clauses allow you to define one or more named subqueries before the start of the main query. The main query then refers to the subquery by it's alias name, just as if it's a table in the database.

Syntax:
*WITH [alias_name] **AS** ([subquery])*

***SELECT** [main_query]*

To create multiple subqueries with the WITH statement can be done by

WITH
*[alias_name] **AS** ([subquery]),*

```
    [alias_name_2] AS ([subquery_2]),
    [alias_name_3] AS ([subquery_3])

SELECT [main_query]
```

While each subquery can be independent, we can actually use the result of the first subquery in subsequent subqueries, and so on. This can be a useful way of building readable complex queries.

Definition 3.25: SQL CREATE VIEW

In SQL, a view is a virtual table based on the result-set of an SQL statement. A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

If you wish to add the view to a database, one can use

```
CREATE VIEW database_name.view_name AS
    SELECT * FROM database_name.table_name;
```

Definition 3.26: SQL DROP VIEW Command

A view is deleted with the DROP VIEW command.

Syntax:

```
DROP VIEW view_name;
```

Example 3.13

We will create a view called customer_gt_90_dollars:
The view will only contain the columns from customers, in their original order.
The view will only contain customers who have purchased more than $90 in tracks from the store.
After the SQL query that creates the view, we will write a second query to display the newly created view.

```
# CB 1.4.12 #

CREATE VIEW chinook.customer_gt_90_dollars AS
    SELECT c.* FROM invoice i
    LEFT JOIN customer c ON i.customer_id = c.customer_id
    GROUP BY i.customer_id
    HAVING SUM(i.total) > 90;

SELECT * FROM chinook.customer_gt_90_dollars;
```

Output

[18 rows x 13 columns]

customer_id	first_name	last_name	company	address
1	Luís	Gonçalves	Embraer - Empresa Brasileira de Aeron...	Av. Brigadeiro Faria
3	François	Tremblay		1498 rue Bélanger
5	František	Wichterlová	JetBrains s.r.o.	Klanova 9/506
6	Helena	Holý		Rilská 3174/6
13	Fernanda	Ramos		Qe 7 Bloco G
17	Jack	Smith	Microsoft Corporation	1 Microsoft Way
20	Dan	Miller		541 Del Medio Avenue
21	Kathy	Chase		801 W 4th Street
22	Heather	Leacock		120 S Orange Ave
30	Edward	Francis		230 Elgin Street
34	João	Fernandes		Rua da Assunção 53
37	Fynn	Zimmermann		Berger Straße 10
42	Wyatt	Girard		9, Place Louis Barth

Figure 3.16: The Output of Query CB 1.4.12

Definition 3.27: SQL UNION Operator

The UNION operator is used to combine the result-set of two or more SELECT statements.

- *Each SELECT statement within UNION must have the same number of columns.*
- *The columns must also have similar data types.*
- *The columns in each SELECT statement must also be in the same order.*

Syntax:
[select_statement_one]
UNION
[select_statement_two]

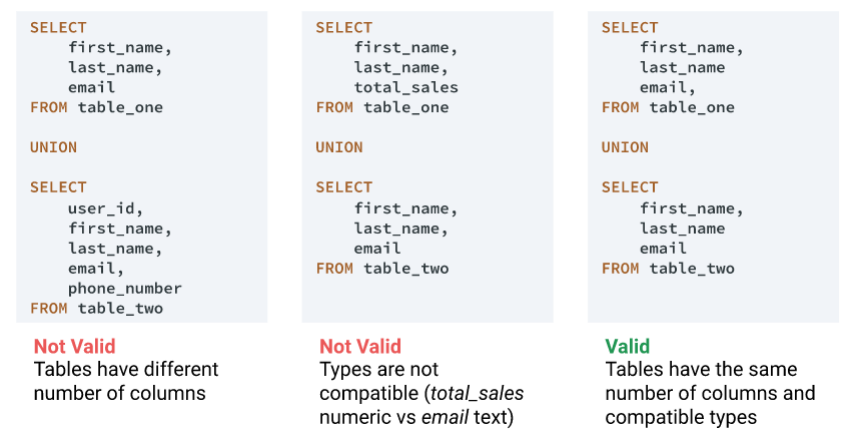

Figure 3.17: Example of SQL Union Validity

Definition 3.28: SQL INTERSECT Operator

The SQL INTERSECT operator is used to return the results of 2 or more SELECT statements. However, it only returns the rows selected by all queries or data sets. If a record exists in one query and not in the other, it will be omitted from the INTERSECT results.

```
Syntax:
[select_statement_one]
INTERSECT
[select_statement_two]
```

Definition 3.29: SQL EXCEPT Operator

Selects rows that occur in the first statement, but don't occur in the second statement.

```
Syntax:
[select_statement_one]
INTERSECT
[select_statement_two]
```

Example 3.14

We will write a query that works out how many customers that are in the USA and have purchased more than $90 are assigned to each sales support agent. For the purposes of this exercise, no two employees have the same name.
The result will have the following columns, in order:

- *employee_name - The first_name and last_name of the employee separated by a space.*
- *customers_usa_gt_90 - The number of customer assigned to that employee that are both from the USA and have have purchased more than $90 worth of tracks.*

The result will include all employees with the title "Sales Support Agent", but not employees with any other title. Lastly, the results will be ordered by the employee_name column.

```
# CB 1.4.13 #

WITH customers_usa_gt_90 AS
    (
     SELECT * FROM customer_usa

     INTERSECT

     SELECT * FROM customer_gt_90_dollars
    )

SELECT
    e.first_name || " " || e.last_name employee_name,
    COUNT(c.customer_id) customers_usa_gt_90
FROM employee e
LEFT JOIN customers_usa_gt_90 c ON c.support_rep_id = e.employee_id
```

```
WHERE e.title = 'Sales Support Agent'
```

Output

[3 rows x 2 columns]

employee_name	customers_usa_gt_90
Jane Peacock	0
Margaret Park	2
Steve Johnson	2

Figure 3.18: The Output of Query CB 1.4.13

Example 3.15

We will create a query to find the customer from each country that has spent the most money at the store, ordered alphabetically by country. The query will return the following columns, in order:

- *country - The name of each country that we have a customer from.*
- *customer_name - The first_name and last_name of the customer from that country with the most total purchases, separated by a space (eg Luke Skywalker).*
- *total_purchased - The total dollar amount that customer has purchased.*

```
# CB 1.4.14 #

WITH
    customer_totals AS
    (
     SELECT
         SUM(i.total) purchase_total,
          c.*
     FROM invoice i
     INNER JOIN customer c ON c.customer_id = i.customer_id
     GROUP BY c.customer_id
    ),
    customer_max AS
    (
    SELECT
        MAX(purchase_total) max_purchase,
         ct.country country,
         ct.first_name || " " || ct.last_name customer_name
    FROM customer_totals ct
    GROUP BY ct.country
    )

SELECT
    cm.country country,
    cm.customer_name customer_name,
    cm.max_purchase total_purchased
FROM customer_max cm
ORDER BY country
```

Output

[24 rows x 3 columns]

country	customer_name	total_purchased
Argentina	Diego Gutiérrez	39.6
Australia	Mark Taylor	81.18
Austria	Astrid Gruber	69.3
Belgium	Daan Peeters	60.38999999999999
Brazil	Luís Gonçalves	108.89999999999998
Canada	François Tremblay	99.99
Chile	Luis Rojas	97.02000000000001
Czech Republic	František Wichterlová	144.54000000000002
Denmark	Kara Nielsen	37.61999999999999
Finland	Terhi Hämäläinen	79.2
France	Wyatt Girard	99.99
Germany	Fynn Zimmermann	94.05000000000001
Hungary	Ladislav Kovács	78.21
India	Manoj Pareek	111.86999999999999

Figure 3.19: The Output of Query CB 1.4.14

3.2.4 Querying SQLite from Python

Definition 3.30: SQLite Connect Function

Once we import the module, we connect to the database we want to query using the connect() function. This function requires a single parameter, which is the database we want to connect to. The connect() function returns a Connection instance, which maintains the connection to the database we want to work with.

Definition 3.31: SQLite Cursor Class

The Cursor class is a SQLite class that allows us to interact with the database such as executing SQL statements and fetching the next row of a query result set [See [15] for further information]. We will use the Cursor class to

- *Run a query against the database.*
- *Parse the results from the database.*
- *Convert the results to native Python objects.*
- *Store the results within the Cursor instance as a local variable.*

Executing Queries w/o Cursor Instance

The SQLite library actually allows us to skip creating a Cursor altogether by using the execute method within the Connection object itself. SQLite will create a Cursor instance for us under the hood and run our query against the database, allowing us to skip a step. Suppose that jobs.db is a database with a recent_grads table, then the code could look like:

```
conn = sqlite3.connect("jobs.db")
query = "select * from recent_grads;"
conn.execute(query).fetchall()
```

Definition 3.32: Cursor Fetching Methods

Each Cursor instance contains an internal counter that updates every time we retrieve results. When we call the fetchone() method, the Cursor instance will return a single result, and then increment its internal counter by 1. This means that if we call fetchone() again, the Cursor instance will actually return the second tuple in the results set (and increment by 1 again).

The fetchmany() method takes in an integer (n) and returns the corresponding results, starting from the current position. It then increments the Cursor instance's counter by n.

Example 3.16

In this example, we want to write and run a query that returns the Major and Major_category columns from recent_grads. We then fetch the first five results and store them as five_results.

```
# CB 1.4.15 #

import sqlite3
conn = sqlite3.connect("jobs.db")
cursor = conn.cursor()

query = "SELECT Major, Major_category FROM recent_grads"
five_results = cursor.execute(query).fetchmany(5)
```

```
Variables
▼ five_results  list (<class 'list'>)
    [('PETROLEUM ENGINEERING', 'Engineering'),
     ('MINING AND MINERAL ENGINEERING', 'Engineering'),
     ('METALLURGICAL ENGINEERING', 'Engineering'),
     ('NAVAL ARCHITECTURE AND MARINE ENGINEERING', 'Engineering'),
     ('CHEMICAL ENGINEERING', 'Engineering')]
```

Figure 3.20: The Output of CB 1.4.15

3.2.5 Guided Project: Answering Business Questions Using SQL

Definition 3.33: Context Manager

A context manager is an object that defines the runtime context to be established when executing a with statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the with statement (described in section The with statement), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc. [See [16] for a good article on this].

Definition 3.34: Python With Statement

The with statement is used to wrap the execution of a block with methods defined by a context manager (see section With Statement Context Managers). This allows common try...except...finally usage patterns

to be encapsulated for convenient reuse.

```
with something_that_returns_a_context_manager() as my_resource:
    do_something(my_resource)
```

The execution of the with statement with one "item" proceeds as follows:

1. The context expression (the expression given in the with_item) is evaluated to obtain a context manager.
2. The context manager's __enter__() is loaded for later use.
3. The context manager's __exit__() is loaded for later use.
4. The context manager's __enter__() method is invoked.
5. If a target was included in the with statement, the return value from __enter__() is assigned to it.

(Note: The with statement guarantees that if the __enter__() method returns without an error, then __exit__() will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 6 below.)

6. The suite is executed.
7. The context manager's __exit__() method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to __exit__(). Otherwise, three None arguments are supplied.

The following code

```
with EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

Definition 3.35: Pandas read_sql_query()

pandas.read_sql_query(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, chunksize=None

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result set of the query string. Optionally provide an index_col parameter to use one of the columns as the index, otherwise default integer index will be used [See [17] for further information].

PARAMETERS:
***sql**: str SQL query or SQLAlchemy Selectable (select or text object)*
SQL query to be executed.

***con**: SQLAlchemy connectable(engine/connection), database str URI*
or sqlite3 DBAPI2 connection. Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

Example 3.17: Setting up SQLite in Python

The first task is to import the SQLite, pandas and matplotlib modules, and use the magic command %matplotlib inline to make sure any plots render in the notebook.

1. *Create a run_query() function, that takes a SQL query as an argument and returns a pandas dataframe of that query.*
2. *Create a run_command() function that takes a SQL command as an argument and executes it using the sqlite module.*
3. *Create a show_tables() function that calls the run_query() function to return a list of all tables and views in the database.*
4. *Run the show_tables() function.*

```
# CB 1.4.16 #

%matplotlib inline

import sqlite3
import pandas as pd
import matplotlib.pyplot as plt

def run_query(query):
    with sqlite3.connect('chinook.db') as conn:
        return pd.read_sql_query(query, conn)

def run_command(c):
    with sqlite3.connect('chinook.db') as conn:
        conn.isolation_level = None
        conn.execute(c)

def show_tables():
    query = """SELECT name, type FROM sqlite_master
            WHERE type IN ('table', 'view') """
    tables = run_query(query)
    print(tables)

show_tables()
```

	name	type
0	album	table
1	artist	table
2	customer	table
3	employee	table
4	genre	table
5	invoice	table
6	invoice_line	table
7	media_type	table
8	playlist	table
9	playlist_track	table
10	track	table
11	country_cust	view

Figure 3.21: The Output of CB 1.4.16

Example 3.18: Finding Total Tracks Purchased in Each Genre from the Chinook Database

We want to write a query that returns each genre, with the number of tracks sold in the USA:
- *in absolute numbers.*
- *in percentages.*

We'll create a plot to show this data.

```
# CB 1.4.17 #

query_genre = """
WITH usa_tracks AS
    (
    SELECT
        i.invoice_id invoice_id,
        il.track_id track_id,
        il.quantity
    FROM invoice i
    INNER JOIN invoice_line il ON il.invoice_id = i.invoice_id
    WHERE i.billing_country = 'USA'
    ),
    usa_genre_tracks AS
    (
    SELECT
        g.name genre,
        SUM(ut.quantity) total_tracks_purchased
    FROM usa_tracks ut
    INNER JOIN track t ON t.track_id = ut.track_id
    INNER JOIN genre g ON g.genre_id = t.genre_id
    GROUP BY 1
    )

SELECT
    ugt.*,
    (
    CAST(ugt.total_tracks_purchased AS float) /
    CAST
    (
    (
```

```
    SELECT SUM(ugt.total_tracks_purchased)
    FROM usa_genre_tracks ugt
    )
    AS float
    )
    )*100 total_tracks_percent
FROM usa_genre_tracks ugt
GROUP BY 1
ORDER BY total_tracks_purchased DESC
"""

invoice_line = run_query(query_genre)
print(invoice_line)
```

	genre	total_tracks_purchased	total_tracks_percent
0	Rock	561	53.377735
1	Alternative & Punk	130	12.369172
2	Metal	124	11.798287
3	R&B/Soul	53	5.042816
4	Blues	36	3.425309
5	Alternative	35	3.330162
6	Latin	22	2.093245
7	Pop	22	2.093245
8	Hip Hop/Rap	20	1.902950
9	Jazz	14	1.332065
10	Easy Listening	13	1.236917
11	Reggae	6	0.570885
12	Electronica/Dance	5	0.475737
13	Classical	4	0.380590
14	Heavy Metal	3	0.285442
15	Soundtrack	2	0.190295
16	TV Shows	1	0.095147

Figure 3.22: The Output of CB 1.4.17

We can use the built-in pandas plotting methods to produce a plot for this data. We do this below.

```
# CS 1.4.18 #

invoice_line.plot(x = 'genre', y = 'total_tracks_purchased',
                  title = "Total Number of Tracks Sold vs Genre (USA)",
                  kind = 'bar', legend = None)
```

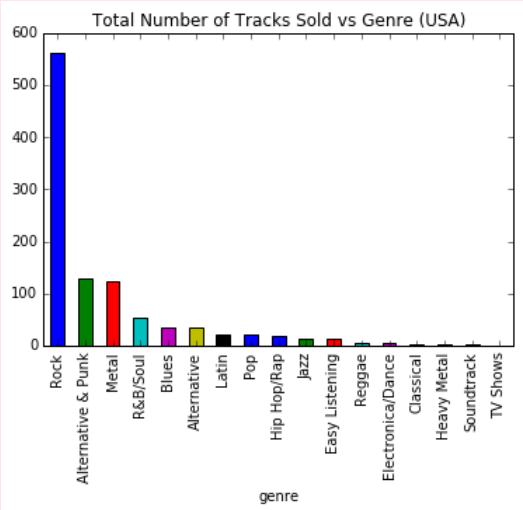

Figure 3.23: The Output of CS 1.4.18

Example 3.19: Finding Transaction Details for Each Country from the Chinook Database

We will write a query that collates data on purchases from different countries. When a country has only one customer, we will collect them into an "Other" group.
The results should be sorted by the total sales from highest to lowest, with the "Other" group at the very bottom.

For each country, we will include:
- *total number of customers.*
- *total value of sales.*
- *average value of sales per customer.*
- *average order value.*

```
# CB 1.4.19 #

c = """
CREATE VIEW country_cust AS
    SELECT
        c.first_name || " " || c.last_name customer_name,
        COUNT(i.customer_id) customer_count,
        c.country country,
        SUM(i.total) total_spent
    FROM customer c
    INNER JOIN invoice i ON i.customer_id = c.customer_id
    GROUP BY 1, 1
    ORDER BY customer_count DESC
"""

d = "DROP VIEW country_cust"

run_command(d)
```

```
run_command(c)

query = """
With country_info AS
    (
    SELECT
        cc.country,
        SUM(cc.customer_count) orders,
        Count(cc.customer_name) unique_customers,
        Sum(total_spent) total_sales
    FROM country_cust cc
    GROUP BY cc.country
    ORDER BY total_sales
    ),
    other_info AS
    (
    SELECT
        CASE
            WHEN ci.unique_customers = 1 THEN "Other"
            ELSE 0
        END
        AS country,
        SUM(ci.orders) orders,
        Count(ci.unique_customers) unique_customers,
        SUM(ci.total_sales) total_sales
    FROM country_info ci
    WHERE ci.unique_customers = 1
    ),
    country_other_info AS
    (
    SELECT *
    FROM
        (
        SELECT
            coi.*,
            CASE
                WHEN coi.country = "Other" THEN "1"
                ELSE 0
            END
            AS count
        FROM
        (
        SELECT * FROM other_info

        UNION

        SELECT * FROM country_info ci
        WHERE ci.unique_customers != 1
        ) coi
        ORDER BY count ASC
        ) coi_new
    )
```

```
SELECT
    country,
    unique_customers,
    total_sales,
    total_sales / CAST(unique_customers as float) average_sale_per_customer,
    total_sales / CAST(orders as float) average_order_value
FROM country_other_info

"""
querydf = run_query(query)
print(querydf)
```

	country	unique_customers	total_sales	average_sale_per_customer	average_order_value
0	Brazil	5	427.68	85.536000	7.011148
1	Canada	8	535.59	66.948750	7.047237
2	Czech Republic	2	273.24	136.620000	9.108000
3	France	5	389.07	77.814000	7.781400
4	Germany	4	334.62	83.655000	8.161463
5	India	2	183.15	91.575000	8.721429
6	Portugal	2	185.13	92.565000	6.383793
7	USA	13	1040.49	80.037692	7.942672
8	United Kingdom	3	245.52	81.840000	8.768571
9	Other	15	1094.94	72.996000	7.448571

Figure 3.24: The Output of CB 1.4.19

The trick in placing the 'Other' column at the very end is to assign some ordered value to 'Other' with all other countries held at some other constant value. This can be seen in the country_other_info subquery in the WITH section of CB 1.4.19.

3.2.6 Table Relations and Normalization

Definition 3.36: SQLite prompt

When you launch the SQLite shell, you will be shown the SQLite prompt, seen below by sqlite>.

```
$ sqlite3 chinook.db
-- Loading resources from /home/dq/.sqliterc
SQLite version 3.21.0 2017-10-24 18:55:49
Enter ".help" for usage hints.
sqlite>
```

Definition 3.37: SQLite Dot Commands

SQLite has a number of dot commands which you can use to help you work with databases. When you use a dot command, you don't need to use a semicolon. Some common dot commands are

- *.help - Displays help text showing all dot commands and their function.*
- *.tables - Displays a list of all tables and views in the current database.*
- *.shell [command] - Run a command like ls or clear in the system shell.*

- *.mode - Allows us to select from a few different display modes. We'll use .mode column to allow for easier to read outputs.*
- *.quit - Quits the SQLite shell.*

Example 3.20: Querying on SQLite Shell

We can send query instructions to the SQLite shell so that we can interact with the database.

```
sqlite> SELECT * FROM album LIMIT 10;
album_id    title                                artist_id
----------  -----------------------------------  ----------
1           For Those About To Rock We Salute You  1
2           Balls to the Wall                    2
3           Restless and Wild                    2
4           Let There Be Rock                    1
5           Big Ones                             3
6           Jagged Little Pill                   4
7           Facelift                             5
8           Warner 25 Anos                       6
9           Plays Metallica By Four Cellos       7
10          Audioslave                           8
sqlite>
```

Figure 3.25: SQLite Shell Query

Definition 3.38: CREATE TABLE

The CREATE TABLE statement is used to create a new table in a database.

```
CREATE TABLE [table_name] (
    [column1_name] [column1_type],
    [column2_name] [column2_type],
    [column3_name] [column3_type],
    [...]
);
```

Each column in SQLite must have a type. While some database systems have as many as 50 distinct data types, SQLite uses only 5 behind the scenes:

- *TEXT*
- *INTEGER*
- *REAL*
- *NUMERIC*
- *BLOB*

Type	Commonly Used For	Equivalent Types

`TEXT`	Names
	Email Addresses
	Dates and Times
	Phone Numbers
		`CHARACTER`
		`VARCHAR`
		`NCHAR`
		`NVARCHAR`
		`DATETIME`

`INTEGER`	IDs
	Quantities
		`INT`
		`SMALLINT`
		`BIGINT`
		`INT8`

`REAL`	Weights Averages	`DOUBLE` `FLOAT`
`NUMERIC`	Prices Statuses	`DECIMAL` `BOOLEAN`
`BLOB`	Binary Data	`BLOB`

Figure 3.26: SQLite Types and Their Common Uses

Definition 3.39: Primary Key & Foreign Key

*A **primary key** is a unique identifier for each row - you cannot have two rows in a table with the same value for the primary key column(s).*

*When two tables have a relation, there will be a column in one table that is a primary key in another table. For instance, from the Chinook database, in the invoice_line table, the invoice_id column is the primary key from the invoice table. This is known as a **foreign key**. By defining a foreign key, our database engine will prevent us from adding rows where the foreign key value doesn't exist in the other table, which helps to prevent errors in our data (note that by default SQLite doesn't force foreign key constraints). If we wanted create a table called user with the user_id column serving as the primary key, we would use the syntax:*

```
CREATE TABLE user (
    user_id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT
);
```

Let's say we wanted to create a new table purchase which tracks basic information about a purchase made by one of our users. Our create statement might look like this:

```
CREATE TABLE purchase (
    purchase_id INTEGER PRIMARY KEY,
    user_id INTEGER,
    purchase_date TEXT,
    total NUMERIC,
    FOREIGN KEY (user_id) REFERENCES user(user_id)
);
```

By adding a FOREIGN KEY clause, we can define one of our columns as a foreign key and specify the table and column that it references.

Definition 3.40: Database Normalization

When we created our wishlist table, we didn't include a track_id column to store which tracks are in the users wishlist. To understand why, let's take a look at what the table might look like if we stored all the data in a single table.

wishlist_id	customer_id	name	track_id
1	34	Joao's awesome wishlist	1158
1	34	Joao's awesome wishlist	2646
1	34	Joao's awesome wishlist	1990
2	18	Amy loves pop	3272
2	18	Amy loves pop	3470

Figure 3.27: Wishlist Table with track_id Column

There are are some drawbacks to storing the data this way:

- ***Data Duplication*** *- we are storing the name of each wishlist multiple times.*
- ***Data Modification*** *- If we want to change the name of one of the wishlists, we have to modify multiple rows.*
- ***Data Integrity*** *- There is nothing to stop a row being added with the wrong wishlist name, and if that happened we wouldn't know which was the correct name.*

The process of optimizing the design of databases to minimize these issues is called database normalization [See [18]].

Definition 3.41: Compound Primary Key

When two or more columns combine to form a primary key it is called a compound primary key. To create a compound primary key, you use the PRIMARY KEY clause:

```
CREATE TABLE [table_name] (
    [column_one_name] [column_one_type],
    [column_two_name] [column_two_type],
    [column_three_name] [column_three_type],
    PRIMARY KEY (column_one_name, column_two_name)
);
```

Definition 3.42: SQL INSERT INTO

The INSERT INTO statement is used to insert new records in a table. To add rows to a SQL table, we'll use the INSERT statement:

```
INSERT INTO [table_name] (
    [column1_name],
    [column2_name],
    [column3_name]
) VALUES (
    [value1],
    [value2],
    [value3]
);
```

If you are inserting values into every column in a table, you don't need to list the column names:

```
INSERT INTO [table_name] VALUES ([value1], [value2], [value3]);
```

Additionally, you can insert multiple rows in a single statement:

```
INSERT INTO [table_name]
VALUES
    ([value1], [value2], [value3]),
    ([value4], [value5], [value6]),
    [...]
```

Definition 3.43: SQL DELETE Statement

The DELETE statement is used to delete existing records in a table.

```
DELETE FROM table_name WHERE condition;
```

If you omit the WHERE statement, SQL will delete all rows from the table.

Definition 3.44: SQL ALTER TABLE

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
The ALTER TABLE statement is also used to add and drop various constraints on an existing table. To add a column, we can use the following syntax:

```
ALTER TABLE [table_name]
ADD COLUMN [column_name] [column_type];
```

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Definition 3.45: SQL UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

The WHERE clause is optional, and can contain any expression that would be valid in a SELECT statement.

Example 3.21

We will first launch the SQLite shell and connect to the chinook.db database. This will be followed by adding two new columns, with values, to the invoice table:

- *tax, with type NUMERIC.*
  The value for all existing rows should be 0.
- *subtotal, with type NUMERIC.*
  The value for each row should be the same as that row's value for total.

We will then quit the SQLite shell.

```
# CB 1.4.20 #

$sqlite3 chinook.db
sqlite> ALTER TABLE invoice
   ...> ADD COLUMN tax NUMERIC;
sqlite> ALTER TABLE invoice
   ...> ADD COLUMN subtotal NUMERIC;
sqlite> UPDATE invoice
   ...> SET tax = 0, subtotal = total;
sqlite> SELECT
   ...>     invoice_id,
   ...>     subtotal,
   ...>     tax,
   ...>     total
   ...> FROM invoice
   ...> LIMIT 5;
sqlite> .quit
```

invoice_id	subtotal	tax	total
1	15.84	0	15.84
2	9.9	0	9.9
3	1.98	0	1.98
4	7.92	0	7.92
5	16.83	0	16.83

Figure 3.28: The Output of the SELECT Query in CB 1.4.20

3.3 SQL and Databases: Advanced

3.3.1 Using PostgreSQL

Definition 3.46: PostgreSQL

PostgreSQL, also known as Postgres, is a free and open-source relational database management system (RDBMS) emphasizing extensibility and technical standards compliance. It is designed to handle a range of workloads, from single machines to data warehouses or Web services with many concurrent users. It is the default database for macOS Server, and is also available for Linux, FreeBSD, OpenBSD, and Windows.

PostgreSQL features transactions with Atomicity, Consistency, Isolation, Durability (ACID) proper-

ties, automatically updatable views, materialized views, triggers, foreign keys, and stored procedures.

Definition 3.47: Psycopg

Psycopg is the most popular PostgreSQL adapter for the Python programming language. Its core is a complete implementation of the Python DB API 2.0 specifications. Several extensions allow access to many of the features offered by PostgreSQL.

SQL Transactions

With PostgreSQL, we're dealing with multiple users who could be changing the database at the same time. If one of these fail, it may leave the database in a state that doesn't match the intention of some set of queries.

Transactions prevent this type of behavior by ensuring that all the queries in a transaction block are executed at the same time. If any of the transactions fail, the whole group fails, and no changes are made to the database at all.

Whenever we open a Connection in psycopg2, a new transaction will automatically be created. All queries run up until the commit method is called will be placed into the same transaction block. When commit is called, the PostgreSQL engine will run all the queries at once.

If we don't want to apply the changes in the transaction block, we can call the rollback method to remove the transaction. Not calling either commit or rollback will cause the transaction to stay in a pending state, and will result in the changes not being applied to the database.

Definition 3.48: Psycopg commit() and rollback() Methods

commit()
Commit any pending transaction to the database. [See [19] for further information].

rollback()
Roll back to the start of any pending transaction. Closing a connection without committing the changes first will cause an implicit rollback to be performed.

Example 3.22

We want to first connect to the dq database as the user dq.
We will then write a SQL query that creates a table called notes in the dq database, with the following columns and data types:

- *id: integer data type, and is a primary key.*
- *body: text data type.*
- *title: text data type.*

We'll execute this query using the execute method.
We will use the commit method on the Connection object to apply the changes in the transaction to the database. We now want to execute a SQL query that inserts a row into the notes table with the following values:

- *id: 1*

- *body: 'Do more missions on Dataquest.'*
- *title: 'Dataquest reminder'.*

We will then execute a SQL query that selects all of the rows from the notes table. We will finally fetch all of the results, print them out and end it off with committing to changes and close the Connection.

```
# CB 1.4.21 #

import psycopg2

conn = psycopg2.connect("dbname = dq user = dq")
cur = conn.cursor()
query = """
CREATE TABLE notes (
id integer PRIMARY KEY,
body TEXT,
title TEXT);
"""
cur.execute(query)
conn.commit()
query = """
INSERT INTO notes (
id, body, title )
VALUES (
1, 'Do more missions on Dataquest.', 'Dataquest reminder');
"""
cur.execute(query)
queryrows = """
SELECT * from notes;
"""
cur.execute(queryrows)
conn.commit()
print(cur.fetchall())
conn.close()
```

Output:

[(1, 'Do more missions on Dataquest.', 'Dataquest reminder')]

Definition 3.49: SQL CREATE DATABASE

The CREATE DATABASE statement is used to create a new SQL database.

***CREATE** DATABASE database_name OWNER database_owner_name*

We can specify the user who will own the database when we create it as well, using the OWNER statement. The database owner is a user that has implied permissions to perform all activities in the database.

Definition 3.50: SQL DROP DATABASE

We can delete a database using the DROP DATABASE statement. The DROP DATABASE statement will immediately remove a database, provided the user executing the query has the right permissions.

```
DROP DATABASE dbName;
```

3.4 APIs and Web Scraping

3.4.1 Working with APIs

Limitations of Working With Entire Databases

While they're popular resources, there are many cases where it's impractical to use one. Here are a few situations where data sets don't work well:

- *The data changes frequently. It doesn't really make sense to regenerate a data set of stock prices, for example, and download it every minute. This approach would require a lot of bandwidth, and be very slow.*
- *You only want a small piece of a much larger data set. Reddit comments are one example. What if you want to pull just your own comments from reddit? It doesn't make much sense to download the entire reddit database, then filter it for a few items.*
- *It involves repeated computation. For example, Spotify has an API that can tell you the genre of a piece of music. You could theoretically create your own classifier and use it to categorize music, but you'll never have as much data as Spotify does.*

Definition 3.51: APIs

In cases like those listed above, an application program interface (API) is the right solution. An API is a set of methods and tools that allows different applications to interact with each other. Programmers use APIs to query and retrieve data dynamically (which they can then integrate with their own apps). A client can retrieve information quickly and effectively through an API.

Definition 3.52: API Endpoints

*An **endpoint** is a server route for retrieving specific data from an API. For example, the /comments endpoint on the reddit API might retrieve information about comments, while the /users endpoint might retrieve data about users*

Definition 3.53: Requests Library

The requests library is the de facto standard for making HTTP requests in Python. It abstracts the complexities of making requests behind a beautiful, simple API so that you can focus on interacting with services and consuming data in your application

Definition 3.54: Requests.get() Method

Makes a request to a web page, and returns the status code.
There are many different types of requests. The most common is a GET request, which we use to retrieve data.

Definition 3.55: Status Codes

Web servers return status codes every time they receive an API request. A status code provides information about what happened with a request. Here are some codes that are relevant to GET requests:

- *200 - Everything went okay, and the server returned a result (if any).*
- *301 - The server is redirecting you to a different endpoint. This can happen when a company switches domain names, or an endpoint's name has changed.*
- *401 - The server thinks you're not authenticated. This happens when you don't send the right credentials to access an API.*
- *400 - The server thinks you made a bad request. This can happen when you don't send the information the API requires to process your request, among other things.*
- *403 - The resource you're trying to access is forbidden; you don't have the right permissions to see it.*
- *404 - The server didn't find the resource you tried to access.*

We can access the status code of a request with the following syntax:

```
response = requests.get(url)
status_code = response.status_code
```

Definition 3.56: JSON Module

Python has a built-in package called json, which can be used to work with JSON data. The JSON library has two main methods:

- *dumps: Takes in a Python object, and converts it to a string*
- *loads: Takes a JSON string, and converts it to a Python object.*

Example 3.23: ISS over San Francisco Request

Get the duration value of the ISS' first pass over San Francisco and assign the value to first_pass_duration. We can get the content of a response as a Python object by using the .json() method on the response.

```
# CB 1.4.? #

import requests

parameters = {"lat": 37.78, "lon": -122.41}
response = requests.get("http://api.open-notify.org/iss-pass.json",
params=parameters)

# Get the response data as a Python object.  Verify that it's a dictionary.
json_data = response.json()
first_pass_duration = json_data['response'][0]['duration']
print(type(json_data))
print(json_data)
```

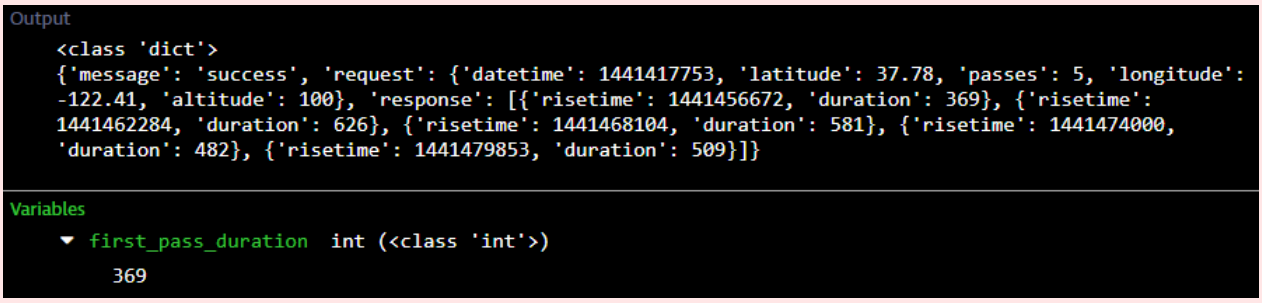

Figure 3.29: The Output of CB 1.4.

3.4.2 Intermediate APIs

Definition 3.57: Access Tokens for APIs

Access Tokens are used in token-based authentication to allow an application to access an API. The application receives an Access Token after a user successfully authenticates and authorizes access, then passes the Access Token as a credential when it calls the target API. The passed token informs the API that the bearer of the token has been authorized to access the API and perform specific actions specified by the scope that was granted during authorization.

Pagination

Sometimes, a request can return a lot of objects. This might happen when you're doing something like listing out all of a user's repositories, for example. Returning too much data will take a long time and slow the server down. For example, if a user has 1,000+ repositories, requesting all of them might take 10+ seconds. This isn't a great user experience, so it's typical for API providers to implement ***pagination****. This means that the API provider will only return a certain number of records per page. You can specify the page number that you want to access. To access all of the pages, you'll need to write a loop.*

Example 3.24

We want to first make an authenticated request to https://api.github.com/users/VikParuchuri/orgs. This will give us a list of the organizations a GitHub user belongs to. We'll then assign the JSON content of the response to orgs (which can be done with response.json()).

```
# CB 1.4.? #

import requests
# Create a dictionary of headers containing our Authorization header.
headers = {"Authorization": "token 1f36137fbbe1602f779300dad26e4c1b7fbab631"}

# Make a GET request to the GitHub API with our headers.
# This API endpoint will give us details about Vik Paruchuri.
response = requests.get("https://api.github.com/users/VikParuchuri/orgs",
headers=headers)
orgs = response.json()
# Print the content of the response. As you can see, this token corresponds
to the account of Vik Paruchuri.
```

```
print(response.json())
```

```
Output
[{'events_url': 'https://api.github.com/orgs/dataquestio/events', 'repos_url':
'https://api.github.com/orgs/dataquestio/repos', 'id': 11148054, 'avatar_url':
'https://avatars.githubusercontent.com/u/11148054?v=3', 'login': 'dataquestio', 'public_members_url':
'https://api.github.com/orgs/dataquestio/public_members{/member}', 'members_url':
'https://api.github.com/orgs/dataquestio/members{/member}', 'description': None, 'url':
'https://api.github.com/orgs/dataquestio'}]
```

Figure 3.30: The Output of CB 1.4.

Definition 3.58: Requests.post() Method

The post() method sends a POST request to the specified url. The post() method is used when you want to send some data to the server. POST requests to send information (instead of retrieve it), and to create objects on the API's server.

Github API POST Requests

Check out GitHub's API documentation for repositories [20] to see a full list of what data we can pass in with this POST request. Here are just a couple data points:

- *name: Required, the name of the repository.*
- *description: Optional, the description of the repository.*

A successful POST request will usually return a 201 status code indicating that it was able to create the object on the server. Sometimes, the API will return the JSON representation of the new object as the content of the response.

Definition 3.59: Requests.patch() & Requests.put() Methods

Sometimes we want to update an existing object, rather than create a new one. This is where PATCH and PUT requests come into play.

We use PATCH requests when we want to change a few attributes of an object, but don't want to resend the entire object to the server. Maybe we just want to change the name of our repository, for example.

We use PUT requests to send the complete object we're revising as a replacement for the server's existing version.

In practice, API developers don't always respect this convention. Sometimes API endpoints that accept PUT requests will treat them like PATCH requests, and not require us to send the whole object back.

Example 3.25: Patch Request to Github Repository

In this example, we want to make a PATCH request to the ***https://api.github.com/repos/VikParuchuri/learning-about-apis*** *endpoint that changes the description to "Learning about requests!". Assign the status code of the response to status.*

```
# CB 1.4.? #

payload = {"description": "Learning about requests!", "name":
"learning-about-apis"}
response = requests.patch("https://api.github.com/repos/VikParuchuri/
learning-about-apis", json=payload, headers=headers)
status = response.status_code
print(response.status_code)
```

Output
200

Definition 3.60: Requests.delete() Method

The final major request type is the DELETE request. The DELETE request removes objects from the server. We can use the DELETE request to remove repositories. A successful DELETE request will usually return a 204 status code indicating that it successfully deleted the object. Use DELETE requests carefully - it's very easy to remove something important by accident.

3.4.3 Challenge: Working with the Reddit API

3.4.4 Web Scraping

Definition 3.61: Web Scraping

A lot of data aren't accessible through data sets or APIs. They may exist on the Internet as Web pages, though. One way to access the data without waiting for the provider to create an API is to use a technique called Web scraping.

Web scraping allows us to load a Web page into Python and extract the information we want. We can then work with the data using standard analysis tools like pandas and numpy.

Definition 3.62: HTML

Hypertext Markup Language (HTML) is the standard markup language for documents designed to be displayed in a web browser. It can be assisted by technologies such as Cascading Style Sheets (CSS) and scripting languages such as JavaScript.

Definition 3.63: Beautiful Soup Library

Beautiful Soup [21] is a Python library designed for quick turnaround projects like screen-scraping. Three features make it powerful:

1. *Beautiful Soup provides a few simple methods and Pythonic idioms for navigating, searching, and*

modifying a parse tree: a toolkit for dissecting a document and extracting what you need. It doesn't take much code to write an application.

2. *Beautiful Soup automatically converts incoming documents to Unicode and outgoing documents to UTF-8. You don't have to think about encodings, unless the document doesn't specify an encoding and Beautiful Soup can't detect one. Then you just have to specify the original encoding.*
3. *Beautiful Soup sits on top of popular Python parsers like lxml and html5lib, allowing you to try out different parsing strategies or trade speed for flexibility.*

Example 3.26: Fetching HTML items with Particular Tag and Class

We consider the following HTML code:

```
<html>
    <head>
        <title>A simple example page</title>
    </head>
    <body>
        <div>
            <p class="inner-text">
                First inner paragraph.
            </p>
            <p class="inner-text">
                Second inner paragraph.
            </p>
        </div>
        <p class="outer-text">
            <b>
                First outer paragraph.
            </b>
        </p>
        <p class="outer-text">
            <b>
                Second outer paragraph.
            </b>
        </p>
    </body>
</html>
```

Figure 3.31: HTML Code for Simple Inner-Outer Paragraph Classes

We want to get the text in the second inner paragraph, and assign the result to second_inner_paragraph_text. We then want to get the text of the first outer paragraph, and assign the result to first_outer_paragraph_text.

```
# CB 1.4.? #

# Get the website that contains classes.
response = requests.get("http://dataquestio.github.io/web-scraping-pages/
simple_classes.html")
content = response.content
parser = BeautifulSoup(content, 'html.parser')

second_inner_paragraph_text = parser.find_all("p", class_ = "inner-text")[1].text
first_outer_paragraph_text = parser.find_all("p", class_ = "outer-text")[0].text
```

```
print(second_inner_paragraph_text)
print(first_outer_paragraph_text)
```

Output:

```
Second inner paragraph.

First outer paragraph.
```

Definition 3.64: CSS

Cascading Style Sheets, or CSS, is a language for adding styles to HTML pages. You may have noticed that our simple HTML pages from the past few screens didn't have any styling; all of the paragraphs had black text and the same font size. Most Web pages use CSS to display a lot more than basic black text. CSS uses selectors to add styles to the elements and classes of elements you specify. You can use selectors to add background colors, text colors, borders, padding, and many other style choices to the elements on HTML pages

Example 3.27: Nesting CSS Selectors for 2014 Superbowl Data

We consider the following HTML code:

```
<html>
    <head lang="en">
        <meta charset="UTF-8">
        <title>2014 Superbowl Team Stats</title>
    </head>
    <body>
        <table class="stats_table nav_table" id="team_stats">
            <tbody>
                <tr id="teams">
                    <th></th>
                    <th>SEA</th>
                    <th>NWE</th>
                </tr>
                <tr id="first-downs">
                    <td>First downs</td>
                    <td>20</td>
                    <td>25</td>
                </tr>
                <tr id="total-yards">
                    <td>Total yards</td>
                    <td>396</td>
                    <td>377</td>
                </tr>
                <tr id="turnovers">
                    <td>Turnovers</td>
                    <td>1</td>
                    <td>2</td>
                </tr>
                <tr id="penalties">
                    <td>Penalties-yards</td>
                    <td>7-70</td>
                    <td>5-36</td>
                </tr>
                <tr id="total-plays">
                    <td>Total Plays</td>
                    <td>53</td>
                    <td>72</td>
                </tr>
                <tr id="time-of-possession">
                    <td>Time of Possession</td>
                    <td>26:14</td>
                    <td>33:46</td>
                </tr>
            </tbody>
        </table>
    </body>
</html>
```

Figure 3.32: HTML for Simple 2014 Superbowl Web Page

We want to find the Total Plays for the New England Patriots, and assign the result to patriots_total_plays_count. We also want to find the Total Yards for the Seahawks, and assign the result to seahawks_total_yards_count. We're going to do this by way of nesting CSS selectors.[a]

```
# CB 1.4.? #

# Get the Superbowl box score data.
response = requests.get("http://dataquestio.github.io/web-scraping-pages/
2014_super_bowl.html")
content = response.content
parser = BeautifulSoup(content, 'html.parser')

patriots_total_plays_count = parser.select("#total-plays_td")[2].text
seahawks_total_yards_count = parser.select("#total-yards_td")[1].text

print(patriots_total_plays_count)
print(seahawks_total_yards_count)
```

Output:
72
396

[a] To specify an HTML item with id *identity* and tag *tag_name* can be done by the selector **#identity tag_name**.

4 Machine Learning Introduction

4.1 Machine Learning Fundamentals

4.1.1 Introduction to K-Nearest Neighbours

Definition 4.1: Nearest Neighbours

In pattern recognition, the k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression.

Definition 4.2: Pandas apply() Method

*DataFrame.apply(self, func, axis=0, raw=False, result_type=None, args=(), **kwds)*

Apply a function along an axis of the DataFrame. Objects passed to the function are Series objects whose index is either the DataFrame's index (axis=0) or the DataFrame's columns (axis=1). By default (result_type=None), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the result_type argument.

Definition 4.3: Pandas sort_values() Method

DataFrame.sort_values(self, by, axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last', ignore_index=False)

Sort by the values along either axis [See [22] for further documentation].

PARAMETERS:
by: *str or list of str*
Name or list of names to sort by.

- *If axis is 0 or 'index' then by may contain index levels and/or column labels.*
- *If axis is 1 or 'columns' then by may contain column levels and/or index labels.*

Changed in version 0.23.0: Allow specifying index or column level names.

Example 4.1: Predicting AirBnB Rental Price with Nearest Neighbours

*The **dc_listings** dataframe has some of the following column information:*

- ***accommodates**: the number of guests the rental can accommodate*
- ***price**: nightly price for the rental*

We will write a function named predict_price that can use the k-nearest neighbors machine learning technique to calculate the suggested price for any value for accommodates. This function should:

- *Take in a single parameter, new_listing, that describes the number of bedrooms.*
- *Code has been added that assigns dc_listings to a new Dataframe named temp_df. We used the pandas.DataFrame.copy() method so the underlying dataframe is assigned to temp_df, instead of just a reference to dc_listings.*

- *Calculate the distance between each value in the accommodates column and the new_listing value that was passed in. Assign the resulting Series object to the distance column in temp_df.*
- *Sort temp_df by the distance column and select the first 5 values in the price column. Don't randomize the ordering of temp_df.*
- *Calculate the mean of these 5 values and use that as the return value for the entire predict_price function.*
- *Use the predict_price function to suggest a price for a living space that:*
  - *accommodates 1 person, assign the suggested price to acc_one.*
  - *accommodates 2 people, assign the suggested price to acc_two.*
  - *accommodates 4 people, assign the suggested price to acc_four.*

```
# CB 1.6.1 #

dc_listings = pd.read_csv('dc_airbnb.csv')
stripped_commas = dc_listings['price'].str.replace(',', '')
stripped_dollars = stripped_commas.str.replace('$', '')
dc_listings['price'] = stripped_dollars.astype('float')
dc_listings = dc_listings.loc[np.random.permutation(len(dc_listings))]

def predict_price(new_listing):
    temp_df = dc_listings.copy()
    temp_df['distance'] = temp_df['accommodates'].apply(lambda x:
    np.abs(x - new_listing))
    temp_df = temp_df.sort_values('distance')
    new_listing = temp_df['price'].head(5).mean()

    return(new_listing)

acc_one = predict_price(1)
acc_two = predict_price(2)
acc_four = predict_price(4)
print(acc_one)
print(acc_two)
print(acc_four)

Output:
68.0
112.8
124.8
```

4.1.2 Evaluating Model Performance

Definition 4.4: Error Metric

We now need a metric that quantifies how good the predictions were on the test set. This class of metrics is called an error metric. As the name suggests, an error metric quantifies how inaccurate our predictions were from the actual values

Definition 4.5: Mean Absolute Error (MAE)

Consider the context of machine learning. Let y denote a $1 \times n$ vector, comprised of the actual values / labels for a given entry and $\hat{y}$ denote the $1 \times n$ vector, comprised of the predicted values / labels for each entry. Let y_k and $\hat{y}_k$ denote the k^{th} entries for y and $\hat{y}$ respectively. Then, the MAE is defined as

$$MAE(y, \hat{y}) = \frac{1}{n} \sum_{k=1}^{n} |y_k - \hat{y}_k| \tag{4.1}$$

Definition 4.6: Mean Squared Error (MSE)

Consider the context of machine learning. Let y denote a $1 \times n$ vector, comprised of the actual values / labels for a given entry and $\hat{y}$ denote the $1 \times n$ vector, comprised of the predicted values / labels for each entry. Let y_k and $\hat{y}_k$ denote the k^{th} entries for y and $\hat{y}$ respectively. Then, the MSE is defined as

$$MSE(y, \hat{y}) = \frac{1}{n}(y - \hat{y})^T(y - \hat{y}) = \frac{1}{n} \sum_{k=1}^{n} (y_k - \hat{y}_k)^2 \tag{4.2}$$

The mean squared error assigns greater penalty towards predictions that are more deviated from the actual values.

Definition 4.7: Root Mean Squared Error (RMSE)

An immediate extension of Mean Squared Error is the Root Mean Sqaured Error (RMSE). Let y and $\hat{y}$ denote two $1 \times n$ vectors. Then, RMSE is defined as

$$RMSE(y, \hat{y}) = \sqrt{MSE(y, \hat{y})} = \sqrt{\frac{1}{n} \sum_{k=1}^{n} (y_k - \hat{y}_k)^2} \tag{4.3}$$

Proposition 4.1: RMSE Bounds

Let $y, \hat{y} \in \mathbb{R}^n$. Then, the following inequalities hold:

$$MAE(y, \hat{y}) \leq RMSE(y, \hat{y}) \leq \sqrt{n}\ MAE(y, \hat{y}) \tag{4.4}$$

Example 4.2: Evaluating Nearest Neighbour Performance with Error Metrics

We use the same predict_price function as in CB 1.6.1 but change the Dataframe that temp_df is assigned to. We will change it from dc_listings to train_df, so only the training set is used.

- *Use the Series method apply to pass all of the values in the bathrooms column from test_df through the predict_price function.*
- *Assign the resulting Series object to the predicted_price column in test_df.*
- *Apply the function to test_df and assign the resulting Series object containing the predicted price values to the predicted_price column in test_df.*
- *Calculate the squared error between the price and predicted_price columns in test_df and assign the resulting Series object to the squared_error column in test_df.*

- *Calculate the mean of the squared_error column in test_df and assign to mse. Take the square root of mse and assign it to rmse.*
- *Use the print function to display the RMSE value.*

```
# CB 1.6.2 #

import pandas as pd
import numpy as np

dc_listings = pd.read_csv("dc_airbnb.csv")
stripped_commas = dc_listings['price'].str.replace(',', '')
stripped_dollars = stripped_commas.str.replace('$', '')
dc_listings['price'] = stripped_dollars.astype('float')
train_df = dc_listings.iloc[0:2792]
test_df = dc_listings.iloc[2792:]

def predict_price(new_listing):
    ## DataFrame.copy() performs a deep copy
    temp_df = train_df.copy()
    temp_df['distance'] = temp_df['bathrooms'].apply(lambda x:
    np.abs(x - new_listing))
    temp_df = temp_df.sort_values('distance')
    nearest_neighbor_prices = temp_df.iloc[0:5]['price']
    predicted_price = nearest_neighbor_prices.mean()

    return predicted_price

test_df['predicted_price'] = test_df['bathrooms'].apply(lambda x:
predict_price(x))
test_df['squared_error'] = (test_df['predicted_price'] - test_df['price'])**(2)
mse = test_df['squared_error'].mean()
rmse = np.sqrt(mse)
print(rmse)

Output:
135.66666532952246
```

4.1.3 Multivariate K-Nearest Neighbors

Definition 4.8: Nominal Data

A nominal scale describes a variable with categories that do not have a natural order or ranking. You can code nominal variables with numbers if you want, but the order is arbitrary and any calculations, such as computing a mean, median, or standard deviation, would be meaningless.

Examples of nominal variables include:

- *genotype*
- *blood type*

- *zip code*
- *race*
- *political party*

Definition 4.9: Ordinal Data

An ordinal scale is one where the order matters but not the difference between values.

Examples of ordinal variables include:

- *socio-economic status ("low income","middle income","high income")*
- *education level ("high school","BS", "MS","PhD")*
- *satisfaction rating ("extremely dislike", "dislike", "neutral", "like", "extremely like").*

Definition 4.10: Scikit-Learn Library

Scikit-learn (formerly scikits.learn and also known as sklearn) is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

Scikit-Learn Workflow

The scikit-learn workflow consists of 4 main steps:

- *instantiate the specific machine learning model you want to use*
- *fit the model to the training data*
- *use the model to make predictions*
- *evaluate the accuracy of the prediction*

Definition 4.11: Scikit-Learn Fit Method

We can fit the model to the data using the fit method. For all models, the fit method takes in 2 required parameters:

- *matrix-like object, containing the feature columns we want to use from the training set.*
- *list-like object, containing correct target values.*

Matrix-like object means that the method is flexible in the input and either a Dataframe or a NumPy 2D array of values is accepted. This means you can select the columns you want to use from the Dataframe and use that as the first parameter to the fit method.

When the fit() method is called, scikit-learn stores the training data we specified within the chosen model instance. If you try passing in data containing missing values or non-numerical values into the fit method, scikit-learn will return an error. Scikit-learn contains many such features that help prevent us from making common mistakes.

Definition 4.12: Scikit-Learn Predict Method

Once we have specified the training data we want to use to make predictions, we can use the predict method to make predictions on the test set. The predict method has only one required parameter:

- *matrix-like object, containing the feature columns from the dataset we want to make predictions on the number of feature columns you use during both training and testing need to match or scikit-learn will return an error.*

Definition 4.13: Scikit-Learn KNeighborsRegressor Class

class sklearn.neighbors.KNeighborsRegressor(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs)

Regression based on k-nearest neighbors.
The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set [See [23] for further documentation].

PARAMETERS:

- ***n_neighbors:*** *int, optional (default = 5)*
  Number of neighbors to use by default for kneighbors queries.
- ***algorithm:*** *{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional*
  Algorithm used to compute the nearest neighbors:
  - *'ball_tree' will use BallTree.*
  - *'kd_tree' will use KDTree.*
  - *'brute' will use a brute-force search.*
  - *'auto' will attempt to decide the most appropriate algorithm based on the values passed to fit method.*
- ***metric:*** *string or callable, default 'minkowski'*
  The distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a Glossary, in which case only "nonzero" elements may be considered neighbors.

Example 4.3: K-Nearest Neighbours for Regression

In this exercise, we want to normalize all of the feature columns in dc_listings and assign the new Dataframe containing just the normalized feature columns to normalized_listings. Afterwards, we will add the price column from dc_listings to normalized_listings.

We will then create a new instance of the KNeighborsRegressor class with the following parameters:

- *n_neighbors: 5*
- *algorithm: brute*
- *metric: euclidean*

We will use all of the columns, except for the price column, to train a k-nearest neighbors model using the above parameters. We will then

- *Use the model to make predictions on the test set and assign the resulting NumPy array of predictions to all_features_predictions.*
- *Calculate the MSE and RMSE values and assign to all_features_mse and all_features_rmse accordingly.*
- *Use the print function to display both error scores.*

```
# CB 1.6.3 #

from sklearn.neighbors import KNeighborsRegressor

normalized_listings = (dc_listings- dc_listings.mean())/dc_listings.std()
normalized_listings['price'] = dc_listings['price']

train_df = normalized_listings.iloc[0:2792]
test_df = normalized_listings.iloc[2792:]

features = train_df.drop('price', axis=1).columns

knn = KNeighborsRegressor(n_neighbors = 5, algorithm = 'brute',
metric = 'euclidean')

knn.fit(train_df[features], train_df['price'])

all_features_predictions = knn.predict(test_df[features])

all_features_mse = mean_squared_error(all_features_predictions, test_df['price'])
all_features_rmse = np.sqrt(all_features_mse)
print(all_features_mse)
print(all_features_rmse)
```

Output:
15455.275631399316
124.31924883701363

Interestingly enough, when we only included the columns {accommodates, bathrooms, bedrooms, number_of_reviews}, we obtained an RMSE value of 115.4, a score lower than what was found when we included all features. Hence, the principle we should really invoke is the following:

> *Select the relevant attributes the model uses to calculate similarity when ranking the closest neighbors,*

which encodes the essence of feature selection.

Definition 4.14: Feature Selection

The process of selecting features to use in a model is known as feature selection. You want to select the relevant attributes for the model to use when computing error metrics such as similarity in closest neighbors.

4.1.4 Hyperparameter Optimization

Definition 4.15: Hyperparameter Optimization

Values that affect the behavior and performance of a model that are unrelated to the data that's used are referred to as hyperparameters. The process of finding the optimal hyperparameter value is known as hyperparameter optimization.

Definition 4.16: Grid Search

A simple but common hyperparameter optimization technique is known as grid search, which involves:

- *selecting a subset of the possible hyperparameter values*
- *training a model using each of these hyperparameter values*
- *evaluating each model's performance*
- *selecting the hyperparameter value that resulted in the lowest error value.*

Example 4.4: Hyperparameter Tuning for K-Nearest Neighbor Regression

While using only the accommodates and bathrooms columns:

- *Train a model for each k value between 1 and 20 using the training data.*
- *Use each model to make predictions on the test set (using just the accommodates and bathrooms columns).*
- *Calculate each model's MSE value by comparing each set of predictions to the true price values.*
- *Find the k value that obtained the lowest MSE value.*
- *Create a dictionary named two_hyp_mse that contains 1 key-value pair:*
  - *key: k value that resulted in lowest MSE value.*
  - *value: corresponding MSE value.*

Repeat this process while using only the accommodates, bathrooms, and bedrooms columns:

- *Create a dictionary named three_hyp_mse that contains 1 key-value pair:*
  - *key: k value that resulted in lowest MSE value.*
  - *value: corresponding MSE value.*

  Display both two_hyp_mse and three_hyp_mse using the print() function.

  Lastly, we will use the scatter() method from matplotlib.pyplot to generate a line plot with:
  - *hyper_params on the x-axis*
  - *mse_values on the y-axis*

  for both two features and three features figures.

```
# CB 1.6.4 #

from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

two_features = ['accommodates', 'bathrooms']
three_features = ['accommodates', 'bathrooms', 'bedrooms']
hyper_params = [x for x in range(1,21)]
# Append the first model's MSE values to this list.
two_mse_values = list()
# Append the second model's MSE values to this list.
three_mse_values = list()
two_hyp_mse = dict()
three_hyp_mse = dict()

def hyper_search(features, hyper_params, mse_values, hyp_mse):

    for hp in hyper_params:
        knn = KNeighborsRegressor(n_neighbors = hp, algorithm = 'brute')
        knn.fit(train_df[features], train_df['price'])
        predictions = knn.predict(test_df[features])
        mse = mean_squared_error(predictions, test_df['price'])
        mse_values.append(mse)

        if hp == hyper_params[0]:
            opt_mse = mse
            opt_hyp = hp
        else:
            if mse < opt_mse:
                opt_hp = hp
                opt_mse = mse
            else:
                continue

    hyp_mse[opt_hp] = opt_mse

    return mse_values, hyp_mse

two_mse_values, two_hyp_mse = hyper_search(two_features, hyper_params,
two_mse_values, two_hyp_mse)
three_mse_values, three_hyp_mse = hyper_search(three_features, hyper_params,
three_mse_values, three_hyp_mse)

print(two_hyp_mse)
print(three_hyp_mse)

fig = plt.figure(figsize = (10,5))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)

ax1.scatter(x = hyper_params, y = two_mse_values)
```

```
ax1.set_title('Two_Features')

ax2.scatter(x = hyper_params, y = three_mse_values)
ax2.set_title('Three_Features')

plt.show()

Output
{5: 14790.314266211606}
{7: 13518.769009310208}
```

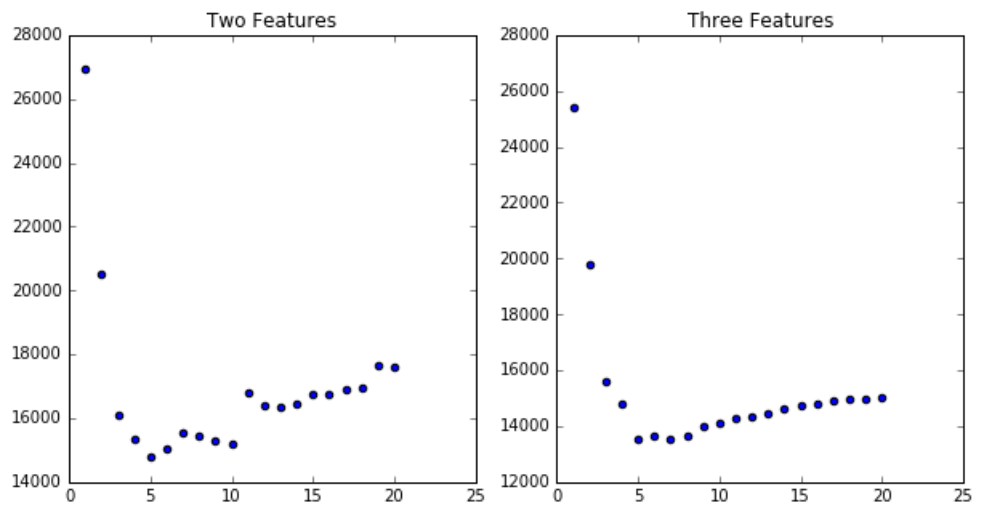

Figure 4.1: The Figure Output of CB 1.6.4

4.1.5 Cross Validation

Definition 4.17: K-Fold Cross Validation

Here's the algorithm from k-fold cross validation:

- *splitting the full dataset into k equal length partitions.*
  - *selecting k-1 partitions as the training set and*
  - *selecting the remaining partition as the test set*
- *training the model on the training set.*
- *using the trained model to predict labels on the test fold.*
- *computing the test fold's error metric.*
- *repeating all of the above steps k-1 times, until each partition has been used as the test set for an iteration.*
- *calculating the mean of the k error values.*

Example 4.5: Five-Fold Cross Validation

In the following figure, we consider what K-Fold cross validation could look like for the $k = 5$ case.

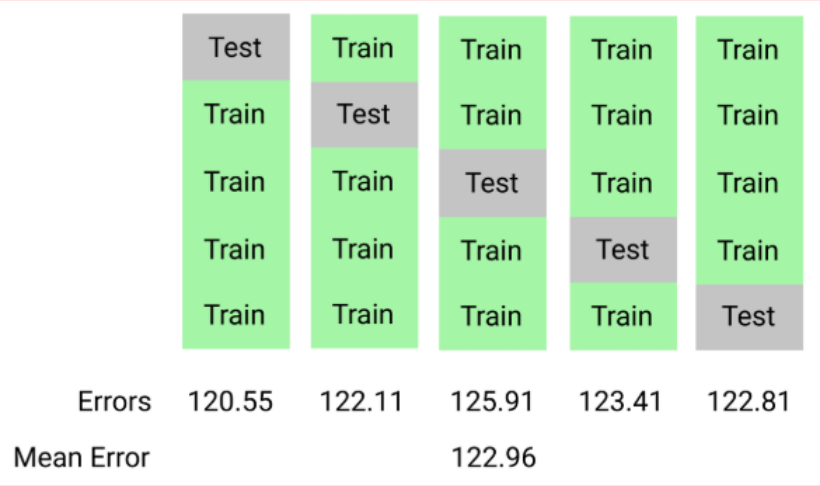

Figure 4.2: 5-Fold Cross Validation

Example 4.6: Custom 5-Fold Cross Validation for KNeighborsRegressor

We begin by partitioning our data into 5 roughly equal parts. To do this, we will add a new column to dc_listings named fold that contains the fold number each row belongs to:

- *Fold 1 should have rows from index 0 up to 745, not including 745.*
- *Fold 2 should have rows from index 745 up to 1490, not including 1490.*
- *Fold 3 should have rows from index 1490 up to 2234, not including 2234.*
- *Fold 4 should have rows from index 2234 up to 2978, not including 2978.*
- *Fold 5 should have rows from index 2978 up to 3723, not including 3723.*

We will now write a function named train_and_validate that takes in a dataframe as the first parameter (df) and a list of fold values (1 to 5 in our case) as the second parameter (folds). This function should:

- *Train n models (where n is number of folds) and perform k-fold cross validation (using n folds). Use the default k value for the KNeighborsRegressor class.*
- *Return a list of RMSE values, where the first element is the RMSE for when fold 1 was the test set, the second element is the RMSE for when fold 2 was the test set, and so on.*

We will then use the train_and_validate function to return the list of RMSE values for the dc_listings Dataframe and assign it to rmses. Lastly, we will calculate the mean of these values and assign to avg_rmse.

```
# CB 1.6.5 #

from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
import numpy as np

fold_ids = [1,2,3,4,5]
```

```
dc_listings.loc[dc_listings.index[0:745], 'fold'] = 1
dc_listings.loc[dc_listings.index[745:1490], 'fold'] = 2
dc_listings.loc[dc_listings.index[1490:2234], 'fold'] = 3
dc_listings.loc[dc_listings.index[2234:2978], 'fold'] = 4
dc_listings.loc[dc_listings.index[2978:3723], 'fold'] = 5

def train_and_validate(df, folds):

    rmses = list()
    model = KNeighborsRegressor()

    for fold in folds:

        train_df = df[df['fold'] != fold]
        test_df = df[df['fold'] == fold]

        model.fit(train_df[['accommodates']], train_df['price'])
        prediction = model.predict(test_df[['accommodates']])

        mse = mean_squared_error(prediction, test_df['price'])
        rmse = np.sqrt(mse)
        rmses.append(rmse)

    return rmses

rmses = train_and_validate(dc_listings, fold_ids)

avg_rmse = np.mean(rmses)

print(rmses, avg_rmse)

Output:
[123.64816897663778, 104.90933995950148, 164.72575286188246, 102.32103626510822,
148.42036980986353]
128.8049335745987
```

Definition 4.18: Sklearn KFold Class

The K-Folds cross-validator provides train/test indices to split data in train/test sets. Split dataset into k consecutive folds (without shuffling by default). Each fold is then used once as a validation while the k - 1 remaining folds form the training set.

```
from sklearn.model_selection import KFold
kf = KFold(n_splits, shuffle=False, random_state=None)
```

PARAMETERS:

- ***n_splits*** *is the number of folds you want to use*
- ***shuffle*** *is used to toggle shuffling of the ordering of the observations in the dataset*
- ***random_state*** *is used to specify the random seed value if shuffle is set to True.*

Definition 4.19: Sklearn cross_val_score() Function

The cross_val_score function evaluates a score by cross-validation [See [24] for further documentation].

```
from sklearn.model_selection import cross_val_score
cross_val_score(estimator, X, Y, scoring=None, cv=None)
```

PARAMETERS:

- ***estimator*** *is a sklearn model that implements the fit method (e.g. instance of KNeighborsRegressor)*
- ***X*** *is the list or 2D array containing the features you want to train on*
- ***y*** *is a list containing the values you want to predict (target column)*
- ***scoring*** *is a string describing the scoring criteria (list of accepted values here)*
- ***cv*** *describes the number of folds. Here are some examples of accepted values:*
  - *an instance of the KFold class,*
  - *an integer representing the number of folds.*

Example 4.7: 5-Fold Cross Validation with KFold and Cross_val_score

We will create a new instance of the KFold class with the following properties:

- *5 folds,*
- *shuffle set to True,*
- *random seed set to 1 (so we can answer check using the same seed), assigned to the variable kf.*

In addition, we will also create a new instance of the KNeighborsRegressor class and assign to knn.

Use the cross_val_score() function to perform k-fold cross-validation:

- *using the KNeighborsRegressor instance knn,*
- *using the accommodates column for training,*
- *using the price column as the target column,*
- *returning an array of MSE values (one value for each fold).*

Assign the resulting list of MSE values to mses. Then, take the absolute value followed by the square root of each MSE value. Then, calculate the average of the resulting RMSE values and assign to avg_rmse.

```
# Cb 1.6.6 #

from sklearn.model_selection import cross_val_score, KFold

kf = KFold(n_splits = 5, shuffle = True, random_state = 1)
knn = KNeighborsRegressor()
mses = cross_val_score(knn, X = dc_listings[['accommodates']], cv = kf,
                       y = dc_listings['price'], scoring = "neg_mean_squared_error")

rmses = np.sqrt(np.abs(mses))
avg_rmse = np.mean(rmses)
```

```
print(avg_rmse)
```

Output:
130.57004998596955

Choosing the right k value when performing k-fold cross validation is more of an art and less of a science. ***Through lots of trial and error, data scientists have converged on 10 as the standard k value.***

4.2 Linear Regression for Machine Learning

4.2.1 The Linear Regression Model

Definition 4.20: Parametric Machine Learning

We need to instead learn about parametric machine learning approaches, like linear regression and logistic regression. Unlike the k-nearest neighbors algorithm, the result of the training process for these machine learning algorithms is a mathematical function that best approximates the patterns in the training set. In machine learning, this function is often referred to as a model.

Example 4.8: Univariate Relationships in AmesHousing Data

In this example, we will read AmesHousing.txt into a dataframe using the tab delimiter (\t) and assign it to 'data'. Afterwards, we will

- *Select the first 1460 rows from data and assign them to 'train'.*
- *Select the remaining rows from data and assign to 'test'.*

To explore the data, we will create a plot using the train dataframe. We will then create a figure with dimensions 15 x 7 containing three scatter plots in a single row:

- *The first plot should plot the 'Garage Area' column on the x-axis against the SalePrice column on the y-axis.*
- *The second one should plot the 'Gr Liv Area' column on the x-axis against the SalePrice column on the y-axis.*
- *The third one should plot the 'Overall Cond' column on the x-axis against the SalePrice column on the y-axis.*

```
# CB 1.6.7 #

import matplotlib.pyplot as plt
import pandas as pd

data = pd.read_csv('AmesHousing.txt', delimiter = '\t')

train = data[0:1460]
test = data[1460:]

fig = plt.figure(figsize=(15,7))

ax1 = fig.add_subplot(1,3,1)
```

```
ax2 = fig.add_subplot(1,3,2)
ax3 = fig.add_subplot(1,3,3)

train.plot(x = 'Garage Area', y = 'SalePrice', ax = ax1, kind = 'scatter')
train.plot(x = 'Gr Liv Area', y = 'SalePrice', ax = ax2, kind = 'scatter')
train.plot(x = 'Overall Cond', y = 'SalePrice', ax = ax3, kind = 'scatter')

plt.show()
```

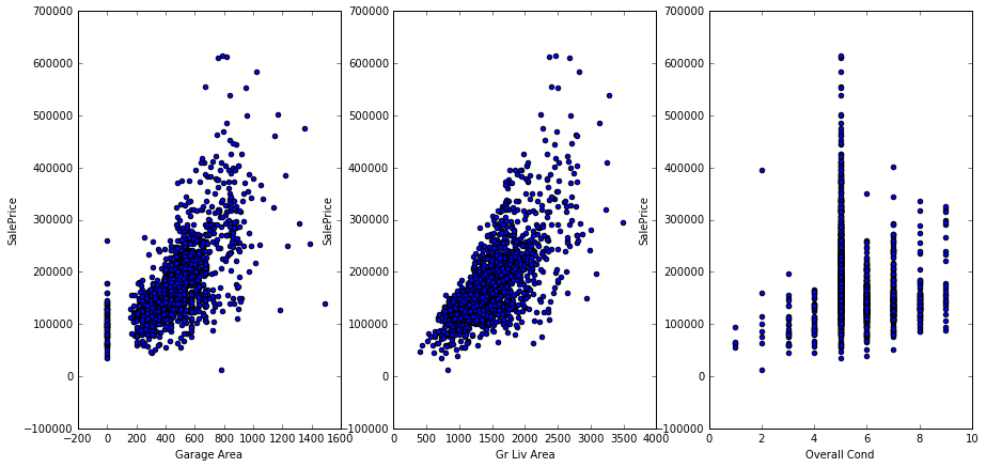

Figure 4.3: The Output Plots of CB 1.6.7

Example 4.9: Multivariate Linear Regression with Scikit-Learn

In this exercise, we will train a linear regression model using the columns 'Overall Cond', 'Gr Liv Area' and assign it to cols. We will then perform the following:

1. *Use the fitted model to make predictions on both the training and test dataset.*
2. *Calculate the RMSE value for the predictions on the training set and assign to train_rmse_2.*
3. *Calculate the RMSE value for the predictions on the test set and assign to test_rmse_2.*
4. *Retrieve the coefficients of the linear regression model and display it.*

```
# CB 1.6.8 #

import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

cols = ['Overall Cond', 'Gr Liv Area']

model = LinearRegression()
model.fit(train[cols], train['SalePrice'])

train_predictions = model.predict(train[cols])
```

```
test_predictions = model.predict(test[cols])

train_rmse_2 = np.sqrt(mean_squared_error(train_predictions, train['SalePrice']))
test_rmse_2 = np.sqrt(mean_squared_error(test_predictions, test['SalePrice']))

a1, a2 = model.coef_
a0 = model.intercept_

print("Linear Model: f(x1,x2) = " + str(a0) + " + " + str(a1) + " x1"
+ " + " + str(a2) + " x2\n")
print(train_rmse_2)
print(test_rmse_2)

Output:
Linear Model: f(x1,x2) = 7858.691146390454 + -409.56846611223847 x1 +
116.73118338867957 x2

56032.39801525867
57066.90779448559
```

4.2.2 Feature Selection

Example 4.10: Correlation Matrix Heatmap

In this exercise, we will read 'AmesHousing.txt' into a dataframe named 'data' with the \t delimiter. Afterwards, we will create a dataframe called 'train', which contains the first 1460 rows of data and a dataframe called 'test', which contains the rest of the rows of data. We only want to keep the numeric columns so we'll select the integer and float columns from train and assign them to the variable 'numerical_train'. We'll drop the following columns from numerical_train:

- *PID (place ID isn't useful for modeling)*
- *Year Built*
- *Year Remod/Add*
- *Garage Yr Blt*
- *Mo Sold*
- *Yr Sold*

We'll calculate the number of missing values from each column in numerical_train. We'll create a Series object where the index is made up of column names and the associated values are the number of missing values. We'll assign this Series object to null_series. Select the subset of null_series to keep only the columns with no missing values, and assign the resulting Series object to full_cols_series.

We'll compute the pairwise correlation coefficients between all of the columns in train_subset and just select the SalePrice column from the resulting data frame. We'll compute the absolute value of each term, sort the resulting Series by the correlation values, and assign to sorted_corrs.

We'll only select the columns in sorted_corrs with a correlation above 0.3 and assign to strong_corrs. We'll filter train_subset using the indexes of strong_corrs and store the correlations to corrmat. Lastly,

we'll use the seaborn.heatmap() function to generate a correlation matrix heatmap for the columns in strong_corrs.

```
# CB 1.6.9 #

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

data = pd.read_csv('AmesHousing.txt', delimiter="\t")
train = data[0:1460]
test = data[1460:]

numerics = ['float', 'int']
numerical_train = train.select_dtypes(include = numerics)

drop_cols = ['PID', 'Year Built', 'Year Remod/Add', 'Garage Yr Blt', 'Mo Sold',
'Yr Sold']
numerical_train = numerical_train.drop(labels = drop_cols, axis = 1)

null_series = numerical_train.isna().sum()
full_cols_series = null_series[null_series == 0]

train_subset = train[full_cols_series.index]
correlation_matrix = train_subset.corr()
sorted_corrs = correlation_matrix['SalePrice'].apply(lambda x: np.abs(x))
.sort_values()

strong_corrs = sorted_corrs[sorted_corrs > 0.3]

corrmat = train_subset[strong_corrs.index].corr()
sns.heatmap(corrmat)
```

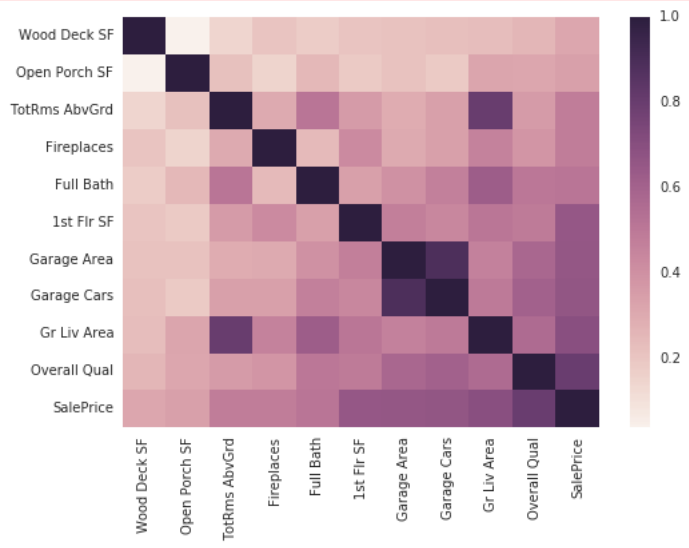

Figure 4.4: The Output Plot of CB 1.6.9

Definition 4.21: Rescaling through Min-Max Normalization

Let $A \subset \mathbb{R}$ be a finite set with at least two distinct values; required so that $max(A) \neq min(A)$. Then, we define a function $f : A \to [0, 1]$ by

$$f(x) = \frac{x - min(A)}{max(A) - min(A)}. \tag{4.5}$$

The listed co-domain is intentional, as to emphasize that $0 \leq f(x) \leq 1 \ \forall x \in A$. This function is known as min-max normalization for a min-max of $[0, 1]$.

Example 4.11: Min-Max Normalization

In this exercise, we select the columns in 'features' from the 'train' data frame. Rescale each of the columns so the values range from 0 to 1, by using train[features] instead of in the formula above. Assign the result to unit_train.

```
# CB 1.6.10 #

unit_train = (train[features] - train[features].min()) /
(train[features].max() - train[features].min())
```

Example 4.12: Training Multivariate Linear Regression Model on Selected Features

In this exercise we'll filter the 'test' data frame so it only contains the columns from final_corr_cols.index. Then, drop the row containing missing values and assign the result to clean_test.

We'll now remove the 'Open Porch SF' feature and build a linear regression model using the remaining features. We will calculate the RMSE on the test and train sets and assign the train RMSE to train_rmse_2 and the test RMSE to test_rmse_2.

```
# CB 1.6.11 #

from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

final_corr_cols = strong_corrs.drop(['Garage Cars', 'TotRms AbvGrd'])
features = final_corr_cols.drop(['SalePrice']).index
target = 'SalePrice'

clean_test = test[final_corr_cols.index].dropna()
features = features.drop('Open Porch SF')

model = LinearRegression()

model.fit(train[features], train['SalePrice'])

test_predictions = model.predict(clean_test[new_features])
train_predictions = model.predict(train[features])

test_rmse_2 = np.sqrt(mean_squared_error(test_predictions,
clean_test['SalePrice']))
train_rmse_2 = np.sqrt(mean_squared_error(train_predictions,
train['SalePrice']))
```

4.2.3 Gradient Descent

Definition 4.22: Model Fitting

In this mission and the next, we'll discuss the two most common ways for finding the optimal parameter values for a linear regression model. Each combination of unique parameter values forms a unique linear regression model, and the process of finding these optimal values is known as ***model fitting****. In both approaches to model fitting, we'll aim to minimize the following function:*

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{4.6}$$

Definition 4.23: Optimization Problems

Mathematical optimization or mathematical programming is the selection of a best element (with regard to some criterion) from some set of available alternatives. Optimization problems of sorts arise in all quantitative disciplines from computer science and engineering to operations research and economics, and the development of solution methods has been of interest in mathematics for centuries.

In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function.

Gradient Descent for Single Parameter Linear Regression Model

Here's an overview of the gradient descent algorithm for a single parameter linear regression model:

- *select initial values for the parameter:* a_1
- *repeat until convergence (usually implemented with a max number of iterations):*
  - *calculate the error (MSE) of model that uses current parameter value:*

$$MSE(a_1) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{4.7}$$

  - *calculate the derivative of the error (MSE) at the current parameter value:* $\frac{d}{da_1}MSE(a_1)$
  - *update the parameter value by subtracting the derivative times a constant (*α*, called the learning rate)*

$$a_1 \leftarrow a_1 - \alpha\frac{d}{da_1}MSE(a_1) \tag{4.8}$$

For the single parameter linear regression model, we define $\hat{y}_i = a_1 x_i$*. The derivative is given as*

$$\frac{d}{da_1}MSE(a_1) = \frac{2}{n}\sum_{i=1}^{n}(a_1 x_i - y_i)x_i \tag{4.9}$$

Proposition 4.2: Univariate Linear Regression for Least Squares

Let $x, y \in \mathbb{R}^n$ *with components* x_i, y_i *and* $f : \mathbb{R} \to \mathbb{R}$ *be defined as*

$$f(u) = \frac{1}{n}\sum_{i=1}^{n}(ux_i - y_i)^2. \tag{4.10}$$

Then, the minimum value is achieved at

$$u_{min} = \frac{y^T x}{x^T x} = \frac{\sum_{i=1}^{n} y_i x_i}{\sum_{i=1}^{n}(x_i)^2} \tag{4.11}$$

Example 4.13: Univariate Gradient Descent

In this exercise, we implement the derivative() function:

- *This function should return the derivative at the current value of* a_1

In addition, our first block of code computes the optimal a_1 *value as determined by Proposition 4.2. We use the 'Gr Liv Area' column for our* x *values. We run the gradient_descent() function and assign the list of iterations for the parameter to param_iterations. We assign the last iteration for* a_1 *to final_param.*

We show that 20 iterations for the starting value of 150 and learning rate of 3×10^{-7} *is numerically sufficient to converge onto the optimal value.*

```
# CB 1.6.12 #

import numpy as np
```

```
x1_array = np.array(train['Gr Liv Area'])
y1_array = np.array(train['SalePrice'])

opt_a1 = np.dot(y1_array.T, x1_array)/np.dot(x1_array.T, x1_array)
print("Optimal a1 value: ", opt_a1)

def derivative(a1, xi_list, yi_list):

    n = len(xi_list)
    deriv = 2/n * np.sum(np.multiply(a1*xi_list-yi_list, xi_list))
    return deriv

def gradient_descent(xi_list, yi_list, max_iterations, alpha, a1_initial):
    a1_list = [a1_initial]

    for i in range(0, max_iterations):
        a1 = a1_list[i]
        deriv = derivative(a1, xi_list, yi_list)
        a1_new = a1 - alpha*deriv
        a1_list.append(a1_new)
    return(a1_list)

max_iterations = 20
param_iterations = gradient_descent(train['Gr Liv Area'], train['SalePrice'],
max_iterations, .0000003, 150)
final_param = param_iterations[-1]
print("a1 parameter after " + str(max_iterations) +
" iterations of Gradient Descent: " + str(final_param))

Output:
Optimal a1 value:  120.14218464950861
a1 parameter after 20 iterations of Gradient Descent: 120.14219147202736
```

Proposition 4.3: Multivariate Linear Regression for Least Squares

Let $x_i, y \in \mathbb{R}^n$ $\forall i \in \mathbb{Z}_p$ and $x_i^{(j)}$ denote the j^{th} component for x_i. Let's define a function $f : \mathbb{R}^{p+1} \to \mathbb{R}$ by

$$f(a_0, a_1, .., a_p) = \frac{1}{n}\sum_{j=1}^{n}(\hat{y}^{(j)} - y^{(j)})^2, \tag{4.12}$$

where

$$\hat{y}^{(j)} = a_0 + \sum_{i=1}^{p} a_i x_i^{(j)}. \tag{4.13}$$

Then, we have that

$$\frac{\partial f}{\partial a_0} = \frac{2}{n}\sum_{j=1}^{n}(\hat{y}^{(j)} - y^{(j)}), \qquad \frac{\partial f}{\partial a_i} = \frac{2}{n}\sum_{j=1}^{n}(\hat{y}^{(j)} - y^{(j)})x_i^{(j)} \quad \forall i \neq 0 \tag{4.14}$$

Example 4.14: Multivariate Gradient Descent

In this exercise we will implement the a0_derivative() function, which implements the gradient for a_0 as well as the a1_derivative() function that implements the gradient for a_1. We also define a gradient_descent() function that keeps updating our a_0, a_1 parameters over some number of iterations given the x_i, y_i, α and initial values.

```
# CB 1.6.13 #

import numpy as np

def a1_derivative(a0, a1, xi_list, yi_list):
    len_data = len(xi_list)
    error = 0
    for i in range(0, len_data):
        error += xi_list[i]*(a0 + a1*xi_list[i] - yi_list[i])
    deriv = 2*error/len_data
    return deriv

def a0_derivative(a0, a1, xi_list, yi_list):

    len_data = len(xi_list)
    deriv = 2/len_data * np.sum(a0 + a1*xi_list - yi_list)
    return deriv

def gradient_descent(xi_list, yi_list, max_iterations, alpha,
a1_initial, a0_initial):
    a1_list = [a1_initial]
    a0_list = [a0_initial]

    for i in range(0, max_iterations):
        a1 = a1_list[i]
        a0 = a0_list[i]

        a1_deriv = a1_derivative(a0, a1, xi_list, yi_list)
        a0_deriv = a0_derivative(a0, a1, xi_list, yi_list)

        a1_new = a1 - alpha*a1_deriv
        a0_new = a0 - alpha*a0_deriv

        a1_list.append(a1_new)
        a0_list.append(a0_new)
    return(a0_list, a1_list)

a0_params, a1_params = gradient_descent(train['Gr Liv Area'], train['SalePrice'],
20, .0000003, 150, 1000)

print("a0: " + str(a0_params[-1]) + ", a1: " + str(a1_params[-1]))
```

Output:
a0: 999.986114052572, a1: 119.53179462379771

4.2.4 Ordinary Least Squares

Proposition 4.4: Ordinary Least Squares

Let X be a $n \times p$ matrix and y be a $n \times 1$ vector. Suppose that X has full column rank, then $X^T X$ is invertible. We define $f : \mathbb{R}^p \to \mathbb{R}$ by

$$f(a) = \frac{1}{n}(Xa - y)^T(Xa - y). \tag{4.15}$$

Then, f is minimized at the point

$$a_{min} = (X^T X)^{-1} X^T y, \tag{4.16}$$

which is known as OLS estimation.

Example 4.15: OLS Estimation

In this exercise we select just the columns in 'features' from the training set and assign to X. Afterwards, we select the SalePrice column from the training set and assign to y. We will then use the OLS estimation formula given by a_{min} in Proposition 4.4 to return the optimal parameter values and store the estimation to the variable ols_estimation.

```
# CB 1.6.14 #

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

data = pd.read_csv('AmesHousing.txt', delimiter="\t")
train = data[0:1460]
test = data[1460:]

features = ['Wood Deck SF', 'Fireplaces', 'Full Bath', '1st Flr SF',
    'Garage Area', 'Gr Liv Area', 'Overall Qual']

X = train[features]
y = train['SalePrice']

train_mat_inv = np.linalg.inv(np.dot(X.T,X))
ols_estimation = np.dot(train_mat_inv, np.dot(X.T, y))
print(ols_estimation)

Output:
[53.75693376, 18232.31375751, -6434.65300989,    22.53151963,
    86.81522574,    28.08976713, 11397.64135314]
```

4.2.5 Processing and Transforming Features

Definition 4.24: Feature Engineering

Feature engineering is the process of using domain knowledge to extract features from raw data via data mining techniques. These features can be used to improve the performance of machine learning algorithms. Feature engineering can be considered as applied machine learning itself

Definition 4.25: Dummy Coding

In statistics and econometrics, particularly in regression analysis, a dummy variable is one that takes only the value 0 or 1 to indicate the absence or presence of some categorical effect that may be expected to shift the outcome.They can be thought of as numeric stand-ins for qualitative facts in a regression model, sorting data into mutually exclusive.

Example 4.16: Categorical Data types and Dummy Coding

In this exercise we will convert all of the text columns in 'train' to the categorical data type. We'll select the Utilities column, return the categorical codes, and display the unique value counts for those codes: train['Utilities'].cat.codes.value_counts().

We will then convert all of the columns in text_cols from the 'train' data frame into dummy columns and delete the original columns from text_cols from the 'train' data frame.

```
# CB 1.6.15 #

import pandas as pd
import numpy as np

text_cols = df_no_mv.select_dtypes(include=['object']).columns

for col in text_cols:
    train[col] = train[col].astype('category')

print(train['Utilities'].cat.codes.value_counts())

for col in text_cols:
    dummy_cols = pd.get_dummies(train[col])
    train = pd.concat([train, dummy_cols], axis = 1)
    del train[col]
```

```
Output:
0     1457
2        2
1        1
```

Imputation

In the next few screens, we'll focus on handling columns with missing values. When values are missing in a column, there are two main approaches we can take:

- *Remove rows containing missing values for specific columns*
  - *Pro: Rows containing missing values are removed, leaving only clean data for modeling*
  - *Con: Entire observations from the training set are removed, which can reduce overall prediction accuracy*
- *Impute (or replace) missing values using a descriptive statistic from the column*
  - *Pro: Missing values are replaced with potentially similar estimates, preserving the rest of the observation in the model.*
  - *Con: Depending on the approach, we may be adding noisy data for the model to learn*

Example 4.17: Imputing Missing Values into Dataset

In this exercise we will only select the columns from 'train' that contain more than 0 missing values but less than 584 missing values and assign the resulting data frame to df_missing_values.

We will then impute the missing values from float_cols with the column's mean. Check for any missing values in float_cols by displaying the number of missing values for each column in df_missing_values.

```
# CB 1.6.16 #

import pandas as pd

data = pd.read_csv('AmesHousing.txt', delimiter="\t")
train = data[0:1460]
test = data[1460:]

train_null_counts = train.isnull().sum()
train_null_miss_cols = train_null_counts[train_null_counts.between(1,583)].index
df_missing_values = train[train_null_miss_cols]

float_cols = df_missing_values.select_dtypes(include=['float'])
float_cols = float_cols.fillna(float_cols.mean())
float_cols.isna().sum()

Output:
Lot Frontage       0
Mas Vnr Area       0
BsmtFin SF 1       0
BsmtFin SF 2       0
Bsmt Unf SF        0
Total Bsmt SF      0
Bsmt Full Bath     0
Bsmt Half Bath     0
Garage Yr Blt      0
dtype: int64
```

4.3 Machine Learning for Python: Intermediate

4.3.1 Logistic Regression

Classification Problem

Linear regression works well when the target column we're trying to predict, the dependent variable, is ordered and continuous. If the target column instead contains discrete values, then linear regression isn't a good fit.

In this mission, we'll explore how to build a predictive model for these types of problems, which are known as classification problems. In classification, our target column has a finite set of possible values which represent different categories a row can belong to. We use integers to represent the different categories so we can continue to use mathematical functions to describe how the independent variables map to the dependent variable

Example 4.18: Fitting a Model for Prediction with LogisticRegression

In this exercise we will import the LogisticRegression class and instantiate a model named logistic_model. We'll use the LogisticRegression method fit to fit the model to the data. We're only interested in constructing a model that uses gpa values to predict admit values. We'll use the LogisticRegression method predict_proba to return the predicted probabilities for the data in the gpa column. We'll assign the returned probabilities to pred_probs.

Create and display a scatter plot using the Matplotlib scatter function where:

- *the x-axis is the values in the gpa column,*
- *the y-axis is the probability of being classified as label 1.*

We'll use the LogisticRegression method predict to return the predicted label for each row in the training set. The parameter for the predict method matches that of the predict_proba method:

- *X: rows of data to use for prediction.*

We'll assign the result to fitted_labels. Lastly, we'll create and display a scatter plot using the Matplotlib scatter function where:

- *the x-axis is the values in the gpa column,*
- *the y-axis is fitted_labels*

```
# CB 1.6.17 #

from sklearn.linear_model import LogisticRegression

logistic_model = LogisticRegression()
logistic_model.fit(admissions[["gpa"]], admissions["admit"])

pred_probs = logistic_model.predict_proba(admissions[['gpa']])
fitted_labels = logistic_model.predict(admissions[['gpa']])

fig = plt.figure(figsize = (10,5))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)
```

```
ax1.scatter(x = admissions['gpa'], y = pred_probs[:,1])
ax1.set_xlabel('GPA')
ax1.set_ylabel('Probability')

ax2.scatter(x = admissions['gpa'], y= fitted_labels)
ax2.set_xlabel('GPA')
ax2.set_ylabel('Admission_Prediction')

plt.show()
```

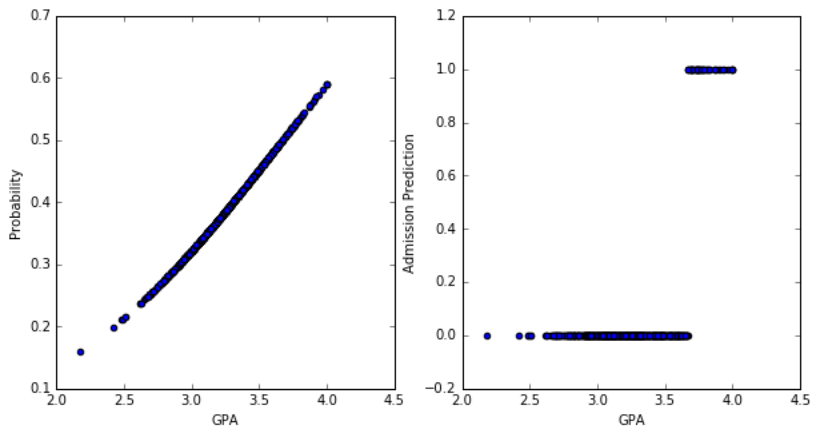

Figure 4.5: The Output Plot of CB 1.6.17

4.3.2 Introduction to Evaluating Binary Classifiers

Definition 4.26: Discrimination Threshold

In logistic regression, recall that the model's output is a probability between 0 and 1. To decide who gets admitted, we set a threshold and accept all of the students where their computed probability exceeds that threshold. This threshold is called the discrimination threshold and scikit-learn sets it to 0.5 by default when predicting labels. If the predicted probability is greater than 0.5, the label for that observation is 1. If it is instead less than 0.5, the label for that observation is 0.

Definition 4.27: Sensitivity / True Positive Rate

Let t_p denote the number of True Positives and f_n denote the number of False Negatives. Then, we define the sensitivity or True Positive Rate of the model as

$$TPR = \frac{t_p}{t_p + f_n} \tag{4.17}$$

Definition 4.28: Specificity / True Negative Rate

Let t_n denote the number of True Negatives and f_p denote the number of False Positives. Then, we define the specificity or True Negative Rate of the model as

$$TNR = \frac{t_n}{t_n + f_p} \tag{4.18}$$

Example 4.19: Sensitivity for a LogisticRegression Model

In this exercise we'll use the LogisticRegression method 'predict' to return the label for each observation in the dataset, admissions. We'll assign the returned list to labels and add a new column to the admissions Dataframe named predicted_label that contains the values from labels. We'll then use the Series method value_counts and the print function to display the distribution of the values in the predicted_label column.

We'll calculate the number of false negatives (where the model predicted rejected but the student was actually admitted) and assign to false_negatives. Finally, we'll compute the sensitivity and assign the computed value to sensitivity.

```
# CB 1.6.18 #

import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression

admissions = pd.read_csv("admissions.csv")
model = LogisticRegression()
model.fit(admissions[["gpa"]], admissions["admit"])

labels = model.predict(admissions[['gpa']])

admissions['predicted_label'] = labels
admissions = admissions.rename(mapper = {'admit': 'actual_label'}, axis = 1)

true_positive_filter = (admissions["predicted_label"] == 1) &
(admissions["actual_label"] == 1)
true_positives = len(admissions[true_positive_filter])

false_negative_filter = (admissions["predicted_label"] == 0) &
(admissions["actual_label"] == 1)
false_negatives = len(admissions[false_negative_filter])

sensitivity = true_positives / (true_positives + false_negatives)
```

Prediction	Observation	
	Admitted (1)	**Rejected (0)**
Admitted (1)	True Positive (TP)	False Positive (FP)
Rejected (0)	False Negative (FN)	True Negative (TN)

Figure 4.6: Table for True/False Positives/Negatives

4.3.3 Multiclass Classification

Definition 4.29: Multiclass Classification

When we have 3 or more categories, we call the problem a multiclass classification problem. There are a few different methods of doing multiclass classification and in this mission, we'll focus on the one-versus-all method.

The one-versus-all method is a technique where we choose a single category as the Positive case and group the rest of the categories as the False case. We're essentially splitting the problem into multiple binary classification problems

Example 4.20: One-vs-All for Car Manufacture Origin Predictions

In the one-vs-all approach, we're essentially converting an n-class (in our case n is 3) classification problem into n binary classification problems. For our case, we'll need to train 3 models:

- *A model where all cars built in North America are considered Positive (1) and those built in Europe and Asia are considered Negative (0).*
- *model where all cars built in Europe are considered Positive (1) and those built in North America and Asia are considered Negative (0).*
- *A model where all cars built in Asia are labeled Positive (1) and those built in North America and Europe are considered Negative (0).*

In this exercise, we will train a logistic regression model for each value in unique_origins. Each model will be trained with the following parameters:

- *X: Dataframe containing just the cylinder & year binary columns.*
- *y: list (or Series) of Boolean values:*
  - *True if observation's value for origin matches the current iterator variable.*
  - *False if observation's value for origin doesn't match the current iterator variable.*

We will add each model to the models dictionary with the following structure:

- *key: origin value (1, 2, or 3),*
- *value: relevant LogisticRegression model instance.*

```
# CB 1.6.19 #

from sklearn.linear_model import LogisticRegression

unique_origins = cars["origin"].unique()
unique_origins.sort()

models = {}
features = [c for c in train.columns if c.startswith("cyl") or
c.startswith("year")]

X = train[features]

for origin in unique_origins:
```

```
y = train['origin'] == origin
model = LogisticRegression()
model.fit(X,y)
models[origin] = model
```

4.3.4 Overfitting

Definition 4.30: Bias-Variance Tradeoff

In statistics and machine learning, the bias–variance tradeoff is the property of a set of predictive models whereby models with a lower bias in parameter estimation have a higher variance of the parameter estimates across samples, and vice versa. The bias–variance dilemma or bias–variance problem is the conflict in trying to simultaneously minimize these two sources of error that prevent supervised learning algorithms from generalizing beyond their training set.

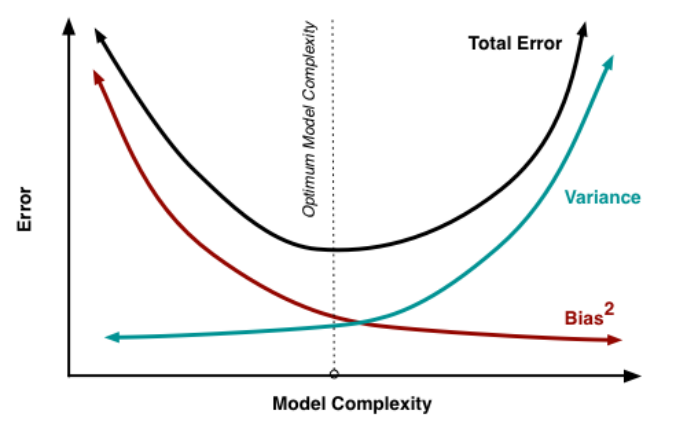

Figure 4.7: Bias-Variance Tradeoff

In-Sample and Out-of-Sample Error

A good way to detect if your model is overfitting is to compare the in-sample error and the out-of-sample error, or the training error with the test error. So far, we calculated the in sample error by testing the model over the same data it was trained on. To calculate the out-of-sample error, we need to test the data on a test set of data. We unfortunately don't have a separate test dataset and we'll instead use cross validation.

If a model's cross validation error (out-of-sample error) is much higher than the in sample error, then your data science senses should start to tingle. This is the first line of defense against overfitting and is a clear indicator that the trained model doesn't generalize well outside of the training set.

Example 4.21: Computing MSE and Variance with 10-Fold Validation for Linear Regression Models

In this exercise we will create a function named train_and_cross_val that:

- *takes in a single parameter (list of column names),*

- *trains a linear regression model using the features specified in the parameter,*
- *uses the KFold class to perform 10-fold validation using a random seed of 3 (we use this seed to answer check your code),*
- *calculates the mean squared error across all folds and the mean variance across all folds.*
- *returns the mean squared error value then the variance using a multiple return statement (e.g. return(avg_mse, avg_var)).*

We'll then use the train_and_cross_val function to train linear regression models using the following columns names in features: {'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'model year', 'origin'}.

- *Our first set of values will be obtained by using the 'cylinders' and 'displacement' columns for a LinearRegression model and assign the resulting mean squared error value to two_mse and the resulting variance value to two_var.*
- *We'll continue this process by taking the first k columns in the features list and assign the resulting mean squared error value to k_mse and variance value to k_var.*

```
# CB 6.20 #

from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error
import numpy as np
import matplotlib.pyplot as plt

def train_and_cross_val(features, target = 'mpg'):

    kf = KFold(n_splits = 10, shuffle = True, random_state =3)

    mse_list = []
    var_list = []

    car_features = filtered_cars[features]
    car_target = filtered_cars[target]

    for train_idx, test_idx in kf.split(X = filtered_cars[features],
    y = filtered_cars[target]):

        model = LinearRegression()

        X_train, X_test = car_features.iloc[train_idx],
        car_features.iloc[test_idx]

        y_train, y_test = car_target.iloc[train_idx],
        car_target.iloc[test_idx]

        model.fit(X_train, y_train)
        predictions = model.predict(X_test)

        mse = mean_squared_error(predictions, y_test)
        var = np.var(predictions)
```

```
            mse_list.append(mse)
            var_list.append(var)

        avg_mse = np.mean(mse_list)
        avg_var = np.mean(var_list)

        return (avg_mse, avg_var)

features = ['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration',
'model_year', 'origin']

two_mse, two_var = train_and_cross_val(features[0:2])
three_mse, three_var = train_and_cross_val(features[0:3])
four_mse, four_var = train_and_cross_val(features[0:4])
five_mse, five_var = train_and_cross_val(features[0:5])
six_mse, six_var = train_and_cross_val(features[0:6])
seven_mse, seven_var = train_and_cross_val(features[0:7])

mse_vals = [two_mse, three_mse, four_mse, five_mse, six_mse, seven_mse]
var_vals = [two_var, three_var, four_var, five_var, six_var, seven_var]

plt.scatter(x = range(2,8), y = mse_vals, c = 'red', label = 'MSE')
plt.scatter(x = range(2,8), y = var_vals, c = 'blue', label = 'VAR')
plt.legend(loc = 'upper_left')

plt.show()
```

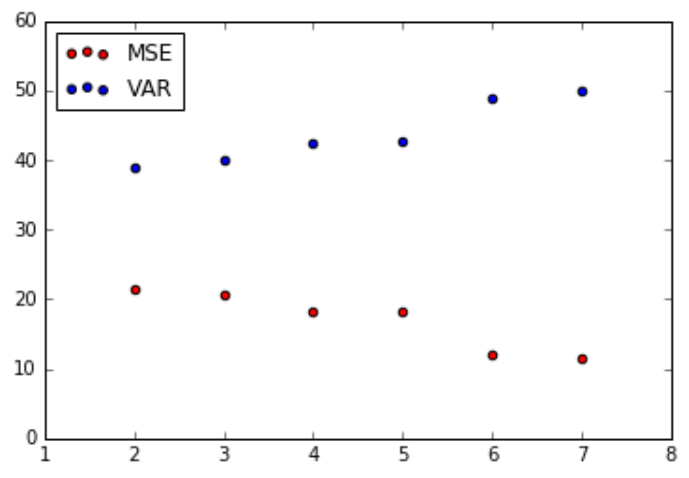

Figure 4.8: The Output Plot of CB 6.20

Alphabetical Index

References

[1] URL: https://docs.scipy.org/doc/numpy/reference/generated/numpy.genfromtxt.html.

[2] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html.

[3] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html.

[4] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html.

[5] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.map.html.

[6] URL: https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.figure.html.

[7] URL: https://matplotlib.org/api/axes_api.html#matplotlib-axes.

[8] URL: https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html.

[9] URL: https://en.wikipedia.org/wiki/Kernel_density_estimation.

[10] URL: https://matplotlib.org/basemap/api/basemap_api.html#mpl_toolkits.basemap.Basemap.

[11] URL: https://www.w3schools.com/sql/sql_operators.asp.

[12] URL: https://www.sqlservertutorial.net/sql-server-aggregate-functions/.

[13] URL: https://www.cia.gov/library/publications/the-world-factbook/.

[14] URL: https://www.sqlstyle.guide/.

[15] URL: https://docs.python.org/3/library/sqlite3.html#cursor-objects.

[16] URL: https://jeffknupp.com/blog/2016/03/07/python-with-context-managers/.

[17] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_sql_query.html.

[18] URL: http://www.bkent.net/Doc/simple5.htm.

[19] URL: https://www.psycopg.org/docs/connection.html#connection.commit.

[20] URL: https://developer.github.com/v3/repos/.

[21] URL: https://www.crummy.com/software/BeautifulSoup/.

[22] URL: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html.

[23] URL: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html#sklearn.neighbors.KNeighborsRegressor.

[24] URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html.