Machine Learning with Python

Jupyter Notebook Templates

GSCM 510/410

Summer 2021

David Gerbing
The School of Business
Portland State University

Table of Contents

Course Text

Introduction to Machine Learning

http://web.pdx.edu/~gerbing/Books/ML/

Prepare for Python Data Analysis

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Set-up

Open a notebook, either a Jupyter Notebook on your computer from the Anaconda download or a Colab Notebook on Google Colab. Then do the following, adapted to a specific analysis.

Python consists of the core language plus many packages that extend the core language with additional functions. However, most of the analyses we pursue are *not* part of the core language, so we first import packages of needed functions before analysis begins.

Time Stamp

Not needed, but useful to know the time and date at which you conducted the analysis. The `now()` function that provides this information is not part of core Python, but is available from the `datetime` package, abbreviated here as `dt`. The `datetime` package is not included with the original Python distribution, so separately access with the `import` method.

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-08-13 at 12:42
```

Refer to the package with its designated abbreviation `dt`. To run the `now()` function, refer to it as `dt.run()`, that is, the package name or abbreviation, a period, then the function name. This convention lets Python know where to locate the corresponding function that is not included with the original Python distribution.

Import Data Analysis Packages

The core language itself does not contain specialized data analysis functions, so first reference the packages that contain these functions before doing data analysis.

> Every data analysis invokes functions from at least two packages, `pandas` and `numpy`.

- `pandas`, a general package for reading, storing, and manipulating data in data frames, standard row x column data tables
- `numpy`, a general package for numerical computations, upon which many other packages rely, including `pandas`

Data analysis usually involves some form of data visualization as well.

> Find the basic data visualization functions in the `matplotlib` package, with increasing reliance upon the more elegant `seaborn`, which is based on `matplotlib`.

- `matplotlib`, basic plotting library for data visualizations
- `seaborn`, more elegant, higher-level plotting library based on `matplotlib`

Access these packages with the `import` method. The optional `as` parameter provides an abbreviation from which to refer to the package functions.

In this Jupyter Notebook, we only need the `pandas` package for reading a data file, but typically reference all four packages in a complete data analysis. The abbreviations listed are traditional for data analysis with Python.

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

Read and Display Data

There are two different situations when reading the contents of data files into a working Python program. Read the data from ...

- the web
- a file on a user's file directory, either locally or networked

To read data files from the web, specify the file's URL (web address) when calling a read function. This section explains the default location of the reading and writing data files on your local or related computer file system.

To read and write data files, each notebook begins with a default location.

> *Current working directory*: Folder (directory) in your computer's file system that is the default location where Python reads and writes data files.

To read and write data files, the concept of a current working directory allows simplified references to data files without the tedium and error-prone process of listing the full path name from the root level of your computer, such as C: on Windows. Instead, reference the files relative to the folder (directory) from which the Python program is being run.

Place your data files directly into your current working directory. Then just specify the file name enclosed within quotes when reading the file. Or, create a `data` folder inside your working directory, then place your data file inside that folder. Reference the file with `"data/file_name"`.

File Directory

On Your Computer

Before opening a Jupyter Notebook, first create a folder to store all notebooks and data files. A suggestion is a folder called Python in your Documents folder. The first screen after opening the Jupyter Notebook app is a listing of folders and files. Click on the `Documents` folder and then the Python folder to begin analysis from that location.

Path names that specify where to read a file and write a file begin with the current working directory as the "home" location. For example, it may be convenient to store your data files in a folder named data, located at the top-level of your current working directory. Then all path names to locate a data file stored in this data folder begin with data. The forward slash indicates a sub-folder, that is, sub-directory.

List the current working directory with the `getcwd()` function from the os package.

```
import os
```

```
os.getcwd()
```

```
'/content'
```

This example was run on Google Colab, described next, so the current directory has neither a Windows or Macintosh file reference.

On Google Colab

Colab is Google's free cloud computing enviornment. All that is needed to access is a Google account.

Colab Notebooks, essentially Jupyter Notebooks, do not access the files on your personal computer. Instead, they use Google Drive as your file directory. For convenience, create a folder called `data` on your Google Drive and copy one or more data files to that location. Colab will automatically create a folder called `Colab Notebooks` on your Google Drive to store your Jupyter Notebook files.

To access your Google Drive files requires the use of the `mount()` function in the `google.colab` package, imported here with the abbreviation of `drive`, so the function reference is `drive.mount()`. Mount your Google Drive in the `content` directory (folder) and the nested directory called `drive`. When you request to mount the drive you will be provided a link to obtain an authorization code.

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Mange the files on your Google Drive, such as storing a data file in a folder called `data`.

To read or write files to or from Google Drive, also include `\MyDrive` in the path name because that reference is to your Google Drive. The full reference to your Google Drive essentially your current working directory:

'/content/drive/MyDrive/'

If you have a folder on the top level of `MyDrive` called `data`, and a file named `employee.xlsx` in that folder, here is the full path name to locate that file:

'/content/drive/MyDrive/data/employee.xlsx'

Note the path name is included with quotes.

Once `drive` is mounted so that you can access the data file, as well as any other existing files, you can also view the path name for any file in the system. Click on the folder icon in the extreme left margin to explore your `drive` file directory (folder). Click on `MyDrive` to access your Google Drive files. To can locate any specific file in that directory, hover the mouse over it, and then click on the three displayed dots. From the displayed choices, choose `Copy path` to get the exact path name to reference that specific file.

Read

The pandas package defines the concept of a data frame and provides the needed functions for reading `csv` or Excel data files (or many other formats) into a data frame. You are free to use any valid name for the data frame, but recommend to read data into a data frame named *d*. The *d* stands for data, entered with only one keystroke. Within a Python analysis, reference the data frame, not the file from which the data were read.

Read an Excel data file from your local file system into the `pandas` data frame named *d* with the `pandas` function `read_excel()`. Comment out other lines of code with the Python (and R) comment symbol: `#`.

- In this example, run on Google Colab, the uncommented read statement is for the Excel data file stored in the `data` folder created on Google Drive.
- The first commented read statement is to read the file on the web.
- The second commented read statement is for the same Excel data file, but for Python running on the user's computer, stored on the user's `data` folder relative to folder that contains the running Python notebook.

```
d = pd.read_excel('/content/drive/MyDrive/data/employee.xlsx')
#d = pd.read_excel('http://web.pdx.edu/~gerbing/data/employee.xlsx')
#d = pd.read_excel('data/employee.xlsx')
```

Display

When using Excel you always see your data. That part is good, but there is a huge downside from mixing data with code. Much easier to write error-free, reproducible code when you separate your data from your code to manipulate that data. Data analysis programming languages such as Python and R provide that separation.

However, when running Python you can still often view a summary of your data.

1. When first reading the data
2. After any data transformation
3. Whenever you encounter a programming error.

When in doubt, always LOOK! For example, after reading data into a data frame, *always* check what was just read.

Here, get the dimensions of the corresponding data frame. The dot notation, the period after the *d* indicates that the corresponding reference to `shape` is for the *d* data frame. Each instance of the data frame object, here *d*, has an attribute (property) called `shape`. Here retrieve the shape of *d*, rows by columns.

```
d.shape
```

```
(37, 9)
```

Most references to pre-programmed functions provide the option of passing specific parameter values within parentheses. But even if all the default values are accepted, the parentheses must still be provided, even if empty. The `head()` function has only one parameter, `n`, which is the number of rows of data to display. The default value is five. So following function calls provide the same result:

- head()
- head(5)
- head(n=5)

```
d.head()
```

	Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
0	Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
1	Wu, James	NaN	M	SALE	94494.58	low	1	62	74
2	Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
3	Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
4	Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

Note that unfortunately (in my opinion) Python starts row numbering (and everything else) with 0 instead of 1. For example, the row of data numbered 4 is the fifth row of data.

Also note that in the corresponding Excel file with the data read into the *d* data frame, the cell for variable *Years* for James Wu is missing, it is blank. When read into the data frame, the corresponding cell displays the missing data code for `pandas, NaN`, abbreviated from Not a Number.

Data Types

Computers store different types of data values differently. The biggest distinction is between numeric and non-numeric data. With numeric data, further distinguish between data values that have no decimal digits, integers, and those with decimal digits (even if all 0's), floating point values.

> *Data type*: The way in which data values are stored in computer memory.

Other data types exist, but in this example, read each variable into one of three Python data types:

- integer (int64) for 64-bit integers, can be very, very large numbers
- floating-point (float64) for potentially very large numbers with decimal digits
- object for non-numeric values

When working with data frames, organize data by variables (columns). Show the type of each variable with the `info()` function. In Python, methods and functions generally perform the same purpose, but methods are specified without trailing parentheses.

```
d.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 37 entries, 0 to 36
Data columns (total 9 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   Name    37 non-null     object 
 1   Years   36 non-null     float64
 2   Gender  37 non-null     object 
 3   Dept    36 non-null     object 
 4   Salary  37 non-null     float64
 5   JobSat  35 non-null     object 
 6   Plan    37 non-null     int64  
 7   Pre     37 non-null     int64  
 8   Post    37 non-null     int64  
dtypes: float64(2), int64(3), object(4)
memory usage: 2.7+ KB
```

We see, for example, that *Name* is a variable that has non-numeric characters as data values. *Salary* is a variable with data values that have decimal digits, and *Pre* is a variable with integer data values. For some reason, the variable *Years* also only has integer values, but is represented as a `float64` variable, that is, data values with decimal digits. Perhaps the missing data value, 36 instead of 37 data values, triggered that designation.

Data Pre-Processing

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Preliminaries

Packages

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-06-25 at 01:11
```

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

Read

```
#d = pd.read_excel('data/employee.xlsx')
d = pd.read_excel('http://lessRstats.com/data/employee.xlsx')
```

The data values in the Name column are not values per se to analyze, but instead serve as row identifiers, ID's. As such, replace the default integer row labels with the values of the column Name. Do so with the `set_index()` function.

```
d = d.set_index('Name')
d.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

Create Dummy Variables

designateMachine learning procedures cannot directly process categorical variables with non-numeric values. For example, consider a data set with two values of *Gender*, coded as M and F. The variable *Gender* with this non-numerical coding cannot be entered into a machine learning analysis, which requires numerical variables only.

Categorical variables with non-numerical values, however, can be converted to numerical representations. Many such conversions are possible. Here we consider the most widely used conversion.

> *Dummy Variable*: A numerically encoded variable for each level of a categorical variable, with a value of 1 if the level is present and 0 if not.

The Gender variable becomes two dummy variables, *Gender_F* and *Gender_M*. For example, if a person's Gender is listed as F, then *Gender_F* is 0 and *Gender_M* is 1.

Pandas provides the function `get_dummies()` to covert a categorical variable to a corresponding set of dummy values, one for each category. The parameter `columns` designates the variables to be converted.

One adjustment is needed. If you know the value of *Gender_F* for an individual is 1, then you also know that *Gender_M* is 0. So the value of either one of two dummy variables implies the value of the other. To avoid redundancy, in general, for k levels of the categorical variables, the number of dummy variables retained in the analysis is $k - 1$. For two levels of *Gender*, arbitrarily retain $2 - 1 = 1$ of the dummy variables in the analysis.

With `get_dummies()`, drop the first dummy variable with the `drop_first` parameter set to `True`. Alphabetically, F comes before M, so in the following analysis, *Gender_F* is dropped. The original *Gender* variable is replaced with *Gender_M*.

For *JobSat* with three levels -- High, Low, and Med -- create three dummy variables, each corresponding to the one of the three values. *JobSat* is then replaced by two dummy variables for the Low and Med values. For example, if you know the values of *JobSat_low* and *JobSat_med* are both 0, then you know that the value of *JobSat_High* is 1. Knowing two values implies the third, so retain only two dummy variables for *JobSat* in the analysis.

```
d = pd.get_dummies(d, columns=["Gender", "JobSat"], drop_first=True)
d.head()
```

Name	Years	Dept	Salary	Plan	Pre	Post	Gender_M	JobSat_low	JobSa
Ritchie, Darnell	7.0	ADMN	53788.26	1	82	92	1	0	
Wu, James	NaN	SALE	94494.58	1	62	74	1	1	

Usually which particular dummy variable is dropped for each categorical variable is irrelevant. If, however, it is desired to drop a dummy variable other than the first, then run `get_dummies()` without the `drop_first` parameter and manually drop the specified dummy variable from the data frame.

Missing Data

Assess Amount of Missing Data

Machine learning functions generally do *not* work in the presence of missing data. Before machine learning analysis, examine the data for missing data and adjust accordingly, either delete the row or column or impute the value.

A missing data value is indicated by the notation `NaN`, an abbreviation for Not a Number. Sometimes functions or discussion of missing data refer to missing data as `na`, which means Not Available.

Here James Wu has a missing value for the number of years he worked at the company. Data values for James Wu occupy the second row of data, identified by row index 1. The row definition of 1:2 (confusingly) also refers to the second row. However, specifying the row as a range results in the output's more visually appealing horizontal placement.

```
d.iloc[1:2, 0:5]
```

Name	Years	Dept	Salary	Plan	Pre
Wu, James	NaN	SALE	94494.58	1	62

The `isna()` function indicates if a data value is missing. Follow with the `sum()` function to sum the number of missing values for a variable, here all variables in the *d* data frame because no specific variable is specified. Follow with a second `sum()` function to sum the sums, that is, the total number of missing values in the entire data frame.

```
print (d.isna().sum())
print ('\nTotal Missing:', d.isna().sum().sum())
```

```
Years         1
Dept          1
Salary        0
Plan          0
Pre           0
Post          0
Gender_M      0
JobSat_low    0
JobSat_med    0
dtype: int64

Total Missing: 2
```

As a programming note, without using the `print()` function, the last row of code in a Jupyter cell that specifies output generates the default output. If there is more than a single line of code that generates output, or if customization of the output is desired, such as adding a descriptive label, then invoke `print()`, as in this example.

Show Rows with Missing Data

The code for viewing all rows of missing data begins with the `isna()` function, which returns `True` if a data value is missing. The `any()` function evaluates the data frame column-by-column and then returns `True` if there are any `True` values in the corresponding row. Putting the expression within `d[ ]` selects only the rows with `True`, that is, with missing according to `isna()`.

```
d[d.isna().any(axis='columns')]
```

Name	Years	Dept	Salary	Plan	Pre	Post	Gender_M	JobSat_low	JobSat_m
Wu, James	NaN	SALE	94494.58	1	62	74	1	1	
Jones,	5.0	NaN	53772.58	1	65	62	0	0	

Delete rows with Missing Data

The simplest method to address missing data deletes a row if it contains any missing data, what is called *case deletion*, or *list-wise deletion*. The `dropna()` function deletes rows with missing data from *d*. It is also possible to apply the function to columns with parameter `axis`, which indicates if the analysis applies to rows or columns. Often in Python coding people use 0 instead of the more descriptive 'rows'.

The process in this example removes the three rows with missing data, from 37 rows to 34 rows.

```
d.shape
```

```
(37, 9)
```

```
d = d.dropna()
d.shape
```

```
(35, 9)
```

The problem with dropping rows that contain missing data is that for some data sets much or most of the data can be deleted. Appropriate if many data values in an entire row are missing, but perhaps not if just one missing data value across data for many variables. Or, sometimes a single variable may contain many missing values, so better to delete the bad variable than delete so may corresponding rows of data (cases).

To illustrate, re-display the variable names and the first five rows of data. In the original data frame, James Wu is missing Years worked for the company, and Alissa Jones is missing the Dept worked in as well and the Job Satisfaction rating. Both rows of data are now deleted from the revised data frame.

```
d.head()
```

Name	Years	Dept	Salary	Plan	Pre	Post	Gender_M	JobSat_low	JobSa
Ritchie, Darnell	7.0	ADMN	53788.26	1	82	92	1	0	
Hoang, Binh	15.0	SALE	111074.86	3	96	97	1	1	
Downs, Deborah	7.0	FINC	57139.90	2	90	86	0	0	

Impute Missing Data

A data frame can contain many variables, including variables not relevant to a particular analysis. Later we show that when doing machine learning, we isolate a relevant set of variables for a given model in their own data structure. In machine learning, by tradition name this data structure X, the uppercase representation to indicate more than a single variable in general.

To make this code more applicable to subsequent machine learning analyses, we subset the variables from the original data frame into a data frame named X, the data that contains just the features (predictor variables) for the machine learning analysis.

```
X = d.loc[:, ['Years', 'Salary', 'Pre', 'Post']]
X.head()
```

Name	Years	Salary	Pre	Post
Ritchie, Darnell	7.0	53788.26	82	92
Hoang, Binh	15.0	111074.86	96	97
Downs, Deborah	7.0	57139.90	90	86
Afshari, Anbar	6.0	69441.93	100	100
Knox, Michael	18.0	99062.66	81	84

```
type(X)
```

```
pandas.core.frame.DataFrame
```

Instead of deleting rows or columns with missing data, an alternative approach *imputes* missing data values. Provide some reasonable guess regarding a missing data value and then set this value in place of the missing data code. Impute with the `SimpleImputer()` function. The mean of each variable is the default value to replace missing data for a variable. A typical better choice replaces the mean with the median to avoid the impact of potential outliers.

Specify the median with the `strategy` parameter. The `missing_values` parameter indicates the value defined as a missing value, here `NaN` as indicated by the numpy array value of `nan`. To impute the median, only select variables with numerical values.

Remember, first we have the Python language, then we have the `numpy` package built on top of base Python, then we have the `pandas` package built on top of `numpy`. So when doing machine learning, we encounter functions from the original Python plus from both `numpy` and `pandas`.

The `fit()` function computes the values to be used to fill in missing values. The `transform()` does the imputation.

```
from sklearn.impute import SimpleImputer
```

```
imp_med = SimpleImputer(missing_values=np.nan, strategy='median')
imp_med = imp_med.fit(X)
X = imp_med.transform(X)
```

The `transform()` function outputs an array from the original input data frame.

```
type(X)
```

```
numpy.ndarray
```

The missing data for James Wu for Years now has a value of 9.0, that is, the value in the second row and first column of *d*.

```
X[1,0]
```

```
15.0
```

The best way to impute missing data involves predicting each missing data value from the remaining non-missing data values. This is more complex and requires much more computer time, so maybe not practical for very large data sets.

Search for Outliers

Sometimes some data values do not appear to part of the same distribution as the other data values.

> *Outlier*: Value considerably different from most remaining values of the distribution.

Note that this definition of an outlier applies to the analysis of a single variable and so identifies what is called a univariate outlier. However, the concept can also be applied to considering several variables simultaneously.

One motivation for outlier detection is that an outlier could represent a simple data collection or transcription error. Alternatively, an outlier could represent a data value sampled from a population distinct from the population that generated the remaining data values and therefore would bias the analysis regarding generalizations to what was intended to be the population of interest. So, an essential aspect of the data analytic process first identifies and then explains the process that generated the anomalous values.

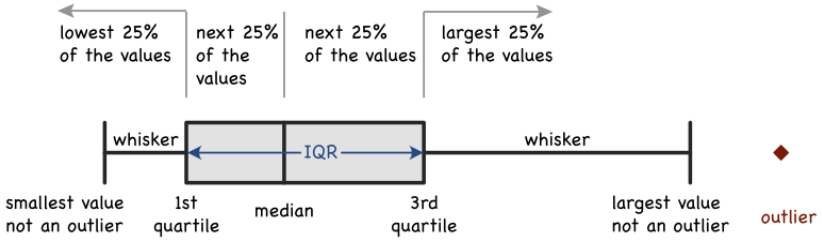
One approach to identify outliers for a single variable follows from the interquartile range (IQR). To compute, sort the values of the distribution from smallest to largest.

> *Quartiles*: Three values that divide the entire sorted distribution of values into quarters, four-equal size groups.

The first one-quarter of the values lie between the smallest value and the first quartile. The second quartile is the median, which occupies the middle spot between the smallest and largest values in the sorted distribution. The third quartile separates the largest 25% of the values from the smaller values.

> *Interquartile Range* (IQR): Range of values that contains the middle 50% of the values in magnitude, the positive difference between the 3rd and 1st quartiles.

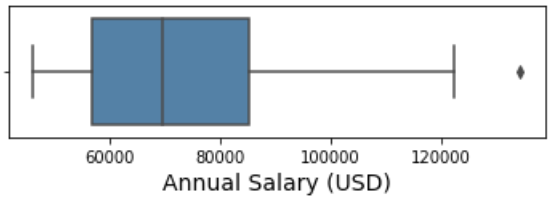
> *Boxplot*: The body of the box extends from approximately the 1st to the 3rd quartiles, with a line through the median and perpendicular lines called whiskers extending out from the edges, with outliers plotted beyond the whiskers.



Values far from the edges of the box are labeled as outliers. The classic definition is that data values beyond 1.5 IQR's and 3 IQR's beyond the 1st and 3rd quartiles are labeled as potential outliers and outliers.

To plot a boxplot with `seaborn`, use the function `boxplot()`. By default, the boxplot is quite high. Can use the `figure` function with the `figsize` parameter to specified a more narrow height.

```
plt.figure(figsize=(6,1.5))
sns.boxplot(x=d['Salary'], color='steelblue')
plt.xlabel('Annual Salary (USD)', fontsize=14)
plt.show()
```



Identify the row of data that has the maximum value for Salary with the `idmax()` function. The `index` is the pandas name for the row names, which in the *d* data frame are the actual employee names, not integers. and display just the value of Salary instead of all the data values for that row (case). (Another approach would be to use filter rows expression from the data wrangingly notebook and list all records larger than about 125,000 or something.)

```
d['Salary'].idxmax()
```

```
    'Correll, Trevon'
```

```
round(d['Salary'].loc[d['Salary'].idxmax()], 2)
```

```
    134419.23
```

The implication for data analysis is to examine the process that generated this anomalous data value. Is the person making that large salary a high-level manager with the other salaries are for hourly workers? If different processes generate the salaries, then the outlier should be deleted from further analysis, and the results of further analyses generalized to the appropriate population, that of hourly workers, in this example.

Transform Variables to Similar Scale

Machine learning algorithms tend to generate more useful results when the variables on which they operate are all on about the same scale. That means the data values for each of the variables have about the same range at least, and maybe even about the same mean and standard deviation. Variables with the values as they originally exist do not generally arrive for

analysis sharing the same scale. Weight in pounds, for example, is scaled in entirely different units than Height in inches, or Annual_Income in USD.

Transform the data values of variables with different scaling to obtain similar scales. The re-scalings discussed here are all *linear*.

> *Linear transformation*: Does not change the shape of the underlying distribution, nor its relations with other variables, just its scale.

After the re-scaling, an approximately normal distribution remains approximately normal, and a skewed distribution remains skewed.

For a linear transformation, multiply each data value by a constant and add another constant. That is, transform variable x to y with $y = a + bx$. An example is converting measurement of length in feet to inches in which the data values are divided by 12, that is, $b = 1/12$ and $a = 0$. That is, to convert feet to inches, divide each data value expressed in feet by 12.

The Python package for machine learning, `sklearn`, module `preprocessing`, provides several rescaling possibilities. Need to import the module.

```
from sklearn import preprocessing
```

Specify the variables to transform in their own data frame.

As a programming note, subsets (slices) of data frames by default do not create clean copies. Changes to the values of X can lead back to changes in the original data frame from which X was derived. To make X completely independent of *d*, invoke the subset slice with the `copy()` function.

```
X = d[['Years', 'Salary', 'Pre']].copy()
X.loc[:,'Pre'] = X.loc[:,'Pre'].astype('float64')
X.head()
```

Name	Years	Salary	Pre
Ritchie, Darnell	7.0	53788.26	82.0
Hoang, Binh	15.0	111074.86	96.0
Downs, Deborah	7.0	57139.90	90.0
Afshari, Anbar	6.0	69441.93	100.0
Knox, Michael	18.0	99062.66	81.0

How do know what (if any) preprocessing methods to choose? There are some guidelines, but the most general answer follows the typical machine learning approach: Try the various possibilities and choose the algorithm that provides the most accurate forecasting.

Suppose we wish to explain a score on a post test from years worked at the company, annual salary, and scores on a corresponding pre-test. In the following examples, isolate those three variables into a data structure called X.

Min-Max Scaling

A common rescaling used in machine learning transforms the variables to have the same minimum and maximum values.

> *Min-Max Scaling*: Convert all data values for a variable so that the minimum value is 0 and the maximum value is 1.

Write this transformation for the value of x in the i^{th} row of data values of variable x as:

$$y_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

Express as a linear function with $a = -(min(x)/range(x))$ and $b = 1/range(x)$.

The `MinMaxScaler` provides the needed transformation from the minimum and maximum values of each variable to which the transformation is applied. Here instantiate as *mm_scaler*.

```
from sklearn.preprocessing import MinMaxScaler
mm_scaler = preprocessing.MinMaxScaler()
```

Apply to Original Data

At this point, X is a (pandas) data frame.

```
type(X)
```

```
pandas.core.frame.DataFrame
```

The `fit()` function calculates the needed information to provide the rescaling: the minimum and maximum needed for each variable. The `transform()` function performs the rescaling using this information. Combine both methods with a single invocation of `fit_transform()`.

The `MinMaxScaler` works with and retains missing data. Any entered non-numeric variables, however, cannot be processed and cause the routine to terminate.

```
Xmm = mm_scaler.fit_transform(X)
```

After processing by the `MinMaxScaler` and transformed, X is now a (numpy) array. This is one of the complications of Python applied to data analysis, the need for external packages, here both `numpy` and `pandas`. Unfortunately, sometimes, as in this example, data structures are created that conform to those of another package than what was input. A `numpy` array is the equivalent data structure to a `pandas` data frame.

```
type(Xmm)
```

```
numpy.ndarray
```

Some statistical computations use `panda` data frames instead of `numpy` arrays. The default display of the data frame also is more aesthetic than the array. Convert the transformed X back to a data frame. The column names also have to be manually added as the `numpy` array deletes the original names.

Examine the first five rows of the data frame. The values of each of the variables *Years*, *Salary*, and *Pre* were initially on discordant scales, but now have values that range from 0 to 1.

```
Xmm = pd.DataFrame(Xmm, columns=['Years', 'Salary', 'Pre'])
Xmm.head()
```

	Years	Salary	Pre
0	0.260870	0.086793	0.560976
1	0.608696	0.735607	0.902439
2	0.260870	0.124753	0.756098
3	0.217391	0.264082	1.000000
4	0.739130	0.599560	0.536585

Confirm the success of the transformations of the three variables by using the pandas data frame methods `min` and `max`.

```
Xmm.min()
```

```
Years     0.0
Salary    0.0
Pre       0.0
dtype: float64
```

```
Xmm.max()
```

```
Years     1.0
Salary    1.0
Pre       1.0
dtype: float64
```

Can transform any data with the same values from the previous application of `fit()`. Here manually transform the second row of data in X.

```
mm_scaler.transform([[15.0, 111074.86, 96.0]])
```

```
array([[0.60869565, 0.73560716, 0.90243902]])
```

You can also transform manually. The linear equation by which each data value is transformed is available from the computed `scale_` and `min_` data structures. Each computed data structure contains one value for each variable in the data frame that is transformed.

```
print('Multiplier:', mm_scaler.scale_.round(3))
print('Additive:', mm_scaler.min_.round(3))
```

```
Multiplier: [0.043 0.    0.024]
Additive: [-0.043 -0.522 -1.439]
```

To illustrate, manually compute the transformed value of Salary for the second row of data in X, which equals the corresponding value in the above display portion of the X data frame.

```
mm_scaler.scale_[1]*111074.86 + mm_scaler.min_[1]
```

```
0.7356071617792598
```

Apply to New Data

The purpose of supervised learning is to develop a model for forecasting from the values of the feature variables. If the data have undergone a scaling transformation such as Min-Max, then any new data must undergo that same transformation before generating a forecast.

The new data can be transformed manually, as in the immediately previous example. Or, apply the information already obtained from the `fit()` function encoded in `mm_scaler` to the transformation with the `transform()` function.

In this example, suppose an employee has worked at the company for 8 years, has a salary of 76,492 USD, and scored an 89 on the pre-test. Next, compute the person's forecasted salary.

```
X_new = [[8,76492, 89]]
X_new = mm_scaler.transform(X_new)
X_new
```

```
array([[0.30434783, 0.34392983, 0.73170732]])
```

In this example, only a single line of values for the three features was processed. Of course, the number of rows of X_new can be a full data frame of new values, all processed at once.

Standardization Scaling

Another linear transformation of the data values converts the original data values to z-scores, this one taught in all introductory stat courses.

> *Standard score*: The number of standard deviations a data value is from the mean.

Write the transformation of the data values for variable x as:

$$z_i = \frac{x_i - m}{s}$$

To standardize, for each data value of a variable, for the i^{th} row of data, subtract the mean of the data and divide by the standard deviation of the data. The result, a distribution of z-scores, has a mean of 0 and a standard deviation of 1.

For this linear transformation, set $a = -(m/s)$ and $b = 1/s$, where s is the sample standard deviation and m is the sample mean.

The `StandardScaler` provides the computations for standardization of variables. IF (not a requirement for standardization) a variable is normal, then most values will be within 2.5 or 3 standard deviations from the mean, that is, standard scores of less than 3.0 and greater than -3.0.

```
from sklearn.preprocessing import StandardScaler
s_scaler = preprocessing.StandardScaler()
```

Get the mean and standard deviation of each variable with the `fit()` function. Do the rescaling with the `transform()` function. Combine both with the `fit_transform()` function.

```
Xst = s_scaler.fit_transform(X)
Xst = pd.DataFrame(Xst, columns=['Years', 'Salary', 'Pre'])
Xst.head()
```

The success of the transformation is shown by examining the mean and standard deviation of the three transformed, now standardized, variables.

```
round(Xst.mean(), 4)
```

```
Years     0.0
Salary    0.0
Pre      -0.0
dtype: float64
```

```
round(Xst.std(), 4)
```

```
Years     1.0146
Salary    1.0146
Pre       1.0146
dtype: float64
```

The range of the data roughly approximates that of normal data, though skewed right. In a perfectly normal distribution the standardized values would range from about -2.5 to 2.5.

```
Xst.min()
```

```
Years    -1.500617
Salary   -1.282158
Pre      -1.779224
dtype: float64
```

```
Xst.max()
```

```
Years     2.553063
Salary    2.811947
Pre       1.752154
dtype: float64
```

Can transform any data with the same values from `fit()`.

```
s_scaler.transform([[15.0, 111074.86, 96.0]])
```

```
array([[0.96684042, 1.72949472, 1.4076293 ]])
```

Can also transform manually. The computed values by which each data value is transformed is available from the computed `scale_` and `mean_` data structures. Each computed data structure contains one value for each variable in the data frame that is transformed.

```
print('mean:', s_scaler.mean_)
print('sd:', s_scaler.scale_)
```

```
mean: [9.51428571e+00 7.37762411e+04 7.96571429e+01]
sd: [5.67385698e+00 2.15661941e+04 1.16101996e+01]
```

To illustrate, compute the transformed value of *Salary* for the second row of data in X.

```
(111074.86 - s_scaler.mean_[1]) / s_scaler.scale_[1]
```

```
1.7294947196040262
```

```
X.head()
```

Name	Years	Salary	Pre
Ritchie, Darnell	7.0	53788.26	82.0
Hoang, Binh	15.0	111074.86	96.0
Downs, Deborah	7.0	57139.90	90.0
Afshari, Anbar	6.0	69441.93	100.0
Knox, Michael	18.0	99062.66	81.0

Robust Scaling

Robust scaling resembles standardization, except it is more robust to the presence of outliers. The presence of outliers does not dramatically change the resulting scaled values as much as standardization in which an outlier can have a significant impact on the mean and an even bigger impact on increasing the size of the standard deviation (which depends on squared deviation scores).

Robust scaling accomplishes this robustness by replacing the mean in the standard score formula with the more robust median and the standard deviation with the more robust interquartile range. The median is the second quartile, and the IQR is the difference between the third and first quartiles. Unlike the mean and standard deviation, no matter how extreme a few values are in a distribution, the quartiles remain the same.

> *Robust scale score*: The number of IQR's a data value is from the median.

Write the transformation of the data values for variable x as:

$$robustscore = \frac{x_i - median}{IQR}$$

That is, to do a robust scaling, for each data value of a variable, for the i^{th} row of data, subtract the median of the data and divide by the IQR of the data.

```
from sklearn.preprocessing import RobustScaler
r_scaler = preprocessing.RobustScaler()
```

```
Xrb = r_scaler.fit_transform(X)
Xrb = pd.DataFrame(Xrb, columns=['Years', 'Salary', 'Pre'])
Xrb.head()
```

	Years	Salary	Pre
0	-0.250	-0.554057	0.108108
1	0.750	1.459989	0.864865
2	-0.250	-0.436222	0.540541
3	-0.375	-0.003715	1.081081
4	1.125	1.037672	0.054054

The specific characteristics of the transformed variables differ from standardization, but the general results remain. The means are somewhat close to 0. The standard deviations are less than 1 but certainly much closer to 1 than from the original

distributions. The minimum and maximum values are less than the range of the standardized variables but roughly similar,

```
round(Xrb.mean(), 4)
```

```
Years     0.0643
Salary    0.1487
Pre      -0.0185
dtype: float64
```

```
round(Xrb.std(), 4)
```

```
Years     0.7196
Salary    0.7693
Pre       0.6367
dtype: float64
```

```
round(Xrb.min(), 4)
```

```
Years    -1.0000
Salary   -0.8235
Pre      -1.1351
dtype: float64
```

```
round(Xrb.std(), 4)
```

```
Years     0.7196
Salary    0.7693
Pre       0.6367
dtype: float64
```

Data Wrangling

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Data in the real world typically does not arrive ready for analysis. Instead usually manipulate the data in various ways to derive a nice, clean, tidy data frame of rows by columns with all the data values in a column of the same type, such as character strings, integers, or floating point numbers (i.e., with decimal digits).

> *Data Wrangling*: The process of cleaning, tidying, and otherwise preparing data for analysis.

The following examples demonstrate some useful data manipulations that are applicable to all data analysis: subsetting a data table, converting variable types, and variable transformations. Merging data frames, another common data manipulation, is shown elsewhere.

This task of data wrangling is where most data scientists spend most of their time, up to 80% is a common understanding. Larger organizations have added a new job category called data engineer, a specialist in data wrangling and other aspects of handling data, such as data collection.

The material in this and related notebooks covers some of the basic, and most common, data wrangling procedures.

Preliminaries

Packages

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-06-27 at 14:06
```

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

Read

```
#d = pd.read_excel('data/employee.xlsx')
d = pd.read_excel('http://lessRstats.com/data/employee.xlsx')
```

```
d.shape
```

```
(37, 9)
```

```
d.head()
```

	Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
0	Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
1	Wu, James	NaN	M	SALE	94494.58	low	1	62	74
2	Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
3	Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
4	Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

Data Frame Row Identifiers

Columns in a data frame represent the variables in the analysis. Columns can be identified by the corresponding variable name or by their numerical position in the data frame.

> *Index*: An integer that specifies the numerical row or column position.

Unfortunately (in my opinion), Python begins all counting with 0 instead of 1. However, the principle of identifying a row or a column by the corresponding integer remains.

In the *d* data frame that contains the read data, as shown above, the default row identifiers are the row indices. However, if there is column of unique identifiers in the data table, that column can be designated as the row identifiers.

In the above *d* data table, the *Name* column appears as any other variable in the data frame. However, *Name* is not a variable per se to analyze but an ID field, with a unique value for each row. Replace the default integer row labels with the values of the column *Name* with the `set_index()` function.

Note that data manipulation methods typically do not change the original data frame. To save changes, explicitly save the manipulation into a variable, such as below, where the change to the *d* data frame is saved back into the *d* data frame. If there is no assigned variable for the output, the output is directed to the console. You will be able to view the output, but no changes are saved.

```
d = d.set_index('Name')
d.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

Or, set the row names as the data values directly when read with a function such as `read_excel()`. Use the parameter `index_col` to set the column index. The variable *Name* in the original data frame is in the first column, that is, Column 0.

```
d = pd.read_excel('http://lessRstats.com/data/employee.xlsx', index_col=0)
#d = pd.read_excel('data/employee.xlsx', index_col=0)
d.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97

Subset Rows and Columns

A data frame is the Python representation of a rectangular data table with rows and columns. In many situations, we wish to view or analyze just a portion of the entire data frame, a *subset*, an extraction from the original. Many possibilities of subsetting exist: a single data value (cell), a single row, a single column, and a range of rows or columns.

Express all references to data values in a data frame in terms of those rows and columns. Reference data values within data frame *d* by its rows and columns:

```
d[row_reference, column_reference]
```

The reference can be a corresponding name of a row or column or its associated integer index. If referencing location by name, use the `.loc()` method for "location". Reference the integer index with the `.iloc()` method for "integer location" or "index location". The "weird" part, as with all Python counting, is that counting rows or columns starts with 0 instead of starting at 1.

Note: There is no assignment of the sub-setted information to the new data frame in the examples below, just a display of the requested information.

Select a Single Cell

We see from the original data table that Binh Hoang's salary is $111,074.86.

To display the data value stored in a single, specific cell, specify a single row reference and a single column reference. Here reference the location of the data value by the row name and column name, so use the `loc` method for "location".

Note that the output is not assigned to another data frame, so the output is directed to the space right below where the cell is located.

```
d.loc['Hoang, Binh', 'Salary']
```

```
111074.86
```

Or, reference the location of the data value by its row and column index according to their integer positions in the data frame with `iloc`. The data for Binh Hoang are in the third row, Row 2 because Python starts counting at 0. The variable *Salary* is in the fourth column, Column 3.

```
d.iloc[2,3]
```

```
111074.86
```

Select Multiple Cells

A colon, `:`, indicates a range of either rows (before the comma) or columns (after the comma).

```
d2 = d.loc['Wu, James':'Jones, Alissa', 'Gender':'JobSat']
d2.head()
```

Name	Gender	Dept	Salary	JobSat
Wu, James	M	SALE	94494.58	low
Hoang, Binh	M	SALE	111074.86	low
Jones, Alissa	F	NaN	53772.58	NaN

When subsetting row or column indices in `pandas` with the colon operator, `:`, always reference the counting system that begins with 0. And then reference the row after the last row you wish to select.

For example, to select the second through the fourth row, the `pandas` row indices are 1 through 3. So specify a row range of 1:4. To select the second through fourth columns, specify a column range of 1:5. (Why do the Python people have to complicate such a simple issue as counting?)

To illustrate, show here again the first five rows of the *d* data frame.

```
d.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

With `iloc` now select the second through the fourth rows, getting three rows indicated by indices 1 through 3, and the second through the fifth columns, getting four columns, indicated by indices 1 through 4.

```
d2 = d.iloc[1:4, 1:5]
d2
```

Name	Gender	Dept	Salary	JobSat
Wu, James	M	SALE	94494.58	low
Hoang, Binh	M	SALE	111074.86	low
Jones, Alissa	F	NaN	53772.58	NaN

Subset Rows

Select a Single Row

The `:` by itself *after* the comma, that is, no columns specified, indicates to retrieve *all* columns for the specified row(s). The colon for columns is optional, but good practice to include.

```
d.loc['Hoang, Binh', :]
```

```
Years         15
Gender         M
Dept        SALE
Salary    111075
JobSat       low
Plan           3
Pre           96
Post          97
Name: Hoang, Binh, dtype: object
```

Unfortunately, Python's specification of row and column ranges with the `:` is unnecessarily complex. The straightforward way to proceed is to identify each row or column with its ordinal position, that is, counting from 1. Instead, Python starts counting from 0, so the index of the first column and of the first row is 0.

Another complication is that the range itself indicates the beginning index but does not include the ending index. To illustrate, a range of 2:3, if placed before the comma, references only Row Index 2, which is the third row.

```
d.iloc[2:3, :]
```

	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Name								
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97

Specify a row without a range results in the vertical listing of the information for that row, here Row 3.

```
d.iloc[2, :]
```

```
Years         15
Gender         M
Dept        SALE
Salary    111075
JobSat       low
Plan           3
Pre           96
Post          97
Name: Hoang, Binh, dtype: object
```

Subset Rows by Data Values

Retrieve just those rows of data that match a logical condition. Here identify all rows of data for employees with a Salary more than $100,000 per year. The first example only displays the result. The second example creates a new data frame with the result, and then displays the subset data frame.

This first example invokes the most straightforward method of selecting rows that satisfy a logical condition: The `query()` function. Note the logical condition is enclosed in quotes.

```
d.query('Salary > 100000')
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Correll, Trevon	21.0	M	SALE	134419.23	low	1	97	94
James, Leslie	18.0	F	ADMN	122563.38	low	3	70	70

The `query()` function may be more straightforwared, but there is another expression for the extraction that many people use. This expression involves repeating the name of the data frame, which becomes awkward for long data frame names and a logical expression that involves multiple variables. Still, this form is frequently encountered in `pandas` data manipulation.

```
dd = d[d['Salary'] > 100000]
dd
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Correll, Trevon	21.0	M	SALE	134419.23	low	1	97	94
James, Leslie	18.0	F	ADMN	122563.38	low	3	70	70
Capelle, Adam	24.0	M	ADMN	108138.43	med	2	83	81

Here query all Males in Finance. The amperssand, `&`, indicates "and". Because the entire expression is enclosed in single quotes, `'`, enclose character constants within the string with double quotes, `"`.

```
d.query('Dept == "FINC"  &  Gender == "M"')
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Sheppard, Cory	14.0	M	FINC	95027.55	low	3	66	73
Link, Thomas	10.0	M	FINC	66312.89	low	1	83	83
Cassinelli, Anastis	10.0	M	FINC	57562.36	high	1	80	87

The `str.contains()` function selects values that contain a specified character string. However, the method only works with complete data, so first remove missing data values in *d* with the `dropna()` function.

```
d2 = d.dropna()
d2[d2['Dept'].str.contains('FI')]
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86
Sheppard, Cory	14.0	M	FINC	95027.55	low	3	66	73
Link, Thomas	10.0	M	FINC	66312.89	low	1	83	83
Cassinelli, Anastis	10.0	M	FINC	57562.36	high	1	80	87

Subset Columns

The easiest way to subset variables, select columns, is with the `filter()` function.

```
d.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

In many situations in data analysis, we wish to refer simultaneously to multiple values instead of a single value. With Python in general and `pandas` specifically, define a vector of values using square brackets `[ ]`.

> *Vector*: A variable that consists of multiple values.

Note to R users: R uses the `c( )` function to accomplish the same result.

In this example, create a vector of two variable names. To emphasize the definition of a vector, define the vector separately from the call to the `filter()` function.

```
d2 = d.filter(['Gender', 'Salary'])
d2.head()
```

Name	Gender	Salary
Ritchie, Darnell	M	53788.26
Wu, James	M	94494.58
Hoang, Binh	M	111074.86
Jones, Alissa	F	53772.58
Downs, Deborah	F	57139.90

To emphasize the definition of a vector, define the vector of variable names separately from the call to the `filter()` function.

```
my_vector = ['Gender', 'Salary']
d2 = d.filter(my_vector)
d2.head()
```

Name	Gender	Salary
Ritchie, Darnell	M	53788.26

To select variables by their names can also use the more general `loc()` function. Need to specify both rows and columns with `loc`, separated by a colon `:`. If not simulateounsly selecting rows, put nothing before the `:`.

```
d2 = d.loc[:, ['Gender', 'Salary']]
d2.head()
```

Name	Gender	Salary
Ritchie, Darnell	M	53788.26
Wu, James	M	94494.58
Hoang, Binh	M	111074.86
Jones, Alissa	F	53772.58
Downs, Deborah	F	57139.90

Or, can use `iloc` for "index location" if specifying the relevant numerical index.

```
d2 = d.iloc[:, [1,3]]
d2.head()
```

Name	Gender	Salary
Ritchie, Darnell	M	53788.26
Wu, James	M	94494.58
Hoang, Binh	M	111074.86
Jones, Alissa	F	53772.58
Downs, Deborah	F	57139.90

Select by Variable Type

The select function for selecting by variable types is `select_dtypes()`, either with the parameter `exclude` or `include`.

```
d_num = d.select_dtypes(exclude=['object'])
d_num.head()
```

	Years	Salarv	Plan	Pre	Post

```
d_obj = d.select_dtypes(include=['object'])
d_obj.head()
```

Name	Gender	Dept	JobSat
Ritchie, Darnell	M	ADMN	med
Wu, James	M	SALE	low
Hoang, Binh	M	SALE	low
Jones, Alissa	F	NaN	NaN
Downs, Deborah	F	FINC	high

Chained Functions

Likely the most straightforward to accomplish data frame subsets is with the `query()` and `filter()` functions. However, when doing multiple function calls, one after the other, you can chain these calls into a single call. This chaining elucidates a complex, multi-step data manipulation process with highly readable, structured code.

To specify a set of chained functions, include the entire expression within parentheses ( ), then separate each function call on its own line. In this example, subset by rows, then by columns, then sort on the *Salary* column with the `sort_values()` function.

```
(d
 .query('Salary > 100000')
 .filter(['Gender', 'Salary'])
 .sort_values(['Salary'], ascending=False)
)
```

Name	Gender	Salary
Correll, Trevon	M	134419.23
James, Leslie	F	122563.38
Hoang, Binh	M	111074.86
Capelle, Adam	M	108138.43

Delete Rows or Columns

Delete a Row

Start with 37 rows of data.

```
d.shape
```

```
(37, 8)
```

Use the `drop()` function to delete a row by row name, to result in 36 rows.

```
d2 = d.drop('Wu, James')
d2.shape
```

```
(36, 8)
```

Delete a row by row index, here the second row, to result in 36 rows.

```
d2 = d.drop([d.index[1]])
```

```
d2.shape
```

```
(36, 8)
```

The data for James Wu is gone.

```
d2.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86
Afshari, Anbar	6.0	F	ADMN	69441.93	high	2	100	100

Delete a Column

The function `drop()` deletes a row or column from the data frame, as specified by the `axis` parameter. The default value of `axis` is `'rows'`, so if dropping a column, need to explicitly specify. Here drop the variable Plan from the resulting data frame of only numeric variables because it is an integer coded categorical variable. Dropping that variable leaves only continuous variables.

```
d_num = d2.drop(['Plan'], axis='columns')
d_num.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Pre	Post
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	82	92
Hoang, Binh	15.0	M	SALE	111074.86	low	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	65	62
Downs, Deborah	7.0	F	FINC	57139.90	high	90	86
Afshari, Anbar	6.0	F	ADMN	69441.93	high	100	100

Other Issues

A kind of "weird" issue exists (especially if you are used to R). Subsets (slices) of data frames by default do not create clean copies. The new data frame is still linked to the original data frame. Changes to the newly created data frame change the original as well! Modifications to the data or indices of the copy are reflected back to the original object.

To make the subset data frame completely independent of *d*, what pandas calls a *deep copy*, invoke the subset extraction with the `copy()` function. This function is not needed if there will not be further manipulation of the contents of the newly created data frame. Unless memory is an issue for very large data sets, or you know that there will be no further modification, using copy() is a good practice.

```
d2 = d.loc[:, 'Salary'].copy()
d2.head()
```

```
    Name
    Ritchie, Darnell     53788.26
    Wu, James            94494.58
    Hoang, Binh         111074.86
    Jones, Alissa        53772.58
    Downs, Deborah       57139.90
    Name: Salary, dtype: float64
```

A short-hand specification to select a column does not call any method or function and only includes the names of the relevant columns. When learning a language, better to focus on the more complete implementation of a concept. However, there is a need to know this abbreviated form because it appears often in real-world applications.

```
d2 = d['Salary']
d2.head()
```

```
    Name
    Ritchie, Darnell     53788.26
    Wu, James            94494.58
    Hoang, Binh         111074.86
    Jones, Alissa        53772.58
    Downs, Deborah       57139.90
    Name: Salary, dtype: float64
```

Variable Transformation

Transform the values of a numerical variable with an equation, a function that specifies how each value is to be transformed. Transform the values of a categorical variable with a recoding that specifies how each individual value is to be replaced with a new value.

Numeric Variable

To transform the values of a numeric variable is straightforward: Enter the corresponding equation that defines the transformation. Refer to a variable within a data frame with the data frame name, such as *d*, and then the variable name within quotes and square brackets. For example, the refer to the *Salary* variable in the *d* data frame: `d['Salary']`.

This example creates a new variable, Salary000, defined as the original Salary variable with values divided by 1000, rounded to two decimal digits. The new variable is added to the already existing variables in the *d* data frame. Running the equation creates the values of the new variable for all rows of data in the data frame.

```
d['Salary000'] = round(d['Salary'] / 1000, 2)
d.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post	Salary000
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92	53.79
Wu, James	NaN	M	SALE	94494.58	low	1	62	74	94.49
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97	111.07
Jones,	5.0	F	NaN	59779.59	NaN	1	85	89	59.77

Categorical Variable

The function `replace()` recodes individual values of a categorical variable. Parameter `to_replace` indicates the values to be replaced, and parameter `value` indicates the replacement value. Here replace across the entire data frame. Here recode both values of Gender with one statement.

```
d.dtypes
```

```
Years        float64
Gender        object
Dept          object
Salary       float64
JobSat        object
Plan           int64
Pre            int64
Post           int64
Salary000    float64
dtype: object
```

This way to recode with `replace()` replaces any 'F' and 'M' in the entire data table, which turns out to be just for the values of *Gender* in this example.

```
d_obj = d.replace(to_replace=['F', 'M'], value=['Female', 'Male'])
d_obj.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post	Salary000
Ritchie, Darnell	7.0	Male	ADMN	53788.26	med	1	82	92	53.79
Wu, James	NaN	Male	SALE	94494.58	low	1	62	74	94.49
Hoang, Binh	15.0	Male	SALE	111074.86	low	3	96	97	111.07
Jones,	5.0	Female	NaN	59779.59	NaN	1	85	89	59.77

This following use of `replace()` targets just values of the variable *Gender*. The curly brackets `{` and `}` indicate a specific `pandas` data type called a dictionary. As with any dictionary, there is a keyword, and then the meaning. For example, the keyword 'F' has meaning 'Female'.

This example presents a dictionary within a dictionary. The keyword *Gender* as meaning 'F' and 'M', which each has their own meaning.

```
d_obj = d.replace(to_replace={'Gender': {'F': 'Female', 'M': 'Male'}})
```

```
dd = d.replace({'Gender': {'F': 'Female', 'M': 'Male'}})
dd.head()
```

Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post	Salary000
Ritchie, Darnell	7.0	Male	ADMN	53788.26	med	1	82	92	53.79
Wu, James	NaN	Male	SALE	94494.58	low	1	62	74	94.49
Hoang, Binh	15.0	Male	SALE	111074.86	low	3	96	97	111.07
Jones,	5.0	Female	NaN	58778.58	NaN	1	65	69	58.77

Rename a Variable

Use the `rename()` function, which allows to specify both the old name and the new name. If defining a new data frame, safer to add the `copy()` function to make sure it is a clean copy with no relation to the original data frame. Or, if wishing to just change the names in the current data frame, no need to set *d* to another data frame. Set the `inplace` parameter to `True` instead.

In this example, a Python dictionary defines the values to replace each specified individual value. That is, replace the variable name Dept with Section, and JobSat with Satisfaction.

```
d2 = d.rename(columns = {'Dept': 'Section',
                         'JobSat': 'Satisfaction'}).copy()
d2.head()
```

Name	Years	Gender	Section	Salary	Satisfaction	Plan	Pre	Post	Sal
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92	
Wu, James	NaN	M	SALE	94494.58	low	1	62	74	
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97	
Jones,	5.0	F	NaN	58778.58	NaN	1	65	69	

Convert to Category Variable Type

In any data analysis, always be aware of two fundamental types of variables: Numerical and categorical.

> *Categorical variable*: A variable that consists of only a relatively small number of unique values.

An example of a categorical variable is the department in which an employee works, such as sales, marketing, etc. Another example is eye color, with values of Blue, Green, Brown, Black, and Gray. The values of categorical variables may also be integers, such as the responses to an attitude question on a survey from Strongly Disagree to Strongly Agree encoded as as integers from 1 to 7.

To represent the values of a categorical variable, Python presents the `category` variable type. Variables with character string values are initially read as type `object`. Variables with integer values are initially read as type `int64`. If the variable is categorical, then best to convert the variable to type `category`. This conversion saves memory, and can change the order of the display of the categories (values).

Assign a new variable type with the `astype()` function.

```
d = pd.read_excel('http://lessRstats.com/data/employee.xlsx', index_col=0)
#d = pd.read_excel('data/employee.xlsx')

d.Gender = d.Gender.astype('category')
d.Dept = d.Dept.astype('category')
d.JobSat = d.JobSat.astype('category')
d.Plan = d.Plan.astype('category')
d.dtypes
```

```
Years      float64
Gender    category
Dept      category
Salary     float64
JobSat    category
Plan      category
Pre          int64
Post         int64
dtype: object
```

> *Loop*: A computer instruction that repeats until a specified condition is reached.

Python has a type of loop called a `for` loop, which processes one or more expressions over a range of values.

Here use a `for` loop to consider each variable one at a time instead of a separate equation for each variable, as was done in the previous cell. The loop applies the `astype()` function to each of the specified variables, beginning with *Gender*. The expression `d[col]` refers to each specified variable, one at a time, beginning with *Gender* and ending with *Plan*.

```
d = pd.read_excel('http://lessRstats.com/data/employee.xlsx')
#d = pd.read_excel('data/employee.xlsx')
for col in ['Gender', 'Dept', 'JobSat', 'Plan']:
    d[col] = d[col].astype('category')
d.dtypes
```

```
Name        object
Years      float64
Gender    category
Dept      category
Salary     float64
JobSat    category
Plan      category
Pre          int64
Post         int64
dtype: object
```

```
d.head()
```

	Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
0	Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
1	Wu, James	NaN	M	SALE	94494.58	low	1	62	74
2	Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
3	Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
4	Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

Binning

Pandas function `qcut()` bins a continuous variable into discrete categories according to the specified quantiles. The optional `labels` parameter provides names for the bins, otherwise integers numbered from 0.

```
d2['Sal_bin'] = pd.qcut(d2['Salary'], q=[0,.25,.50,.75,1],
                        labels=['Low', 'Med', 'High', 'Top'])
d2.head(6)
```

Name	Years	Gender	Section	Salary	Satisfaction	Plan	Pre	Post	Sal
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92	
Wu, James	NaN	M	SALE	94494.58	low	1	62	74	
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97	
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62	

Summarize Data

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Preliminaries

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-07-01 at 19:02
```

After the data are prepared as a tidied data table and read into the analysis system such as Python, the first analysis step summarizes the data with descriptive statistics and visualizations. As always in data analysis, distinguish between *categorical variables* and *continuous variables* in the analysis.

If reading data from your computer's file system or a networked computer, identify the current working directory. This folder is the reference point in your file system for which file references begin. Then import the needed libraries and read the data for analysis.

As an option, list the current working directory, which, by default, is where this Jupyter notebook file is located. Your current working direction is the default location for reading and writing files. The working directory of `/content` is from running on Google Colab.

```
import os
os.getcwd()
```

```
'/content'
```

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

Display matplotlib visualizations in the notebook. Not needed for Colab. If using, remove the `#` comment, the first character in the following code.

```
#%matplotlib inline
```

Can also display standard HTML links. For example, here view a list of all available color names to use when creating a visualization.
named colors with a sample of each color

Can read the data from the Excel data file *employee.xlsx* on the web. Or, download to your computer or Google Drive and read from there as described the previous week, especially if going to be working on your computer without internet.

```
d = pd.read_excel('http://lessRstats.com/data/employee.xlsx')
```

```
#d = pd.read_excel('data/employee.xlsx')
```

Always verify that the data were read as you intended.

```
d.shape
```

```
(37, 9)
```

```
d.head()
```

	Name	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
0	Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
1	Wu, James	NaN	M	SALE	94494.58	low	1	62	74
2	Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
3	Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
4	Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

Categorical Variables

Counts

One Categorical Variable

Compute the counts of categorical data values with `pandas` function `value_counts()`, here categorical variable *Dept* within data frame *d*. Multiple ways to specify a variable as part of a data frame. First specify with the `.` notation.

```
d.Dept.value_counts()
```

```
SALE    15
ADMN     6
MKTG     6
ACCT     5
FINC     4
Name: Dept, dtype: int64
```

Alternatively, use the `[]` notation.

```
d['Dept'].value_counts()
```

```
SALE    15
ADMN     6
MKTG     6
ACCT     5
FINC     4
Name: Dept, dtype: int64
```

Yet a third notation to get the counts. Illustrated here to highlight that virtually any operation can be done multiple ways. If you are used to one way of coding an analysis, do not be surprised if someone else does it differently.

```
d.value_counts('Dept')
```

```
Dept
SALE    15
MKTG     6
ADMN     6
ACCT     5
FINC     4
dtype: int64
```

And a fourth way to get the counts, just to emphasize that there is typically no standard one way to do an analysis. Also, note that `value_counts()` is a `pandas` function, but when preceded by a data frame name and a dot in the function call, the leading `pd.` is not needed. A data frame is a `pandas` object, so the reference to `pandas` is understood. But when by itself, the `pd.` must precede the function call.

```
pd.value_counts(d.Dept)
```

```
SALE    15
ADMN     6
MKTG     6
ACCT     5
FINC     4
Name: Dept, dtype: int64
```

To obtain the relative frequencies or proportions instead of the frequencies or counts, set the `normalize` paramaeter to `True`.

```
d.Dept.value_counts(normalize=True)
```

```
SALE    0.416667
ADMN    0.166667
MKTG    0.166667
ACCT    0.138889
FINC    0.111111
Name: Dept, dtype: float64
```

The categorical variable *JobSat* has categories 'low', 'med', and 'high'. Python does not understand the English language, and so has no way of knowing how these categories should be ordered. Python does not know that 'low' is less than 'high', for example. Instead, use the `Categorical()` function to explicitly define a variable as categorical with the option to specify the ordering of the categories when accessed in subsequent analyses.

In this example, *JobSat* is converted from type `object` as read to type `category`.

```
d['JobSat'] = pd.Categorical(d['JobSat'], categories=['low', 'med', 'high'], ordered=True)
d.dtypes
```

```
Name        object
Years      float64
Gender      object
Dept        object
Salary     float64
JobSat    category
Plan         int64
Pre          int64
Post         int64
dtype: object
```

Two Categorical Variables

Use the `pandas` function `crosstab()` for cross-tabulation, to compute two-way frequency distributions for categorical variables. Can refer to the variables with the `.` notation.

```
pd.crosstab(d.Dept,d.Gender)
```

Gender	F	M
Dept		
ACCT	3	2
ADMN	4	2
FINC	1	3
MKTG	5	1
SALE	5	10

Alternatively, use the [ ] notation in the call to `crosstab()`.

```
ct = pd.crosstab(d['Dept'], d['Gender'])
ct
```

Gender	F	M
Dept		
ACCT	3	2
ADMN	4	2
FINC	1	3
MKTG	5	1
SALE	5	10

Bar Charts

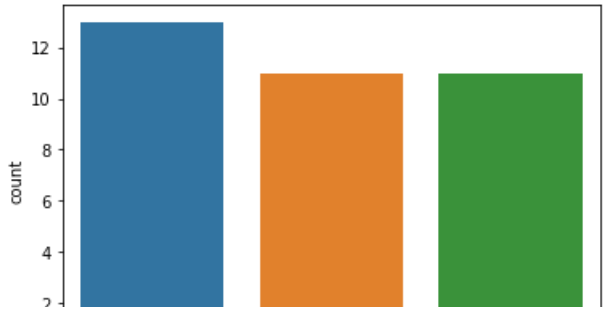
The primary data visualization package for Python data analysis is `matplotlib`. A more recent development is the `seaborn` package which builds upon `matplotlib`, designed to provide more elegant graphics with less work. Most of the examples presented here are from `seaborn`, with the abbreviation `sns`. They could also be replicated with `matplotlib`, continuing the theme that almost always, there is more than one way to proceed.

`seaborn` offers the function `countplot()` to count the number of occurrences for each category of a categorical variable. The function requires to include the parameter name with the value of each parameter, not relying upon the position of the value in the parameter list. For example, a positional system would not require the parameter name for the first parameter listed in the function definition.

One Categorical Variable

```
sns.countplot(x='JobSat', data=d)
```

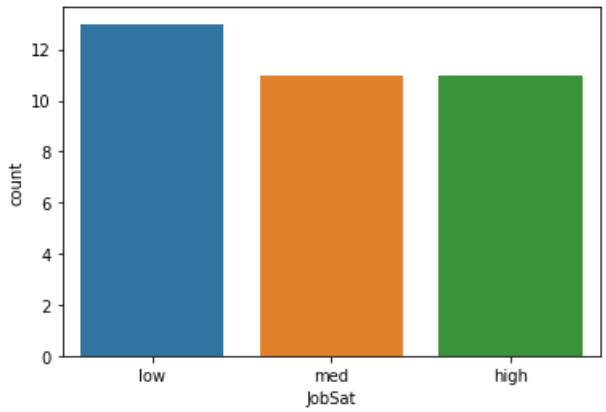
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff22a56e390>
```



If the categorical variable was not defined as an ordered categorical variable with the function `Categorical()`, you can specify the ordering of the categories in the call to `countplot()`.

```
sns.countplot(x=d['JobSat'], order=['low', 'med', 'high'])
```

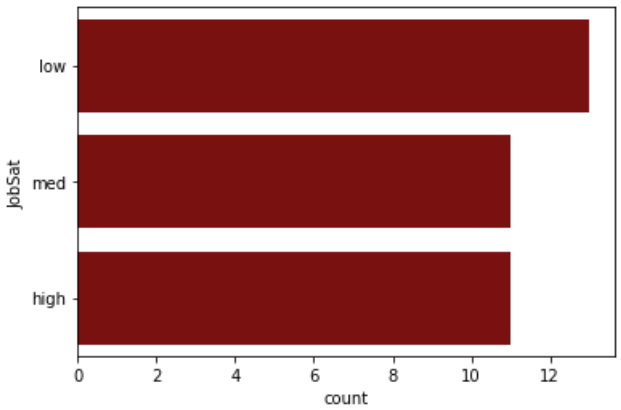
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff229479090>
```



With parameter `y`, request that the categories be placed on the y-axis, that is, a horizontal bar chart. With parameter `color`, specify a custom color.

```
sns.countplot(y='JobSat', data=d, color='darkred')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228ff8410>
```

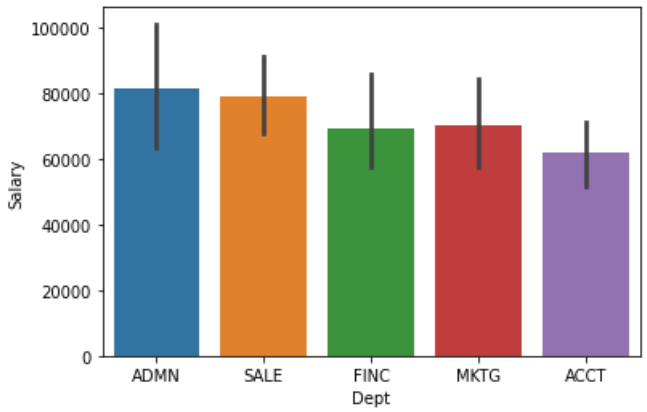


The `seaborn` package has multiple functions that yield bar charts. The previously illustrated `countplot()` function plots the height of the bars according to the count (frequency) or proportion (relative frequency) of each category. The function named `barplot()` applies to one categorical variable and then a numerical variable that defines the height of the bars.

By default, the mean of the numerical variable for each category is plotted, along with a corresponding error bar that illustrates the 95% confidence interval.

```
sns.barplot(x='Dept', y='Salary', data=d)
```

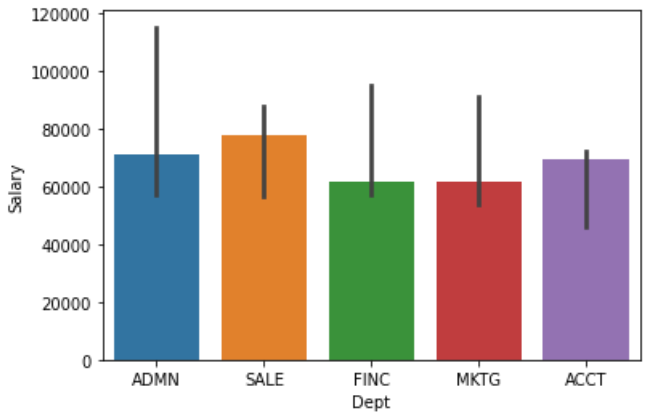
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228f122d0>
```



The default statistic plotted for the height of the bars is the mean of the numerical variable. Use the parameter `estimator` to specify another statistic. The `numpy` package defines these statistics, so precede the statistic's name with `np.` when calling the `barplot()` function.

```
sns.barplot(x='Dept', y='Salary', data=d, estimator=np.median)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff22a05b8d0>
```



Two Categorical Variables

The bar chart of two categorical variables is presented in one of two basic forms, stacked or unstacked. This first example is unstacked, with the bars separated for the levels of the second variable, *Gender*, at each level of the first variable, *Dept*.

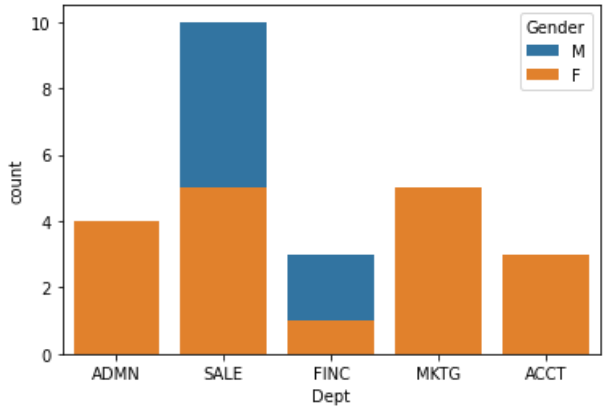
```
sns.countplot(x='Dept', hue='Gender', data=d)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228e0ff50>
```

The `seaborn` version of the stacked form of the bar chart, indicated by the `dodge` parameter, does not appear to work. The bars for categories SALE and FINC work, but the remaining bars do not show the Male counts.


```
sns.countplot(x='Dept', hue='Gender', dodge=False, data=d)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228e8a790>
```

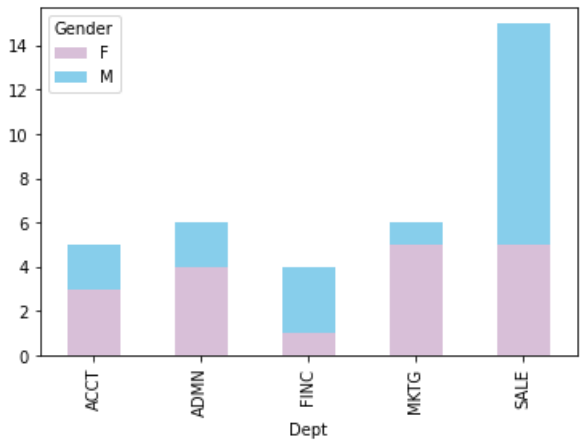


Go to the foundational `matplotlib` with its `plot()` function to get the stacked bar chart to work. Indicate a bar chart with the `kind` parameter and set `stacked` to `True`. For `matplotlib` we need to feed the output of `crosstabs()`, which we previously called *ct*, into the `plot()` function to visualize the cross-tabulation table as a bar chart of two categorical variables. Add some optional colors with the `color` parameter. Find a full list of named colors.

One way to use `plot()` is to enter the cross-tabulation matrix into the analysis. Do so by beginning the function call with the name of the matrix, here *ct*.

```
ct.plot(kind='bar', color=['thistle','skyblue'], stacked=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228cd3490>
```



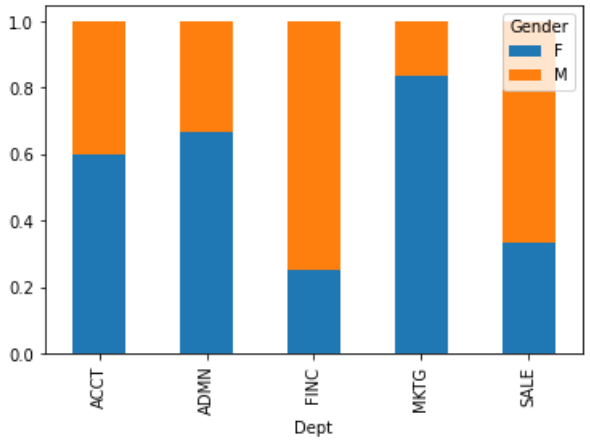
The 100% bar chart for these two categorical variables shows the *percentage* of *Gender* distributed across *each* department. This visualization compares the second categorical variable, here Gender, across the levels of the first variable, particularly useful when there are unequal group sizes.

Specify the grouping of the categorical variables with the `pandas` function `groupby()`. The key to getting the 100% stacked bar chart is to set the `normalize` parameter to `True`, which, as shown previously, converts the counts to frequencies. With

`matplotlib`, plot the 100% stacked bar chart, where each bar goes the complete 100%. This example uses the chain method explained in the previous week, where each function call appears on a separate line.

```
(d
 .groupby('Dept')['Gender']
 .value_counts(normalize=True)
 .unstack('Gender')
 .plot.bar(stacked=True)
)
```

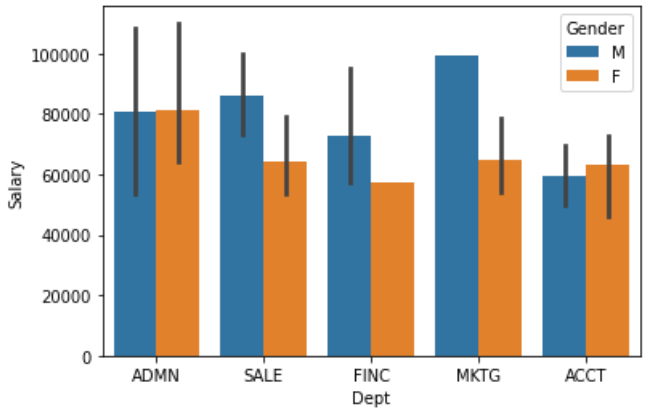
```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228c6d6d0>
```



Here apply `barplot()` to two categorical variables, specified with parameters of `x` and `hue`. Specify the numeric variable for analysis with parameter `y`.

```
sns.barplot(x='Dept', y='Salary', hue='Gender', data=d)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228bfc190>
```



Continuous Variables

Statistics

Methods are available for individual statistics.

Calculate the mean of a variable with the `pandas` function `mean()`.

```
d["Salary"].mean()
```

```
73795.55675675675
```

Usually better to also invoke the `round()` function to not have so many decimal digits. In this example, save the mean in its own variable, *m*, then round to two decimal digits.

```
m = d["Salary"].mean()
round(m, 2)
```

```
73795.56
```

The `pandas` function `describe()` computes summary statistics of continuous variables over the entire data frame. Unmodified it applies to all the continuous variables. The basic summary statistics that describe the sample values of a continuous variable: number of non-missing values (count), mean (mean), standard deviation (std), minimum (min), first quartile (25%), median (50%), third quartile (75%), and maximum (max).

```
d.describe()
```

	Years	Salary	Plan	Pre	Post
count	36.000000	37.000000	37.000000	37.000000	37.000000
mean	9.388889	73795.556757	1.783784	78.783784	81.000000
std	5.723524	21799.533464	0.712396	12.037292	11.592622
min	1.000000	46124.970000	1.000000	59.000000	59.000000
25%	5.000000	56772.950000	1.000000	70.000000	72.000000
50%	9.000000	69547.600000	2.000000	80.000000	84.000000
75%	13.000000	87785.510000	2.000000	90.000000	91.000000
max	24.000000	134419.230000	3.000000	100.000000	100.000000

Or, apply to one or more selected variables, here the variable Salary.

```
d.Salary.describe()
```

```
count        37.000000
mean      73795.556757
std       21799.533464
min       46124.970000
25%       56772.950000
50%       69547.600000
75%       87785.510000
max      134419.230000
Name: Salary, dtype: float64
```

Pivot Tables

Summarizing numerical data over different sub-groups of a data set is a standard analysis technique. Calculate descriptive statistics across groups of data.

> *Aggregation*: Summarize a numerical variable with descriptive statistics computed over combinations of groups defined by one or more categorical variables.

The analysis aggregates statistics across the numerical variables grouped by levels of one or more categorical variables. Excel refers to a table of aggregated statistics as a *pivot table*.

One possibility for aggregation is the `pandas` function `groupby()`, applied to data frames. Here compute the mean of the different groups for all numerical variables in the data frame across the levels of *Dept*.

Note: Available functions include `count()`, `sum()`, `mean()`, `median()`, `min()`, `max()`, `mode()`, `std()`, and `var()`.

Note: This is easier than all that clicking and mousing to get the same result with the Excel pivot table function. (Can also easily write the results back to Excel with function `to_excel()`.)

```
d.groupby('Dept').mean()
```

Dept	Years	Salary	Plan	Pre	Post
ACCT	5.600000	61792.776000	2.00	76.600000	78.600000
ADMN	10.166667	81277.116667	2.00	80.833333	82.000000
FINC	10.250000	69010.675000	1.75	79.750000	82.250000
MKTG	9.833333	70257.128333	2.00	79.666667	84.000000
SALE	10.285714	78830.064667	1.60	79.000000	81.133333

To define a two-way grouping, provide a vector of variable names to `groupby()`.

```
d.groupby(['Dept', 'Gender']).mean()
```

Dept	Gender	Years	Salary	Plan	Pre	Post
ACCT	**F**	4.666667	63237.163333	2.000000	73.333333	77.333333
	M	7.000000	59626.195000	2.000000	81.500000	80.500000
ADMN	**F**	7.500000	81434.002500	2.250000	80.000000	79.750000
	M	15.500000	80963.345000	1.500000	82.500000	86.500000
FINC	**F**	7.000000	57139.900000	2.000000	90.000000	86.000000
	M	11.333333	72967.600000	1.666667	76.333333	81.000000
MKTG	**F**	8.200000	64496.022000	1.800000	79.400000	84.000000
	M	18.000000	99062.660000	3.000000	81.000000	84.000000
SALE	**F**	6.600000	64188.254000	1.600000	76.200000	78.600000
	M	12.333333	86150.970000	1.600000	80.400000	82.400000

Use the `agg()` function to aggregate the values of multiple numerical variables. With this function, need to precede the function names with `np.` as they are functions defined in the `numpy` package.

Here write the code for the aggregation in chain function notation.

```
(d
 .groupby(['Dept', 'Gender'])
 .agg([np.mean, np.median])
)
```

		Years		Salary		Plan		Pre
		mean	median	mean	median	mean	median	mean
Dept	Gender							
ACCT	F	4.666667	3.0	63237.163333	71084.020	2.000000	2.0	73.33
	M	7.000000	7.0	59626.195000	59626.195	2.000000	2.0	81.50
ADMN	F	7.500000	5.0	81434.002500	71058.595	2.250000	2.0	80.00
	M	15.500000	15.5	80963.345000	80963.345	1.500000	1.5	82.50
FINC	F	7.000000	7.0	57139.900000	57139.900	2.000000	2.0	90.00
	M	11.333333	10.0	72967.600000	66312.890	1.666667	1.0	76.33
MKTG	F	8.200000	8.0	64496.022000	61356.690	1.800000	2.0	79.40
	M	18.000000	18.0	99062.660000	99062.660	3.000000	3.0	81.00
SALE	F	6.600000	8.0	64188.254000	56508.320	1.600000	2.0	76.20
	M	12.333333	13.0	86150.970000	82442.740	1.600000	1.0	80.40

Of course, as seen before, there are multiple ways to proceed. Another possibility uses the `pandas` function `pivot_table()` for aggregating (pivoting) the values of one or more continuous variables over different groups defined by one or more categorical variables.

Specify the continuous variables over which to aggregate with the `values` parameter. The `index` parameter specifies the categorical variables that define the groups. The `aggfunc` parameter specifies the statistic for the aggregation, here the mean. Note that the calculation of the mean here is from the `numpy` package, abbreviated `np` upon which `pandas` depends.

```
pd.pivot_table(d, values=['Years', 'Salary', 'Plan', 'Pre', 'Post'],
               index=['Dept', 'Gender'], aggfunc=np.mean)
```

		Plan	Post	Pre	Salary	Years
Dept	Gender					
ACCT	F	2.000000	77.333333	73.333333	63237.163333	4.666667
	M	2.000000	80.500000	81.500000	59626.195000	7.000000
ADMN	F	2.250000	79.750000	80.000000	81434.002500	7.500000
	M	1.500000	86.500000	82.500000	80963.345000	15.500000
FINC	F	2.000000	86.000000	90.000000	57139.900000	7.000000
	M	1.666667	81.000000	76.333333	72967.600000	11.333333
MKTG	F	1.800000	84.000000	79.400000	64496.022000	8.200000
	M	3.000000	84.000000	81.000000	99062.660000	18.000000
SALE	F	1.600000	78.600000	76.200000	64188.254000	6.600000
	M	1.600000	82.400000	80.400000	86150.970000	12.333333

If you do not specify the `values` parameter, then all numeric variables are analyzed.

```
pd.pivot_table(d, index=['Dept', 'Gender'], aggfunc=np.mean)
```

Dept	Gender	Plan	Post	Pre	Salary	Years
ACCT	F	2.000000	77.333333	73.333333	63237.163333	4.666667
	M	2.000000	80.500000	81.500000	59626.195000	7.000000
ADMN	F	2.250000	79.750000	80.000000	81434.002500	7.500000
	M	1.500000	86.500000	82.500000	80963.345000	15.500000
FINC	F	2.000000	86.000000	90.000000	57139.900000	7.000000
	M	1.666667	81.000000	76.333333	72967.600000	11.333333
MKTG	F	1.800000	84.000000	79.400000	64496.022000	8.200000
	M	3.000000	84.000000	81.000000	99062.660000	18.000000
SALE	F	1.600000	78.600000	76.200000	64188.254000	6.600000

Visualizations

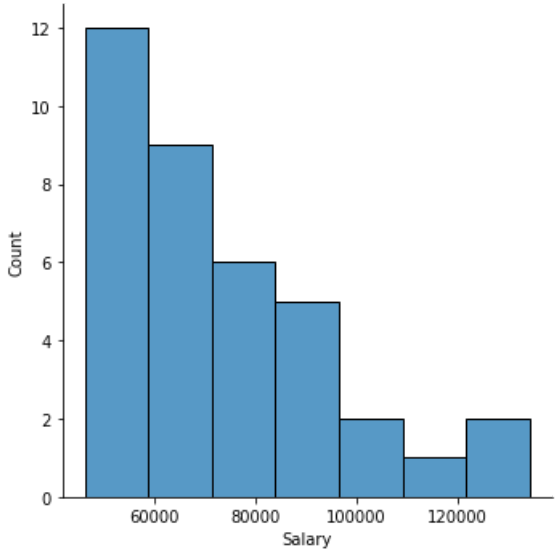
The distribution of a continuous variable can be presented several different ways. Perhaps the two most encountered visualizations are histograms and box plots.

Histogram

`seaborn` provides two functions for generating histograms, `histplot()` and `displot()`. Here we focus on the later.

```
sns.displot(d, x='Salary')
```

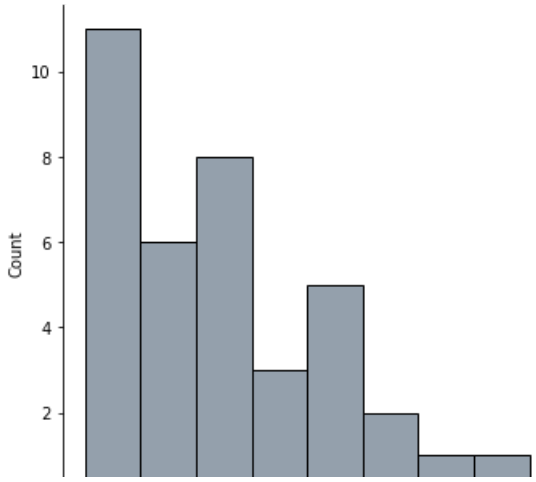
```
<seaborn.axisgrid.FacetGrid at 0x7ff228bf4d10>
```



Reference the parameter *color* for a custom color. Specify the number of bins with the `bins` parameter. Could also use the `binwidth` parameter.

```
sns.displot(d, x='Salary', color='slategray', bins=8)
```

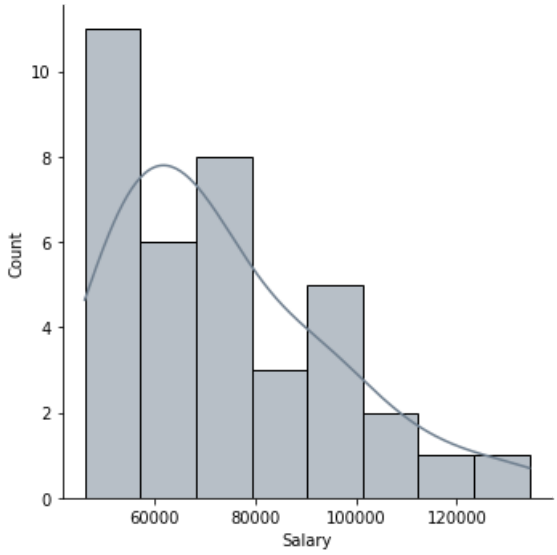
```
<seaborn.axisgrid.FacetGrid at 0x7ff228c03e10>
```



The `seaborn` package provides more advanced plots. For example, easily overlay a smoothed frequency-type curve, called a *density plot*, on the histogram with the function `displot()`.

```
sns.displot(d, x='Salary', color='slategray', bins=8, kde=True)
```

```
<seaborn.axisgrid.FacetGrid at 0x7ff228a9f250>
```



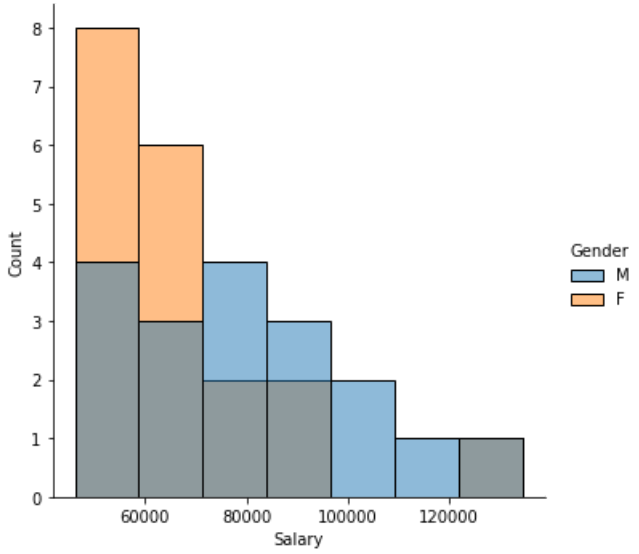

```
from numpy.random import normal
sim_data = normal(size=1000)
sns.displot(x=sim_data)
```

```
<seaborn.axisgrid.FacetGrid at 0x7ff21e16ac90>
```

Throughout the various `seaborn` functions, generate multiple visualizations on the same panel according to different groups with the `hue` parameter.

```
sns.displot(d, x='Salary', hue='Gender')
```

```
<seaborn.axisgrid.FacetGrid at 0x7ff21e15f210>
```

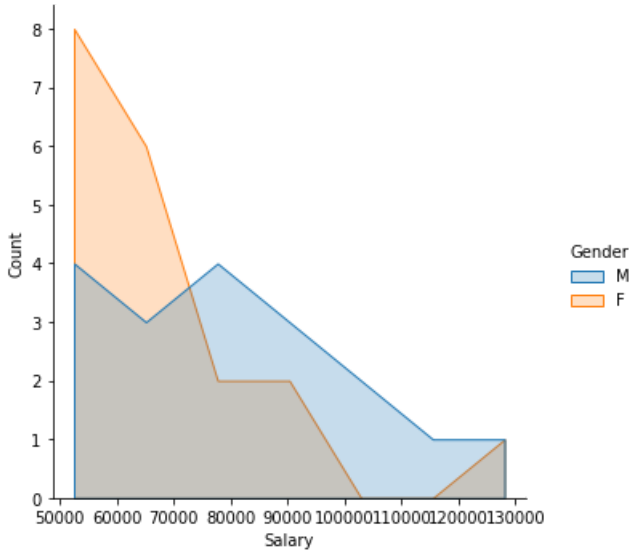


A closely related visualization to a histogram is a frequency polygon, which may provide a helpful portrayal of the distributions for overlapping distributions. For example, in the above histogram, overlapping values are shown in gray. In the frequency polygon version, the polygons overlap as a blending of their respective colors.

Specify with the `element` parameter.

```
sns.displot(d, x='Salary', hue='Gender', element='poly')
```

```
<seaborn.axisgrid.FacetGrid at 0x7ff21e131910>
```

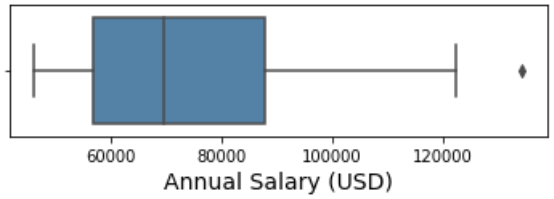


Box Plot

The box plot has already been demonstrated in an earlier notebook, but included here for completeness as a standard visualization of a distribution, emphasizing detecting outliers. Use the `seaborn` function `boxplot()`. The height of the box plot is not relevant, and by default is quite large, so set to a small number, 1.5, with the `matplotlib figure()` function.

```
plt.figure(figsize=(6,1.5))
sns.boxplot(x=d['Salary'], color='steelblue')
plt.xlabel('Annual Salary (USD)', fontsize=14)
```

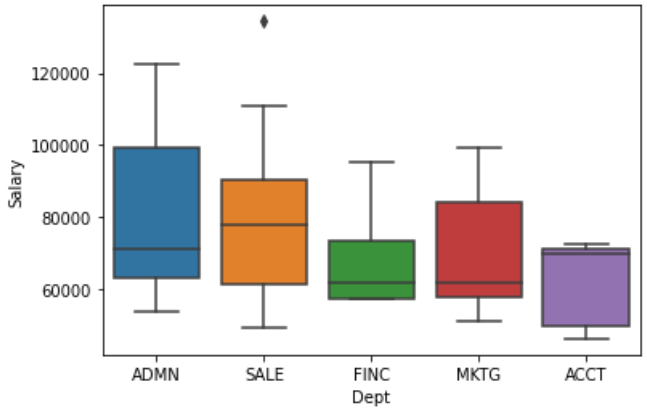
```
Text(0.5, 0, 'Annual Salary (USD)')
```



The box plot is an excellent way to visualize the distribution of a continuous variable across levels of a categorical variable. Here compare *Salary* across the different *Dept* or departments of a company.

```
sns.boxplot(x='Dept', y='Salary', data=d)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff228c73e50>
```



Can go one step further and specify a second categorical variable at each level of the first categorical variable. Here, with the `hue` parameter, show box plots for *Gender* at each *Dept*. Change the colors from the default blue and orange with the `palette` parameter.

```
plt.figure(figsize=(8,6))
sns.boxplot(x='Dept', y='Salary', hue='Gender', palette=['steelblue', 'gold'], data=d)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff21deb3850>
```

Correlational Structure

The *correlational structure* refers to the (linear) relations among the variables, pairwise, that is, two at a time. The sample or population correlation varies between -1 and 1, which respectively indicate perfect – or + linear relationship. A correlation of 0 indicates no linear relationship.

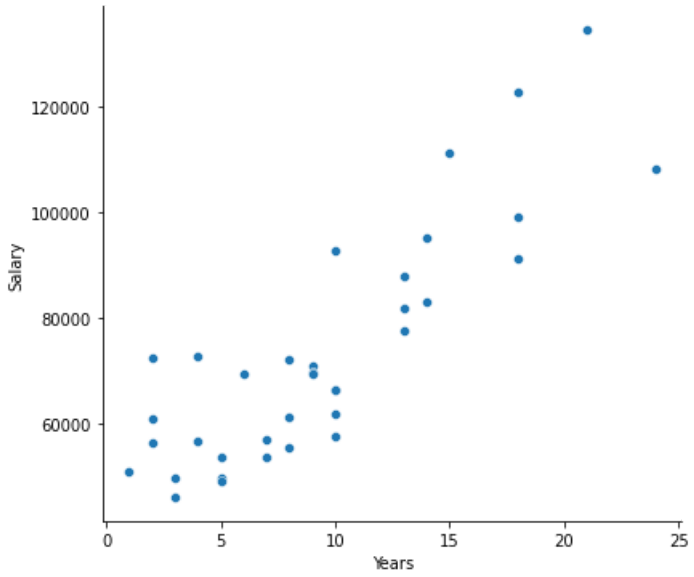
Scatterplot

The classic visualization of the relation between two numerical variables is the *scatterplot*, which plots each pair of values for the two variables for a row of data as a point. The coordinates of the plotted point are the values of the two variables for that row of data.

A `seaborn` scatterplot function is `relplot()`. The `aspect` parameter controls the ratio of height and width.

```
sns.relplot(x='Years', y='Salary', data=d, aspect=1.2)
```

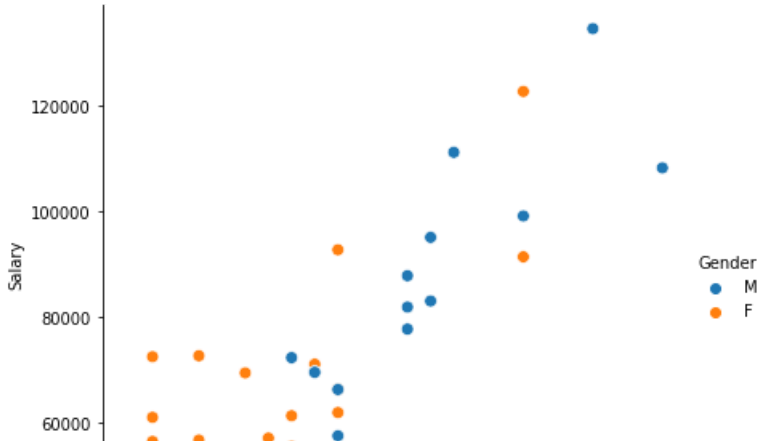
```
<seaborn.axisgrid.FacetGrid at 0x7ff22022e8d0>
```



Consistent with other `seaborn` functions, introduce a grouping variable with the `hue` parameter. Control the size of the points with the `s` parameter.

```
sns.relplot(x='Years', y='Salary', hue='Gender', s=60, data=d, aspect=1.2)
```

```
<seaborn.axisgrid.FacetGrid at 0x7ff21dce4890>
```



Correlation matrix

The correlation coefficient varies from -1 for a perfectly negative or inverse relationship, to 0 for no relationship, to +1 for a perfectly positive relationship.

Get correlations with the `pandas` function `corr()`. Here calculate the correlation between number of *Years* worked for the company and annual *Salary*. Clearly an employee's salary is related to the length of time working for the company.

```
d['Years'].corr(d['Salary']).round(2)
```

```
0.85
```

The correlation matrix in traditional form, a square matrix that, given a list of variables, contains the correlation for each variable in the list, twice, because the correlation coefficient is the same regardless of the order of the two variables: $r_{12} = r_{21}$. 1's run down what is called the *principle diagonal*, indicating that each variable correlates with itself a perfect 1.0. It is distracting and generally useless to display more than two decimal digits for each of the displayed correlations.

```
d.corr().round(2)
```

Heat map

The heat map visualizes the correlation matrix, color coded according to the intensity of each correlation. Get the heat map with the seaborn function `heatmap()`. Specify the parameter `annot` for annotation to be `True` to also provide the numerical value of the correlations. Here use the `loc` method to select the variables for the heat map.

```
keep_vars = ['Years', 'Salary', 'Pre', 'Post']
d2 = d.loc[:, keep_vars]
sns.heatmap(d2.corr().round(2), linewidths=2.0, annot=True)
```

Scatterplot matrix

The *scatterplot matrix* has the form of a correlation matrix, but replaces each correlation with the corresponding scatterplot. The diagonal elements of the correlation matrix are replaced with a plot of the distribution of the variable.

Get the scatterplot matrix with the seaborn function `pairplot()`.

Can also use the `kind` parameter to specify a regression line for each scatterplot, and the `diag_kind` parameter to specify kernel density plots (kde) in the diagonal. If there is no `hue` parameter, the default is a histogram for each diagonal element.

```
sns.pairplot(data=d, vars=["Years", "Salary"], kind="reg",
             diag_kind="kde")
```

The pairplot can be obtained for multiple levels of a categorical variable. The parameter `hue` specifies a categorical variable from which to map its levels to different colors in scatterplot matrix.

```
sns.pairplot(d, vars=["Years", "Salary", "Pre", "Post"], hue='Gender')
```

Regression Analysis with One Predictor

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

This template shows how to do regression analysis with a single predictor with the `statsmodel` package.

Preliminaries

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```



```
:53
```

```
import os
os.getcwd()
```

```
'/content'
```

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

Read the Data

The data consist of variables based on regions of Boston from some decades back, with a focus on houses and housing prices.

```
#d = pd.read_csv('data/Boston.csv')
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/Boston.csv')
```

```
d.shape
```

```
(506, 15)
```

```
d.head()
```

	Unnamed: 0	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptr
0	1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	
1	2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	
2	3	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	
3	4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	
4	5	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	

Do not need the first column, so drop.

```
d = d.drop(["Unnamed: 0"], axis="columns")
d.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	blac
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.9
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.9
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.8
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.6
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.9

Do a missing data check before analysis.


```
zn          0
indus       0
chas        0
nox         0
rm          0
age         0
dis         0
rad         0
tax         0
ptratio     0
black       0
lstat       0
medv        0
dtype: int64
```

No missing data here, so can proceed as is.

Form X and y Data Structures

Build a model that forecasts/explains the median house price, *medv* in terms of the average number of rooms, *rm*.

- *medv*: Median value of owner-occupied homes in $1000's
- *rm*: Average number of rooms per dwelling

Store the features, the predictor variables, of which there is only one in this example, in data structure X. Store the target variable in data structure y. The uppercase X is used because in real-world applications, X invariably contains multiple variables.

```
y = d['medv']
X = d['rm']
```

A technical point, but one worth considering when doing data analysis, is to understand the type of data structures created throughout an analysis. The data as read are read into a `pandas` data structure called a *dataframe*. However ,when the data frame is sub-setted into X and y, both of which consist of only a single variable in this example, the result is a one-dimensional `pandas` data structure called a `series`. Actually, a *dataframe* consists of columns, and each column is a `series`. That is why reduction of the data frame to a single column results in a `series`.

For this particular analysis pursued here, being aware of this distinction is not necessary. But, in general, always good to know the underlying data structures. Thinking the data is of one type, when it is actually of another type, in many situations leads to programming errors.

To check the type of a variable, use the Python function `type()`.

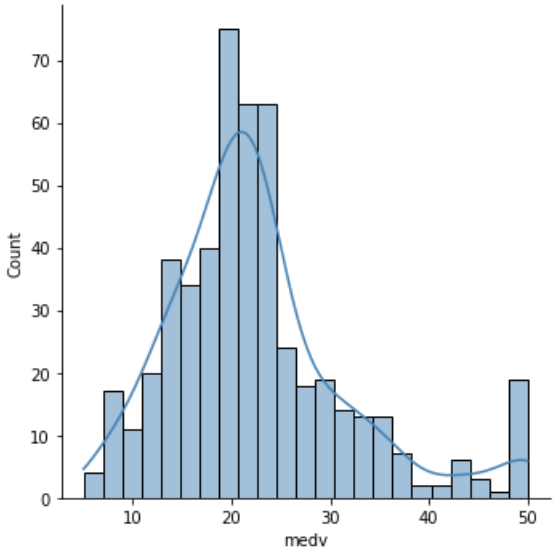
```
print("d: ", type(d))
print("X: ", type(X))
print("y: ", type(y))
```

```
d:  <class 'pandas.core.frame.DataFrame'>
X:  <class 'pandas.core.series.Series'>
y:  <class 'pandas.core.series.Series'>
```

Understand the distribution of the target variable, *medv*, to make sure that the distribution is not too weird. Show the distribution with its histogram and density estimate (smoothed histogram), obtained with the `seaborn` method `distplot()`.

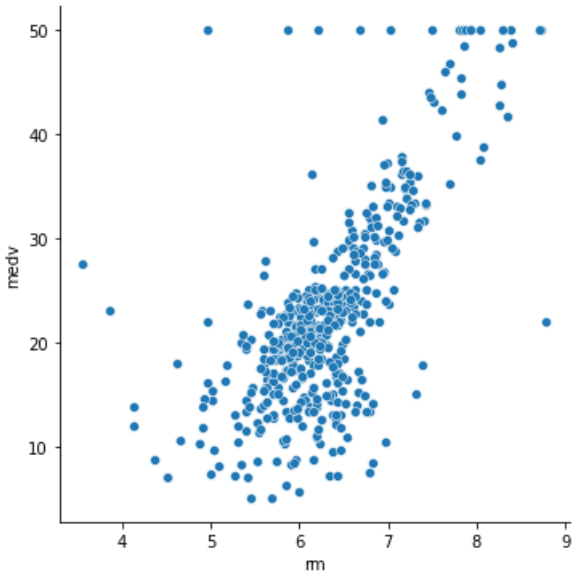

```
sns.displot(x= medv , color= steelblue', kde=True, data=d)
```

```
<seaborn.axisgrid.FacetGrid at 0x7f1527210a10>
<Figure size 324x360 with 0 Axes>
```



Before doing linear regression, first make sure that the relationship between the variables is at least roughly linear. Check via a scatterplot with the `seaborn` function `relplot()`.

```
ax = sns.relplot(x="rm", y="medv", data=d)
```



Can also use the `pandas` function `corr()` to get the correlation between predictor and target.

```
d['rm'].corr(d['medv']).round(2)
```

```
0.7
```

The variables are highly correlated with $r = 0.70$, and the scatterplot indicates an apparent linear relationship. The only "weird" issue is that apparently housing prices over 50,000 USD are truncated and listed at 50,000 USD. Probably a good idea to filter these rows of data out of the data table and generalize the results to houses with less than that value, but we will leave

Model Analysis

Estimation

For some reason, by default, the estimation procedure assumes a y-intercept of 0 unless there is constant value in the feature data. To compensate, before estimating the model, explicitly add a column of 1.0's to the X data structure so that the estimated model will have a y-intercept, and therefore fit better without requiring the assumption of a value of 0. Add the constant with the `statsmodels` package `add_constant()` function.

```
import statsmodels.api as sm
from statsmodels.regression.linear_model import RegressionResults
X = sm.add_constant(X)
X.head()
```

```
/usr/local/lib/python3.7/dist-packages/statsmodels/tools/_testing.py:19: Fu
  import pandas.util.testing as tm
```

	const	rm
0	1.0	6.575
1	1.0	6.421

Specify the model with the `OLS()` function from the `statsmodel` package. OLS means *ordinary least squares*, the default and usual estimation procedure for regression models. Specify the target variable first, followed by the data structure with the features. Do the least-squares regression analysis of the defined model by applying the `fit()` function.

Save the results of this regression analysis to a `statsmodel` data structure we name *results*. The `summary()` function summarizes the main results of the analysis.

```
model = sm.OLS(y, X)
results = model.fit()
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   medv   R-squared:                       0.484
Model:                            OLS   Adj. R-squared:                  0.483
Method:                 Least Squares   F-statistic:                     471.8
Date:                Wed, 30 Jun 2021   Prob (F-statistic):           2.49e-74
Time:                        21:53:24   Log-Likelihood:                -1673.1
No. Observations:                 506   AIC:                             3350.
Df Residuals:                     504   BIC:                             3359.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        -34.6706      2.650    -13.084      0.000     -39.877     -29.465
rm             9.1021      0.419     21.722      0.000       8.279       9.925
==============================================================================
Omnibus:                      102.585   Durbin-Watson:                   0.684
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              612.449
Skew:                           0.726   Prob(JB):                    1.02e-133
Kurtosis:                       8.190   Cond. No.                         58.4
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
```


The primary results of the model are in this section:

```
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        -34.6706      2.650    -13.084      0.000     -39.877     -29.465
rm             9.1021      0.419     21.722      0.000       8.279       9.925
```

The estimated values of the intercept, b_0 and b_1 are -34.67 and 9.10, respectively. So write the estimated model as:

$$\hat{y}_{medv} = -34.67 + 9.10(x_{rm})$$

In *this data set* only, for each increase in the average number of rooms, the average selling price increases by 9,102 USD. We do, however, need inferential statistics to generalize to the population as a whole.

Hypothesis test: The p-values for the t-statistics for each of the two estimated coefficients, `P>|t|`, are well below the cutoff of $\alpha = 0.05$. For the slope, assuming that there is no relationship between *rm* and *medv*, then the probability of getting an estimated slope coefficient as large as 9.10 is an extremely improbable event, indistinguishable from 0 to three decimal digits.

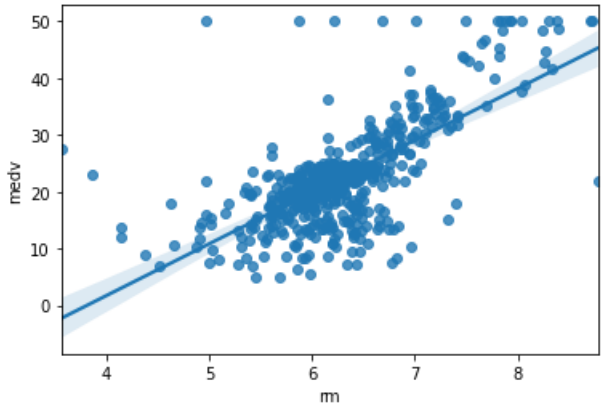
So reject the null hypothesis of no relationship, and conclude there is a positive relationship between *rm* and *medv*. As *rm* increases, so does *medv*.

Confidence interval: Accordingly, the 95% confidence interval for the slope, which contains the plausible values of the true, population value of the slope coefficient, is, with 95% confidence, in the interval from 8.279 to 9.925. That is, with 95% confidence, for each increase in the average number of rooms, the average selling price increases somewhere between 8,279

Fit

The `seaborn` plotting function `regplot()` automatically plots the scatterplot and the regression line through the scatterplot. Also provided is the confidence interval of the regression line (values computed over hypothetical repeated samples). Values of X futher from the middle have more variability, analogous to a teeter-tottor in which each end swings further than a place on the teeter-tottor closer to the middle, the fulcrum.

```
ax = sns.regplot(x="rm", y="medv", data=d)
```



From the line through the scatterplot, it looks reasonable. There is, of course, scatter about the line, but not so much.

To evaluate fit of the model with statistics, access various values computed by the `fit()` function, here stored in the structure called *results*. Use the functions `ssr()`, `mse_resid()`, and `rsquared()`. The RMSE or standard deviation of the residuals is not provided directly, so compute as the square root of the MSE. Use the function `sqrt()` from the `numpy` package. All values are rounded to two decimal digits with the `round()` function.

```
print("Sum of squared residuals:", results.ssr.round(2))
print("Mean squared error:", results.mse_resid.round(2))
RMSE = np.sqrt(results.mse_resid)
print("Stdev of residuals:", RMSE.round(2))
res_range = 4 * RMSE
print("95% range of residuals:", res_range.round(2))
print("R-squared:", results.rsquared.round(2))
```

```
Sum of squared residuals: 22061.88
Mean squared error: 43.77
Stdev of residuals: 6.62
95% range of residuals: 26.46
R-squared: 0.48
```

The sum of the squared residuals is provided for reference, upon which the more meaningful fit indices are derived. The standard deviation of the residuals, consistent with not too much scatter about the scatterplot, is fairly small, indicating reasonable fit, with 95% of residuals spanning a range of about 26 and 1/2 USD.

R^2 is almost 0.5, which indicates room for improvement, but a demonstration that the model improves much over the null model. That is, using *rm* to predict *medv* does much better than simply using the mean of *medv* to predict *medv*.

Postscript

In practice, regression models, and all machine learning models, typically involve much more than a single feature, predictor variable. Subsequent material expands this model to multiple regression, that is, multiple features.

Feature Selection

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

Feature selection is not always necessary for building machine learning models, but it is typically a helpful process to pursue. The goal is to reduce the number of predictor variables (features) in the model, to keep predictive accuracy at or about the same level, but with a much simpler model, with fewer predictor variables.

Two reasons to pursue feature selection:

1. Data costs money. The fewer the predictors, the less data needs to be collected.
2. Understanding the underlying relationships between predictors and target variable, which indirectly often leads to the construction of better models.

Preliminaries

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-07-12 at 14:28
```

```
import os
os.getcwd()
```

```
'/content'
```

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

Data

```
#d = pd.read_csv('data/Boston.csv')
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/Boston.csv')
```

```
d.shape
```

```
(506, 15)
```

```
d.head()
```

	Unnamed: 0	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptr
0	1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	
1	2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	
2	3	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	
3	4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	
4	5	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	

Do not need the first column, so drop.

```
d = d.drop(["Unnamed: 0"], axis="columns")
d.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	blac
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.9
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.9
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.8
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.6
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.9

Store the features, the predictor variables, in data structure X. Store the target variable in data structure y. To run multiple regression with all possible predictor variables, one possibility defines X as the entire data frame with *medv* dropped, as in

X = d.drop(['medv'], axis="columns")

Alternatively, use the procedure below that manually defines a vector of the predictor variables (features) names, and then define X as the subset of *d* that contains just these variables.

```
y = d['medv']
pred_vars = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad',
             'tax', 'ptratio', 'black', 'lstat']
X = d[pred_vars]
```

Not necessary, but see how many features in the model, and observe the data type of the X and y data structures. The function `len()` provides the length of a vector, that is, the number of elements of a vector.

```
n_pred = len(pred_vars)
print("Number of predictor variables:", n_pred)
```

```
Number of predictor variables: 13
```

X and y are created as two different pandas data types: X is a data frame, y is a one-dimensional array called a `series`. A data frame can have a single column, but, somewhat confusingly (in my opinion), subsetting a data frame down to a single variable is no longer is a data frame.

Feature Selection

Features, the predictor variables, should be ...

- *relevant*: Predictors each correlate with the target
- *unique*: Predictors do not correlate much with each other

The problem of *collinearity* is the problem of correlated predictor variables, the features. Too much correlation and redundancy make estimating the slope coefficients difficult, though it does not harm predictive accuracy per se. Generally, improve model fit by adding new information in the form of a new predictor variable to the model to the extent that the new predictor is relevant and unique.

Do, however, be aware of the problem of *data leakage*. When testing the model on data previously unseen by the model, all aspects of that data must have been unseen, just as in a real-world forecasting scenario. Otherwise, the data is said to leak from training to testing data. Making decisions regarding the model based on *all* the data then by definition includes both training and testing data. Best to make decisions regarding model estimation only from the training data.

Manual Selection

Base selection of the predictor variables on satisfying the two criterion: relevance and uniqueness. The goal here is to produce a single output, a table, that displays numerical indices for both criterion.

Uniqueness. Besides the correlation coefficient of two predictor variables, a more general indicator of collinearity is the *variance inflation factor* or *VIF*. The *VIF* assesses the linear redundancy of one predictor variable not just with one other predictor variable, but all the other predictor variables.

Relevance. Compute the correlation of each predictor with the target.

```
print("X is a: ", type(X))
print("X.values is a: ", type(X.values))
```

```
X is a:  <class 'pandas.core.frame.DataFrame'>
X.values is a:  <class 'numpy.ndarray'>
```

Use the `statsmodels` function `variance_inflation_factor()` to compute the variance inflation factor for each predictor. The VIF's are a property only of the X's, so the target y is not part of this analysis. The `variance_inflation_factor()` function does not compute all the VIF's, but only one at a time. Create a data frame named *vif*, then fill each row of the data frame with the corresponding name of the predictor variable and its corresponding variance inflation factor.

To systematically calculate and retrieve the VIF's, one for each feature, traverse through the variables in X one at a time with a programming structure known as a `for` loop, from the first X variable through the last X variable, where `X.shape[1]` is the number of rows of the data frame.

Because the loop cannot traverse through the original data frame, transfer the X data frame to a more primitive data structure, a `numpy` structure of a numeric matrix, obtained with the `values` method.

1. To begin, create an empty data frame with any valid name. Here we use *vif*. Then define a variable called *Predictor* in the data frame, filled with the names of the columns of the X data structure using the `columns` method.

2. Then create a variable called *VIF*, the variance inflation factor for each predictor variable. Loop through the data matrix (not data frame) with the `values` method for each predictor variable.
3. Calculate the correlation of each predictor (feature) with the target and store in the variable called *Relevance*. Store in the data series *cr*, then loop through *cr* for each variable to copy the value to the new *Relevance* variable.
4. Finally, display the contents of the created *vif* data frame by listing its name as the last line of code in the cell. (If we wish to display information before the last line, then need the `print()` function.)

```
from statsmodels.stats.outliers_influence import variance_inflation_factor
vif = pd.DataFrame()
vif['Predictor'] = X.columns

vif['VIF'] = [variance_inflation_factor(X.values, i)
              for i in range(X.shape[1])]

cr = d.corr()['medv'].round(3)
vif['Relevance'] = [cr[i]
              for i in range(X.shape[1])]
vif
```

```
/usr/local/lib/python3.7/dist-packages/statsmodels/tools/_testing.py:19: Fu
  import pandas.util.testing as tm
```

	Predictor	VIF	Relevance
0	crim	2.100373	-0.388
1	zn	2.844013	0.360
2	indus	14.485758	-0.484
3	chas	1.152952	0.175
4	nox	73.894947	-0.427
5	rm	77.948283	0.695
6	age	21.386850	-0.377
7	dis	14.699652	0.250
8	rad	15.167725	-0.382
9	tax	61.227274	-0.469
10	ptratio	85.029547	-0.508
11	black	20.104943	0.333
12	lstat	11.102025	-0.738

There is much collinearity in the data, consistent with the correlation matrix that shows many feature correlations far from 0. Many features could be deleted to yield a more parsimonious model that would be just as effective if not more so. Although *rm* has one of the highest VIF's, it is also strongly related to the target as shown by the regression coefficients' analysis and has one of the highest correlations with the target. A high VIF does not mean a feature should be deleted because perhaps a relevant feature is correlated with other, less relevant features that, when deleted, lower the VIF on the more relevant feature.

Automated Feature Selection

The pure machine learning approach seeks to automate everything. This approach makes the most sense when there are many, tens if not hundreds, of features. Otherwise, best to perform feature selection manually, analyzing correlations, variance inflation factors, p-values from the regression analysis of all features, and all possible subset regressions. And there is always understanding the meaning of the individual features (predictor variables), favoring the most understandable and meaningful, and perhaps easiest or cheapest for which to collect the data.

Let's proceed as if we have too many features to model effectively or we wish to rely only on influential predictor variables. So we pare down our model here using automated feature selection. We begin with all 13 features.

If you have the computation time, do this after the analysis with all the features. If computation time is limited, do at least some feature selection before the model evaluation.

Automated Univariate Feature Selection

There is one simple `sklearn` feature selection module called `SelectKBest` that selects the specified number of features according to relevance, the correlation of each feature with the target. It simply selects those features with the highest correlations with the target. Specify the number of retained features with the `k` parameter.

Here the logical array we name `selected` indicates which of the `k` values in the X feature data structure are to be retained.

```
from sklearn.feature_selection import SelectKBest, f_regression
selector = SelectKBest(f_regression, k=5).fit(X,y)
selected = selector.get_support()
selected
```

```
array([False, False,  True, False, False,  True, False, False, False,
       True,  True, False,  True])
```

Select the selected variables by subsetting the original X data structure.

```
X2 = X.iloc[:, selected]
X2.head()
```

	indus	rm	tax	ptratio	lstat
0	2.31	6.575	296	15.3	4.98
1	7.07	6.421	242	17.8	9.14
2	7.07	7.185	242	17.8	4.03
3	2.18	6.998	222	18.7	2.94
4	2.18	7.147	222	18.7	5.33

Automated Multivariate Feature Selection

A more sophisticated, though more costly in CPU time procedure, is the `sklearn` module `RFE`, for *recursive feature elimination*. First, specify the estimation procedure by which to initially assign weights to the features, such as linear regression as in this example. The `RFE` procedure then evaluates the features and identifies the single weakest feature on the basis of model fit, which is then pruned from the model. This assumes the parameter *step* is set at 1, which is the number of features pruned at each step.

To apply the estimator, invoke the `fit()` function on the specified feature and target data structures, X and y. The process is recursively repeated until the requested number of features, *n_features_to_select*, is obtained. In this example, retain the top 5 features.

This method generally produces a better model than `SelectKBest`, but the issue is computation time. If the CPU time is available, `RFE` is preferred.

```
from sklearn.linear_model import LinearRegression
estimator = LinearRegression()
from sklearn.feature_selection import RFE
selector = RFE(estimator, n_features_to_select=5, step=1).fit(X,y)
```

The features are selected, but now pare down the X data frame of feature data to just include the selected features. Rely upon two variables that `RFE()` created. The output vector *support_* indicates by `True` or `False` the selected variables. The output *ranking_* vector ranks the features, with all the selected variables ranked at 1.

```
print(selector.support_)
print(selector.ranking_)
```

```
[False False False  True  True  True False  True False False  True False
 False]
[4 6 5 1 1 1 9 1 3 7 1 8 2]
```

Use the *support_* output structure from `RFE()`. Subset the data with `iloc()` to redefine the feature data frame.

```
X2 = X.iloc[:, selector.support_]
X2.head()
```

	chas	nox	rm	dis	ptratio
0	0	0.538	6.575	4.0900	15.3
1	0	0.469	6.421	4.9671	17.8
2	0	0.469	7.185	4.9671	17.8
3	0	0.458	6.998	6.0622	18.7
4	0	0.458	7.147	6.0622	18.7

We see that the five feature variables selected by the more sophisticated `RFE()` differ from the five chosen features by `SelectKBest()`.

To view the rankings of all the features, to show the order of the variables that did not make the final 5, access the output *ranking_* variable. Note that one of the two features most highly correlated with the target, *lstat*, did not make the cut.

The crucial information not shown here is how much higher is R^2, or how much lower is MSE, for a five-feature model. No answer from this analysis. To test, the model would need to be re-run.

```
rnk = pd.DataFrame()
rnk['Feature'] = X.columns
rnk['Rank']= selector.ranking_
rnk.sort_values('Rank').transpose()
```

	3	4	5	7	10	12	8	0	2	1	9	11	6
Feature	chas	nox	rm	dis	ptratio	lstat	rad	crim	indus	zn	tax	black	age
Rank	1	1	1	1	1	2	3	4	5	6	7	8	9

Postscript

The model should also be analyzed with standardized variables to put everything on a common scale. Further, at least one outlier should be removed. Given the high degree of collinearity, the model can likely be reduced to about 3 or 4 features with little if any lose in predictive power.

Also, the model should be developed on one set of data, the training data, and then evaluated on testing data, apart from the training data. If no new data is available, then split the original data up into 75%/25% samples and then estimate (learn) on the 75% sample and test on the 25% sample.

The most useful statistical information, in my opinion, for feature selection is what is called *all subset regression*, which evaluates R^2 for all (or many) possible subsets of feature combinations (actually, the adjusted version). Then it becomes straightforward to see which core set of predictors are best combined to achieve one of the best models among the available alternatives.

On a bit of a tangent here, but in terms of the most general advice, my R `Regression()` function provides this subset regression analysis automatically (though little time if any) to discuss feature selection in that class. I prefer that program in my `lessR` R package to anything I have seen in Python when doing regression analysis. Very straightforward to use and does cross-validation as well. Read the data and run the function. Not part of this course per se, but helpful to apply in real-world contexts. Here is the R code that gives all of the above, plus all subsets regressions.

```
library(lessR)
d = Read("http://web.pdx.edu/~gerbing/data/Boston.csv")
Regression(medv ~ crim + zn + indus + chas + nox + rm + age + dis + rad + tax + ptratio + black + lstat)
```

As a bonus, add the parameter `Rmd="house"` (or named whatever), and you will get a complete written report.

Regression Outliers

 David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

Preliminaries

In [2]:
```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2020-07-13 at 16:46
```

In [3]:
```
import os
os.getcwd()
```

Out[3]:
```
'/Users/davidgerbing/Documents/000/410/Code'
```

In [4]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Data

In [5]:
```
d = pd.read_csv('data/Boston.csv')
#d = pd.read_csv('http://web.pdx.edu/~gerbing/data/Boston.csv')
```

In [6]:
```
d.shape
```

Out[6]:
```
(506, 15)
```

In [7]:
```
d.head()
```

Out[7]:

	Unnamed: 0	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat	medv
0	1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	3	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	5	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

Do not need the first column, so drop.

In [8]:
```
d = d.drop(["Unnamed: 0"], axis="columns")
d.head()
```

Out[8]:

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lstat	medv
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

Store the features, the predictor variables, in data structure X. Store the target variable in data structure y. To run multiple regression with all possible predictor variables, define X as the entire data frame with *medv* dropped, as in

X = d.drop('medv'], axis="columns")

or, use the procedure below that manually defines a vector of the predictor variables (features) names, and then define X as the subset of *d* that contains just these variables.

In [9]:
```
y = d['medv']
pred_vars = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax',
             'ptratio', 'black', 'lstat']
X = d[pred_vars]
```

Not necessary, but see how many features in the model, and observe the data type of the X and y data structures. The function `len()` provides the length of a vector, that is, the number of elements of a vector.

In [10]:
```
n_pred = len(pred_vars)
print("Number of predictor variables:", n_pred)
```

```
Number of predictor variables: 13
```

X and y end up as two different pandas data types: X is a data frame, y is a one-dimensional array called a `series`. The `numpy` package, upon which `pandas` is built, has its own data type for arrays, such as created with the `np.array()` function. Some distinctions are that numpy arrays can have multiple dimensions, whereas Pandas series can have any set of characters for the corresponding indices.

Influence and Outliers

Run `statsmodels` function `OLS()` to do the basic regression analysis. This includes adding a column of 1's to the X structure of the data for the feature variables.

In [19]:
```
import statsmodels.api as sm
from statsmodels.regression.linear_model import RegressionResults
X = sm.add_constant(X)
model = sm.OLS(y, X)
results = model.fit()
```

Calculate the values of Cook's Distance, *D*, and other residual based statistics, with the function `get_influence()`. To be useful, sort the data with `sort_values()` according to `cooks_d`, then list the first 15 or so rows of `cooks_d` and related indices. Set the parameter `ascending` to `False` to begin the sort with the largest values, that is, a descending sort.

In [20]:
```
from statsmodels.stats.outliers_influence import OLSInfluence
infl = results.get_influence()
smry = infl.summary_frame()
smry = smry.loc[:, ['standard_resid', 'student_resid', 'cooks_d']]
smry.sort_values(by="cooks_d", ascending=False).head(15)
```

Out[20]:

	standard_resid	student_resid	cooks_d
368	5.713855	5.907411	0.165674
372	5.180330	5.322247	0.094097
364	-3.420118	-3.457995	0.069430
365	2.933770	2.956763	0.067184
369	3.756430	3.807609	0.055263
412	3.505841	3.546859	0.050041
367	2.696070	2.713448	0.045412
370	3.332992	3.367841	0.044196
214	2.754234	2.772893	0.042925
371	5.335291	5.491079	0.042555
414	2.470361	2.483299	0.034770
253	2.789298	2.808758	0.033492
380	-1.004220	-1.004229	0.031755
374	2.811972	2.831962	0.028228
166	2.752133	2.770745	0.022481

In this example, Observation #368 with $D = 0.166$ is likely an outlier. Its values should be examined and likely deleted from the analysis. Any written description of the results should include a description of this observation, its abnormal values of X that lead to the large value of *D*, and why it was deleted from further analysis.

In real life analysis we would list the data values for the outlier and then figure out why that set of data values lead to that row of data being an outlier. Here is the listing. Certainly age is way high, but we will not be concerned with more analysis here.

In [21]:
```
d.iloc[368, :]
```

Out[21]:
```
crim          4.89822
zn            0.00000
indus        18.10000
chas          0.00000
nox           0.63100
rm            4.97000
age         100.00000
dis           1.33250
rad          24.00000
tax         666.00000
ptratio      20.20000
black       375.52000
lstat         3.26000
medv         50.00000
Name: 368, dtype: float64
```

To drop the row of data, use the `drop` method as applied to indices since we never assigned actual row names to the rows of the data frame.

In [22]:
```
d.shape
```

Out[22]:
```
(506, 14)
```

In [23]:
```
d2 = d.drop([d.index[368]])
```

In [24]:
```
d2.shape
```

Out[24]:
```
(505, 14)
```

To continue the regression analysis without the outlier, we would repeat the preceding steps, beginning with forming the X and y data structures.

Regression with *sklearn* Machine Learning

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

Preliminaries

Misc

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-07-12 at 13:49
```

```
import os
os.getcwd()
```

```
'/content'
```

Import Standard Data Analysis Libraries

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

```
import matplotlib.pyplot as plt
import seaborn as sns
```

Access Solution Algorithm

The `sklearn` package provides many different solution algorithms to accommodate many different types of machine learning models, each in its own module called a *class*. The `LinearRegression` module provides the functions for doing linear regression. Access an algorithm by creating a specific instance of the algorithm, referred to by a specific name in the analysis. This process is called *instantiation*.

Here instantiate `LinearRegression()` with the name *reg_model*, accepting all default parameters, not passing any parameter values between the parentheses. All subsequent references to the linear regression algorithm below are then implemented via this name *reg_model*.

```
from sklearn.linear_model import LinearRegression
reg_model = LinearRegression()
```

Data

Boston Housing Data Set

- *crim*: per capita crime rate by town
- *zn*: proportion of residential land zoned for lots over 25,000 sq.ft.
- *indus*: proportion of non-retail business acres per town.
- *chas*: Charles River dummy variable (1 if tract bounds river; 0 otherwise)
- *nox*: nitric oxides concentration (parts per 10 million)
- *rm*: average number of rooms per dwelling
- *age*: proportion of owner-occupied units built prior to 1940
- *dis*: weighted distances to five Boston employment centres
- *rad*: index of accessibility to radial highways
- *tax*: full-value property-tax rate per 10,000 USD
- *ptratio*: pupil-teacher ratio by town
- *b*: 1000(Bk*: 0.63)^2 where Bk is the proportion of blacks by town
- *lstat*: % lower status of the population
- *medv*: Median value of owner-occupied homes in 1000's USD

```
#d = pd.read_csv('data/Boston.csv')
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/Boston.csv')
```

```
d.shape
```

```
    (506, 15)
```

```
d.head()
```

	Unnamed: 0	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptr
0	1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	

Do not need the first column, so drop.

```
d = d.drop(['Unnamed: 0'], axis="columns")
d.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	blac
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.9
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.9
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.8
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.6
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.9

Check for missing data to determine if any action such as row or column deletion or any data imputation is needed.

```
print (d.isna().sum())
print ('\nTotal Missing:', d.isna().sum().sum())
```

```
crim       0
zn         0
indus      0
chas       0
nox        0
rm         0
age        0
dis        0
rad        0
tax        0
ptratio    0
black      0
lstat      0
medv       0
dtype: int64

Total Missing: 0
```

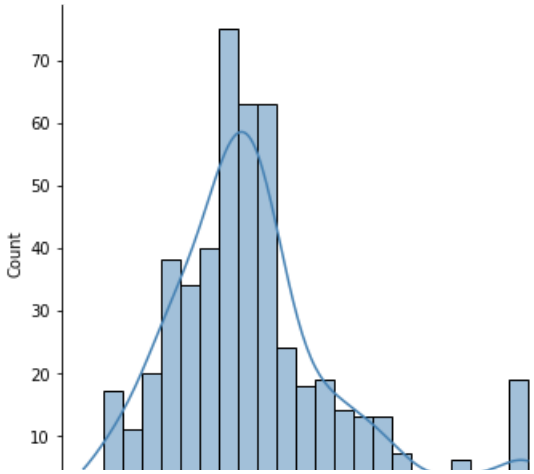
No missing data.

Check out the distribution of the target with seaborn `displot()`. Set parameter `kde` to `True` to show the smoothed summary, called a *density plot*. If going to run a model focused on forecasting the target, one should also understand the nature of the target variable. The main purpose is to understand the distribution, not necessarily to show normality per se. Look for skewness, outliers, etc.

Data Exploration

```
plt.figure(figsize=(6,6))
sns.displot(d.medv, kde=True, color='steelblue')
```

```
<seaborn.axisgrid.FacetGrid at 0x7f8385b12e50>
<Figure size 432x432 with 0 Axes>
```



More or less normal, with some large values beyond normality. It appears prices more than 50,000 USD are truncated to 50,000 USD for some reason.

Examine the *relevance* of each feature according to its correlation with the target. Use `pandas` function `corr()` to calculate just the correlations of the variables with `medv`. Use function `sort_values()` to sort from smallest to largest. Correlations of large magnitude, regardless of sign, indicate relevance.

Feature *chas* appears the least relevant with a correlation of the target of only 0.18. Even so, not 0, so with the small data set, will retain for the initial model analysis.

The most relevant features are *lstat* and *rm*.

```
(d
 .corr()['medv']
 .sort_values()
 .round(2)
)
```

```
lstat     -0.74
ptratio   -0.51
indus     -0.48
tax       -0.47
nox       -0.43
crim      -0.39
rad       -0.38
age       -0.38
chas       0.18
dis        0.25
black      0.33
zn         0.36
rm         0.70
medv       1.00
Name: medv, dtype: float64
```

> *Data leakage*: Feedback from data analysis from which the model is trained is used to evaluate the model used for forecasting.

We need to avoid data leakage. Test the final proposed forecasting model on data that has not in any way been used to estimate the model.

In practice, however, you might need to reduce computation time if you have a huge data set and a model with many predictors, particularly with a more complicated model and solution algorithm than for linear regression. In that situation,

without doing any model estimation, perhaps eliminate some features that violate the two properties of *relevance* and *uniqueness* before model estimation.

```
plt.figure(figsize=(12,10))
sns.heatmap(d.corr().round(2), linewidths=2.0, annot=True,
            cmap=sns.diverging_palette(5, 250, as_cmap=True))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f8381d45fd0>
```

Create Feature and Target Data Structures

Store the features, the predictor variables, in data structure X. Store the target variable in data structure y.

To run multiple regression with all possible predictor variables, define X as the entire data frame with *medv* dropped, as in

X = d.drop(['medv'], axis="columns")

or,

use the procedure below that manually defines a vector of the predictor variables (features) names, and then define X as the subset of *d* that contains just these variables.

```
y = d['medv']
pred_vars = ['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad',
             'tax',  'ptratio', 'black', 'lstat']
X = d[pred_vars]
```

Useful to see how many features in the model with the Python `len()` function for length, and observe the data type of the X and y data structures. Because this function is part of the original Python language, no package prefix is needed, just the

function name.

```
n_pred = len(pred_vars)
print("Number of predictor variables:", n_pred)
```

```
Number of predictor variables: 13
```

```
print("X: ", type(X))
print("y: ", type(y))
```

```
X:  <class 'pandas.core.frame.DataFrame'>
y:  <class 'pandas.core.series.Series'>
```

Model Validation with One Hold-Out Sample

Now for Python machine learning!

The `sklearn` package, named as an abbreviation for the full name `scikit-learn`, provides many machine learning algorithms, all implemented with the same general procedure illustrated here. Many support functions such as dividing data into training and testing partitions are also supported.

The underlying goal is to provide "Simple and efficient tools for predictive data analysis". A primary reason Python has become the leading platform for machine learning is because of `scikit-learn`. The name `scikit-learn` is a *sci*entific tool*kit* for machine *learn*ing.

Split data into train and test sets

Cross-validation tests a model on a new data set, *testing data* different from the data on which the model was estimated, *training data*. The concept of cross-validation has applied to regression analysis for many decades, though perhaps often recommended more than actually accomplished. The machine learning framework provides for easily accessible cross-validation methods, and is a necessary component of the analysis.

The `sklearn` function `train_test_split()`, from the `model_selection` module, randomly shuffles the original data into two sets, training data and testing data, here called *X_train* and *X_test* for the features and *y_train* and *y_test* for the target.

- Parameter `test_size` specifies the percentage of the original data set allocated to the test split.
- Parameter `random_state` specifies the initial seed (or starting point) from which the process of number generation begins so that the sequence can be repeated.

The input into the `train_test_split()` function are the X and y data structures. The function provides four outputs from a single function call: X training and testing data, and y training and testing data. Python has the convention of listing the names for multiple outputs on the left side of the equals sign, separated by commas, in the correct order in which the function lists the output.

Optional parameter `random_state` specifies the initial seed so that the same random process, more properly called a pseudo-random process, can be repeated at some future time with the same data split is obtained from `train_test_split()`. Re-run with the same value, here 7, obtain the same random split.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.25, random_state=7)
```

The `shape` method displays the dimensions of each of the resulting two data sets, *X_train* and *X_test*. The first number is the number of rows in the corresponding data structure. Here with the size of the testing data set at 25% of all the data, there are 379 rows of data in the two training data structures and 127 rows of data in the two testing data structures. The y data structures have only one column. The y structures are not data frames, so their number of columns is not specified.

```
print("Size of X data structures: ", X_train.shape, X_test.shape)
print("Size of y data structures: ", y_train.shape, y_test.shape)
```

```
Size of X data structures:  (379, 13) (127, 13)
Size of y data structures:  (379,) (127,)
```

Estimate the model parameters

All `sklearn` solution algorithms fit the model, that is, estimate the model parameters, with the `fit()` function, presumably first applied only to the training data. Expressed yet another way, the machine (i.e., algorithm implemented on the machine) learns from the training data. Only use the training data at this point in the analysis.

Here apply the `fit()` function for linear regression by applying our *reg_model* instantiation of `LinearRegression`.

```
reg_model.fit(X_train, y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

The `fit()` function creates several different data structures as output, each structure stored with a pre-defined name. The name of a data structure whose values that the analysis procedure creates ends in an underline.

The estimated model coefficients are stored in the `intercept_` and `coef_` structures. To reference, precede each name, in this example, with the model's name and a period. The coefficients of the final, validated model, estimated with all of the data, are needed to apply the model to other situations.

The machine learning implementation of regression is typically not primarily directed towards understanding and interpreting the model coefficients. Instead, focus on evaluating the extent of forecasting error. The estimated coefficients are not even displayed by default. The analysis does not provide the usual regression model output with the coefficients listed along with their corresponding *t*-tests of the null hypothesis of 0, and the associated confidence interval, such as obtained from the `OLS()` function in the `statsmodels` package.

The corresponding output structures are not `pandas` data frames, but rather `numpy` arrays, which do not display as nicely. To make the output more readable, convert the `numpy` array output format to a `pandas` data frame.

In the `print()` function, the `%.3f` is a format that indicates to display a floating-point number, that is, one with decimal digits, and to display three decimal digits.

```
print("intercept: %.3f" % (reg_model.intercept_), "\n")

cdf = pd.DataFrame(reg_model.coef_, X.columns, columns=['Coefficients'])
print(cdf)
```

```
intercept: 23.957

         Coefficients
crim        -0.129373
zn           0.029590
indus        0.022293
chas         2.837446
nox        -15.395420
rm           5.275573
age         -0.010538
dis         -1.301708
rad          0.266393
```

```
tax         -0.010969
ptratio     -0.964830
black        0.010860
lstat       -0.378363
```

Calculate $\hat{y}$

Given the estimated model, generate forecasts. The standard `sklearn` function to calculate a fitted value from the estimated model is `predict()`.

Here compute two sets of $\hat{y}$ values: *y_fit* when the model is applied (fitted) to the data on which it trained, and, for model evaluation, *y_pred* when the model is applied to the test data.

```
y_fit = reg_model.predict(X_train)
y_pred = reg_model.predict(X_test)
```

> Evaluate the descriptive analysis of fit by comparing y to $\hat{y}$ for the training data.

> Evaluate true forecasting fit by comparing y to $\hat{y}$ for the testing data.

Assess Fit

Visual assessment of fit

If there is only one predictor variable, plot the scatter plot of X and y and the least-squares regression line through the scatterplot. If this multiple regression, then this code is not run.

The Python syntax for an `if` statement uses the double equal sign, ==, to evaluate the equality, and a single equal sign, =, to create equality by assigning the value on the right to the variable on the left. Indicate the end of the conditional statement, here n_pred==1, with a colon, : . Indent two spaces for the statements that are run if the conditional statement is true.

```
if n_pred == 1:
  plt.scatter(X_train, y_train, color='gray')
  plt.plot(X_train, y_fit, color='black', linewidth=2)
  plt.xlabel("Prices: $X_i$")
  plt.title("Y and Fitted $\hat{Y}_i$ Plotted Against X")
```

The basis of the assessment of the model is the comparison of the actual data values of y in the testing data, *y_test*, to the values of y calculated from the model, *y_pred*.

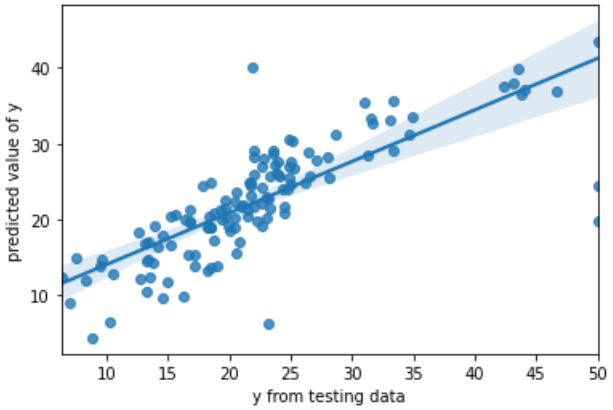
Visualize the overall fit by plotting the actual values of y in the test data, *y_test*, with the corresponding values of the forecasted y's, $\hat y$, or *y_pred*. If the forecasting is perfect, then $y=\hat y$, and all points lie on the 45-degree line through the origin. By default, the horizontal axis started numbering at 10, which was explicitly overridden to start at 0 with the `xlim()` function so that both axes begin at 0.

To obtain a scatter plot with the regression line and associated confidence interval, use the `seaborn` function `regplot()`. The variables to be plotted are not in a data frame, so there is no `data` parameter. To label the axes requires the `pandas` function `Series()` to name the associated series. In my opinion, it's a bit of contortion just to label the axes, but it works.

```
y_test = pd.Series(y_test, name="y from testing data")
y_pred = pd.Series(y_pred, name="predicted value of y")
```

```
sns.regplot(x=y_test, y=y_pred)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f83762b7790>
```



We can see that the predicted values closely match with the actual data values from the testing data.

Fit metrics

This first application is not always done. It evaluates the fit of the model to the training data, comparing the actual data values, *y_train*, to the corresponding values computed by the model, *y_fit*. This is *not* the official evaluation of model fit and performance. It is useful, however, to compare the fit indices for the training data to the testing data. A large drop indicates *overfitting* the model to the training data.

The `metrics` module in the `sklearn` package provides the computations for the fit indices. The module provides the mean squared error, MSE, and $R^2$$R^2$ fit indices with the functions `mean_squared_error()` and `r2_score()`. To get the standard deviation of the residuals, manually take the square root of the variance MSE with the `numpy` function `sqrt()`.

The %.3f formatting code instructs the Python `print()` function to print a floating-point number (numeric with decimal digits) with three decimal digits.

```
from sklearn.metrics import mean_squared_error, r2_score
mse = mean_squared_error(y_train, y_fit)
rsq = r2_score(y_train, y_fit)
print("MSE: %.3f" % mse)
se = np.sqrt(mse)
range95 = 4 * se
print("Stdev of residuals: %.3f " % se)
print("Approximate 95 per cent range of residuals: %.3f " % range95)
print("R-squared: %.3f" % rsq)
```

```
MSE: 20.266
Stdev of residuals: 4.502
Approximate 95 per cent range of residuals: 18.007
R-squared: 0.767
```

For pedagogy, here compute the standard deviation of the residuals from the data. Define the residuals as *e*. Note that the mean squared residual, both here and from the previous cell, is calculated with the full sample size, not the technically correct degrees of freedom.

```
e = y_train - y_fit
print("stdev of residuals: %.3f " % np.sqrt(np.mean(e**2)))
```

```
stdev of residuals: 4.502
```

Here we do the actual evaluation of model performance. Evaluate how well the actual data values for $y$$y$, y_test, match the forecasted or predicted values of $y$$y$, $\hat y$$\hat y$. From this split of data, the value of $R^2$$R^2$ typically drops from that obtained from the training data. Sometimes, however, by chance, the testing data may outperform the training data, again due to chance.

```
mse_f = mean_squared_error(y_test, y_pred)
rsq_f = r2_score(y_test, y_pred)
print('Forecasting Mean squared error: %.3f' % mse_f)
print('Forecasting Standard deviation of residuals: %.3f' % np.sqrt(mse_f))
print('Forecasting R-squared: %.3f' % rsq_f)
```

```
Forecasting Mean squared error: 29.515
Forecasting Standard deviation of residuals: 5.433
Forecasting R-squared: 0.617
```

We see that when applied to new data, the standard deviation of residuals, $s_e$$s_e$, increased from 4.502 to 5.433, still a small number. $R^2$$R^2$ decreased from 0.767 from the training data to 0.617 applying the model to the testing data. Regardless, good fit is obtained even with the forecasting model.

Model Validation with Multiple Hold-Out Samples

As an alternative to the one hold-out cross-validation in the previous section, here evaluate model fit with cross-validation on *multiple* samples. The `sklearn KFold` module performs the cross-validation in which the model is estimated using $k-1$$k-1$ folds and then tested on the remaining fold. The process automatically repeats for each fold.

Here pass specific parameter values to `Kfold`.

- *n_splits*: Number of splits (folds) of the training data.
- *shuffle*: Randomly shuffle the data before splitting into the folds.
- *random_state*: Set the seed to recover the same "random" data set in a future analysis.

The number of splits can vary from 2 to $n-1$$n-1$, where $n$$n$ is the total number of rows in the training data. Values of 3 and 5 are the most common. Larger data sets support a larger number of splits. Usually, shuffle the data first to keep the entire process entirely random.

Here instantiate the `KFold` module with *kf*, invoking the desired parameter values.

```
from sklearn.model_selection import KFold, cross_validate
kf = KFold(n_splits=5, shuffle=True, random_state=1)
```

The `cross_validate()` conveniently provides for multiple evaluation scores from the same cross-validation folds without manually repeating the computations for each score. Plus, computation times are also provided.

To estimate the model for each fold, here five different estimates from five different samples, specify the estimation algorithm. We have already instantiated the `LinearRegression()` estimator earlier as *reg_model*, but repeat here for clarity. The *scoring* parameter specifies to obtain $R^2$$R^2$ and MSE scores for each of the true forecasts of applying the model, for each split, from the k-1 folds data to the hold-out fold.

Weirdly, MSE is reported in the negative. The reason is that the best score is always the largest across all scoring procedures and all estimation algorithms and is thus consistent with other model-tuning algorithms that expect this behavior and

consistent with the internal code of the related `sklearn` functions. So here, the least negative is the largest value, the most desirable value. In reality, MSE must be a non-negative number, so the sign of the real MSE is just flipped to go negative.

The training data evaluations are not needed for the evaluation per se, which occurs on the testing data, but sometimes helpful to compare to the corresponding testing scores. Training scores much larger than the related testing scores indicates overfitting. Obtain the training information with the parameter *return_train_score*.

```
scores = cross_validate(reg_model, X, y, cv=kf,
                        scoring=('r2', 'neg_mean_squared_error'),
                        return_train_score=True)
```

Our *scores* array contains much information regarding the fit of each model over the five different analyses, but not so directly readable. To make it more readable, convert *scores* to a data frame, rename the long column names to more compact versions, convert the MSE scores to positive numbers, and average the results. The display includes the time to fit the training data for each fold and the time to calculate the evaluation scores, which includes getting the predicted values.

The parameter `inplace` set to `True` means making the change in the specified data frame and saving the data frame with those changes. This parameter removes the need to copy to a new data frame.

```
ds = pd.DataFrame(scores)
ds.rename(columns = {'test_neg_mean_squared_error': 'test_MSE',
                     'train_neg_mean_squared_error': 'train_MSE'},
          inplace=True)

ds['test_MSE'] = -ds['test_MSE']
ds['train_MSE'] = -ds['train_MSE']
print(ds.round(4))
```

```
   fit_time  score_time  test_r2  train_r2  test_MSE  train_MSE
0    0.0048      0.0047   0.7634    0.7294   23.3808    21.8628
1    0.0020      0.0014   0.6468    0.7582   28.6143    20.5029
2    0.0019      0.0015   0.7921    0.7262   15.1606    23.7937
3    0.0019      0.0013   0.6508    0.7580   27.2082    20.8185
4    0.0022      0.0014   0.7353    0.7409   23.3712    21.6071
```

A fit index averaged over all the folds is the best summary of how well the model fits, either to the training data, or more interestingly to the testing data.

```
print('Mean of test R-squared scores: %.3f' % ds['test_r2'].mean())
print('\n')
print('Mean of test MSE scores: %.3f' % ds['test_MSE'].mean())


se = np.sqrt(ds['test_MSE'].mean())
print('Standard deviation of mean test MSE scores: %.3f' % se)
```

```
Mean of test R-squared scores: 0.718


Mean of test MSE scores: 23.547
Standard deviation of mean test MSE scores: 4.853
```

This 13-predictor model fits well, with an average $R^2$$R^2$ across the five folds of 0.72. (Note that we never see the actual estimated model from each fold.) The average MSE and $s_e$$s_e$ is also low in terms of the more interpretable standard deviation of the residuals. Once the model is validated, fit it to the entire, full data set.

Strategy to Obtain the Final Model

Begin data preparation by deleting any unnecessary features, removing any obvious univariate outliers, and converting any categorical variables to indicator/dummy variables if included in the model as features. Also check for missing data as machine learning solution algorithms do not run if missing data are present.

If CPU time is an issue, cross-validate with only one hold-out sample. Otherwise, cross-validate with 3 or 5 or more hold-out samples, depending on CPU time and the size of the original data set.

> All that is needed for model validation if computation time permits is the $k$$k$-fold cross-validation with multiple-scores.

The only advantage of the one train-test split approach is that the model coefficients can be obtained, but they are not of primary interest because the final model has not yet been estimated on all of the data. Cross-validation with $k$$k$-fold does what the one train-test split approach does, but now $k$$k$ times. The train-test one split approach almost becomes pedagogical as a way to learn how the $k$$k$-fold procedure works.

The initial model is usually pared down to a more parsimonious model, retaining a smaller set of relevant features that each provide unique information. Obvious candidates for features to delete can be deleted before model validation begins, that is, those with low correlations with the target and/or high correlations with other features.

More sophisticated feature deletion can occur after the model if the model is validated. Then use the `statsmodels` regression function `OLS()` for ordinary least squares to estimate the model on all of the data to get the estimated model on the largest sample possible. Do a more sophisticated feature selection procedure using your own judgement, based on $p$$p$-values for individual features and VIF values for individual features. Also, use Cook's distance to investigate and possibly eliminate any rows of data that are outliers with respect to the regression model.

Once a final model is selected, re-run the cross-validation on the smaller number of features to make sure the reduced model still evaluates well. Ideally, this analysis would be done on a completely new data set, but that may not be practical.

When completed, with the final `statsmodels` run you have the $b$$b$ coefficients -- $b_0, b_1, b_2,$$b_0, b_1, b_2,$ etc. -- that define the model that you now, in another context, put into production.

Logistic Regression with *sklearn*

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

Preliminaries

Misc

```
from datetime import datetime as dt
```

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-08-02 at 19:06
```

```
import os
os.getcwd()
```

```
'/content'
```

Import Standard Data Analysis Libraries

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

Data

An online retailer offers clothes in women's and men's styles, but sometimes the value of Gender is missing from the online order.

To provide a prediction equation, use logistic regression, which forecasts membership into a group, that is, a *label*. Here forecast values of the variable Gender, binary in this data set, from physical body dimensions as it relates to body type.

Read and Verify Data

```
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/BodyMeas.csv')
#d = pd.read_csv('data/BodyMeas.csv')
```

```
d.shape
```

```
(340, 8)
```

```
d.head()
```

	Gender	Weight	Height	Waist	Hips	Chest	Hand	Shoe
0	F	200	71	43	46	45	8.5	7.5
1	F	155	66	31	43	37	8.0	8.0
2	F	145	64	35	40	40	7.5	7.5
3	F	140	66	31	40	36	8.0	9.0
4	M	230	76	40	43	44	9.0	12.0

Pre-Process Data

This is the place to check for outliers, though not doing now.

Apply pandas `get_dummies()` to *Gender* to create two new binary variables scored 0 and 1, *Gender_F* and *Gender_M*. We could specifically target *Gender* with the `columns` parameter but can also convert all the categorical variables at one time, of which the only one is *Gender* in this example. To target all the variables, enter the data frame's name instead of a specific list of variables.

We only need and want one of the newly created indicator (dummy) variables, $k - 1$ indicator variables, where k is the number of categories. Use the `drop_first` parameter to delete the first dummy variable, *Gender_F* arbitrarily. For the remaining *Gender_M*, a 1 means Male, and a 0 means Female.

```
d = pd.get_dummies(d, drop_first=True)
```

Verify the data are correctly read.

```
d.head()
```

	Weight	Height	Waist	Hips	Chest	Hand	Shoe	Gender_M
0	200	71	43	46	45	8.5	7.5	0
1	155	66	31	43	37	8.0	8.0	0
2	145	64	35	40	40	7.5	7.5	0
3	140	66	31	40	36	8.0	9.0	0
4	230	76	40	43	44	9.0	12.0	1

If we had more variables, we could transpose the output of `head()` to list the variables vertically. Not needed here, but illustrated to show the effect. Use the `transpose()` function.

Verify a data frame with many variables using `transpose()`.

```
d.head().transpose()
```

	0	1	2	3	4
Weight	200.0	155.0	145.0	140.0	230.0
Height	71.0	66.0	64.0	66.0	76.0
Waist	43.0	31.0	35.0	31.0	40.0
Hips	46.0	43.0	40.0	40.0	43.0
Chest	45.0	37.0	40.0	36.0	44.0
Hand	8.5	8.0	7.5	8.0	9.0
Shoe	7.5	8.0	7.5	9.0	12.0
Gender_M	0.0	0.0	0.0	0.0	1.0

Check for missing data with `isna()`.

```
print (d.isna().sum())
print ('\nTotal Missing:', d.isna().sum().sum())
```

```
    Weight      0
    Height      0
```

```
Waist       0
Hips        0
Chest       0
Hand        0
Shoe        0
Gender_M    0
dtype: int64

Total Missing: 0
```

No missing data, so proceed as is.

Pre-Analysis Understanding and Feature Selection

There is no strict requirement to study your data before beginning building the model. However, typically better to first develop some intuition for the data before attempting to construct the model. Unless the goal is to reduce the computation time of model analysis, avoid *data leakage* by modifying the data *before* dividing into training and testing data sets. Otherwise, changes are made to all of the data, some of which later become testing data, now no longer entirely independent of the training phase.

Target Variable

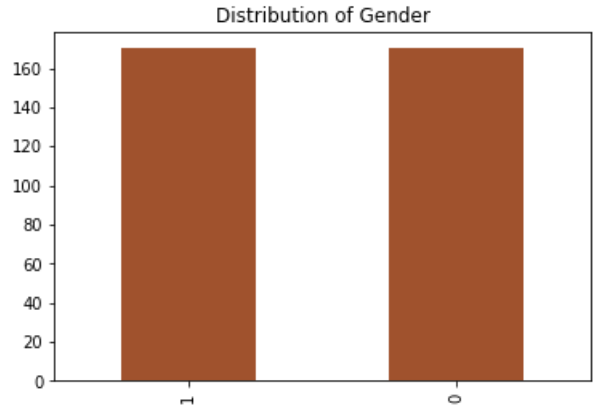
Check out the distribution of the target. If going to run a model focused on forecasting the target, should also understand the nature of the target variable. Choose a named color.

```
freq = d['Gender_M'].value_counts()
freq
```

```
1    170
0    170
Name: Gender_M, dtype: int64
```

```
plt.title('Distribution of Gender', fontsize=12)
freq.plot(kind='bar', color="sienna")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe5a464e310>
```



In this data set M (y=1) and F (y=0) are evenly distributed. There is plenty of variability in the target variable that a set of related features could explain.

Feature Relevance

Are all the features relevant? Examine the means of M and F across the features. All the numerical variables appear to differ depending on Gender, so prediction accuracy should be good.

The `pandas` function `groupby()` analyzes the data according to the specified statistic, here `mean()`, for *each level* of the specified categorical variable, here *Gender_M*. Without specifying specific variables to analyize, all numerical variables in the data frame are analyzed.

```
d.groupby('Gender_M').mean()
```

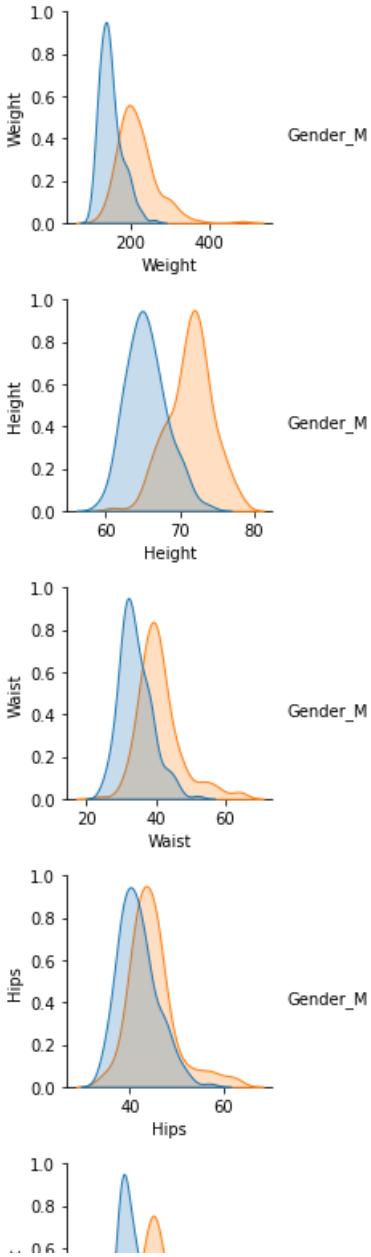
Gender_M	Weight	Height	Waist	Hips	Chest	Hand	Shoe
0	148.411765	65.464706	34.111765	41.629412	38.929412	7.728529	7.981176
1	215.758824	71.352941	40.941176	44.652941	45.005882	9.145588	10.670588

Examine the overlap in the distributions of Male and Female for each of the features. If the feature is relevant so that it relates to the target *Gender*, then it should differentiate men from women. Visualize the extent of this differentiation according to the separation between men and women on each feature.

To visualize overlap, use the `seaborn` function `pairplot()` that generates a scatterplot matrix (table). However, only plot the diagonal elements of this matrix, which visualize the smoothed histograms of the corresponding variable. Specify only a single variable for `pairplot()` with the `vars` parameter. The resulting scatterplot "matrix" consists of only a single diagonal element of the distribution of the variable.

To avoid entering a separate `pairplot()` call for each variable, use a `for` loop that loops through the specified columns of the data frame one-by-one in the call to `pairplot()`, here for the potential features.

```
pred_vars = ['Weight', 'Height', 'Waist', 'Hips', 'Chest', 'Hand', 'Shoe']
for column in d[pred_vars]:
    sns.pairplot(d, vars=[column], hue='Gender_M')
```



The less the overlap for a feature, the more accurate the feature classifies M and F. Hip size appears to be the least useful. Height and Hand size appear to be the most useful.

Still, the full multiple logistic regression specifies an interaction among all the variables, so complete model selection is best accomplished *after* deriving an initial model, and then preferably only on testing data to avoid *data leakage*. In practice, however, for large models with huge data sets, some obvious culling of features at this point can be worthwhile to same computation time.

Feature Redundancy

Check for collinearity by examining the correlation of the features with each other. We could use `pairplot()` to generate the full scatterplot matrix, but a heatmap requires less space. Generate the heatmap with the `seaborn` function `heatmap()`.

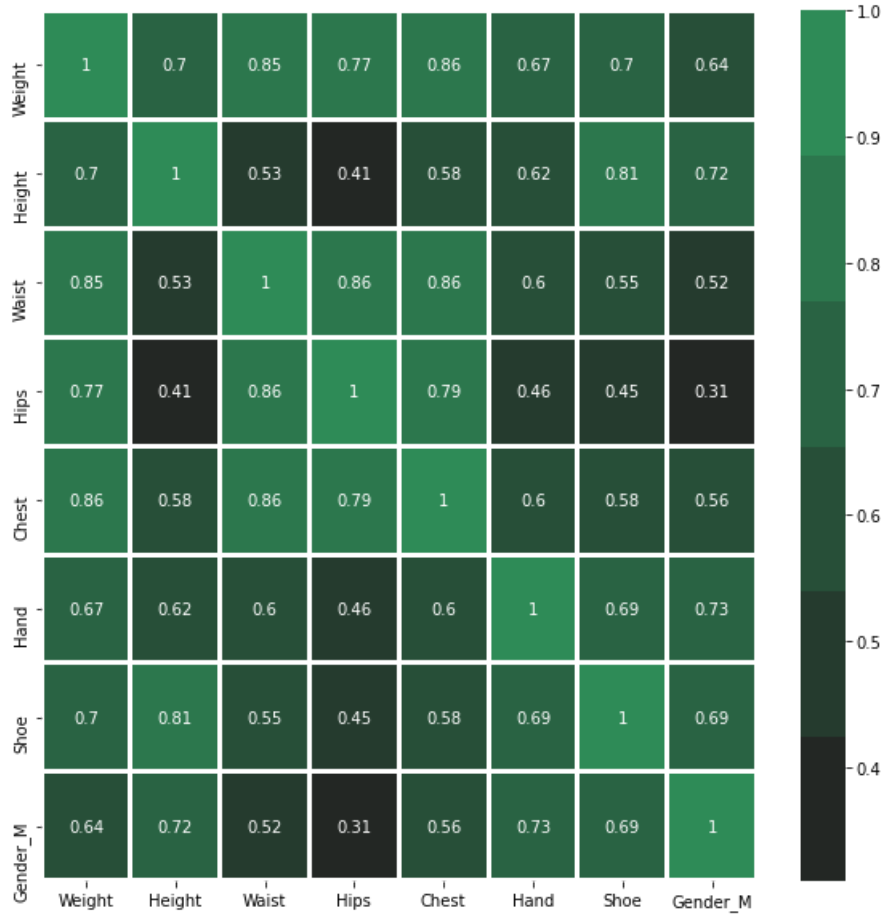
Not needing to drop features before model estimation as CPU time is not an issue, it is helpful to explore relations of the features with each other and with the target. The goal is to understand how the model will perform and how many features will be needed.

The `cmap` parameter for the color map specifies the assignment of colors to correlation values, with many possibilities include many palettes from `matplotlib` obtained just be entering a color name. Here use a `seaborn` palette called `dark_palette`.

This palette is *sequential*, from lighter to darker of the same hue, appropriate since all correlations are above 0. To show positive and negative correlations, choose a *diverging* palette, with different hues on both sides of the midpoint, such as from `cmap=sns.diverging_palette(5, 250, as_cmap=True))`

```
plt.figure(figsize=(10,10))
sns.heatmap(d.corr().round(2), linewidths=2.0, annot=True,
            cmap=sns.dark_palette("seagreen"))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe59bffa650>
```



There are some high feature correlations. *Shoe* size correlates with *Height* at r=0.81, and *Chest* correlates with *Weight* at 0.86, both very high correlations. Because of this redundancy, the final model will not require all 7 features, probably more like 3 or 4 at the maximum.

Create Feature and Target Data Structures

Define all the variables that are added to X, as below. y is a column of 0's and 1's

```
y = d['Gender_M']
X = d[pred_vars]
X.shape
```

```
(340, 7)
```

Optional, but see how many features are in the model, and observe the data type of the X and y data structures.

```
n_pred = len(pred_vars)
print("Number of predictor variables:", n_pred)
```

```
Number of predictor variables: 7
```

```
print("X: ", type(X))
print("y: ", type(y))
```

```
X:  <class 'pandas.core.frame.DataFrame'>
y:  <class 'pandas.core.series.Series'>
```

It might be useful to study rescaling the features so that each has approximately the same scale, either minimum of 0 and a maximum of 1, or standardization. However, will not pursue that here.

Fit Model, then Predict, Evaluate with *One* Hold-Out Sample

This analysis follows the standard `sklearn` machine learning paradigm. The logistic regression analysis outlined below is similar to the standard least-squares regression analysis pursued previously. Ultimately we prefer k-fold cross-validation but first shown is the one-split vesion instead of k splits.

Split Data into Train and Test sets

Cross-validation is testing a model a new data set different from the data on which the model was estimated. The concept of cross-validation has applied to regression analysis for many decades, though perhaps often recommended more than actually accomplished. The machine learning framework provides for easily accessible cross-validation methods, and is considered a necessary component of the analysis.

The key component of a cross-validation, or more simply, validation, is the *hold-out sample*, the portion of the original data set aside as the testing data. If there is much data with a model that requires much training time, only one hold-out sample may be practical, the approach here.

Function `train_test_split()` randomly shuffles the original data into two sets, training data and testing data, here called X_train and X_test for the features and y_train and y_test for the target. Parameter `test_size` specifies the percentage of the original data set allocated to the test split.

The input into the `train_test_split()` function are the X and y data structures. With a single function call, the function provides four outputs, X training and testing data, and y training and testing data. Python has the convention of listing the names for multiple outputs on the left side of the equals sign, separated by commas, in the correct order of the output.

Keep the group membership balanced in the created data sets with the parameter `stratify`, set equal to the variable in the y target data structure. We want randomization of the data into training and testing structures but with the same proportion of group members across the structures. That keeps the null model the same for each data set, to have the same baseline of comparison for the testing data.

Optional parameter `random_state` specifies the initial seed so that the same random process, more appropriately called a pseudo-random process, can be repeated in the future with the same data split is obtained from `train_test_split()`. Re-run with the same value, here 9, get the same random split.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.30,
                                    stratify=d['Gender_M'], random_state=9)
```

In the full data set, M and F are equally represented. Show that the `stratify` parameter worked correctly by maintaining the same balance in the created training and testing data sets.

```
y_train.value_counts()
```

```
1    119
0    119
Name: Gender_M, dtype: int64
```

```
y_test.value_counts()
```

```
1    51
0    51
Name: Gender_M, dtype: int64
```

Verify the data are as expected for the target variable. Here view an excerpt from the testing data for the target. As expected, a series of 0's and 1's.

```
pd.DataFrame(y_test).head(10).transpose()
```

	255	15	280	238	306	236	138	309	26	281
Gender_M	1	1	1	0	1	1	0	1	0	0

The `shape` method displays the dimensions of each of the resulting two data sets, X_train and X_test. The first number is the number of rows in the corresponding data structure. Here, with the testing data set size at 25% of all the data, there are 379 rows of data in the two training data structures and 127 rows of data in the two testing data structures. The y data structures have only one column and are not data frames, so their number of columns is not specified.

```
print("size of X data structures: ", X_train.shape, X_test.shape)
print("size of y data structures: ", y_train.shape, y_test.shape)
```

```
size of X data structures:  (238, 7) (102, 7)
size of y data structures:  (238,) (102,)
```

Access Solution Algorithm

The `sklearn` package provides many machine learning algorithms, all implemented with the same general procedure. Here invoke `LogisticRegression` module, which provides the functions for the logistic regression analysis. Implement a general algorithm by making a specific instance of the algorithm, referred to by a specific name in the analysis of your choosing. This process is called *instantiation*. Here instantiate with the reference *logistic_model*. All subsequent references to the linear regression algorithm are implemented via the name *logistic_model*.

The `solver` parameter indicates the specific solution algorithm. Several solution methods are available, all of which employ gradient descent, which iterates to a solution step-by-step from an intial, arbitrary solution. Convergence for the solution algorithm was not obtained with 100 iterations, so increase the value of parameter `max_iter` to achieve convergence.

```
from sklearn.linear_model import LogisticRegression
logistic_model = LogisticRegression(solver='lbfgs', max_iter=500)
```

Fit the Model to the Training Data

Employ the standard `sklearn` machine learning functions. Fit the model, that is, estimate the parameters, with the `fit()` function, here applied on the training data applied to a logistic regression model, previously instantiated as *logistic_model*. Expressed yet another way, have the machine (i.e., algorithm) learn from the training data. Only fit the training data at this point in the analysis.

```
logistic_model.fit(X_train, y_train)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=500,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

Showing the model at this point in the analysis is for pedagogy. In actual practice no need to examine this model because first, the model has not yet been validated on testing data, and second, it is based only on the training data. This training data-derived model will be applied to the testing data to evaluate the extent of the forecasting error.

If the model validates, then we view the model to employ in future forecasts at the end of the analysis. Repeat the technique for viewing the model shown here at the end of a successful analysis.

Optionally display the estimated model coefficients, stored in the `lm.coef_` object, a data structure created by the `fit()` function — the name of a variable whose values that the procedure creates ends in an underline. The coefficients of the final model, estimated with all of the data, are needed to apply the model to other data sets.

The pure machine learning implementation of regression is not directed towards understanding and interpreting the model coefficients but instead focuses on evaluating the extent of forecasting error. As such, the analysis does not provide the usual regression output. Traditionally, list the coefficients along with their corresponding t-tests of the null hypothesis of 0, and the associated confidence interval (as obtained from the `statsmodels` package).

```
print("intercept %.3f" % logistic_model.intercept_, "\n")
cf = pd.DataFrame()
cf['Feature'] = X.columns
cf['Coef']= np.transpose(logistic_model.coef_).round(3)
cf.transpose()
```

```
intercept -25.125
```

	0	1	2	3	4	5	6
Feature	Weight	Height	Waist	Hips	Chest	Hand	Shoe
Coef	0.046	0.232	0.248	-0.689	0.091	1.457	0.591

The estimated coefficients for variables *Height*, *Waist*, *Chest*, *Hand*, and *Shoe* all associate with an increase in the probability of the relevant group, $y = 1$, whereas *Hips* has a negative coefficient. That is, *Hips* goes in the opposite direction indicating a lower probability of Male, and so an increased probability of Female, as Hip size increases.

Model:

$$\hat{y}_{Gender} = -25.121 + 0.046(x_{Weight}) + 0.232(x_{Height}) + 0.248(x_{Waist}) - 0.690(x_{Hips}) + 0.091(x_{Chest}) + 1.458(x_{Ha}$$
$$+ 0.591(x_{Shoe})$$

This model is then used to calculate $\hat{y}$ from the testing data to evaluate fit.

Evaluate Fit

Predicted Values

Use the `sklearn` function `predict()` to calculate the values of y from the previously fitted model using `fit()`. Calculate the fitted values $\hat{y}$ for both the training data and testing data. Name the $\hat{y}$ values for the training data y_fit. The values of $\hat{y}$ for the testing data are true forecasts or predictions, data never before seen by the model, so name y_pred.

```
y_fit = logistic_model.predict(X_train)
y_pred = logistic_model.predict(X_test)
```

The resulting y_fit and y_pred data structures are numpy arrays. Here display the first 15 values of y_pred to show a string of forecasted positive and negative outcomes, relative to value Male. Verify that the predicted values have the correct structure.

```
print(y_pred[1:25])
```

```
    [0 1 0 1 1 0 1 0 0 0 0 1 1 1 1 0 1 1 1 0 1 0 1 0]
```

Yes, all predicted values are 0's and 1's, the assignment to one of the two target groups.

Probabilities for Prediction

How were those predictions derived? Not needed as part of the analysis, but here presented for understanding.

The model predicts group membership from the calculated probability of being a Male. Calculate the probability of class membership with the `predict_proba()` function. It returns a column for each value of the target, here probability of 0 and then probability of 1. The expression `i[1]` restricts the output to just the second column, the probabilities of being Male given the values of the feature variables.

If probability of Male > 0.5, then forecast as Male, otherwise Female.

Here list the first 15 rows of data with `head()`, transposed to take up less vertical space.

Baseline prediction is predicting membership in the group with the highest probability of the two target classes, the prediction in the absence of all information regarding X, the *null model*. Before we can evaluate fit, we need the baseline probability.

To calculate the baseline probabilities, compute the proportion of rows of data for each group. The mean of a column of 0's and 1's is the proportion of 1's (males). The group with the largest proportion is the baseline probability.

```
probs = [i[1] for i in logistic_model.predict_proba(X_test)]
pred_df = pd.DataFrame({'true_values': y_test,
                        'pred_values': y_pred,
                        'pred_probs':probs})
pred_df.head(15).transpose().style.format("{:.3}")
```

	255	15	280	238	306	236	138	309	26	281	204	48	330	188	172
true_values	1.0	1.0	1.0	0.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0
pred_values	0.0	0.0	1.0	0.0	1.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0
pred_probs	0.431	0.0228	0.99	0.00313	0.991	0.804	0.0011	0.682	0.0023	0.0729	0.00112	0.0339	0.924	0.991	1.0

Here are two forecasts from the probability calculations. The data have been randomly shuffled in the creation of the training and test data sets, so the row numbers are arbitrarily ordered.

- First column, Row 255: predicted probability of Male is 0.431, so forecast Female
- Third entry, Row 280: predicted probability of Male is 0.990, so forecast Male

Fit Metrics

The `metrics` module in the `sklearn` package provides the computations for the fit indices. The basis of the assessment of the model compares the actual data values of y in the testing data, *y_test*, to the values of y calculated from the model, *y_pred*. For a binary target variable, this comparison reduces the two correct classifications and the two incorrect classifications, summarized by the *confusion matrix*.

Obtain forecasting accuracy with `sklearn` function `accuracy_score()`. First, compare the values to see if the model overfit the training data by applying to both the training data and the testing data.

```
from sklearn.metrics import accuracy_score
print ('Accuracy for training data:  %.3f' % accuracy_score(y_train, y_fit))
print ('Accuracy for testing data:  %.3f' % accuracy_score(y_test, y_pred))
```

```
    Accuracy for training data:  0.933
    Accuracy for testing data:  0.902
```

The testing performance from training to testing did not drop much, only 3.1%, so not much overfitting here. Also evaluate if the testing accuracy is high for model validation. At 90.2% accuracy for the testing data, the model is reasonably accurate, and considerably larger than the 50% baseline prediction (analogous to the R^2 statistic).

Now a more in-depth analysis of the testing model performance, beginning with the confusion matrix, the numerical basis for all the classification fit indices. `sklearn` provides two functions for the confusion matrix. One function, `confusion_matrix()`, counts the correct classifications and the mis-classifications given the y and $\hat{y}$ values, the actual and forecasted 0's and 1's.

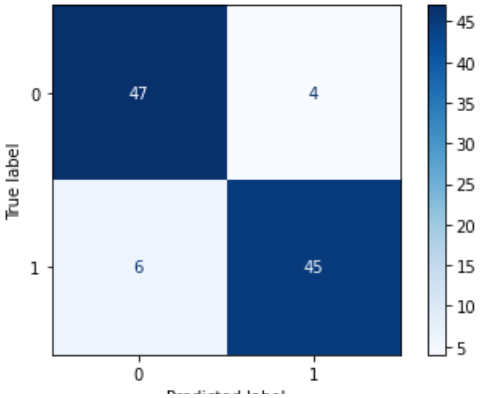
To display the results, and to access them for later reference, save the confusion matrix in a data frame, here named *dc*.

```
from sklearn.metrics import confusion_matrix
dc = pd.DataFrame(confusion_matrix(y_test, y_pred))
dc
```

	0	1
0	47	4
1	6	45

The second confusion matrix function, `plot_confusion_matrix()`, provides a heat map that illustrates the extent of the four outcome numbers for the 102 customers in the testing data. This function works directly from the model, implicitly computing the $\hat{y}$'s from the features, X, and then comparing to the given values of y.

```
from sklearn.metrics import plot_confusion_matrix
confmat = plot_confusion_matrix(logistic_model, X_test, y_test, cmap="Blues")
```



Not needed, but here explicitly label each of the four outcomes, referring to the previously computed data frame *dc*.

```
print("True Negatives: ", dc.iloc[0,0])
print("True Positives: ", dc.iloc[1,1])
print("False Negatives: ", dc.iloc[1,0])
print("False Positives: ", dc.iloc[0,1])
```

```
True Negatives:  47
True Positives:  45
False Negatives:  6
False Positives:  4
```

With these data, there is no indication that misclassifying a Male as Female or a Female as Male is more costly. So the asymmetric fit indices, recall (sensitivity), and precision are likely not needed.

For completeness, however, calculate the fit indices recall, precision, and F1, as with any fit index, compare the actual value so of y, named *y_test*, to the values fitted by the model, $\hat{y}$, named *y_pred*. Pass each set of values to the respective `sklearn` functions `recall_score()`, `precision_score()`, and `f1_score()`.`

```
from sklearn.metrics import recall_score, precision_score, f1_score
print ('Recall for testing data:  %.3f' % recall_score(y_test, y_pred))
print ('Precision for testing data:  %.3f' % precision_score(y_test, y_pred))
print ('F1 for testing data:  %.3f' % f1_score(y_test, y_pred))
```

```
Recall for testing data:  0.882
Precision for testing data:  0.918
F1 for testing data:  0.900
```

The lowest fit index, *recall* (sensitivity), is still high, at 88.1%. That means that the model correctly forecasts 88.2% of the Males as Males (true positive). So the model mislabels almost 12% of actual Male body types as Female. This 12% comes from the 6 false negatives from the confusion matrix.

Precision is even higher, which means that of those the model forecasted as Male, 91.8% are Male. Less than 8% of those predicted as Male are indicated as Female in the data, a false positive. As seen in the confusion matrix, only 4 false positives.

By definition, the *F1* statistic is between recall and precision, their harmonic average, at 91.8%.

Baseline Probability

Did the estimated model do better than the null model in which the forecast is to the group with the largest membership? That is, the null model forecasts without any knowledge of X.

```
my = y.mean()
max_my = np.max([y.mean(), 1-y.mean()])
```

```
print('Proportion of 0\'s (female): %.3f' % (1-my))
print('Proportion of 1\'s (male): %.3f' % my)
print('Null model accuracy: %.3f' % max_my)
```

```
Proportion of 0's (female): 0.500
Proportion of 1's (male): 0.500
Null model accuracy: 0.500
```

Here the proportion of M's and F's are equal, so the baseline probability is 0.5. If all customers are predicted to be either Male or Female, accuracy is 50%. Obtained forecasting accuracy on the testing data of 0.902 is much larger than the 0.500 accuracy of the null model.

Validate with *Multiple* Hold-Out Samples

Double-click (or enter) to edit

Here use a version of `Kfold` designed for models with categorical target variables called `StratifiedKFold`. The distinction is that `StratifiedKFold` generates the test set of each fold with the same proportion of samples in each group (class, level) as in the whole population. Here substantiate with *skf*, invoking the desired parameters. That way, the test data resemble the overall data set in terms of the distribution of the target variable, which keeps the baseline or null error rate the same across folds.

```
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=1)
```

Get the accuracy, recall, and precision scores for each of the true forecasts of applying the model, for each split, from the k-1 folds data to the hold-out fold.

To estimate the model (here five different estimates from five different samples) we need to specify the estimation algorithm. We have already instantiated the *LogisticRegression()* module as *logistic_model*.

The `cross_validate()` function provides for obtaining multiple evaluation scores from the same cross-validation folds without having to repeat the computations for each score. Computation times are also provided, of importance for massive data sets.

The evaluations performed on the training data are not needed for model evaluation per se, which occurs on the testing data, but are useful to compare to the corresponding testing scores. Training scores much larger than the corresponding testing scores indicates overfitting. Obtain the training evaluations with the parameter `return_train_score`.

```
from sklearn.model_selection import cross_validate
scores = cross_validate(logistic_model, X, y, cv=skf,
                        scoring=('accuracy', 'recall', 'precision'),
                        return_train_score=True)
```

The *scores* array contains much information, but not so directly readable. Here convert to a data frame, rename the long column names, convert the MSE scores to positive numbers, and average the results. The display includes the time to fit the training data for each fold, and also the time to calculate the evaluation scores, which includes getting the predicted values.

The parameter `inplace` set to `True` means to make the change in the specified data frame and the save the data frame with those changes. That is, do not need to copy to a new data frame.

The result is a slight increase in recall compared indicated by our single train/test sample for recall, which averages 91.8% across the five samples. A slight decrease in precision results, here with an average of 92.1%. No change in the fundamental conclusion of a good-fitting model.

These results also indicate that to study a specific model and confusion matrix, the single train/test split can be useful, but to power through the formal model evaluation, if CPU time is not an issue `StratifiedKFold` makes it easy to avoid a single arbitrary split for which to rely upon for model evaluation. Instead, go for many such splits.

```
ds = pd.DataFrame(scores).round(3)
print(ds)
```

```
   fit_time  score_time  ...  test_precision  train_precision
0     0.080       0.003  ...           0.938            0.927
1     0.041       0.003  ...           0.917            0.941
2     0.064       0.003  ...           0.868            0.940
3     0.049       0.003  ...           0.943            0.919
4     0.033       0.003  ...           0.967            0.934

[5 rows x 8 columns]
```

```
print('Mean of test accuracy: %.3f' % ds['test_accuracy'].mean())
print('Mean of test recall: %.3f' % ds['test_recall'].mean())
print('Mean of test precision: %.3f' % ds['test_precision'].mean())
```

```
Mean of test accuracy: 0.927
Mean of test recall: 0.930
Mean of test precision: 0.927
```

The model does well on all three basic fit classification indices. Given the high collinearity among the predictors, probably not all predictor variables are needed.

Automated Feature Selection

The pure machine learning approach seeks to automate everything. This approach makes the most sense when there are many, tens if not hundreds, of features. Otherwise, feature selection is best performed "hands on", analyzing correlations, variance inflation factors, *p*-values from the regression analysis of all features. And there is always understanding the meaning of the individual features (predictor variables), favoring those that are the most understandable and meaningful, and perhaps easiest or cheapest for which to collect the data.

Anyhow, let's proceed as if we have too many features to effectively model, and so we need to pare down our model, here using automated feature selection. We begin with all 13 features.

If you have the computation time, do this after the analysis with all the features. If computation time is limited, do at least some feature selection *before* the model evaluation.

Univariate Selection Procedure

There is one simple feature selection procedure called `SelectKBest()` that selects the specified number of features according to the statistical test of the difference in group means of each feature across the two levels of the target.

The `get_support()` function identifies the selected features by listing a `True` for each selected feature across the varaibles in a vector of `True` and `False` values.

```
from sklearn.feature_selection import SelectKBest, f_classif
selector = SelectKBest(k=3).fit(X,y)
selected = selector.get_support()
```

```
selected
```

```
array([False,  True, False, False, False,  True,  True])
```

Verify the sub-setted data frame.

```
X2 = X.iloc[:, selected]
X2.head()
```

	Height	Hand	Shoe
0	71	8.5	7.5
1	66	8.0	8.0
2	64	7.5	7.5
3	66	8.0	9.0
4	76	9.0	12.0

Multivariate Selection Procedure

A more sophisticated, though more costly in CPU time procedure, is called `RFE`, for recursive feature elimination. First, specify the estimation procedure by which to initially assign weights to the features, such as linear regression, as in this example. The `RFE` procedure then evaluates the features and identifies the weakest feature, which is then pruned from the model. This description assumes the parameter `step` is set at 1, which is the number of features pruned at each step. To apply the estimator: invoke the `fit()` function on the specified feature and target data structures, X and y. Recursively repeat the process until the requested number of features, `n_features_to_select`, is obtained. In this example, retain the top 3 features.

This method generally produces a better model than `SelectKBest` but the issue is computation time. If the CPU time is available, `RFE` is preferred.

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import RFE
selector = RFE(logistic_model, n_features_to_select=3, step=1).fit(X,y)
```

The features are selected, but now the X data frame of feature data must be pared down to just include the selected features. For this there are two variables that `RFE` created. The `support_` vector indicates by True or False the selected variables. The `ranking_` vector ranks the features, with all the selected variables ranked at 1.

```
print(selector.support_)
print(selector.ranking_)
```

```
[False False False False  True  True  True]
[5 3 4 2 1 1 1]
```

Use the `support_` created data variable with the `iloc` method to redefine the feature data frame. Here we return to the full data set of the features, the X data frame.

```
X_reduced = X.iloc[:, selector.support_]
X_reduced.head()
```

	Chest	Hand	Shoe
0	45	8.5	7.5
1	37	8.0	8.0
2	40	7.5	7.5
3	36	8.0	9.0
4	44	9.0	12.0

Optional, but to view the rankings of all the features, to show the order of the variables that did not make the final 3, access the `ranking_` variable.

Display in a more readable format by converting output to a data frame, sorting the values, and transpose the data frame to list horizontally.

```
rnk = pd.DataFrame()
rnk['Feature'] = X.columns
rnk['Rank']= selector.ranking_
rnk.sort_values('Rank').transpose()
```

	4	5	6	3	1	2	0
Feature	Chest	Hand	Shoe	Hips	Height	Waist	Weight
Rank	1	1	1	2	3	4	5

Validate Reduced Model

Now that we have a new model, that is, with fewer features, we need to validate the new model. Validate the reduced model with k-fold cross-validation. Then compare to the full model.

```
scores = cross_validate(logistic_model, X_reduced, y, cv=skf,
                        scoring=('accuracy', 'recall', 'precision'),
                        return_train_score=True)
ds = pd.DataFrame(scores).round(3)
print(ds)
print('\n')
print('Mean of test accuracy: %.3f' % ds['test_accuracy'].mean())
print('Mean of test recall: %.3f' % ds['test_recall'].mean())
print('Mean of test precision: %.3f' % ds['test_precision'].mean())
```

```
   fit_time  score_time  ...  test_precision  train_precision
0     0.011       0.003  ...           0.875            0.912
1     0.010       0.003  ...           0.909            0.917
2     0.011       0.004  ...           0.861            0.923
3     0.013       0.003  ...           0.943            0.910
4     0.010       0.003  ...           0.969            0.899

[5 rows x 8 columns]


Mean of test accuracy: 0.906
Mean of test recall: 0.900
Mean of test precision: 0.911
```

Comparing to the full model with seven predictors, there was a slight decrease in performance, less than 1% on the average of the evaluation statistics. How many features to include in the model is a business decision, which weighs the cost of

additional data and CPU time versus forecasting accuracy. Perhaps the less than 1% decrease is acceptable, or, run with all

Estimate and Apply the Model

Estimate

Once validated, we need the best estimate that we can get for this model. The best estimate of the model is from all of the data, but here from only three features, those that define X_reduced. Here revise *logistic_model* with a model that is fit (estimated parameters) from all the data.

The *logistic_model* construct must have been previously instantiated as a `LogisticRegression`, here fit to all of the data for the specified variables with `fit()`.

```
logistic_model.fit(X_reduced, y)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=500,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)
```

Access the estimated linear coefficients from the `fit()` output structures `intercept_` and `coef_`. To enhance the display of the partial slope coefficients, show them from within a constructed data frame, here called *cf*.

```
print("intercept %.3f" % logistic_model.intercept_, "\n")
cf = pd.DataFrame()
cf['Feature'] = X_reduced.columns
cf['Coef']= np.transpose(logistic_model.coef_).round(3)
cf.transpose()
```

```
intercept -31.871
```

	0	1	2
Feature	Chest	Hand	Shoe
Coef	0.146	2.079	0.912

Model: $$\hat y_{Gender} = -31.871 + 0.146(x_{Chest}) + 2.079(x_{Hand}) + 0.912(x_{Shoe})$$$$\hat y_{Gender} = -31.871 + 0.146(x_{Chest}) + 2.079(x_{Hand}) + 0.912(x_{Shoe})$$

Apply

Now apply this model where it most conveniently fits into the work flow. It can be added to a local app such as Excel, or coded into a web application, available on the Internet. Or, it can be retained in Python, though of course, the entire model would not be re-estimated for each application.

Forecast a M or F from a set of *Chest*, *Hand*, and *Shoe* measurements from new data. Suppose a customer failed to report *Gender*, but did report a *Chest* of of 48 inches, a *Hand* size of 9 inches, and a *Shoe* size of 9.5. The double brackets for the array [[66, 9, 9.5]] of new data from which to make a prediction indicate the creation of a two-dimensional array, which is the form of the input that the sklearn functions expect.

Use the `predict()` function to forecast the *Gender* label, 0 or 1, and the `predict_proba()` function to assign the corresponding probability.

Make sure that when using `predict` to calculate $\hat y_i$$\hat y_i$ that the number of elements of *X_new*, here three, exactly match the elements when *model* was fit using `fit()`.

```
X_new = [[48, 9, 9.5]]
y_new = logistic_model.predict(X_new)
print("Predicted group membership:", y_new)
y_prob = logistic_model.predict_proba(X_new)
print(round(y_prob[0,1], 3))
```

```
Predicted group membership: [1]
0.926
```

From this person's measurements of Chest=48, Hand=9, and Shoe=9.5, the person is predicted to be a member of Group 1, Male, with high probability of 0.926.

This application of the model forecasted group membership, Male or Female, only for a single set of the three feature values in the revised model. Could also read a new data set of just X values and process them to apply the model to forecast future events with `predict()` for multiple observations.

Decision Tree Classification with *sklearn*

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Goal: Define a decision tree to classify according to Gender from other body measurements. The motiviation is to forecast Gender body type when that information is missing from online clothing orders.

Preliminaries

Misc

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-07-25 at 15:59
```

```
import os
os.getcwd()
```

```
'/content'
```

Import Standard Data Analysis Libraries

Only need `pandas` and `matlib` in this analysis, though does not hurt to do the standard import that includes `numpy` and `seaborn`.

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

Access Solution Algorithm

To demonstrate the power and ease of use of the `sklearn` machine learning paradigm, here allow for a variety of classification algorithms. Activate the Decision Tree analysis with a maximum depth of five.

```
from sklearn.tree import DecisionTreeClassifier
dt_model = DecisionTreeClassifier(max_depth=5)
```

Get and Structure Data

```
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/BodyMeas.csv')
#d = pd.read_csv('data/BodyMeas.csv')
d.shape
```

```
d.shape
```

```
(340, 8)
```

```
d.head()
```

	Gender	Weight	Height	Waist	Hips	Chest	Hand	Shoe
0	F	200	71	43	46	45	8.5	7.5
1	F	155	66	31	43	37	8.0	8.0
2	F	145	64	35	40	40	7.5	7.5
3	F	140	66	31	40	36	8.0	9.0
4	M	230	76	40	43	44	9.0	12.0

Create the features and target data structures. Need the target variable with two levels scores 0 and 1. Could use `get_dummies()`, but here manually create our dummy variable with `replace()`. Arbitrarily score 1 for Male.

```
classes = ['Female', 'Male']  # for graph
features = ['Weight', 'Height', 'Waist', 'Hips', 'Chest', 'Hand', 'Shoe']
X = d[features]
y = d['Gender'].replace({'F':0, 'M':1})
```

Evaluate Model with *Multiple* Hold-Out Samples

k-fold cross-validation for *one* set of parameters

Use 5-fold cross-validation to build and estimate the model five different times, all with the same set of parameters. For decision tree analysis, set `max_depth` to 5 as defined in the model instantiation. Use all 7 features as defined in X. Assess the fit of each model with accuracy, recall, and precision.

The `Kfold` module creates the folds, partitions of the data set into five different training sets of data, each with a corresponding testing data set. The `cross_validate` module does the actual cross-validation, running five different models on five different training sets and then testing each model on the corresponding testing data set.

```
from sklearn.model_selection import KFold
kf = KFold(n_splits=5, shuffle=True, random_state=1)

from sklearn.model_selection import cross_validate
scores = cross_validate(dt_model, X, y, cv=kf,
                        scoring=('accuracy', 'recall', 'precision'),
                        return_train_score=True)
```

Convert the *scores* output to a data frame for the appearance of the display.

```
ds = pd.DataFrame(scores).round(3)
print(ds)
```

```
   fit_time  score_time  ...  test_precision  train_precision
0     0.006       0.003  ...           0.846            0.970
1     0.002       0.002  ...           0.946            0.956
2     0.002       0.002  ...           0.933            0.978
3     0.002       0.003  ...           0.912            0.971
4     0.002       0.002  ...           0.839            0.952
```

```
[5 rows x 8 columns]
```

```
print('Mean of test accuracy: %.3f' % ds['test_accuracy'].mean())
print('Mean of test recall: %.3f' % ds['test_recall'].mean())
print('Mean of test precision: %.3f' % ds['test_precision'].mean())
```

```
Mean of test accuracy: 0.897
Mean of test recall: 0.900
Mean of test precision: 0.895
```

The model fits well. Accuracy, recall (sensitivity), and precision, all around 89%.

Grid Search: Hyperparameter Tuning with Cross-Validation

The previous cross-validation is for pedagogy only, as it is just one of a much larger set of results from a full grid search. But what is the best depth for the decision tree? The optimal number of features? To answer those questions, systematically explore a range of different models. Define any one model by a specific depth and number of features. Define many models, each with a different combination of depth and features, called hyper-parameters. Explore the fit of a complete set of related models by systematically varying the depth and number of features with a hyper-parameter grid search.

Here run 4 levels of depth, 4 different numbers of features for 16 different models. For each model, do cross-validations on 3 different folds. So 16x3=48 different analyses in all, conveniently and automatically accomplished with the module `GridSearchCV`, for grid-search cross-validation. The `param_grid` parameter specifies the hyper-parameters to systematically adjust in all possible combinations.

`GridSearchCV` sets up the grid of the 16 models. Here, instantiate the module as `grid_search` and then use `fit()` to fit all the models, each 3 times. Transpose the results to fit on the page.

Much work was accomplished with little code, illustrating the power and convenience of `sklearn`.

```
from sklearn.model_selection import GridSearchCV
kf3 = KFold(n_splits=3, shuffle=True, random_state=1)

params = {'max_depth': [2, 3, 4, 5],
          'max_features': [1, 2, 3, 4]}
grid_search = GridSearchCV(dt_model, param_grid=params, cv=kf3,
                           scoring=('accuracy', 'recall', 'precision'), refit=False,
                           return_train_score=True)
grid_search.fit(X,y)
```

```
GridSearchCV(cv=KFold(n_splits=3, random_state=1, shuffle=True),
             error_score=nan,
             estimator=DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None,
                                              criterion='gini', max_depth=5,
                                              max_features=None,
                                              max_leaf_nodes=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              presort='deprecated',
                                              random_state=None,
                                              splitter='best'),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': [2, 3, 4, 5],
                         'max_features': [1, 2, 3, 4]},
```

```
                pre_dispatch='2*n_jobs', refit=False, return_train_score=True,
```

Here are all the results, for each individual fold and their summaries.

```
d_results = pd.DataFrame(grid_search.cv_results_).round(3)
d_results = d_results.drop(['params'], axis='columns')
d_results.transpose()
```

	0	1	2	3	4	5	6	7	8	
mean_fit_time	0.003	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.002	0.
std_fit_time	0.001	0	0	0	0	0	0	0	0	
mean_score_time	0.003	0.003	0.002	0.002	0.002	0.004	0.004	0.003	0.002	0.
std_score_time	0.001	0	0	0	0	0.003	0.001	0	0	0.
param_max_depth	2	2	2	2	3	3	3	3	4	

In this output we mostly care about the `mean` of each fit index over the five folds, For example, `mean_test_accuracy` provides the mean of the five accuracy scores from the cross-validation. Subset the results data frame, and rename variables to be more compact.

```
d_summary = d_results[['param_max_depth', 'param_max_features', 'mean_test_accuracy',
                       'mean_test_recall', 'mean_test_precision', 'mean_train_accuracy',
                       'mean_train_recall', 'mean_train_precision']]
d_summary = d_summary.rename(columns= {
                       'param_max_depth': 'depth',
                       'param_max_features': 'features',
                       'mean_test_accuracy': 'test_accuracy',
                       'mean_test_recall': 'test_recall',
                       'mean_test_precision': 'test_precision',
                       'mean_train_accuracy': 'train_accuracy',
                       'mean_train_recall': 'train_recall',
                       'mean_train_precision': 'train_precision'})
d_summary
```

	depth	features	test_accuracy	test_recall	test_precision	train_accur
0	2	1	0.856	0.850	0.862	0
1	2	2	0.879	0.915	0.856	0
2	2	3	0.871	0.829	0.905	0
3	2	4	0.897	0.898	0.903	0
4	3	1	0.844	0.873	0.826	0
5	3	2	0.891	0.912	0.880	0
6	3	3	0.894	0.894	0.899	0
7	3	4	0.906	0.901	0.912	0
8	4	1	0.841	0.776	0.890	0
9	4	2	0.882	0.884	0.883	0
10	4	3	0.885	0.897	0.877	0
11	4	4	0.915	0.901	0.928	0
12	5	1	0.868	0.852	0.882	0
13	5	2	0.876	0.856	0.895	0
14	5	3	0.891	0.890	0.893	0
15	5	4	0.882	0.887	0.879	0

All the decision trees provide a similar fit, with very little overfitting except as expected in the more complex models.

Parsimony: Choose the simplest model that provides the best or almost the best fit to the testing data.

As always, strive for parsimony. Choose the model with a depth of two and three or four features. The resulting mean test

Choose Model and Estimate on All Data

Generally obtai the best estimates with the most data. Given sufficient fit from the validation phase, choose and then fit (estimate) the final model on the full data set. Sacrifice a little accuracy for parsimony, which results in a model with 3 features at a depth of 2.

```
dt_model = DecisionTreeClassifier(max_features=3, max_depth=2)
mf = dt_model.fit(X,y)
```

Use the created output variable `feature_importances_` to identify the most important features to choose the best 3, the three with non-zero entries. Instead of writing code individually for each feature, do a for loop to process each feature one-by-one.

```
for name, importance in zip(X.columns, dt_model.feature_importances_):
    print(name, '%.3f' % importance)
```

```
Weight 0.178
Height 0.766
Waist 0.000
Hips 0.000
Chest 0.000
Hand 0.000
Shoe 0.055
```

The model accuracy, recall, and precision have previously been evaluated for forecasting efficiency, each as the mean of the corresponding k-fold values. Here compute the same values for the full data set.

Begin with the $\hat{y}_i$'s with `predict()`.

```
y_fit = dt_model.predict(X)
```

```
from sklearn.metrics import confusion_matrix
pd.DataFrame(confusion_matrix(y, y_fit))
```

	0	1
0	150	20
1	18	152

From the confusion matrix, calculate the basic test indices.

```
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score
print ('Accuracy:  %.3f' % accuracy_score(y, y_fit))
print ('Recall:  %.3f' % recall_score(y, y_fit))
print ('Precision:  %.3f' % precision_score(y, y_fit))
print ('F1:  %.3f' % f1_score(y, y_fit))
```

```
Accuracy:  0.888
Recall:  0.894
Precision:  0.884
F1:  0.889
```

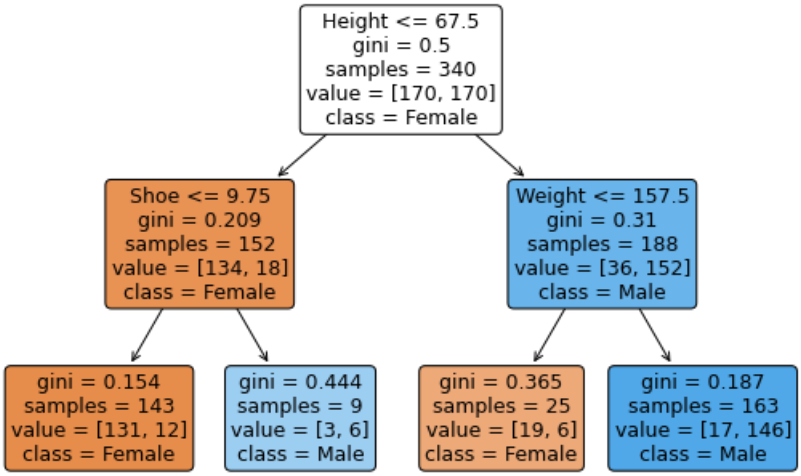
Illustrate the Model

The `sklearn` module `tree` provides a visualization of the obtained decision tree. The visualization shows the stakeholders who have sponsored the analysis (including your salary) how to classify customers with the model. The visualization facilitates model understanding, illustrating the sequence of decisions that arrive at the final classification.

The visualization also shows those classifications with which we have a high degree of confidence according to low Gini values and more intense coloring than those classifications for which we have little confidence, with a Gini coefficient clost to 0.5.

Obtain the visualization waith the `plot_tree()` function from the `tree` module. Specify the features with the `feature_names` parameter, and the groups for which to classify into with the `class_names` parameter. Their respective values for this analysis, *features* and *classes*, have been previously defined much earlier in this notebook.

```
from sklearn import tree
plt.figure(figsize=(9,6))
tree.plot_tree(mf, feature_names=features, class_names=classes, rounded=True, filled=True)
plt.savefig('dt_Gender.png')
```



The `value` output indicates the number of samples classified as Female and Male, in that order. The more extreme the differences in the two numbers, the better the classification accuracy, which yields a more desirable lower *gini coefficient*. For example, customers with a Hand size less than 8.125 inches, and a Weight less than 186 lbs, correctly classify 135 customers as Female and misclassify only five Male customers as Female.

The total number of False Negatives, Males misclassified as Females, is 5 + 8 + 3 = 16, a number directly available from the confusion table. The tree diagram specifies exactly where those misclassifications occurred.

There are 12 False Positives, as indicated from the tree diagram or the confusion matrix.

Apply the Model

Jupyter notebooks are not an optimal production environment, but the following code does generate a model forecast from new data entered into the model, such as in an actual forecasting application. The double brackets for the array of new data from which to make a prediction indicate the creation of a two-dimensional array, which is the form of input for `sklearn`

functions. To apply the model here more efficiently, we could add a read statement that reads X_new values from an external data file and then create a sequence of probabilities and predictions.

This model only uses three features, so it could refit on just a feature data set X that contains only those three features: Weight, Height, and Hand size. Then would only need to enter just those three values to obtain a forecast of Gender.

```
X_new = [[142,66,31,40,36,8,9]]
y_new = dt_model.predict(X_new)
print("Predicted group membership:", y_new)
y_prob = dt_model.predict_proba(X_new)
print(round(y_prob[0,1], 3))
```

```
Predicted group membership: [0]
0.084
```

Cluster Analysis with the *sklearn* ML Framework

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

Return to the data set from the online clothing retailer. This data is actual data from a real online retailer, with customer names and ID's deleted from the rows of data. The only data manipulation prior to analysis is to sample from the larger data set to balance the ratio of Males and Females in the analysis and to delete samples with missing data.

Previous examples of machine learning presented supervised machine learning analyses to build a model to forecast missing Gender body type from an online order form. Here, use unsupervised learning to suggest the underlying structure regarding Gender. Without supervision, does Gender emerge from the clustering customers from other variables?

This template outlines the steps for a complete cluster analysis, with the following caveat.

Validation warning: The 340 samples in this analysis are not enough samples to split the data into training and testing data sets for meaningful cluster analysis. Not doing a proper data split keeps the analysis simpler. Without a testing data set , do *not* select a solution with many clusters as that solution likely results from overfitting.

In general, unsupervised machine learning leads to a solution evaluated for fit, just as with supervised machine learning. As always, that solution should ultimately be validated on new data.

Preliminaries

```
from datetime import datetime as dt
```

```
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-08-02 at 01:16
```

```
import os
os.getcwd()
```

```
'/content'
```

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

Because a cluster solution depends on the initial, somewhat arbitrary initial solution, a different solution results from each analysis. This impact is lessoned by automatically running multiple solutions, each with a different starting set of clusters, the default. Set the parameter `random_state` to an odd integer, again arbitrarily selected to reproduce these solutions.

For convenience only, here set `random_state` to a specific value referenced with two analyses so that the value only needs to be changed once to generate the same solutions.

```
start=47
```

Get and Structure Data

Read and Verify

```
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/BodyMeas.csv')
#d = pd.read_csv('data/BodyMeas.csv')
d.shape
```

```
(340, 8)
```

```
d.head()
```

	Gender	Weight	Height	Waist	Hips	Chest	Hand	Shoe
0	F	200	71	43	46	45	8.5	7.5
1	F	155	66	31	43	37	8.0	8.0
2	F	145	64	35	40	40	7.5	7.5
3	F	140	66	31	40	36	8.0	9.0
4	M	230	76	40	43	44	9.0	12.0

Create the Feature Data Structure

In general, a cluster analysis typically is based on several variables. Here, more for pedagogy, select just two features so that the result can be plotted and the clusters visualized.

```
#X = d[['Weight', 'Height', 'Waist', 'Hips', 'Chest', 'Hand', 'Shoe']]
```

```
X = d[['Hand','Height']]
X.head()
```

	Hand	Height
0	8.5	71
1	8.0	66
2	7.5	64
3	8.0	66
4	9.0	76

```
n_features = X.shape[1]
print('Number of features:', n_features)
```

```
Number of features: 2
```

Missing data check.

```
print (d.isna().sum())
print ('\nTotal Missing:', d.isna().sum().sum())
```

```
Gender    0
Weight    0
Height    0
Waist     0
Hips      0
Chest     0
Hand      0
Shoe      0
dtype: int64

Total Missing: 0
```

Standardize the Variables

Because cluster analysis is based on straight-line (Euclidean) distance, the variables must be on at least approximately the same scale. Here standardize each variable to a mean of 0 and a standard deviation of 1, the z-scores. Most transformed values lie between -3 and 3 IF normally distributed.

```
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
s_scaler = preprocessing.StandardScaler()
X = s_scaler.fit_transform(X)
```

```
X = pd.DataFrame(X, columns=['Hand','Height'])
X.head()
```

	Hand	Height

Use `describe()` to calculate descriptive statistics that verify that the transformations correctly resulted in z-scores for each of the X variables.

```
X.describe().round(3)
```

	Hand	Height
count	340.000	340.000
mean	-0.000	0.000
std	1.001	1.001
min	-2.504	-2.297
25%	-0.706	-0.832
50%	0.065	-0.100
75%	0.578	0.877
max	2.633	2.342

The mean and standard deviation of each standardized variable are 0 and 1, respectively.

Find Optimal Number of Clusters

Each cluster analysis requires first to specify the number of clusters. A single cluster analysis cannot evaluate the optimal number of clusters to obtain the best fit. Instead, run a variety of cluster analyses, treating the number of clusters as a hyper-parameter.

Hyper-Parameter Tuning

Specify the k-means analysis with the `sklearn` function `Kmeans()`. Apply the usual `sklearn` function `fit()` to do the analysis.

However, a specific cluster analysis begins from the specified number of clusters, finding the best fit for that number of clusters. Beyond the best fit for a single model, the more general question is to understand how many clusters exist in the data. Answer that question with a hyper-parameter tuning of the number of clusters, systematically running different cluster models with differing numbers of clusters.

```
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.metrics import pairwise_distances
from sklearn.metrics import silhouette_samples, silhouette_score
```

Set the variables needed to loop through a range of different numbers of clusters. Specify the maximum number of clusters with our own defined variable *max_nc*. Define two empty arrays, *inertia* and *silhouette* for storing the corresponding error indices for each cluster analysis at a specified number of clusters.

```
max_nc = 25
inertia = []
silhouette = []
```

For the `sklearn` function `KMeans()`, specify the number of clusters with parameter `n_clusters`. Use a `for` loop that varies the index *i* to run analyses from 2 to 25 cluster solutions systematically. All lines of indented code are the code that runs in the `for` loop.

By default, try 10 different arbitrary initial cluster solutions for each cluster analysis with the `init` parameter set to `k-means++` to specify the initial cluster seeds for each solution. To provide for a more stable, optimal solution, up this default to 100 with the parameter `n_init`. The variable *start* was previously set as an arbitrary odd number to be able to recover the solution. Or, could just set this as a constant directly.

Evaluate fit of each solution in terms of cluster inertia from created data structure *model.inertia_*, as well as the average silhouette value from *silhouette_score*. Store these fit values in the previously created *inertia* and *silhouette* arrays.

Note that a complete analysis would involve a systematic grid search over the number of clusters and the number of features.

```
for i in range(2, max_nc):
    model = KMeans(n_clusters=i, init='k-means++', n_init=100, random_state=start)
    model.fit(X)
    inertia.append(model.inertia_)
    s_score = metrics.silhouette_score(X, model.labels_, metric='euclidean')
    silhouette.append(s_score)
```

Choose the Optimal Model

To analyze the fit of each cluster solution, display both Silhouette and Inertia for each specified number of clusters. First, print the column headings, followed by a dotted line. A `for` loop specifies the printing of the fit coefficients for each solution.

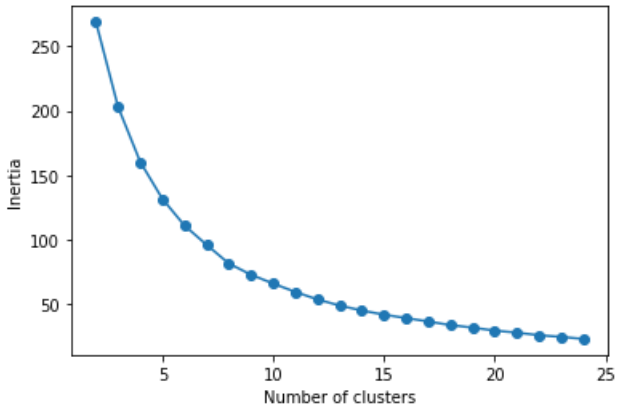
Each `format` specifies how to display the values. For the column headings, set the spacing of each. The `d` format applies to integers for the numerical output, and the `f` format applies to numbers with decimal digits, the width of the entire numerical output, and the number of decimal digits.

```
print('{:>2}{:>11}{:>9}'.format('nc', 'Silhouette', 'Inertia'))
print('-' * 24)
for i in range(2, max_nc):
    print('{:>2d}{:>8.3f}{:>12.3f}'.format(i, silhouette[i-2], inertia[i-2]))
```

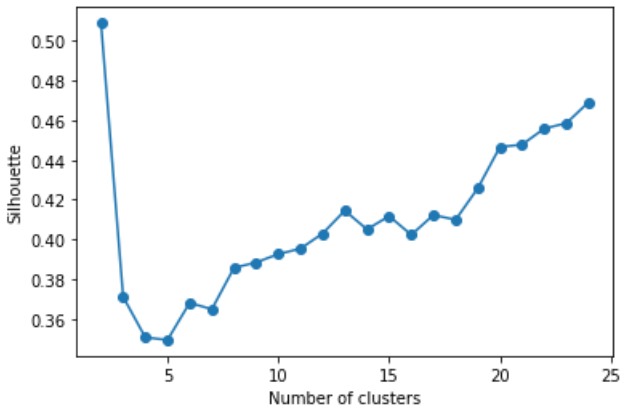
```
 nc Silhouette  Inertia
 ------------------------
  2   0.509     268.921
  3   0.371     203.022
  4   0.351     160.368
  5   0.349     131.704
  6   0.368     110.968
  7   0.365      95.997
  8   0.386      81.539
  9   0.388      72.883
 10   0.393      65.994
 11   0.395      59.540
 12   0.403      53.671
 13   0.415      48.966
 14   0.405      45.091
 15   0.412      41.876
 16   0.402      39.236
 17   0.412      36.593
 18   0.410      33.782
 19   0.426      31.837
 20   0.447      29.498
 21   0.448      27.932
 22   0.456      25.928
```

```
     23   0.459       24.621
```

```
plt.plot(range(2, max_nc), inertia, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```




```
plt.plot(range(2, max_nc), silhouette, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette')
plt.show()
```



For this analysis with two features, the two cluster solutions appear the most appropriate. As always, as the number of clusters increases, cluster inertia, the sums of squared errors of the points in a cluster about its centroid, decreases. But the silhouette score drops much after two clusters. It does slowly increase, so a 25 cluster solution has the best fit in statistical terms, but is not meaningful in interpretative terms.

The question now is to the extent that the two-cluster solution provides an interpretable solution.

Implement Chosen Cluster Model

Do the Cluster Analysis

Focus on the chosen solution of two clusters, according to the value of *n_clusters*.

```
model = KMeans(n_clusters=2, init='k-means++', n_init=100, random_state=start)
model.fit(X)
```

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
       n_clusters=2, n_init=100, n_jobs=None, precompute_distances='auto',
       random_state=47, tol=0.0001, verbose=0)
```

Evaluate Fit

The `fit()` function generates variables *inertia_* and *labels_*. We have these values from the hyper-parmater tuning, but repeat here for reference, and to verify we have the same model.

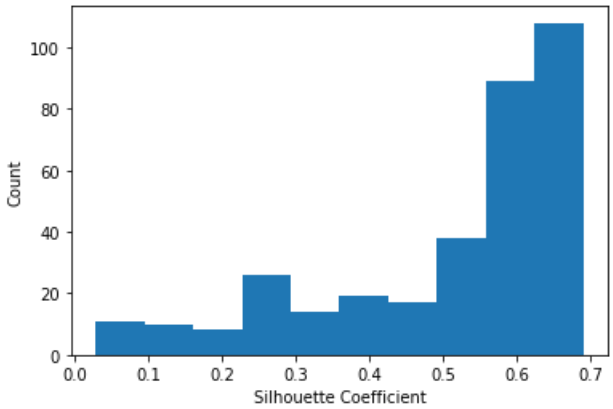
```
s_score = metrics.silhouette_score(X, model.labels_, metric='euclidean')
print('Error: %.3f ' % model.inertia_)
print('Mean silhouette score: %.3f' % s_score)
```

```
Error: 268.921
Mean silhouette score: 0.509
```

View the individual silhouette values from function `silhouette_samples()`. Ww want as many as possible larger than around 0.4 and 0.5.

```
s_values = silhouette_samples(X, model.labels_)
plt.hist(s_values.round(3))
plt.xlabel('Silhouette Coefficient')
plt.ylabel('Count')
```

```
Text(0, 0.5, 'Count')
```



The two-cluster solution appears to fit the data well. Most individual silhouette are above 0.5, and there are no negative values.

Interpret the Cluster Solution

The cluster analysis has no business value unless the cluster solution can be meaningfully interpreted and related to how the world works. To do this interpretation, we need to know the cluster that each sample belongs to and the corresponding unstandardized cluster centroid (center). As usual in data analysis, the center summarizes the constituent parts, the individual rows of data, the samples, that comprise each cluster.

Assign Cluster Stats to Data

Update the original data table for each sample with the cluster assignment and the silhouette score. This enhanced data set allows further exploration of the cluster assignments to other variables in the data.

Obtain the cluster assignment with the output (created) variable *labels_*. There is no "prediction" in this context as there is no y variable as a target in the sense of supervised training only features.

Obtain the silhouette values from the previously computed *s_values* from function `silhouette_samples()`. These values are optional but allow the detection of samples that do not fit well with any cluster, should such an analysis be pursued.

As an option, save the original data retained in the analysis with `to_csv()`, though this option is commented out here with the

```
d['Cluster'] = model.labels_
d['S'] = s_values.round(3)
#d.to_csv("Clustered.csv", header=True)
d.head()
```

	Gender	Weight	Height	Waist	Hips	Chest	Hand	Shoe	Cluster	S
0	F	200	71	43	46	45	8.5	7.5	1	0.408
1	F	155	66	31	43	37	8.0	8.0	0	0.594
2	F	145	64	35	40	40	7.5	7.5	0	0.690
3	F	140	66	31	40	36	8.0	9.0	0	0.594
4	M	230	76	40	43	44	9.0	12.0	1	0.571

Cluster Centroids with Counts

Do a frequency distribution of the cluster membership. The `value_counts()` function only applies to data frame variables, so convert.

```
d_lab = pd.DataFrame(model.labels_, columns=['labels'])
count = d_lab['labels'].value_counts()
count
```

```
0    174
1    166
Name: labels, dtype: int64
```

To interpret the clusters, view their centroids. View the cluster centroids as they exist in terms of the analysis for the cluster solution, that is, standardized. The created data structure `cluster_centers_` provides the centroid for each cluster, though in the metric in which the cluster analysis is conducted, here standardized.

```
dcc = pd.DataFrame(model.cluster_centers_,
          columns=['Hand', 'Height']).round(3)
dcc['Count'] = count
dcc.sort_values('Count', ascending=False)
```

	Hand	Height	Count
0	-0.736	-0.782	174
1	0.772	0.820	166

Negative standardized values indicate the number of standard deviations below the mean. Cluster 0 cutomers have below average hand size and height. The opposite is true for Cluster 1 customers.

In a cluster analysis with more than two features on which to cluster the samples, this table of the cluster centroids (centers) is the primary output for interpreting the meaning of the clusters. (With only two features, we also plot the data and cluster centroids).

To better interpret the clusters, view the cluster centers in terms of the original, unstandardized data. Compute the mean of each feature by cluster with function `groupby()` followed by the function `mean()`. Convert to a data frame and restore the variable names. Sort the clusters by frequency, from largest to smallest.

Can also save the unstandardized cluster centers to a file with the `to_csv()` function in terms of the original data as read from an external file. That option is commented out here with a #.

```
avg = d.groupby(d['Cluster']).mean().round(2)
d_avg = pd.DataFrame(avg, columns=['Hand', 'Height']).round(3)
d_avg['Count'] = count
d_avg = d_avg.sort_values('Count', ascending=False)
#d_avg.to_csv("ClustCenters.csv", header=True)
d_avg
```

Cluster	Hand	Height	Count
0	7.72	65.21	174
1	9.19	71.77	166

```
print('Hand size difference:', (d_avg.loc[1,'Hand'] - d_avg.loc[0,'Hand']).round(3))
print('Height difference:   ', (d_avg.loc[1,'Height'] - d_avg.loc[0,'Height']).round(3))
```

```
Hand size difference: 1.47
Height difference:    6.56
```

Primary interpretation: Cluster 0 samples, on average, have smaller Hand sizes and Heights, and Cluster 1 samples, on average, have larger Hand sizes and Heights.

Cluster Scatterplot

With only two features, we can also create a scatterplot to visualize the location of the samples and the cluster centroids in the plot of Hand sizes and Height.

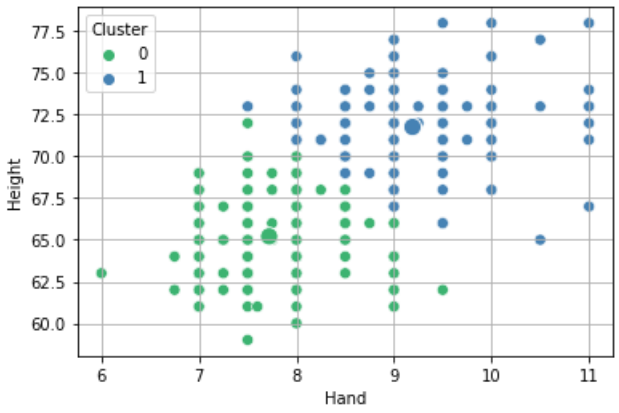
Create a `seaborn` (sns) scatterplot, with also relies upon the original `matplotlib` (plt) visualization system. The resulting scatterplot is actually two super-imposed scatterplots, one of the data and one of the cluster centroids. Parameter `s` specifies the size of the plotted points, a larger value for the centroids. Parameter `hue` sets the grouping variable. Each group is plotted in a different color according to the specified *palette*.

With the parameter `palette`, assign the colors to the points based on the `hue` specification, cluster membership. Use a Python dictionary, indicated by the { and } to set the colors. The names Cluster 0 and Cluster 1 from the cluster analysis solution are arbitrary, and, in this example, go counter to the way that the `scatterplot()` function assigns the colors to the data values. That is why the assignment of colors to the cluster centroids is reversed from the data scatterplot. The initial assignments were arbitrary, so align the centroid and data colors as needed with the dictionary.

The `if` clause indicates to run the following indented code only if *n_features* equals 2, which it does in this example. Standard Python syntax specifies that all lines of code that run when the `if` clause is true must be indented from that statement, as in the following code.

Note: If there are many samples per cluster, the cluster centroid, though displayed as a bigger circle, still blends in with the surrounding points. In that case, for printing the cluster centroids, drop the `hue` parameter, which displays a different color for each centroid, and replace with a single color such as `color='black'` or `color='DimGray'`.

```
if n_features == 2:
    sns.scatterplot(x=d['Hand'], y=d['Height'], s=50,
                hue=d['Cluster'], palette={0: 'MediumSeaGreen', 1: 'SteelBlue'})
    sns.scatterplot(x=d_avg['Hand'], y=d_avg['Height'], s=125, hue=d['Cluster'],
                    palette={1: 'MediumSeaGreen', 0: 'SteelBlue'}, legend=False)
    plt.grid()
```



With two clusters specified, the cluster solution generally recovers the Male/Female body size division based on Hand size and Height.

Evaluate Solution Against Ground Truth

After a cluster analysis, when there are other variables in the analysis, one question often of interest is if the clusters are related to other variables. Such a correspondence can suggest future supervised machine learning targeting the related variable. To uncover these relationships, do cross-tabulations of cluster membership with the values of other variables.

As an example, consider the relationship between the Cluster membership and Gender.

> *Ground truth*: The extent to which the classification of the samples reflects the true reality.

In this case, to what extent did the cluster solution, without any input from Gender, express Gender in terms of body type? Previous analyses of these data showed how supervised learning methods such as logistic regression and decision trees could predict or account for Gender. In this unsupervised analysis, there was no knowledge of Gender available to the clustering algorithm.

Can the clustering algorithm when computing the model recover the classification into groups without awareness of the groups? The previous scatterplot of Hand size and Height indicates a general classification of Male and Female. To what extent do the clusters from the two-cluster solution correspond to Male and Female body types?

The basic display of the classification output is the *confusion matrix*, previously introduced. One way to obtain is with the pandas function `crosstab()` for cross-tabulation. This method works with the non-numeric Gender values as the procudure simply counts the values in each group.

```
ct = pd.crosstab(d['Gender'], d['Cluster'])
ct
```

Cluster	0	1
Gender		
F	155	15
M	19	151

The standard procedure is for all non-numeric variables to be convered to numeric prior to machine learning. Gender was not part of this cluster analysis, and so was not converted. To proceed with `sklearn` functions in this post-analysis, we do, however, need to convert Gender to numeric.

Another way to get the confusion matrix is from the `sklearn` function `confusion_matrix()`. Its use, however, requires only numeric data, so replace the M and F data values with 1 and 0, respectively.

```
d['Gender'] = d['Gender'].replace({'M':1, 'F':0})
pd.DataFrame(metrics.confusion_matrix(d['Gender'], d['Cluster']))
```

	0	1
0	155	15
1	19	151

From the confusion matrix, obtain the traditional binary classification evaluation metrics from `sklearn`.

```
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score
print ('Accuracy: %.3f' % accuracy_score(d['Gender'], d['Cluster']))
print ('Recall: %.3f' % recall_score(d['Gender'], d['Cluster']))
print ('Precision: %.3f' % precision_score(d['Gender'], d['Cluster']))
print ('F1: %.3f' % f1_score(d['Gender'], d['Cluster']))
```

```
Accuracy: 0.900
Recall: 0.888
Precision: 0.910
F1: 0.899
```

With no knowledge of Gender, the cluster analysis obtained 90% accuracy of classification into male and female body types. This classification accuracy is particularly impressive given that it is about as accurate as the decision tree analysis, a supervised method explicitly based on the target Gender. Yet Gender did not even appear in the cluster analysis.

Plot the four values of the confusion matrix from the data structure *ct* with a two-variable bar chart.

```
ct.plot(kind='bar', color=['mediumseagreen','steelblue'])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd75639af90>
```

This cluster analysis of Hand size and Height, that is unsupervised, uncovered the structure of two distinct clusters that identify Male and Female body types. The unsupervised cluster analysis method did not include Gender, yet it was unable to cover the structure provided by the different body types.

Had our investigation of body measurements begun with the cluster analysis, by discovering a pattern, the unsupervised analysis identifies a potential target variable for future supervised analyses.