Course Notes

Course 811M:
Python for Data Scientists

ROI Training
Maximize Your Training Investment
www.roitraining.com
US: 1-800-ROI-9877 | Outside US: +44 (0)20 3287 6485

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

INTRODUCTION

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

The following course materials are copyright protected materials. They may not be reproduced or distributed and may only be used by students attending the *Python for Data Scientists* course.

Welcome!

- ROI leads the industry in designing and delivering customized technology and management training solutions
- Meet your instructor
  - Name
  - Background
  - Contact info
- Let's get started!

Course Objectives

In this course, we will:

- Learn how to use Python for data science and machine learning
- Explore the most common packages such as NumPy, Pandas, and Sklearn
- Learn how to transform data into the correct shape for analysis
- Explore supervised and unsupervised models including Cluster, Classification, and Regression
- Become familiar with graph database concept using network package

Course Contents

Class Schedule

- Start of class _____
- Morning breaks approximately on the hour
- Lunch _____
- Afternoon breaks approximately on the hour
- Class end _____

ROI's Training Curricula

Agile Development	.NET and Visual Studio
Amazon Web Services (AWS)	Networking and IPv6
Azure	Oracle and SQL Server Databases
Big Data and Data Analytics	OpenStack and Docker
Business Analysis	Project Management
Cloud Computing and Virtualization	Python and Perl Programming
Excel and VBA	Security
Google Cloud Platform (GCP)	SharePoint
ITIL® and IT Service Management	Software Analysis and Design
Java	Software Engineering
Leadership and Management Skills	UNIX and Linux
Machine Learning and Neural Networks	Web and Mobile Apps
Microsoft Exchange	Windows and Windows Server

Please visit our website at www.ROItraining.com for a complete list of offerings

Student Introductions

Please introduce yourself stating:

- Name
- Position or role
- How many years of experience you have with Python
- Expectations or a question you'd like answered during this class

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

CHAPTER 1: TOOLSET OVERVIEW

Chapter Objectives

In this chapter, we will introduce:

- What machine learning is
- The basics of Python
- The toolset to be introduced in this course

Chapter Concepts

What Is Machine Learning?

Python Primer

The Common Toolsets

Chapter Summary

Machine Learning

- Machine learning goes beyond simply summarizing data and instead uses complex math to:
  - Make predictions of future values
  - Find hidden patterns
  - Spot outliers or anomalies
- Is more calculation and CPU intensive than traditional reporting
  - Applies mathematical analysis to find how various attributes influence others
  - Requires an understanding of both the raw data and the business in order to explore the deeper meaning of the information
  - Each question asked leads to new questions and involves a lot of trial and error
- Can be used to find actionable business insights

Use Cases

- Predict a future value based on past experience
  - How much of a product should be produced to meet expected demand
  - Probability a disease will respond to treatment based on a mix of patient characteristics
  - What the future price of a commodity is likely to be based on
  - Weather forecast
- Classify a record into a different category
  - Determine if a person should qualify for a credit card
  - Decide if a new card purchase is likely to be legitimate or fraudulent
  - Predict whether a patient is at high or low risk of disease to determine correct treatment
  - Is it likely to rain tomorrow?
- The methods used in machine learning transcend any one field or discipline and can be applied equally in the financial, medical, scientific, entertainment fields, and more

The Science Part

- The Science part of Data Science comes from the fact that the way we approach solving these problems is the same way scientists in general approach their work, the Scientific Method:
  - Ask a question
  - Construct a hypothesis
  - Test the hypothesis by doing experiments and acquiring data
  - Analyze the data and draw a conclusion about whether it supports or rejects the hypothesis
  - Communicate the findings to others
- Models are the key to how scientists accomplish this
  - Representation of an idea, system, object, process, or phenomenon that cannot be directly experienced or observed
  - Simulation of reality that can be used to see how close the model is at describing the real thing we want to understand
  - Always have some level of uncertainty
    - Scientists rarely say always or never, they speak in terms of probabilities

Models

- Even before computers mathematicians were able to identify certain patterns in numbers that are useful in modeling a simulation of reality
- With powerful computers capable of doing huge amounts of calculations these statistical models have been coded into easy to use preprogrammed modules that are more approachable to a wider audience
- The basic ideas behind these models have been coded by different programmers in different ways and on a variety of platforms and languages, but the basic principles remain the same
- To use them, you do not need to understand all the math behind them, but a rudimentary understanding helps
- A big picture understanding of which model can be used to generate a certain result and basic skills in preparing the data is all that is needed to harness the power they offer

Basic Models

- There are a lot of different models and even within a particular type of model, there are subtle variations and implementation differences
- The most fundamental and most widely used models generally fall into one of the following types:
  - Linear Regression
  - Classification
  - Dimension reduction
  - Tree-based methods
  - Clustering
  - Many more

Supervised Models

- Models can be supervised or unsupervised
- Supervised
  - Need to be trained using a dataset with known values we are trying to predict
  - Require a training and testing set to validate how good a job the model does
  - Use a new set of data with unknown values to try to predict what they will be
  - Examples:
    - Regression – good for continuous values like predicting a price
    - Classification – good at identifying the data as a member of a group or category
- Requires you to have well labeled features and known values you are trying to predict

Unsupervised Models

- Unsupervised
  - Can do their analysis on a static set of data without the need to separate it into two sets
- Examples:
  - Cluster Analysis
  - Dimension Reduction
  - Principal Component Analysis
  - Anomaly Detection
  - Association and Recommendation Engines
  - Autoencoders
- You don't give it a set of known values you are trying to predict, instead you give it raw data and let it identify natural groupings that help shed light on what the data may represent
- Difficult to measure how accurate it is, but it's helpful in analyzing data to build towards a supervised classification model

Semi-Supervised and Reinforcement

- A hybrid between supervised and unsupervised
- Useful in cases where it's not easy to label the datasets
  - For example, reading CT scans
- Looks for natural patterns to help label the data then trains itself how to predict based on the patterns it finds
- Generative Adversarial Network invented in 2014
  - Creates two neural networks that compete with each other to see which does a better job
- Low Density Separation is another technique that uses a variation on Support Vector Machines to teach itself about the categories in the dataset
- Reinforcement Learning uses rewards to help an agent accomplish its goal

Chapter Concepts

What Is Machine Learning?

Python Primer

The Common Toolsets

Chapter Summary

Why Python for Data Analysis?

- Many choices of tools for data analysis, exploratory computing, and data visualization
  - R
  - MATLAB
  - SAS
  - Etc.
- Python provides the following which makes it a strong choice:
  - Strong library support for analysis
  - General purpose programming language

Python

- Python is a general-purpose programming language that is well suited to processing data and machine learning
- Some of its basic features are:
  - Interpreted
  - Easy to master syntax
  - Wealth of libraries in a variety of fields
  - Interoperates well with C for performance enhancements
- Python comes in versions 2 and 3
  - Differences are largely unimportant for machine learning
  - Ultimately choose the version that has the libraries you need
  - Most common libraries exist for both versions
  - At this point it makes the most sense to start new projects with Python 3 since Python 2 will not be maintained after 2020

Data Types

- Python has all the basic data types one would expect in a language
  - Strings (`str`) can be enclosed in single, double, or triple quotes
  - Integers (`int`) and floating point (`float`) numbers
  - Booleans (`bool`) for True and False
  - Dates (`date`, `time`, `datetime`)
- There are also complex types used to store collections of data
  - List (`list`) uses square brackets `[]`
    - Flexibly stores an ordered collection of almost any type of objects
  - Set (`set`) uses curly braces `{}`
    - Stores a unique set of any type of objects
  - Dictionary (`dict`) uses curly braces `{}` and colon `:`
    - Key and Value pair to store any type of objects
  - Tuple (`tuple`) uses parentheses `()`
    - Like a list but immutable and slightly faster
    - Usually used to encode row-like object that is a collection of columns

Lists

- Can contain any type of data, are mutable and stored in order
- Created by enclosing values in `[]` or by supplying another collection object as a parameter to `list()`
  - `numbers = [1, 2, 3, 4]`
  - `students = [['Abe', 10], ['Betty', 11], ['Charles', 12]]`
- Can get individual elements or slices using `[]`
  - `numbers[0]` → `1`
  - `numbers[1:3]` → `[2, 3]`
  - `numbers[:3]` → `[1, 2, 3]`
  - `numbers[1:]` → `[2, 3, 4]`
  - `numbers[-2]` → `3`
  - `students[1]` → `['Betty',11]`
  - `students[1][0]` → `'Betty'`
- Has methods to append, sort, remove items, etc.

Sets

- Like a list, but only contains unique elements
  - `numbers = {1, 2, 3, 2}` → `{1, 2, 3}`
- Has methods to add and remove items and to find intersection, difference, and union with another set
  - `numbers.add(5)` → `{ 1, 2, 3, 5}`
  - `numbers.add(3)` → `{1, 2, 3, 5}`
  - `numbers.union({1, 3, 4, 6})` → `{1, 2, 3, 4, 5, 6}`
  - `numbers.remove(5)` → `{1, 2, 3}`
  - `numbers.intersection({1, 2, 4})` → `{1, 2}`
  - `numbers.difference({2, 4})` → `{1, 3}`
- Set is often used to make it unique
  - `names = ['Abe','Betty','Carl','Abe']`
  - `names = list(set(names))` → `['Abe','Betty','Carl']`

Tuples

- Tuples are like lists, but immutable and slightly more efficient
- Generally used to represent a row structure or multiple different values
- Has no methods to modify them, in fact the only real method is `count()`
- `numbers = (1, 2, 3, 2)`
- `numbers[0]` → `1`

Dictionaries

- Like in JSON, dictionaries are used to encode key value pairs
- Useful for storing any kind of data
- Instead of being indexed by position, the value is indexed by a key so it is efficient to look up
  - `person = {'firstname':'John', 'lastname':'Smith', 'age':40}`
  - `people = {101:{'firstname':'John', 'lastname':'Smith', 'Age':40}, 201:{'firstname':'Mary', 'lastname':'Jones', 'Age':35}}`
  - `person['firstname']` → `'John'`
  - `people[201]['lastname']` → `'Jones'`
  - `k = person.keys()` → `['firstname', 'lastname', 'age']`
  - `v = person.values()` → `['John', 'Smith', 40]`
  - `x = person.items()` → `[('firstname', 'John'), ('lastname', 'Smith'), ('Age', 40)]`
  - `zip(k,v)` → `{'firstname':'John', 'lastname':'Smith', 'age':40}`

List Comprehensions

- All collection objects can be queried in a SQL-like manner using list comprehensions
- Enclose an expression in `[]` to return list, `()` to return generator, or `{}` to return set or dictionary
  - `[x * 2 for x in [1, 2, 3, 4]]` → `[2, 4, 6, 8]`
  - `[x for x in [1, 2, 3, 4, 4] if x % 2 == 0]` → `[2, 4, 4]`
- Enclose an expression in `()` to return generator
  - `(x * 2 for x in [1, 2, 3, 4])` → `<generator object <genexpr> at 0x7fdd70291480>`
  - `tuple((x * 2 for x in [1, 2, 3, 4]))` → `(2, 4, 6, 8)`
- Enclose an expression in `{}` to return set
  - `{x for x in [1, 2, 3, 4, 4] if x % 2 == 0}` → `{2, 4}`
- Enclose an expression in `{}` to return dictionary
  - `{ k: v['firstname'] for k,v in people.items()}` → `{101: 'John', 201: 'Mary'}`

Functions and Lambdas

- To create a function in Python is simple:
  - `def func(x):`
    `body`
    `return value`
  - `def double(x):`
    `return x * 2`
- Sometimes a function is so short and just returns a calculation you could abbreviate as a lambda expression
  - `lambda x : x * 2`
  - `double = lambda x : x * 2`
- Some functions can take other functions as a parameters
  - You can either pass a named function or embed a lambda expression in such cases

Collection Functions

- Processing collections is fairly common, so to avoid writing loops there are some helpful functions
  - `data = [10, 2, 30, 4, 5]`
- `map` is a common function used to call a function on each element of a collection. Similar to return value of list comprehension.
  - `map(lambda x : x * 2, data)` → `[20, 4, 60, 8, 10]`
- `filter` can be used to return elements from a collection that meet a condition. Similar to `if` clause of list comprehension.
  - `filter(lambda x : x < 10, data)` → `[2, 4, 5]`
- `sorted` returns a list in a sorted order
  - `sorted(data)` → `[2, 4, 5, 10, 30]`
  - `sorted(data, reverse=True)` → `[30, 10, 5, 4, 2]`
  - `sorted(data, key=lambda x : (x % 2, x))` → `[2, 4, 10, 30, 5]`
- `reduce` can do an aggregation function on a collection
  - `from functools import reduce`
    `reduce(lambda x, y: x + y, data)` → `51`

Modules

- Modules are Python code that has been bundled up for convenience
- Contain libraries full of function and classes programmed for a particular functionality
- You import a module to include the code in your scripts
  - `import module`
  - `import module as alias`
  - `from module import function`
  - `from module import function as alias`
- Once imported, you can use the code in the module using the module name and function name or alias
  - `module.function()`
  - `alias.function()`
  - `function()`
  - `alias()`

Chapter Concepts

What Is Machine Learning?

Python Primer

The Common Toolsets

Chapter Summary

Essential Python Libraries and Tools

- In this course, we will use the following scientific libraries:
  - NumPy
  - Pandas
  - Matplotlib
  - SciPy
  - Scikit-learn
  - Seaborn
- As well as various UIs:
  - IPython
  - Jupyter Notebook
  - Spyder
  - PyCharm
- And explore other resources like:
  - Google Cloud Platform (GCP)
  - Hadoop
  - Spark

NumPy

- Numerical Python is the foundational package for scientific computing in Python
- NumPy provides:
  - Fast and efficient multidimensional array object `ndarray`
  - Functions for performing computations on arrays
  - Reading/writing array-based data sets to disk
  - Linear algebra operations
  - Fourier transforms
- NumPy arrays are much more efficient for storing and manipulating data than other Python structures
- `import numpy as np`

SciPy

- Collection of packages for scientific computing:
  - `integrate`
    - Numerical integration
  - `linalg`
    - Linear algebra and matrix decomposition
  - `optimize`
    - Function optimizers and root finding algorithms
  - `signal`
    - Signal processing tools
  - `sparse`
    - Sparse matrices and sparse linear system solvers
  - `stats`
    - Standard continuous and discrete probability distributions
- `import scipy as sp`

Pandas

- Provides rich data structures and functions
  - Makes working with structured data fast and easy
- Combines high-performance array features of NumPy with flexible data manipulation features of spreadsheets and databases
- Indexing functionality enables reshaping, slice/dicing of data
- Provides time-series functionality useful for financial data
- `import pandas as pd`

Matplotlib and Seaborn

- Used for producing plots and 2D data visualizations
- Provides interactive environment for plotting and exploring data
- Plots are interactive, enabling zoom and pan of plots
- `import matplotlib as mp`
- `from matplotlib import pyplot as plt`
- Seaborn builds on Matplotlib and adds more plots and visualization
- `import seaborn as sns`

Scikit-learn

- Toolset for data mining and data analysis
- Built on NumPy, SciPy, and Matplotlib
- Support for a number of areas, such as:
  - Classification
  - Regression
  - Clustering
- `import sklearn as sk`

User Interfaces

- IPython is an enhanced Python shell
  - Provides interactive environment for scientific computing
- Useful when interactively working with data using matplotlib
- Also provides:
  - Jupyter Notebook for working with web browser to combine markdown and Python code mixed together
  - GUI console with inline plotting, multiline editing, and syntax highlighting
- PyCharm is a popular, general purpose Integrated Development Environment (IDE)
- Spyder is another popular IDE among data scientists

Other Resources

- Hadoop is a cluster of computers that allows for storage and processing of huge multi-terabyte datasets
- Spark is a processing engine that:
  - Works very fast on a cluster
  - Supports a lot of machine learning algorithms
  - Uses Python in addition to Scala and Java as a programming language
- GCP provides the ability to create cloud-based machines or use existing services to do a lot of the machine learning

Installation and Setup

- Simplest way of obtaining tools is to install Anaconda
  - Provided by Continuum Analytics
- You can also use regular Python and install the libraries you need with pip
  - `pip install numpy scipy pandas matplotlib sklearn seaborn`

Chapter Concepts

What Is Machine Learning?

Python Primer

The Common Toolsets

Chapter Summary

Chapter Summary

In this chapter, we have introduced:

- What machine learning is
- The basics of Python
- The toolset to be introduced in this course

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

CHAPTER 2: NUMPY ESSENTIALS: ARRAYS AND VECTORIZED COMPUTATION

Chapter Objectives

In this chapter, we will introduce:

- Universal functions: fast element-wise array functions
- Data processing using arrays
- File input and output with arrays
- Linear algebra
- Random number generation

Chapter Concepts

Array Functions

Data Processing

File Input Output

Linear Algebra

Random Numbers

Chapter Summary

NumPy `ndarray`

- `ndarray` is a N-dimensional array object
  - Fast, flexible container for large data sets in Python
- Easiest way to create an array is to use the `array` function
  - Accepts any sequence-like object and produces `ndarray`

```
import numpy as np

data = [1,2,3,4]

array1 = np.array(data)

array1
array([1, 2, 3, 4])
```

Access `numpy` library

Create `numpy` array

NumPy `ndarray` (continued)

- Nested sequences, e.g., list of lists are converted to a multi-dimensional array

```
data2 = [[1,2,3,4],[5,6,7,8]]

array2 = np.array(data2)

array2
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

- Data type is inferred from array data used
  - Stored in property `dtype`

```
array2.dtype
dtype('int64')
```

Other Functions for Creating Arrays

- Other functions are provided for creating arrays
  - `zeros` – creates array of 0's
  - `ones` creates array of 1's
  - `empty` creates uninitialized array

```
np.zeros(4)
array([ 0.,  0.,  0.,  0.])

np.ones((2,4))
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])

x = np.empty((2,4))
x[:] = 0
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

No guarantee elements will be `0` so do this to guarantee it

Data Types for `ndarrays`

- `dtype` is a special object that defines type of data in array
  - Can be set when creating array
- Full set of data types can be found at
  - https://docs.scipy.org/doc/numpy/user/basics.types.html

```
array1 = np.array([1,2,3,4,5], dtype=np.float64)

array1
array([ 1.,  2.,  3.,  4.,  5.])

array1.dtype
dtype('float64')
```

Specify type of data

Exercise 2.1: Array Creation

- Please turn to the Exercise Manual and complete Exercise 2.1

Chapter Concepts

Array Functions

Data Processing

File Input Output

Linear Algebra

Random Numbers

Chapter Summary

Operations Between Arrays

- Arrays allow operations on elements without writing loops
  - Usually called *vectorization*
- Arithmetic operations between equal sized arrays applies the operation element to element

```
array1 = np.array([[1,2,3],[4,5,6]])

array1

array([[1, 2, 3],
       [4, 5, 6]])

array1+array1
array([[ 2,  4,  6],
       [ 8, 10, 12]])
```

Standard arithmetic operators

Operations Between Arrays and Scalars

- Operations on arrays with scalars propagate the value to each element

```
array1 = np.array([[1.,2.,3.],[4.,5.,6.]])
array1

array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

1/array1
array([[ 1.        ,  0.5       ,  0.33333333],
       [ 0.25      ,  0.2       ,  0.16666667]])

array1*3
array([[  3.,   6.,   9.],
       [ 12.,  15.,  18.]])
```

Operation with scalar

Indexing and Slicing

- One-dimensional arrays are similar to Python lists
- Values applied to a slice are propagated (broadcasted) to the entire selection

```
array1 = np.arange(12)
array1
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

array1[2]
2

array1[2:5]
array([2, 3, 4])

array1[2:5] = 99
array1
array([ 0,  1, 99, 99, 99,  5,  6,  7,  8,  9, 10, 11])
```

Array slice

Value is broadcast to selection

Higher Dimensional Arrays

- In multi-dimensional arrays, elements can be accessed
  - Recursively
  - Comma-separated lists

```
array2d = np.array([[1,2],[3,4],[5,6]])
array2d[1]
array([3, 4])

array2d[1][0]
3

array2d[1,0]
3
```

Recursive index access

Comma-separated access

Higher Dimensional Arrays (continued)

- In multi-dimensional arrays, if later indices are omitted, returned objects are lower-dimensional arrays
- Consider the following 2 x 2 x 4 array

```
array3d
array([[[ 1,  2,  3,  4],
        [ 5,  6,  7,  8]],

       [[ 9, 10, 11, 12],
        [13, 14, 15, 16]]])

array3d[1]
array([[ 9, 10, 11, 12],
       [13, 14, 15, 16]])
```

Returns 2 x 4 array

Transposing Arrays

- Transposing returns a view of underlying data without copying data
- Reshape function will change dimensionality of array

```
array = np.arange(20).reshape((4,5))
array

array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

Reshape to a 4 x 5 array

Transposing Arrays (continued)

- Arrays have the transpose method and the `T` attribute
- `T` can be used to transpose axis

```
array

array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])

array.T
array([[ 0,  5, 10, 15],
       [ 1,  6, 11, 16],
       [ 2,  7, 12, 17],
       [ 3,  8, 13, 18],
       [ 4,  9, 14, 19]])
```

Transpose array

Mathematical and Statistical Methods

- A number of mathematical functions that compute statistics about a complete array along an axis are available
- Can be called on the array or calling top-level NumPy functions

```
array = np.random.randn(2,4)
array
array([[-0.25417597,  0.34146309,  2.52982769,  1.45988811],
       [ 1.50491216, -0.49649801,  0.72392736, -0.24452797]])

array.mean()
0.6956020590526804 7

np.mean(array)
0.6956020590526804 7

array.mean(axis=0)
array([ 0.62536809, -0.07751746,  1.62687753,  0.60768007])
```

Can specify axis for computation

Exercise 2.2: Array Basic Operations

- Please turn to the Exercise Manual and complete Exercise 2.2

Chapter Concepts

Array Functions

Data Processing

File Input Output

Linear Algebra

Random Numbers

Chapter Summary

File Input and Output with Arrays

- NumPy can load and save data from disk in text or binary format
  - By default, files are written in an uncompressed binary format
    - File extension `.npy`

```
array1 = np.arange(10)

np.save('array1.npy', array1)

array2 = np.load('array1.npy')

array2
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Extension added if not provided explicitly

File Archives

- Multiple arrays can be saved to an archive file using `np.savez()`
  - `np.load()` will return dictionary style object
    - Each array is loaded lazily

```
array1 = np.arange(10)

array2 = 2 * array1

np.savez('array_archive.npz', data_set_1=array1,
data_set_2=array2)

archive = np.load('array_archive.npz')

archive['data_set_1']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

archive['data_set_2']
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Data loaded lazily

Saving and Loading Text Files

- NumPy provides `loadtxt()` and `savetxt()` to read and write text files

```
array2d = np.array([[1,2,3],[4,5,6]])

np.savetxt('array_data.txt',array2d, delimiter=',')

array2d_from_file = np.loadtxt('array_data.txt',
delimiter=',')

array2d_from_file

array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

Data loaded lazily

Chapter Concepts

Array Functions

Data Processing

File Input Output

Linear Algebra

Random Numbers

Chapter Summary

Linear Algebra

- With NumPy, multiplying two two-dimensional arrays with * is an element-wise product, not a matrix dot product
- The function dot provides matrix dot product

```
array1 = np.array([[1,2,3],[4,5,6]])

array2 = np.array([[1,2,3],[4,5,6]])

array_multiply = array1 * array2

array_multiply

array([[ 1,  4,  9],
       [16, 25, 36]])
```

Result is element-wise multiplication

Linear Algebra (continued)

```
array1 = np.array([[1,2,3],[4,5,6]])

array2 = np.array([[1,2],[3,4],[5,6]])

array_dot_product = array1.dot(array2)

array_dot_product

array([[22, 28],
       [49, 64]])
```

Dot product of array: `array1 . array2`

numpy.linalg

- Has a standard set of matrix decompositions
  - E.g., `inv()`, `dot()` etc.
- Documentation found at:
  - https://docs.scipy.org/doc/numpy/reference/routines.linalg.html

```
array1 = np.array([[.1,.2,.3],[.4,.5,.6], [.7,.8,.9]])

inv(array1)

array([[ -6.74335773e+15,   1.34867155e+16,  -6.74335773e+15],
       [  1.34867155e+16,  -2.69734309e+16,   1.34867155e+16],
       [ -6.74335773e+15,   1.34867155e+16,  -6.74335773e+15]])
```

Chapter Concepts

Array Functions

Data Processing

File Input Output

Linear Algebra

Random Numbers

Chapter Summary

Random Number Generation

- `numpy.random` provides functions for generating arrays
  - From many kinds of probability distributions
    - Normal
    - Uniform
    - Poisson
    - Many more

```
print (np.random.normal(5, 2, 9)) # mean = 5, std = 2
array([6.81532146, 3.64397936, 6.68626991, 6.24245039,
2.74427372, 6.35545999, 3.19515877, 1.83536618, 2.59710754])

print (np.random.uniform(1, 100, 8)) # low = 1, high = 100
array([88.54188937, 21.03845531, 12.20124916, 64.99097202,
49.20289727, 33.33857889, 46.44605034, 31.57050879])

print (np.random.poisson(10, 10)) # 10 numbers averaging to 10
array([ 3,  8,  8, 12, 13, 14, 11, 10, 14, 11])
```

Chapter Concepts

Array Functions

Data Processing

File Input Output

Linear Algebra

Random Numbers

Chapter Summary

Chapter Summary

In this chapter, we have introduced:

- Universal functions: fast element-wise array functions
- Data processing using arrays
- File input and output with arrays
- Linear algebra
- Random number generation

ROITRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

CHAPTER 3: GETTING STARTED WITH PANDAS

Chapter Objectives

In this chapter, we will introduce:

- Pandas data structures
- Essential functionality of pandas
- Reading from data sources
- Summarizing and computing descriptive statistics
- Handling missing data
- Hierarchical indexing

Chapter Concepts

Introduction to Pandas Data

Essential Functionality

Reading From Data Sources

Summarizing and Computing Descriptive Statistics

Handling Missing Data

Hierarchical Indexing

Chapter Summary

Introducing Pandas

- *Pandas* is an open-source, BSD-licensed library
  - Provides high-performance, easy-to-use data structures and data analysis tools
- Common usages
  - `import pandas as pd`
  - `from pandas import DataFrame`
  - `from pandas import Series`

Pandas `Series` Data Structure

- Pandas provides the `Series` data structure
  - A one-dimensional array-like object
- Contains:
  - Array of data
  - Array of data labels known as the index
- `Series` object has `values` and `index` properties
- Can be thought of as a fixed-length dictionary

Series Example

```
from pandas import Series
data = Series([1,2,3,4])
print (data)
0    1
1    2
2    3
3    4
dtype: int64

print (data.values)
array([1, 2, 3, 4])

print (data.index)
RangeIndex(start=0, stop=4, step=1)

print (data[1])
2
```

Each data value is assigned an index from `N` through to `N-1`

Series Index

- It is possible to create a `Series` with a user-defined index for each data point

```
data = Series([1,2,3,4], index=['a','b','c','d'])
print (data.index)
Index([u'a', u'b', u'c', u'd'], dtype='object')

print (data['c'])
3
```

Index can be used to access data points

`Series` and Dictionaries

- A `Series` can be created by passing a dictionary
  - Dictionary keys are used for `Series` index by default
    - Separate keys can be provided

```
cities = {'Dublin': 200000, 'Athlone': 15000, 'Galway': 700000}
series1 = Series(cities)
print (series1)
Athlone     15000
Dublin     200000
Galway     700000
dtype: int64
```

`Series` and Dictionaries (continued)

- If no key is provided, then `NAN` is used
  - Can use `isnull()` and `notnull()` functions to detect missing data

```
cities = {'Dublin': 200000, 'Athlone': 15000, 'Galway':
700000}

indexes = ['Dublin', 'Athlone', 'Waterford']

series2 = Series(cities, index=indexes)

print (series2)

Dublin       200000.0
Athlone       15000.0
Waterford         NaN
dtype: float64
```

Missing value

Detecting Missing Data

```
print (series2)
Dublin       200000.0
Athlone       15000.0
Waterford         NaN
dtype: float64

print (series2.isnull())
Dublin       False
Athlone      False
Waterford     True
dtype: bool

print (series2.notnull())
Dublin        True
Athlone       True
Waterford    False
dtype: bool
```

Exercise 3.1: Pandas `Series`

- Please turn to the Exercise Manual and complete Exercise 3.1

DataFrame

- `DataFrame` represents a tabular data structure
  - Similar to spreadsheet
  - Contains ordered collection of rows and columns
- Has both a row and column index
- Most common way to construct is from a dictionary of lists or NumPy arrays
  - Must be equal length
  - Index will be provided automatically
  - Columns placed in sorted order by default

DataFrame Example

```
from pandas import DataFrame

data = {'team':['Leicester', 'Manchester City',
'Arsenal'], 'player':['Vardy', 'Aguero', 'Sanchez'],
'goals':[24,22,19]}

football = DataFrame(data)

print (football)
   goals   player              team
0     24    Vardy         Leicester
1     22   Aguero  Manchester City
2     19  Sanchez           Arsenal
```

DataFrame Indexes

- Column order can be specified when creating `DataFrame`

```
data = {'team':['Leicester', 'Manchester City',
'Arsenal'], 'player':['Vardy', 'Aguero', 'Sanchez'],
'goals':[24,22,19]}

football = DataFrame(data,
       columns=['player','team','goals','played'],
       index=['one','two','three'])

print (football)
```

Extra column

Extra column

```
         player              team  goals played
one       Vardy         Leicester     24    NaN
two      Aguero  Manchester City     22    NaN
three   Sanchez           Arsenal     19    NaN
```

Index Objects

- Index objects are immutable and cannot be modified
  - Can be shared across data structures
    - Act as a set

```
print (football)
        player             team  goals played
one      Vardy        Leicester     24    NaN
two     Aguero  Manchester City     22    NaN
three  Sanchez          Arsenal     19    NaN

print ('player' in football.columns)
True

print ('three' in football.index)
True
```

Set operations

- Index has a number of methods found at:
  - http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Index.html

Exercise 3.2: Pandas `DataFrame`

- Please turn to the Exercise Manual and complete Exercise 3.2

Chapter Concepts

Introduction to Pandas Data

Essential Functionality

Reading From Data Sources

Summarizing and Computing Descriptive Statistics

Handling Missing Data

Hierarchical Indexing

Chapter Summary

`Series` Indexing and Selection

- `Series` can be indexed using integers and indexes

```
data = Series(np.arange(4.0), index=['a','b','c','d'])
```

```
print (data)
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64

print (data[2])
2.0
```

```
print (data[['b','d']])
b    1.0
d    3.0
dtype: float64
```

```
print (data<2)
a     True
b     True
c    False
d    False

print (data[data<2])
a    0.0
b    1.0
dtype: float64
```

DataFrame Indexing and Selection

- Indexing retrieves one or more columns

```
data = DataFrame(np.arange(9).reshape((3,3)),
       index=['a','b','c'], columns=['one','two','three'])
```

```
print (data)
   one  two  three
a    0    1      2
b    3    4      5
c    6    7      8
print (data['three'])
a    2
b    5
c    8
Name: three, dtype:
int64
```

```
print (data[:2])
   one  two  three
a    0    1      2
b    3    4      5
```

```
print (data['two']>1)
a   False
b    True
c    True
Name: two, dtype:bool
print (data[data['two']>1])
   one  two  three
b    3    4      5
c    6    7      8
```

DataFrame Indexing with ix, loc, and iloc

- DataFrame has several indexing fields: ix, loc, and iloc
  - Allows selecting subset of rows and columns

```
data = DataFrame(np.arange(9).reshape((3,3)),
       index=['a','b','c'], columns=['one','two','three'])
   one  two  three
a    0    1      2
b    3    4      5
c    6    7      8
```

- All these could be used to retrieve the first row
  - `data.ix[0] data.ix['a'] data.loc['a'] data.iloc[0]`
- All these would retrieve the second column for all rows
  - `data.ix[:,'two'], data.ix[:,1], data.loc[:,'two'], data.iloc[:,1]`
- All these would retrieve the first two rows and the second column
  - `data.ix[['a','b'],'two'], data.ix[0:2,1], data.loc['a':'c':,'two'], data.iloc[0:2,1]`

Series
Arithmetic and Data Alignment

- When adding together objects if the index pairs are not the same, then index in result is the union of the index pairs

```
data1 = Series([1.0,2.0,3.0], index=['a','d','e'])
data2 = Series([2.0,3.0,4.0, 5.0], index=['a','b','c','e'])
```

```
print (data1)
a    1.0
d    2.0
e    3.0
dtype: float64

print (data2)
a    2.0
b    3.0
c    4.0
e    5.0
dtype: float64
```

```
print (data1+data2)
a    3.0
b    NaN
c    NaN
d    NaN
e    8.0
dtype: float64
```

NaN for indices that do not overlap

DataFrame
Arithmetic and Data Alignment

- DataFrame's alignment is performed on columns and rows

```
data1 = DataFrame(np.arange(9.0).reshape((3,3)),
columns=list('abc'), index=['one','two','three'])

data2 = DataFrame(np.arange(12.0).reshape((4,3)),
columns=list('ace'),index=['one','two','three','four'])
```

```
print (data1)
          a    b    c
one     0.0  1.0  2.0
two     3.0  4.0  5.0
three   6.0  7.0  8.0
```

```
print (data2)
          a     c     e
one     0.0   1.0   2.0
two     3.0   4.0   5.0
three   6.0   7.0   8.0
four    9.0  10.0  11.0
```

```
print (data1+data2)
          a   b     c   e
four    NaN NaN   NaN NaN
one     0.0 NaN   3.0 NaN
three  12.0 NaN  15.0 NaN
two     6.0 NaN   9.0 NaN
```

Index and columns are unions of data1 and data2

Arithmetic with Fill Values

- For arithmetic between differently indexed objects, can use `fill_value` to prevent missing values appearing in resultant data structure

```
print (data1)
          a    b    c
one     0.0  1.0  2.0
two     3.0  4.0  5.0
three   6.0  7.0  8.0
```

```
print (data2)
          a    c     d     e
one     0.0  1.0   2.0   3.0
two     4.0  5.0   6.0   7.0
three   8.0  9.0  10.0  11.0
```

```
data1.add(data2, fill_value=0)
            a    b     c     d     e
one       0.0  1.0   3.0   2.0   3.0
two       7.0  4.0  10.0   6.0   7.0
three    14.0  7.0  17.0  10.0  11.0
```

Function Application and Mapping

- A frequent operation is to apply a function to each column or row of `DataFrame`

```
data = DataFrame(np.random.randn(4,4))
print (data)
          0         1         2         3
0  0.546781 -0.862394 -2.384923 -0.098065
1 -0.219738  0.172776 -1.558335 -0.124880
2 -1.016070 -0.670825 -1.602997 -0.018526
3 -0.050491  0.258218  0.073574  0.144686
```

```
f = lambda x: x.max() - x.min()
```

```
print (data.apply(f))
print (data.apply(f, axis=0))
0    1.562851
1    1.120612
2    2.458497
3    0.269566
dtype: float64
```

Apply per column

```
print (data.apply(f, axis=1))
0    2.931705
1    1.731110
2    1.584471
3    0.308709
dtype: float64
```

Along row

Sorting

- Sorting on index on either axis is available, ascending order by default

```
print (data)
          0         1         2         3
0  0.546781 -0.862394 -2.384923 -0.098065
1 -0.219738  0.172776 -1.558335 -0.124880
2 -1.016070 -0.670825 -1.602997 -0.018526
3 -0.050491  0.258218  0.073574  0.144686
```

```
print (data.sort_index(ascending=False))
          0         1         2         3
3 -0.050491  0.258218  0.073574  0.144686
2 -1.016070 -0.670825 -1.602997 -0.018526
1 -0.219738  0.172776 -1.558335 -0.124880
0  0.546781 -0.862394 -2.384923 -0.098065
```

Specify axis = 1 for column sorting

Descending order on rows

- Can also sort by values instead of index

```
print (data.sort_values( by=[1], ascending=False))
```

Ranking

- Ranking assigns ranks from 1 to the number of valid data points in an array

```
data = DataFrame({'b':[1,4,3,2], 'a':[6,9,20,3],
'c':[7,2,8,15]})
print (data)
```

```
    a  b   c
0   6  1   7
1   9  4   2
2  20  3   8
3   3  2  15
```

Rank on column

```
print (data.rank())
```

```
     a    b    c
0  2.0  1.0  2.0
1  3.0  4.0  1.0
2  4.0  3.0  3.0
3  1.0  2.0  4.0
```

Rank on row

```
print (data.rank(axis=1))
```

```
     a    b    c
0  2.0  1.0  3.0
1  3.0  2.0  1.0
2  3.0  1.0  2.0
3  2.0  1.0  3.0
```

Chapter Concepts

Introduction to Pandas Data

Essential Functionality

Reading From Data Sources

Summarizing and Computing Descriptive Statistics

Handling Missing Data

Hierarchical Indexing

Chapter Summary

Data

- To use the tools in this course, we need data to work with
- Data can be used from a variety of sources:
  - Text format
    - CSV
    - JSON
    - XML/HTML
  - Binary formats
    - HDF5
    - Excel
  - Databases
    - MongoDB

Working with Text Formats

- We will introduce pandas during this course
  - It provides a number of features for working with data in a tabular format
    - Known as a `DataFrame`
      - We will use this in our examples here with details to follow
- Pandas provides the following functions for reading data:
  - `read_csv`
    - Load data from delimited file or URL, comma is default delimiter
  - `read_table`
    - Load delimited data from a file or URL, tab is default delimiter
  - `read_fwf`
    - Read fixed-width column formatted file or URL

Reading a Text File Source

- The following code loads a `csv` file

```
data = pd.read_csv('ch02/sample.csv')
print (data)

   1   2   3   4          hello
0  5   6   7   8          world
1  9  10  11  12  some message
```

Returns a `DataFrame`

- `read_table` would work for this file too

```
pd.read_table('ch02/sample.csv',sep=',')

   1   2   3   4          hello
0  5   6   7   8          world
1  9  10  11  12  some message
```

Delimeter to use

Reading Large Files

- It is possible to read large files in smaller fragments
  - Specify a `chunksize` to `read_csv`
    - Size is number of lines to supply

```
fragment = pd.read_csv('ch02/sample.csv', chunksize=1)

for line in fragment:
    print (line)
```

Read a line at a time

```
   1  2  3  4  hello
0  5  6  7  8  world
   1   2   3   4          hello
0  9  10  11  12  some message
```

Writing Data Out to Text Files

- Data can be exported to files in a delimited format

```
print (data)

    1   2   3   4          hello
0   5   6   7   8          world
1   9  10  11  12   some message

data.to_csv('ch02/file1.csv')
```

Write to file

- Can specify a separator too

```
data.to_csv('ch02/file1.csv', sep = '|')
```

Separator parameter

JSON Data

- Can read in JSON data using Python
  - Create `DataFrame` from data
- Consider the following JSON file:

```
{ "name":"jayne",
  "role":"sales",
   "customers" :
     [{"name":"Andersons","product":"Bosch","quantity":100},
      {"name":"ElectricalDirect","product":"Miele",
               "quantity":200}]}
```

```
import json
data = json.loads(open('ch02/example.json').read())
customers = DataFrame(data['customers'])
print (customers)
```

```
                name product  quantity
0          Andersons   Bosch       100
1  Electrical Direct   Miele       200
```

Read JSON file

SQL Data

- Can read from SQL databases using a standard recipe
  - Import the package for the particular database
  - Open a connection
  - Create a cursor
  - Execute a query
  - Iterate through the cursor or fetch the results to a list
  - Close the connection when done

```
import sqlite3
cn = sqlite3.connect('test.sqlite')
curs = cn.cursor()
curs.execute("create table names (id int, name varchar(20))")
curs.execute("insert into names values(1, 'Alice'), (2, 'Bob')")
cn.commit()
curs.execute("select * from names")
names = curs.fetchall()
print (names)
names2 = pd.read_sql_query("select * from names", cn)
print (names2)
cn.close()
```

```
[(1, 'Alice'), (2, 'Bob')]
```

	id	name
0	1	Alice
1	2	Bob

Other Formats

- Python has many libraries for reading and writing different formats/sources
  - HTML and XML
- We will not cover the details here, but at a high level data can be processed from:
  - HTML/XML sources
  - Binary formats
  - Microsoft Excel files
  - Web APIs (JSON)
  - Databases
    - Relational
    - MongoDB

Chapter Concepts

Introduction to Pandas Data

Essential Functionality

Reading From Data Sources

Summarizing and Computing Descriptive Statistics

Handling Missing Data

Hierarchical Indexing

Chapter Summary

Summarizing and Computing Descriptive Statistics

- Pandas objects have a set of common mathematical and statistical methods
- Most are reductions or summary statistics
  - Extract a single value, e.g., `sum()`
  - They exclude missing data

```
data = DataFrame([[1,np.nan],[3,4],[5,np.nan]],
                 columns=['a','b'])
print (data)

   a    b
0  1  NaN
1  3  4.0
2  5  NaN
```

Sums columns

```
print (data.sum())

a    9.0
b    4.0
dtype: float64
```

Sums rows

```
print (data.sum(axis=1))

0    1.0
1    7.0
2    5.0
dtype: float64
```

Pandas Mathematical Methods

- A few methods return multiple values, e.g., `describe()`

```
print (data)

   a    b
0  1  NaN
1  3  4.0
2  5  NaN
```

```
print (data.describe())

          a    b
count   3.0  1.0
mean    3.0  4.0
std     2.0  NaN
min     1.0  4.0
25%     2.0  NaN
50%     3.0  NaN
75%     4.0  NaN
max     5.0  4.0
```

Produces summary statistics

- Full list of DataFrame methods found at:
  - http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.html

Correlation and Covariance

- Correlation and covariance are computed from pairs of arguments
- Consider fetching data from Yahoo! Finance

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker) for ticker in
['AAPL', 'IBM', 'MSFT', 'GOOG']}
print (all_data['AAPL'])
```

Data returned

```
[2418 rows x 6 columns],
'AAPL':
Date        Open        High         Low         Close        Volume

2010-01-04  626.951088  629.511067  624.241073  626.751061   3927000
2010-01-05  627.181073  627.841071  621.541045  623.991055   6031900
....
Date         Adj Close
2010-01-04  313.062468
2010-01-05  311.683844
```

Percentage Price Change

- Consider calculating the percentage change in the daily price of the stocks

```
price = DataFrame({ticker:data['Adj Close'] for
        ticker, data in all_data.items()})

print (price)
```

```
Date          AAPL        GOOG         IBM          MSFT
2010-01-04   27.727039  313.062468  111.405000  25.555485
2010-01-05   27.774976  311.683844  110.059232  25.563741
2010-01-06   27.333178  303.826685  109.344283  25.406859
```

```
returns = price.pct_change()

print (returns.tail())
```

Daily price change

```
                AAPL      GOOG       IBM      MSFT
Date
2017-03-23 -0.003536 -0.014477  0.000229 -0.002460
2017-03-24 -0.001987 -0.003853 -0.005663  0.001696
2017-03-27  0.001707  0.006238 -0.000345  0.001847
```

Correlation and Covariance

- DataFrame's `corr()` and `cov()` methods return a correlation or covariance matrix as a `DataFrame`

```
print (returns.corr())
```

```
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.409814  0.382086  0.389641
GOOG  0.409814  1.000000  0.402671  0.471145
IBM   0.382086  0.402671  1.000000  0.495369
MSFT  0.389641  0.471145  0.495369  1.000000
```

```
print (returns.cov())
```

```
          AAPL      GOOG       IBM      MSFT
AAPL  0.000267  0.000104  0.000075  0.000092
GOOG  0.000104  0.000242  0.000075  0.000106
IBM   0.000075  0.000075  0.000143  0.000085
MSFT  0.000092  0.000106  0.000085  0.000208
```

Chapter Concepts

Introduction to Pandas Data

Essential Functionality

Reading From Data Sources

Summarizing and Computing Descriptive Statistics

Handling Missing Data

Hierarchical Indexing

Chapter Summary

Handling Missing Data

- Missing data is common in most data analysis applications
- Pandas tries to make working with missing data as painless as possible
  - The `NaN (NA)` value is used to represent missing data
- Two approaches to working with missing data:
  - Filter out missing data
  - Fill in missing data

Filtering Out Missing Data: `Series`

```
from numpy import nan as NA
data = Series([1,NA,2,3,4,NA])
print (data)
0    1.0
1    NaN
2    2.0
3    3.0
4    4.0
5    NaN
dtype: float64

print (data.dropna())
0    1.0
2    2.0
3    3.0
4    4.0
dtype: float64

data = data.dropna()
data.dropna(inplace=True)
```

Returns only non-null values and index values but does not modify the original data set

Reassign it back to the same variable to modify it or use `inplace=True`

Filtering Out Missing Data: `DataFrame`

- Useful parameters include:
  - `axis` which defaults to 0 for rows or 1 for columns
  - `how` which drops the row or column if `any` one value is NA or `all` are NA

```
data = DataFrame
([[1,2,3],[NA,5,NA],[NA,NA,NA],[10,11,12]])
```

```
      0     1     2
0   1.0   2.0   3.0
1   NaN   5.0   NaN
2   NaN   NaN   NaN
3  10.0  11.0  12.0
```

```
print (data.dropna(how='all'))
```

```
      0     1     2
0   1.0   2.0   3.0
1   NaN   5.0   NaN
3  10.0  11.0  12.0
```

```
print (data.dropna(how='all',axis=1))
```

```
      0     1     2
0   1.0   2.0   3.0
1   NaN   5.0   NaN
2   NaN   NaN   Nan
3  10.0  11.0  12.0
```

```
print (data.dropna(how='any'))
```

```
      0     1     2
0   1.0   2.0   3.0
3  10.0  11.0  12.0
```

```
print (data.dropna(how='any',axis=1))
```

```
Empty DataFrame
Columns: []
Index {0, 1, 2, 3]
```

Filling In Missing Data

```
print (data)
```

```
     0     1     2
0  1.0   2.0   3.0
1  NaN   5.0   NaN
2  NaN   NaN   NaN
3 10.0  11.0  12.0
```

```
filled = data.fillna(0)
print (filled)
```

```
     0     1     2
0  1.0   2.0   3.0
1  0.0   5.0   0.0
2  0.0   0.0   0.0
3 10.0  11.0  12.0
```

```
filled = data.fillna({0:10, 1:11, 2:12})
print (filled)
```

```
     0     1     2
0  1.0   2.0   3.0
1 10.0   5.0  12.0
2 10.0  11.0  12.0
3 10.0  11.0  12.0
```

Supply dictionary with column:value pair for different fill values per column

Chapter Concepts

Introduction to Pandas Data

Essential Functionality

Reading From Data Sources

Summarizing and Computing Descriptive Statistics

Handling Missing Data

Hierarchical Indexing

Chapter Summary

Hierarchical Indexing

- Allows multiple (at least two) index levels on an axis

Two-level index supplied

```
data = Series(np.random.randn(12),
        index=[['A','A','A','A','B','B','B','B','C','C','C','C'],
                              [1,2,3,4,1,2,3,4,1,2,3,4]])
```

```
print (data)

A  1    0.498512
   2    0.504765
   3    1.173850
   4   -1.336820
B  1   -1.262746
   2   -0.164154
   3   -1.105161
   4   -0.781278
C  1   -1.309080
   2   -1.078479
   3    0.515532
   4    0.052269
dtype: float64
```

```
print (data['A'])

1    0.498512
2    0.504765
3    1.173850
4   -1.336820
dtype: float64
```

```
print (data['A':'C'])

A  1    0.498512
   2    0.504765
   3    1.173850
   4   -1.336820
B  1   -1.262746
   2   -0.164154
   3   -1.105161
   4   -0.781278
C  1   -1.309080
   2   -1.078479
   3    0.515532
   4    0.052269
dtype: float64
```

Stacking and Unstacking

- Reshaping can be performed
  - `Series`: converts to `DataFrame` using `unstack()`
  - `DataFrame` to a `Series` using `stack()`

```
print (data)

A  1    0.498512
   2    0.504765
   3    1.173850
   4   -1.336820
B  1   -1.262746
   2   -0.164154
   3   -1.105161
   4   -0.781278
C  1   -1.309080
   2   -1.078479
   3    0.515532
   4    0.052269
dtype: float64
```

```
print (data.unstack())

          1         2         3         4
A  0.498512  0.504765  1.173850 -1.336820
B -1.262746 -0.164154 -1.105161 -0.781278
C -1.309080 -1.078479  0.515532  0.052269
```

`DataFrame` from hierarchical indexed `Series`

Stacking and Unstacking (continued)

```
data_frame = data.unstack()
print (data_frame)
```

```
          1         2         3         4
A  0.498512  0.504765  1.173850 -1.336820
B -1.262746 -0.164154 -1.105161 -0.781278
C -1.309080 -1.078479  0.515532  0.052269
```

Hierarchical indexed Series from DataFrame

```
print (data_frame.stack())
```

```
A  1    0.498512
   2    0.504765
   3    1.173850
   4   -1.336820
B  1   -1.262746
   2   -0.164154
   3   -1.105161
   4   -0.781278
C  1   -1.309080
   2   -1.078479
   3    0.515532
   4    0.052269
dtype: float64
```

Summary Statistics by Level

- Many statistics methods have a level option allowing you to specify the level to work with

```
print (data)

A  1    0.498512
   2    0.504765
   3    1.173850
   4   -1.336820
B  1   -1.262746
   2   -0.164154
   3   -1.105161
   4   -0.781278
C  1   -1.309080
   2   -1.078479
   3    0.515532
   4    0.052269
dtype: float64
```

Second index level for sum

```
print (data.sum(level=1))

1   -2.073314
2   -0.737869
3    0.584221
4   -2.065829
dtype: float64
```

Sum of all `1` second indexes

Chapter Concepts

Introduction to Pandas Data

Essential Functionality

Reading From Data Sources

Summarizing and Computing Descriptive Statistics

Handling Missing Data

Hierarchical Indexing

Chapter Summary

Chapter Summary

In this chapter, we have introduced:

- Pandas data structures
- Essential functionality of pandas
- Reading from data sources
- Summarizing and computing descriptive statistics
- Handling missing data
- Hierarchical indexing

ROITRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

CHAPTER 4: DATA PREPARATION

Chapter Objectives

In this chapter, we will introduce:

- A basic refresher on statistics
- Basic ETL (extract, transform, and load) and reshaping data for modeling
- Splitting data into training and testing sets
- Free-form text

Chapter Concepts

Statistics Primer

Basic ETL and Reshaping

Splitting Data

Free-Form Text

Chapter Summary

Statistics

- Statistics is a branch of mathematics that deals with collecting, organizing, analyzing, interpreting, and presenting data
- The math calculations behind it can be complex and time consuming to do, but there are some basic concepts that can be understood and applied without knowing all the details
- Data scientists need to understand a few basic big picture ideas in order to apply the computer models
  - Statistical features of a dataset (central tendency, min, max, IQR, deviation)
  - Probability (Normal, Uniform, Poisson, Binomial distributions)
  - Dimension reduction and sampling
  - Independent vs. Dependent variables and correlation
  - Accuracy Analysis

Statistical Features

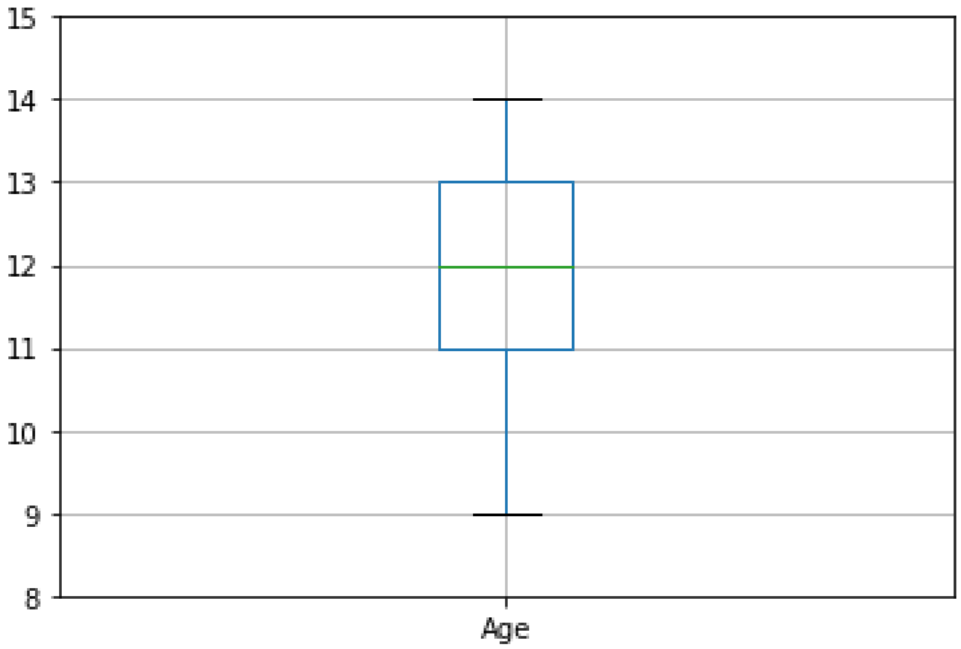
- Sometimes called *Descriptive Statistics* or *Exploratory Data Analysis*
- Gives us an overview of the range of values we find in a dataset
- Central Tendency is a measure of what usually happens and can be expressed with three different measures:
  - Mean or average is the sum of all the values divided by how many values there are
  - Median is the value in the middle of the set when all values are lined up in order
  - Mode is the single value that occurs most often in the set

```
import pandas as pd
df = pd.DataFrame([9,10,10,11,11,11,12,12,12,13,13,13,13,14],
columns=['Age'])
print ("Mean", df.Age.mean(), "Median", df.Age.median(), "Mode",
df.Age.mode()[0], "Count", df.Age.count())
print (df.Age.value_counts())
Mean 11.714285714285714 Median 12.0 Mode 13 Count 14
13    4
12    3
11    3
10    2
14    1
9     1
```

Box Plot

- Very often it is helpful to see these numbers plotted graphically instead

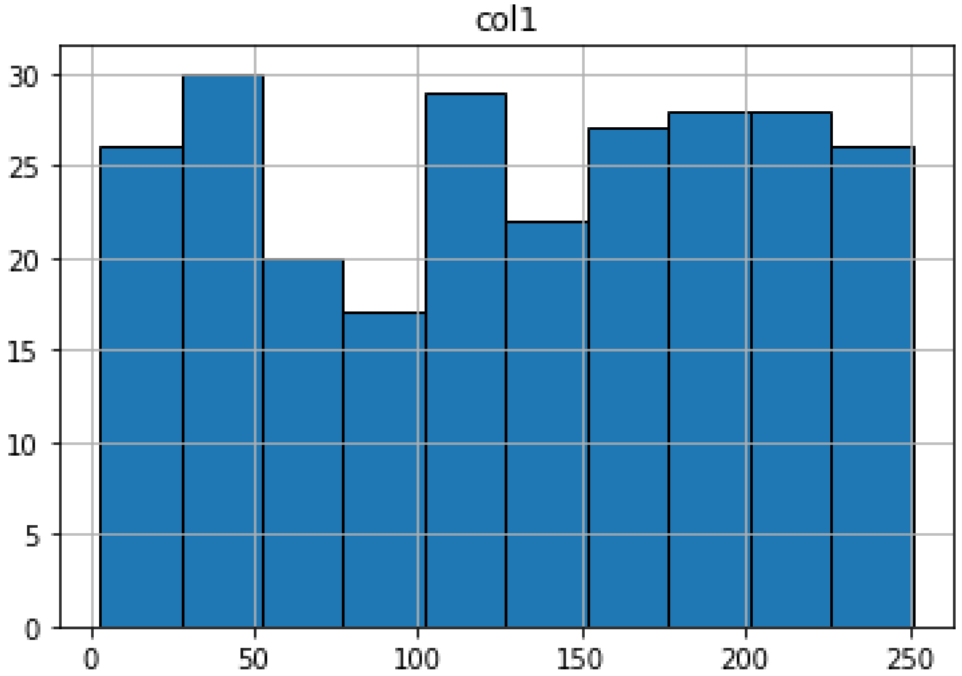
```
import matplotlib as mp
from matplotlib import pyplot as plt
plt.ylim(8,15)
df.boxplot()
```



Histogram

- Useful way to see each value range and how many items are in that range

```
import matplotlib as mp
import numpy as np
df = pd.DataFrame(np.random.rand(253, 1) * 254, columns=['col1'])
df.hist(histtype='bar', ec='black')
```

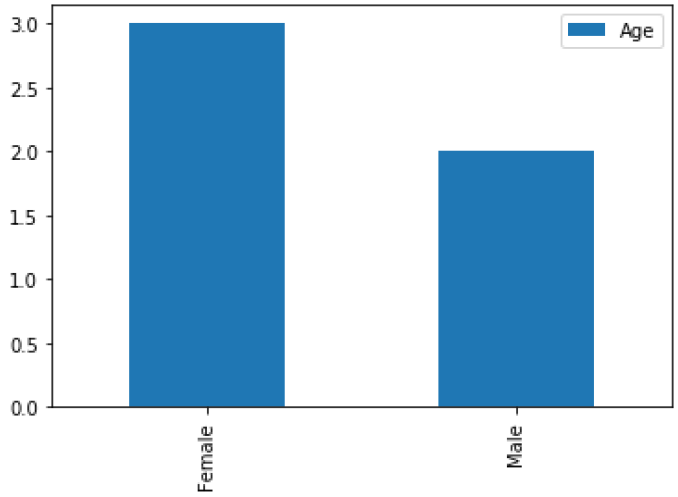


Bar Chart

- For categorical data, a bar chart is a good option

```
df = pd.DataFrame([('Male', 10),('Male', 11), ('Female', 11),
('Female', 12), ('Female', 12)], columns=['Gender','Age'])
x = df.groupby('Gender').count()
print (x)
x.plot(kind='bar')
```

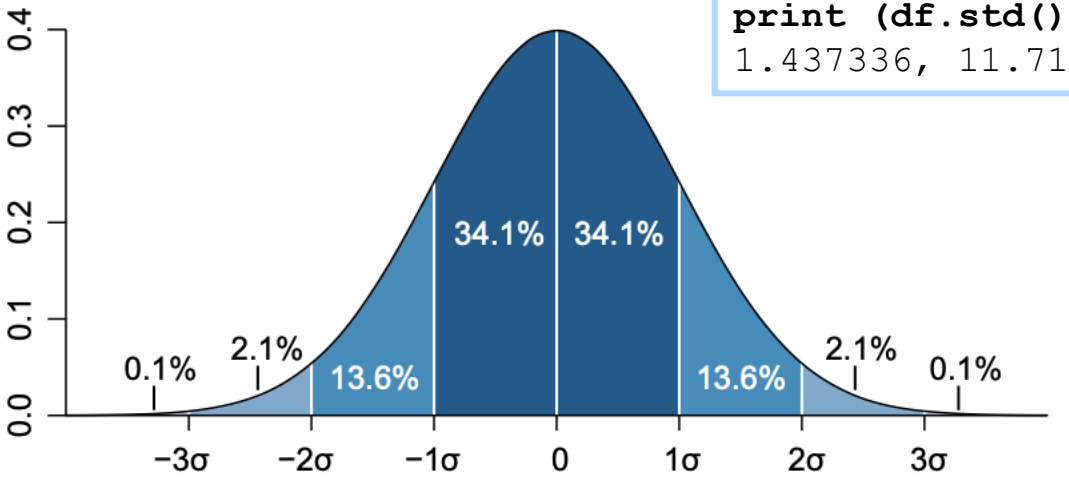
```
        Age
Gender
Female    3
Male      2

<matplotlib.axes._subplots.AxesSubplot at 0x7fed930f1358>
```



Standard Deviation

- Useful for describing how measurements are distributed in the data
- The average IQ is 100 and the standard deviation is 15 points
- A person with an IQ of 115 is one standard deviation above the mean
- A person with an IQ of 85 is one standard deviation below the mean
- A person with an IQ of 130 is two standard deviations above the mean
- By definition, 68% lie within 1 standard deviation, 95% within 2, and 99% within 3

```
print (df.std(), df.mean())
1.437336, 11.714286
```



Chapter Concepts

Statistics Primer

Basic ETL and Reshaping

Splitting Data

Free-Form Text

Chapter Summary

Fixing Up DataFrames

- `insert()` can add a new column to a DataFrame
- `drop()` removes a column from a DataFrame
- Categorical replaces strings with number placeholders
- `astype` converts the data type of a column

```
fatal.insert(11, 'Program', pd.Categorical(fatal['State or Federal
Program']).codes)

print (fatal[['Program', 'State or Federal Program']])

fatal['Number of Fatalities, 2012'].fillna(0).astype(int)

fatal.drop(['State or Federal Program'], axis=1, inplace=True)
```

Basic Data Types

- Most of the models you will use require data to be perfectly clean
  - Cannot have null values
  - Text sometimes needs to be replaced with numbers
  - Numbers need to be on the same scale
- There are some basic data types in machine learning
  - Numeric
    - Continuous – decimal numbers (weight or income)
    - Discreet – whole numbers (number of students)
    - Binned – numbers grouped together to form a category (18-25) (26-35) (36-45)
  - Categorical – text or numeric that represents a category (male/female)
  - Time series – sequence of numbers collected at regular interval (temperature measurement from a weather station)
  - Text – any free-form text needs to be converted to a numeric document term matrix first

Missing Values

- Nulls or missing values can mess up the calculation
  - Need to remove them or replace them
  - Usually replace them with the central tendency (mean, median, mode) or zero
  - NaN is used in Pandas to represent Not a Number
- `isnull()` will return a series of True/False indicating if a value is Null
- `fillna()` replaces nulls with another value

```
import pandas as pd
fatal = pd.read_csv('2012_Workplace_Fatalities_by_State.csv')
print (fatal.columns)

fatal.columns = ['State', 'Number of Fatalities', 'Rate of Fatalities', 'State
Rank', 'Number of Injuries', 'InjuriesRate', 'Penalties Avg', 'Penalties Rank',
'Inspectors', 'Years to Inspect Each Workplace Once', 'StateFederal']

print (fatal['Penalties Rank'].mean())
print (fatal['Penalties Rank'][48:])
print (fatal['Penalties Rank'][48:].isnull())
fatal['Penalties Rank'] = fatal['Penalties Rank'].fillna(fatal['Penalties
Rank'].mean())
print (fatal['Penalties Rank'][48:])
```

Rescaling

- Sometimes numbers in a dataset can be on a different scale in one column vs. another
  - Weight might be measured in pounds, heights in inches
  - Value range in different columns can be wildly different
- Rescaling them to a common scale can be helpful for understanding how to compare data in different scale
- Some algorithms require that all the numbers be on the same scale, others might not care but may perform better if they are rescaled first
- Large trial and error to see what works best

```
from sklearn import preprocessing as pp
x = fatal.NumberOfFatalities
print (x.mean(), x.std(), x.min(), x.max()) → 171 624 0 4628
pp.scale(x, with_mean = False, with_std = False) → [ 60. 218. 149.  88. 137.]
pp.scale(x, with_mean = True, with_std = False) → [-111  46  -22. -83 -34]
pp.scale(x, with_mean = False, with_std = True) → [0.09 0.35 0.24 0.14 0.22]
pp.scale(x, with_mean = True, with_std = True) → [-0.17  0.07 -0.03 -0.13  -0.05]
```

concat

- Often you need to combine two DataFrames into one
  - Read in separate files and append them together like `UNION ALL` in SQL
  - Combine columns from calculation to create a new DataFrame structure
- `concat` is a function that can combine two DataFrames together, either along the row or column axis works best

```
df1 = pd.DataFrame([('Male', 10),('Male', 11), ('Female', 11), ('Female', 12),
('Female', 12)], columns=['Gender','Age'])
df2 = pd.DataFrame([('Male', 20),('Male', 21), ('Female', 21), ('Female', 22)],
columns=['Gender','Age'])
df = pd.concat([df1, df2])
print (df)
df3 = pd.DataFrame([('John', 'Smith'), ('Joe','Average'), ('Jane', 'Doe'),
('Jill', 'Hill')], columns = ['First', 'Last'])
df = pd.concat([df1, df3], axis = 1)
print (df)
```

merge

- When you need a feature like a SQL Join to match two DataFrames on a common column value, use the `merge` command

```
person_data = { 'id': ['1', '2', '3', '4', '5'],
     'first_name': ['John', 'Sue', 'Jack', 'Alice', 'Joe'],
     'last_name': ['Smith', 'Miller', 'Sprat', 'Wonderland', 'Blow']}
df1 = pd.DataFrame(person_data, columns = ['id', 'first_name', 'last_name'])

skill_data = {'id' : ['1', '1', '2', '3', '3', '3', '5', '6'],
   'skill' : ['C++', 'Java', 'Java', 'C++', 'Java', 'Python', 'Python', 'Java']}
df2 = pd.DataFrame(skill_data, columns = ['id', 'skill'])
print (pd.merge(df1, df2, on = 'id'))
print (pd.merge(df1, df2, how = 'left'))
```

Recoding Categorical Data

- Sometimes you have a column of categorical data list Status that could have values of Active, Pending, Cancelled
- In some cases, you need to re-encode this data as a sequential number
  - Categorical function will do that

```
person_data = { 'id': ['1', '2', '3', '4', '5'],
  'first_name': ['John', 'Sue', 'Jack', 'Alice', 'Joe'],
  'status': ['Active', 'Active', 'Pending', 'Cancelled',
  'Cancelled']}
df1 = pd.DataFrame(person_data, columns = ['id', 'first_name',
     'status'])
print (df1)
df1.status = pd.Categorical(df1.status).codes
print (df1)
```

Dummy Coding

- Other models require the data be encoded as multiple columns with a 0 or 1 indicating which value it is
- Sometimes you skip the first column as a baseline and sometimes not
- Also referred to as One Hot Encoding

```
person_data = { 'id': ['1', '2', '3', '4', '5'],
  'first_name': ['John', 'Sue', 'Jack', 'Alice', 'Joe'],
  'status': ['Active', 'Active', 'Pending', 'Cancelled',
  'Cancelled']}
df1 = pd.DataFrame(person_data, columns = ['id', 'first_name', 'status'])
print (df1)
dummies = pd.get_dummies(df1.status, drop_first = True)
df2 = pd.concat([df1[['id','first_name']], dummies], axis = 1)
print (df2)
dummies = pd.get_dummies(df1.status, drop_first = False)
df3 = pd.concat([df1[['id','first_name']], dummies], axis = 1)
print (df3)
```

Dummy Coding Example

- In the first case, we encoded three values into two columns
  - Active becomes the baseline as indicated with zeros in Cancelled and Pending
  - This is usually what we need for regression analysis
- In the second case, we encoded the three values into three columns each with a 1 to indicate it is the value
  - Usually need to do this for Neural Networks
- Models like Naive Bayes and Decision Trees that don't use distance calculations don't usually need to be dummy coded

```
   id first_name     status
0   1       John     Active
1   2        Sue     Active
2   3       Jack    Pending
3   4      Alice  Cancelled
4   5        Joe  Cancelled
   id first_name  Cancelled  Pending
0   1       John          0        0
1   2        Sue          0        0
2   3       Jack          0        1
3   4      Alice          1        0
4   5        Joe          1        0
   id first_name  Active  Cancelled  Pending
0   1       John       1          0        0
1   2        Sue       1          0        0
2   3       Jack       0          0        1
3   4      Alice       0          1        0
4   5        Joe       0          1        0
```

Chapter Concepts

Statistics Primer

Basic ETL and Reshaping

Splitting Data

Free-Form Text

Chapter Summary

Splitting Data

- Supervised models require that they be trained with a set of data first
- After training, you need to test the results using another set of data which has the known values you are trying to predict
- By comparing the predicted values to the known values, you can determine how good a model is at predicting
- Run the same data through multiple different algorithms with different parameters to tweak the result until you find the combination that yields the best results
- Pandas and Scikit-learn offer many ways to split a dataset into a training and testing set
  - `sample`
  - `train_test_split`
- It is important to examine the two sets to make sure they are fairly representative of the whole set and not skewed in some way

Sample

- Sample is a method built into DataFrame objects
- The following recipe can create a random sample and then return the rest to another set

```
train = fatal.sample(frac=0.8,random_state=200)
test = fatal[~fatal.index.isin(train.index)]
x0 = fatal.ProgramType
x1 = train.ProgramType
x2 = test.ProgramTypeprint(x0.value_counts()/x0.count())
print (x1.value_counts()/x1.count())
print (x2.value_counts()/x2.count())
print (fatal.shape, train.shape, test.shape)

1      0.5
0      0.5

1      0.592
0      0.470

1      0.625
0      0.375

(42, 11) (34, 11) (8, 11)
```

train_test_split

- Convenient function to split the sets in one step
- The following recipe can create a random sample and then return the rest to another set

```
from sklearn.model_selection import train_test_split
train, test = train_test_split(fatal, test_size=0.2)
x0 = fatal.ProgramType
x1 = train.ProgramType
x2 = test.ProgramTypeprint(x0.value_counts()/x0.count())
print (x1.value_counts()/x1.count())
print (x2.value_counts()/x2.count())
print (fatal.shape, train.shape, test.shape)

1      0.5
0      0.5

1      0.515
0      0.484

1      0.555
0      0.444

(42, 11) (33, 11) (9, 11)
```

Chapter Concepts

Statistics Primer

Basic ETL and Reshaping

Splitting Data

Free-Form Text

Chapter Summary

Text Processing

- Raw unformatted text can come in many forms
  - Emails
  - Resumes
  - White Papers
  - Tweets
- Words don't process well mathematically, so you need to convert the words into a matrix of numbers that can be run through the algorithms
- Document Term Matrix (DTM) is a restructuring of text data that describes the frequency that words or terms occur in a collection of documents (often called a corpus)
- There are many steps involved in getting the data to this format
  - Split lines into words (tokenization)
  - Removing punctuation, numbers, etc.
  - Standardizing on upper/lower case
  - Removing trivial words (of, and, or, is, etc.) called stop words
  - Stemming versions of words (run, running, runs)

Text Processing (continued)

- The steps for doing this are formulaic and there are many recipes to get there
- The result is a big matrix which can be fed into any of the models just like regular data
- Common usages include:
  - Classify a document into a category (spam/not spam)
  - Determine the overall sentiment of the document
  - Plagiarism detection
  - Finding similar papers for research

Example DTM

```
import pandas as pd
from sklearn.feature_extraction.text
import CountVectorizer

def corpus_from_dir(folder):
   import os
   ret = dict(docs = [open(os.path.join(folder,f)).read() \
         for f in os.listdir(folder)], \
         ColNames = map(lambda x: x.split('.')[0], os.listdir(folder)))
   return ret

def tdm_df(docs, colNames = None, **kwargs):
   vectorizer = CountVectorizer(**kwargs)
   x1 = vectorizer.fit_transform(docs)
   df = pd.DataFrame(x1.toarray().transpose(), \
        index = vectorizer.get_feature_names())
   return df

corpus = corpus_from_dir('text')
print (corpus)
df = tdm_df(docs = corpus['docs'], colNames = corpus['ColNames'], \
     stop_words = 'english')
print (df)
```

Chapter Concepts

Statistics Primer

Basic ETL and Reshaping

Splitting Data

Free-Form Text

Chapter Summary

Next Steps

- ETL can be done at so many levels
  - At the source of the data using SQL
  - There are many ETL tools besides Python
- Explore the different ways to rescale and normalize data
- Text processing has so much more to it than just Document Term Matrix
  - Sentiment analysis
  - Word clouds
  - Term frequency-inverse document frequency
- Even binary data like images and sound can be turned into numeric data that can be run through models
  - Look for APIs to do things like image and facial recognition

Chapter Summary

In this chapter, we have introduced:

- A basic refresher on statistics
- Basic ETL (extract, transform, and load) and reshaping data for modeling
- Splitting data into training and testing sets
- Free-form text

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

CHAPTER 5: PLOTTING AND VISUALIZATION

Chapter Objectives

In this chapter, we will introduce:

- Matplotlib
- Plotting functions in pandas
- Python visualization tool ecosystem

Chapter Concepts

Introducing Matplotlib

Plotting Functions in Pandas

Python Visualization Tool Ecosystem

Chapter Summary

Introducing Matplotlib

- `matplotlib` is a plotting package designed for creating publication quality plots
  - Has a number of add-on toolkits
    - 3D plots
    - Mapping and projections
- `pyplot` is a module built on `matplotlib` usually imported as `plt`
- Run in `pylab` mode in IPython
- In this chapter, we provide enough detail to begin working with `matplotlib`
  - Full documentation including extensive examples can be found at:
    - http://matplotlib.org/2.0.0/index.html

Figures and Subplots

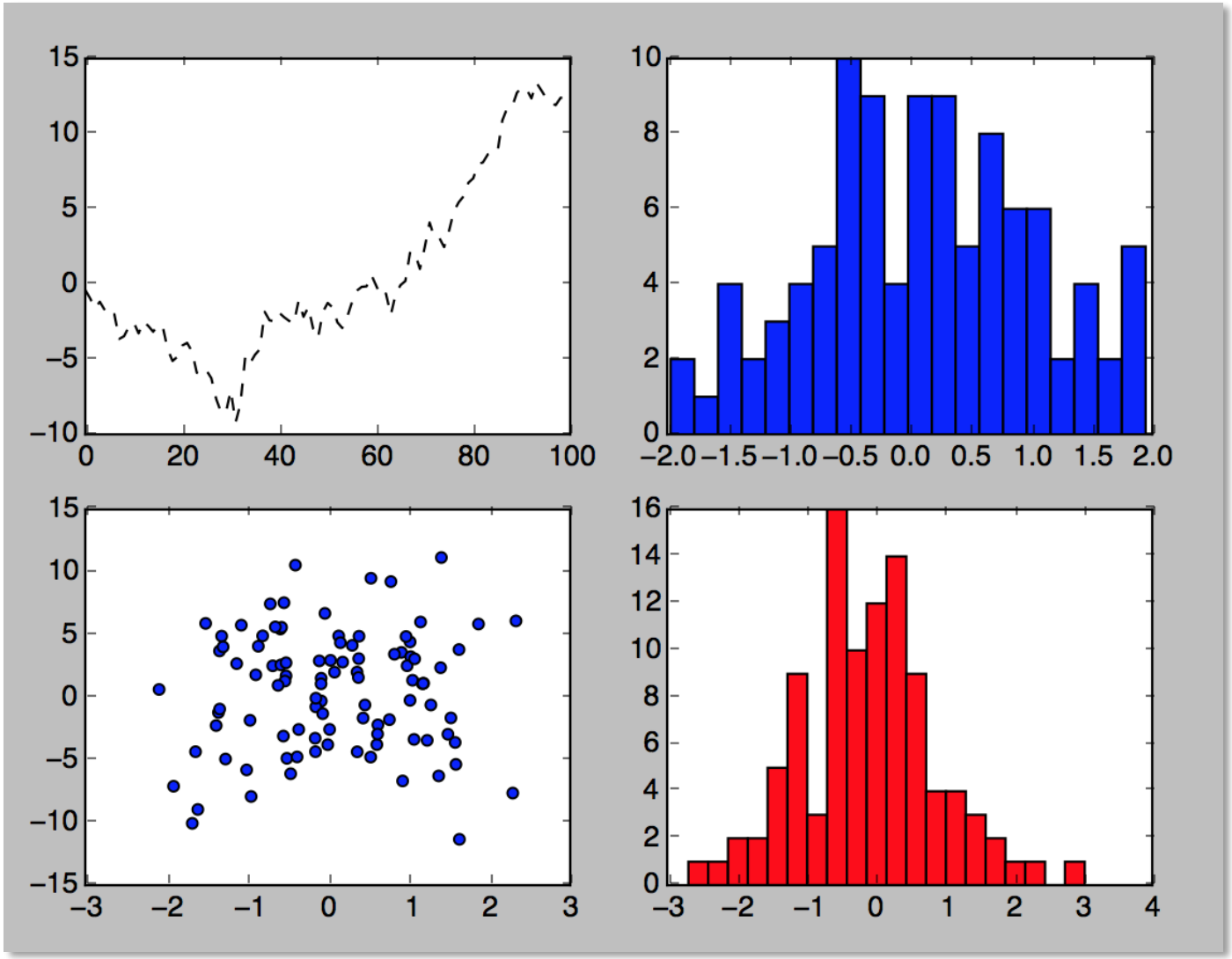
- Plots reside within a `Figure` object
- Subplots are added to a `Figure` object
  - Using `add_subplot(rows, columns, plot number)`
  - Returns `AxesSubplot` objects

```
import matplotlib
from matplotlib import pyplot as plt
figure = plt.figure()

sp1 = figure.add_subplot(2,2,1)
sp2 = figure.add_subplot(2,2,2)
sp3 = figure.add_subplot(2,2,3)
sp4 = figure.add_subplot(2,2,4)

sp1.plot(randn(100).cumsum(), 'k-')
sp2.hist(randn(100), bins=20)
sp3.scatter(randn(100), randn(100)-5*randn(100))
sp4.hist(randn(100), bins=20, color='r')
plt.show()
```

2x2 subplots, subplot 1

2x2 subplots, subplot 2

Figures and Subplots Example



Saving Plot

- Plots can be saved using the `savefig` method
- Various file formats are supported and can be listed with the following command:

```
plt.gcf().canvas.get_supported_filetypes_grouped()

{'Postscript': ['ps'], 'Encapsulated Postscript':
['eps'], 'Portable Document Format': ['pdf'], 'PGF code
for LaTeX': ['pgf'], 'Portable Network Graphics':
['png'], 'Raw RGBA bitmap': ['raw', 'rgba'], 'Scalable
Vector Graphics': ['svg', 'svgz'], 'Joint Photographic
Experts Group': ['jpeg', 'jpg'], 'Tagged Image File
Format': ['tif', 'tiff']}
```

- Using the extension indicates which format to save as

```
plt.savefig('chart1.jpg')
plt.savefig('chart1.pdf'
```

Colors and Styles

- The plot function accepts arrays of x and y coordinates and also an optional string
  - Optional string is for color and style
    - E.g., `sp1.plot(x,y,'r—')`
      - `r` indicates red color and `--` is the dashed style
- More explicit requests for color and style can be made
  - E.g., `sp1.plot(x,y,linestyle='—', color='r')`
- Plots will have continuous line plots and, therefore, will have data interpolated
  - Can request data points to be shown
    - E.g., `sp1.plot(x,y,'ro—')`
    - Or `sp1.plot(x,y,linestyle='—', color='r', marker='o')`

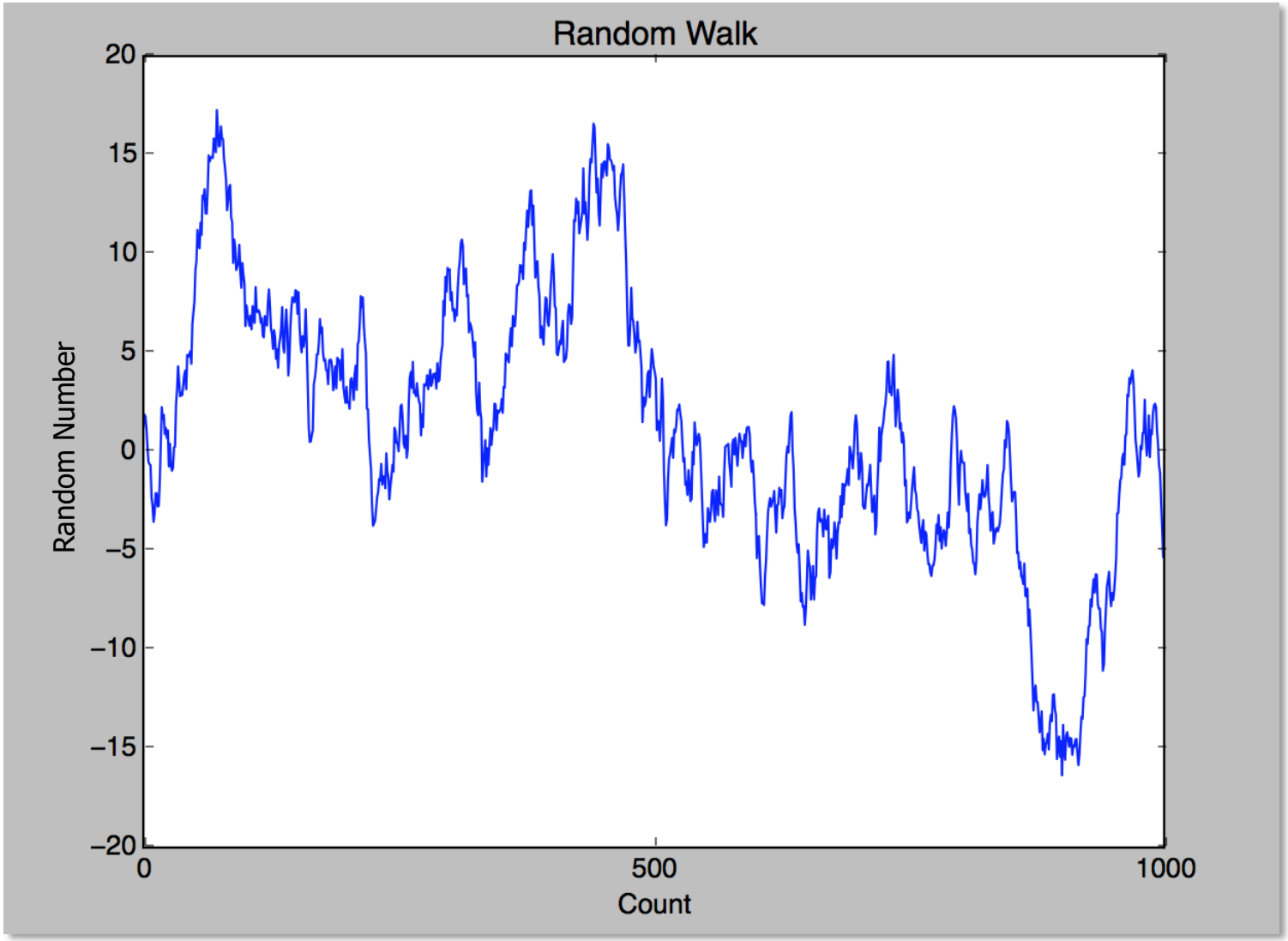
Labels and Legends

- Following example shows how to change axis ticks, labels, and add a title

```
figure = plt.figure()
p1 = figure.add_subplot(1,1,1)

p1.plot(randn(1000).cumsum())
p1.set_title('Random Walk')

p1.set_xticks([0,500,1000])

p1.set_xlabel('Count')

p1.set_ylabel('Random Number')
```

`set_yticks` for Y axis

Labels and Legends Example



Chapter Concepts

Introducing Matplotlib

Plotting Functions in Pandas

Python Visualization Tool Ecosystem

Chapter Summary

Plotting Functions in Pandas

- Pandas objects have built-in plotting functions
  - Simplify working with Matplotlib
    - In particular, for DataFrame objects
- Provide support for a number of different chart types such as:
  - Line plots
  - Bar plots
  - Histograms
  - Density plots
  - Scatter plots
  - Etc.

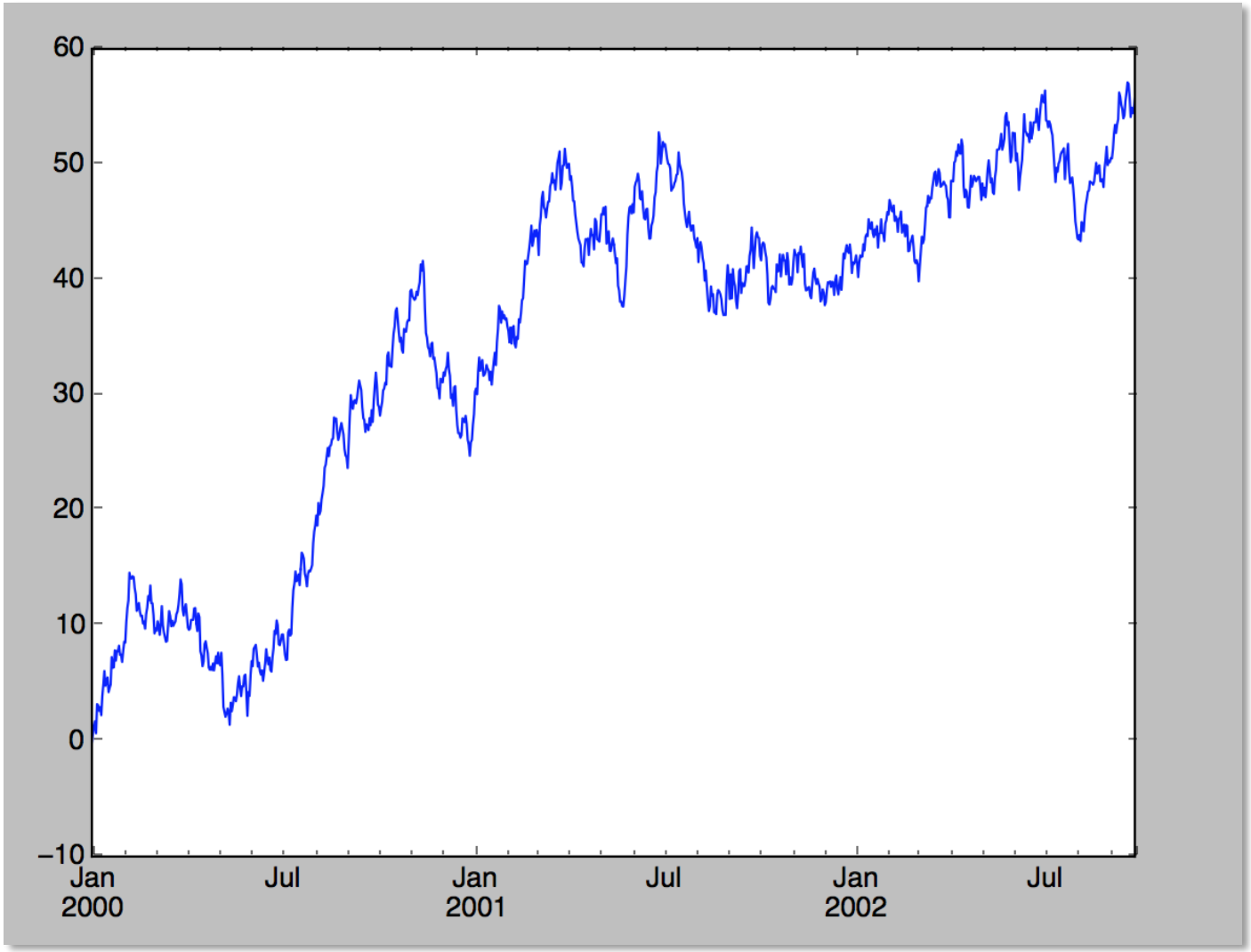
A Simple Example

- Consider plotting the function `f(x)=cos(x)`

```
pylab

import numpy as np

import pandas as pd

from pandas import Series, DataFrame

ts = Series(np.random.randn(1000), \
       index=pd.date_range('1/1/2000', periods=1000))

ts.cumsum()

ts.plot()
```

Built-in function

A Simple Example (continued)



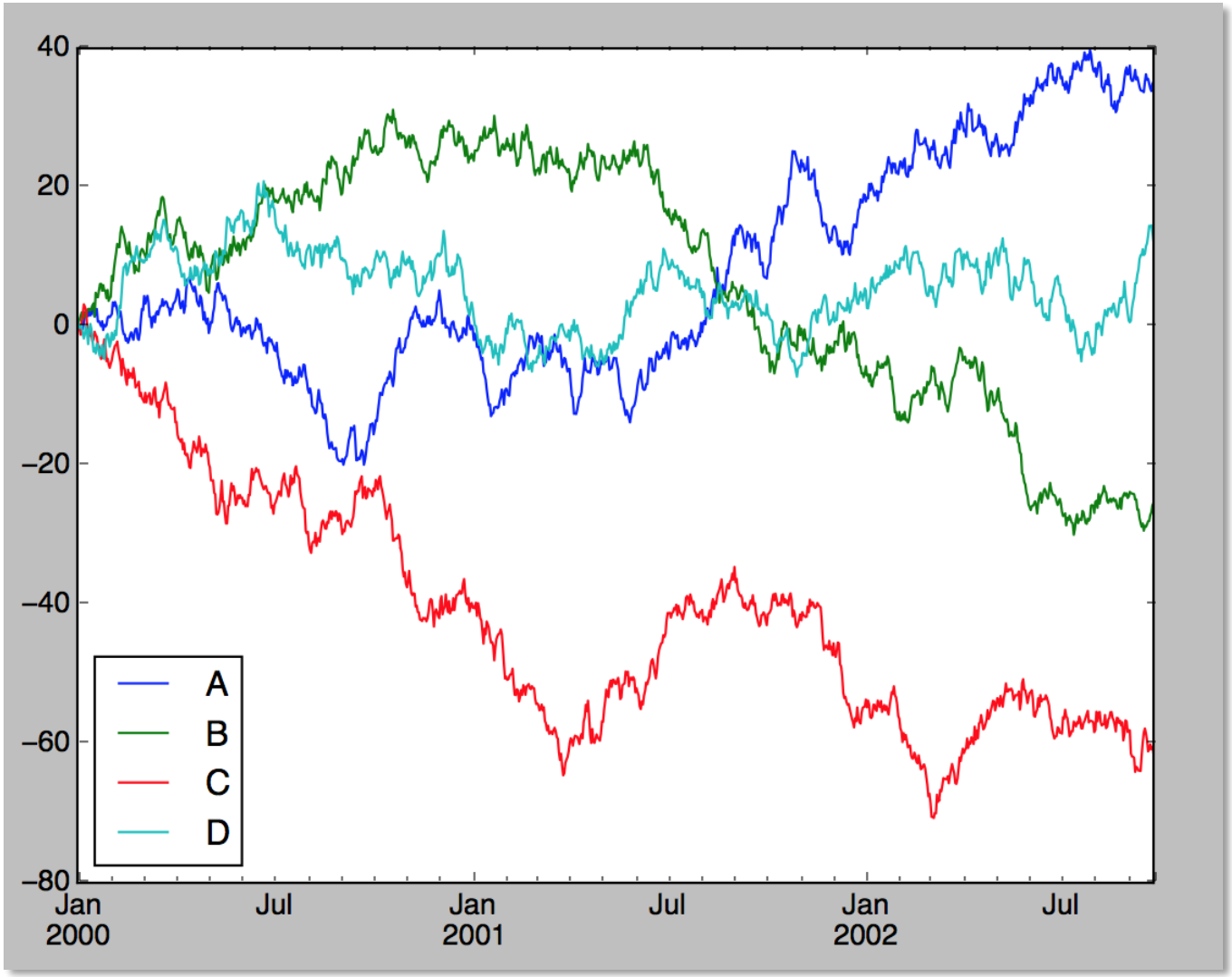
Line Plot with `DataFrame`

- `DataFrame's` plot method plots each of its columns as a different line
  - On the same plot
  - A legend is created automatically

```
ts = Series(np.random.randn(1000), \
           index=pd.date_range('1/1/2000', periods=1000))

df = DataFrame(np.random.randn(1000,4), \
           index= ts.index, columns=list('ABCD'))

df = df.cumsum()

df.plot()
```

Used in legend

Line Plot with `DataFrame` (continued)



`Series plot()` Arguments

- Series plots can be customized using arguments to `plot()`
  - `label`
    - Label for plot legend
  - `style`
    - String such as `'g–'` for Matplotlib
  - `alpha`
    - Fill opacity from 0 to 1
  - `kind`
    - Line, bar, barh, kde
  - `grid`
    - Display axis grid
  - `logy`
    - Use logarithmic scaling on the Y axis
- For full list, see:
  - http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.plot.html

`DataFrame plot()` Arguments

- Series plots can be customized using arguments to `plot()`
  - `subplots`
    - Plot each DataFrame in separate subplot
  - `sharex`
    - Share same x axis for subplots
  - `sharey`
    - Share same y axis for subplots
  - `figsize`
    - Size of figure to create
  - `title`
    - Plot title as a string
  - `legend`
    - Add a subplot legend
  - `sort_columns`
    - Plot columns in alphabetical order using existing column order
- For full list, see:
  - http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html

Histogram Example

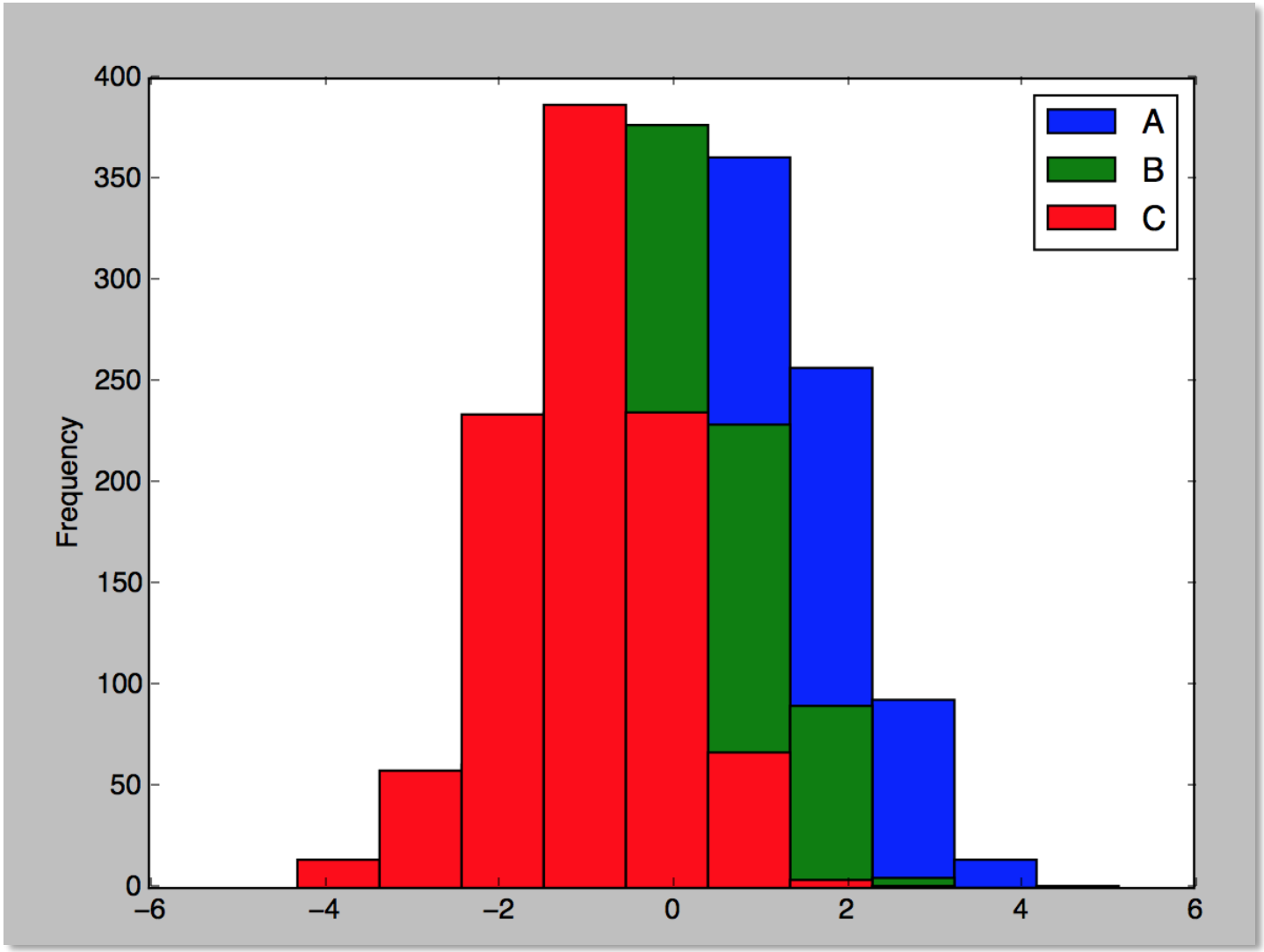
```
df = pd.DataFrame({'A': np.random.randn(1000) + 1, \
        'B': np.random.randn(1000), \
        'C': np.random.randn(1000)-1})

print (df.head())
```

```
          A         B         C
0  0.627152  1.984009  0.785683
1  1.316856  0.318605  0.143795
2 -0.763011 -0.261403 -1.346760
3  1.174517  1.044114  0.556043
4  1.052025 -0.021766 -1.868798
```

```
df.plot(kind='hist')
```

Select histogram

Histogram Example (continued)



Scatter Plots

- Useful way of visualizing relationship between two one-dimensional data series
- `matplotlib` and `pyplot` have a scatter method for plotting charts

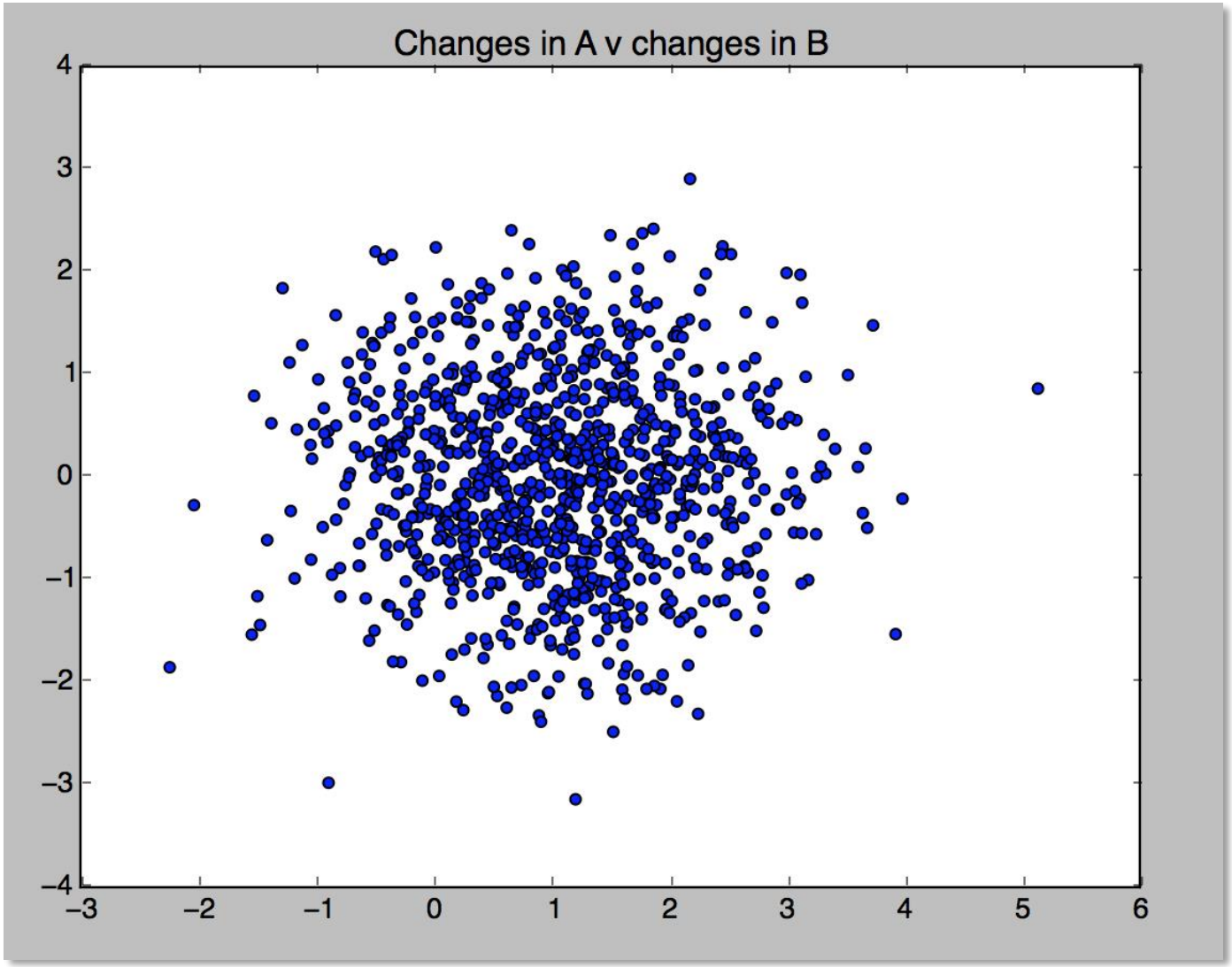
```
df.head()
```

```
          A         B         C
0  0.627152  1.984009  0.785683
1  1.316856  0.318605  0.143795
2 -0.763011 -0.261403 -1.346760
3  1.174517  1.044114  0.556043
4  1.052025 -0.021766 -1.868798
```

Scatter plot

```
plt.scatter(df['A'], df['B'])

plt.title('Changes in A v changes in B')
```

Scatter Plots (continued)



Scatter Plot Matrix

- For exploratory data analysis, it may be helpful to look at all scatter plots amongst a group of variables
  - Known as a pair plot or scatter plot matrix
- Pandas has `scatter_matrix` function
  - Works with a `DataFrame`
  - Also supports placing histograms or density plots of each variable along the diagonal

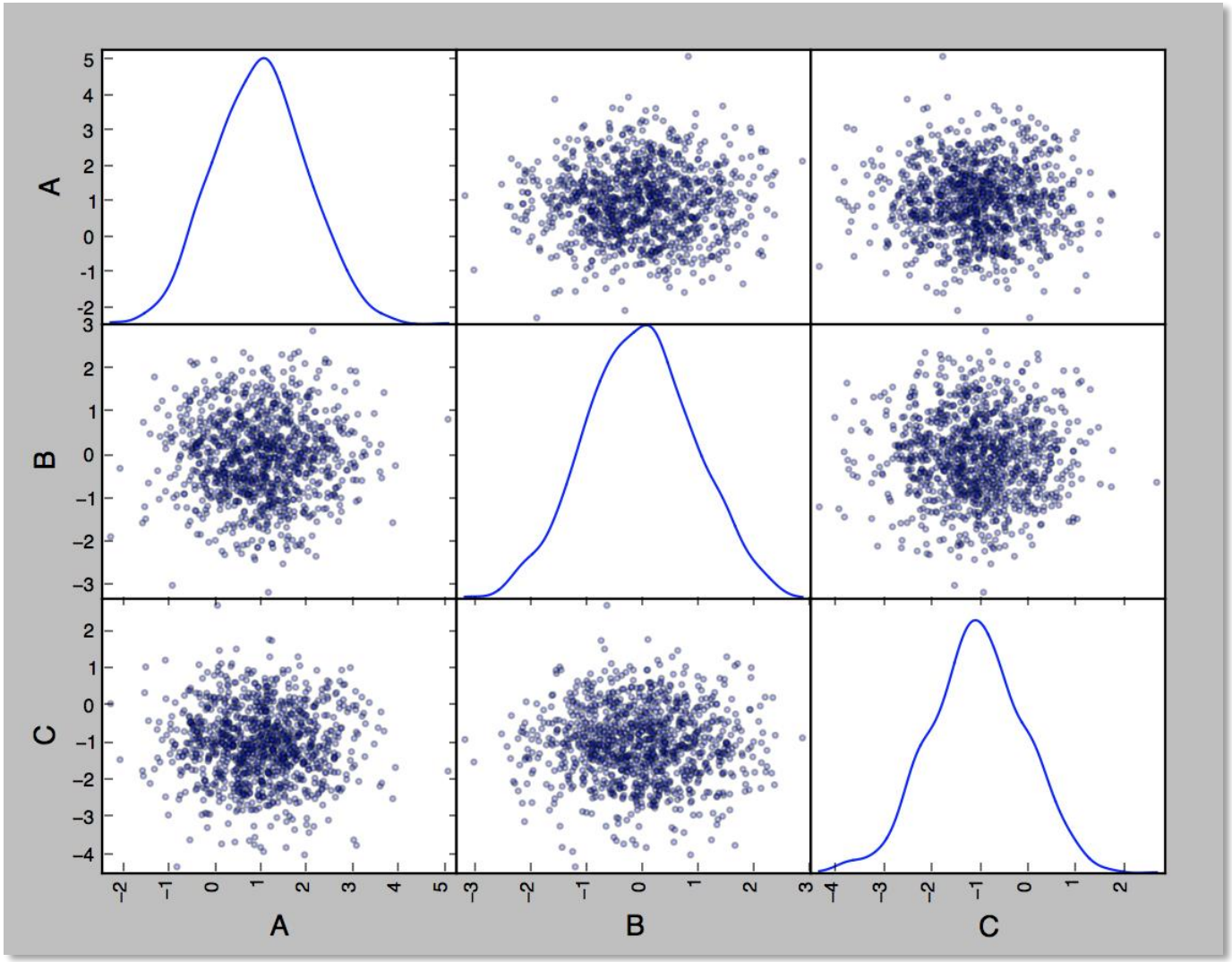
```
df.head()
```

```
          A         B         C
0  0.627152  1.984009  0.785683
1  1.316856  0.318605  0.143795
2 -0.763011 -0.261403 -1.346760
3  1.174517  1.044114  0.556043
4  1.052025 -0.021766 -1.868798
```

```
pd.scatter_matrix(df, diagonal='kde',alpha=0.3)
```

Density plot on diagonal

Scatter Plot Matrix Example



Chapter Concepts

Introducing Matplotlib

Plotting Functions in Pandas

Python Visualization Tool Ecosystem

Chapter Summary

Chapter Concepts

Introducing Matplotlib

Plotting Functions in Pandas

Python Visualization Tool Ecosystem

Chapter Summary

Chapter Summary

In this chapter, we have introduced:

- Matplotlib
- Plotting functions in pandas
- Python visualization tool ecosystem

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

CHAPTER 6: CLUSTER ANALYSIS

Chapter Objectives

In this chapter, we will:

- Explore Cluster Analysis
- Compare two algorithms
  - K-Means
  - Hierarchical

Chapter Concepts

Cluster Analysis

Algorithms

Chapter Summary

Cluster Analysis

- Analysis tool to help make sense of the data before feeding it into other models
- Unsupervised
  - More about discovering patterns in data
  - Not about predicting values for unknown values
- Looks for natural groupings among the data
  - Voter groups (is it just left vs. right, or left, right, center, or more)
  - Species identification (are two groups of organisms different enough to be considered a different species or not)
  - Identify different types of customers we may have
- Often helpful as a preparatory step before classification to determine how many categories we may want to predict

Types of Cluster Analysis

- There are two main approaches to solve this
  - Top down (K-Means)
  - Bottom up (Hierarchical clustering)
- Both rely on the notion of similarity
  - Objects are similar if they share common attributes to others
  - The more similar they are, the closer they are to one another
  - If something is far away in similarity to one thing, it may be closer to something else
- Ultimately the goal is to take a large sample of data and break it up into a small number of meaningful groupings that shed insight as to what the data means

Dataset

- For these examples let's generate some random datasets just because it's easier to analyze

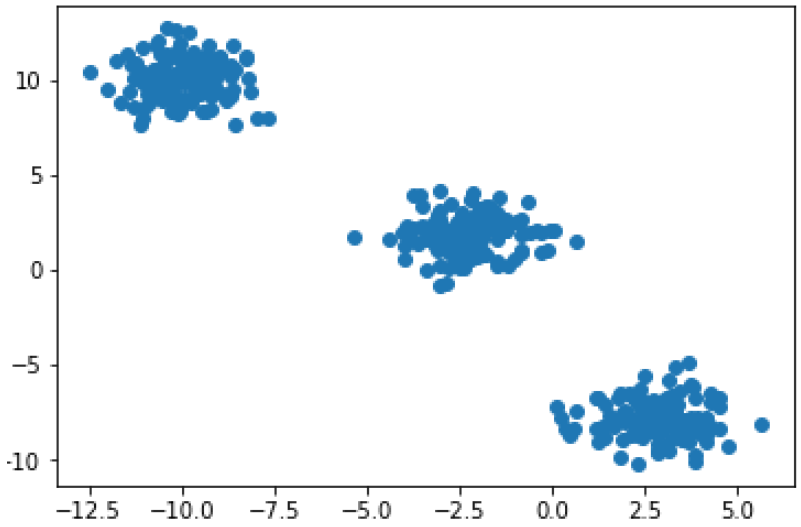
```
import numpy as np
from sklearn.cluster
import Kmeans
from sklearn.datasets import make_blobs
# Creating a sample dataset with 4 clusters
x, y = make_blobs(n_samples=400, n_features=2, centers=3)
print (x[:5]) # shape location
print (y[:5]) # cluster member
```

```
[[-6.10513999 -3.58316594] [-7.6168443   5.40841142] [-
2.06235753 -3.92038777] [-1.8104498  -4.1218467 ] [-5.32915489
-6.17092626]]

[2 1 0 0 2]
```

Visualize the Data

- It is often helpful to visualize the data by plotting it
  - There are only two features in this set so it's easy to plot
  - You can also plot a 3D graph for three features
  - Beyond that, it's hard to visualize more features

```
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (16, 9)
plt.plot(x[:,0],x[:,1],'o')
plt.show()
```



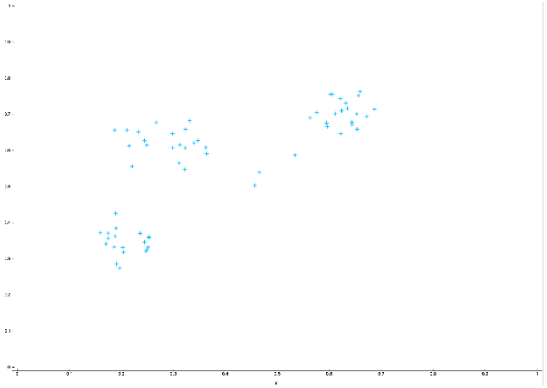
Chapter Concepts

Cluster Analysis

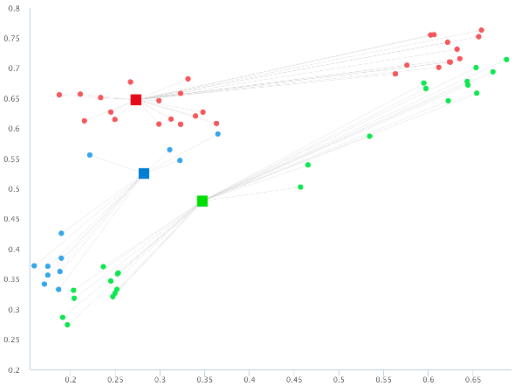
Algorithms

Chapter Summary

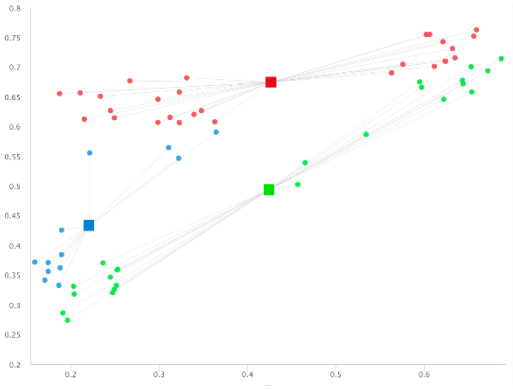
K-Means in Actions



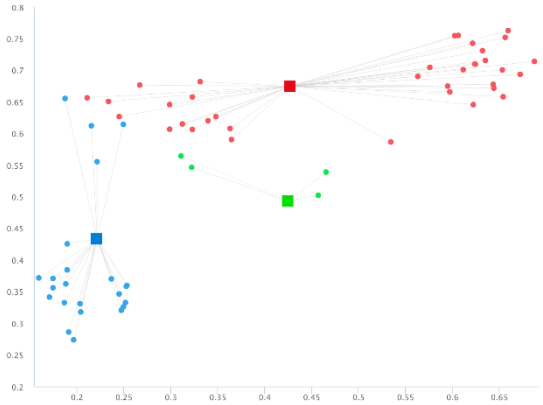
Random Data



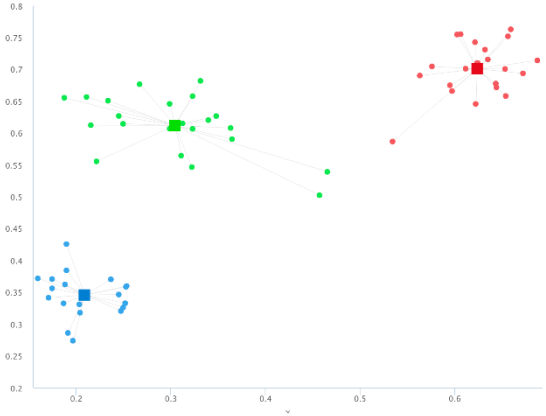
Random Centroids



Adjust Centroids



Reassign Membership

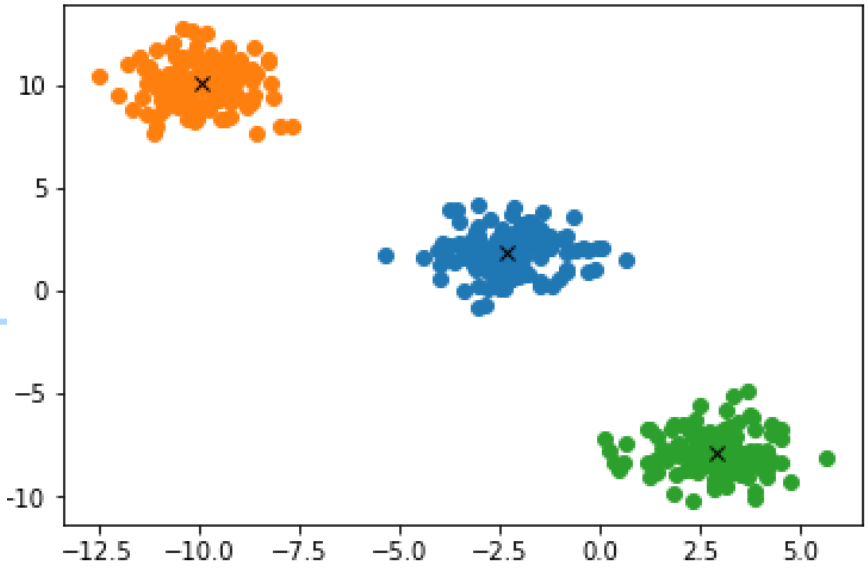


Keep Doing Until Stops Changing

Run K-Means

- Just eyeballing it, let's try out three clusters and plot the results

```
from sklearn import cluster
CLUSTERS = 3
k_means = cluster.KMeans(n_clusters=CLUSTERS)
k_means.fit(x)
labels = k_means.labels_
centroids = k_means.cluster_centers_
for i in range(CLUSTERS):
    ds = x[np.where(labels==i)]
    plt.plot(ds[:,0],ds[:,1],'o')
    lines = plt.plot(centroids[i,0]
            centroids[i,1],'kx')
plt.show()
```



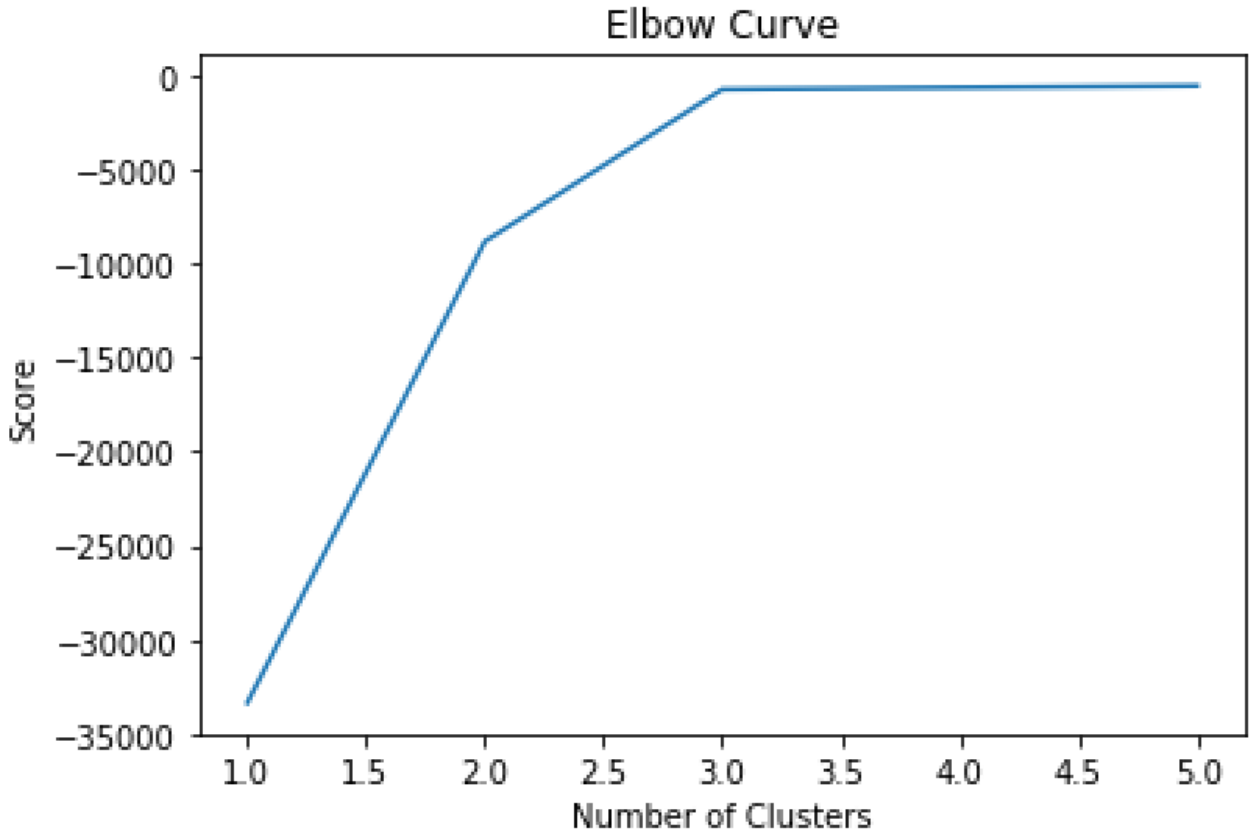
Elbow Chart

- Here the results are very clear cut, but sometimes the data overlap and don't fit nicely into a particular cluster
- It is often helpful to run a chart that helps figure out how many clusters is ideal
  - Too few and the items are too dissimilar
  - Too many and the additional distinctions become trivial
    - Is there much difference between a brown poodle and a chocolate poodle?

```
def plot_elbow(data, cluster_cnt = 6):
    CLUSTERS = range(1, cluster_cnt)
    kmeans = [cluster.KMeans(n_clusters=i) for i in CLUSTERS]
    score = [kmeans[i].fit(data).score(data) \
             for i in range(len(kmeans))]
    plt.plot(CLUSTERS ,score)
    plt.xlabel('Number of Clusters')
    plt.ylabel('Score')
    plt.title('Elbow Curve')
    plt.show()
plot_elbow(x)
```

Elbow Chart (continued)

- In the chart, we can see there is a bend between two to four clusters
- Three feels like the right number to start with in this case

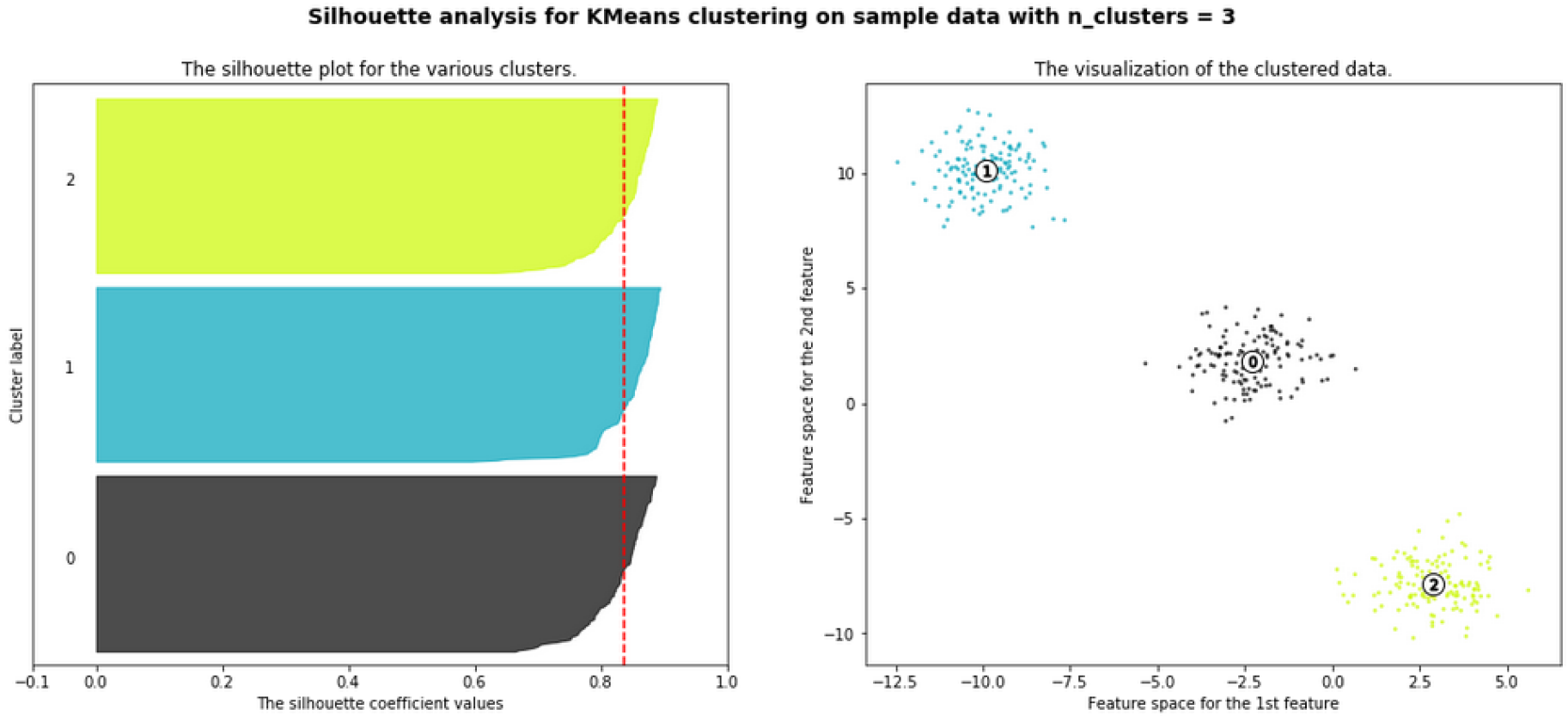


Silhouette Charts

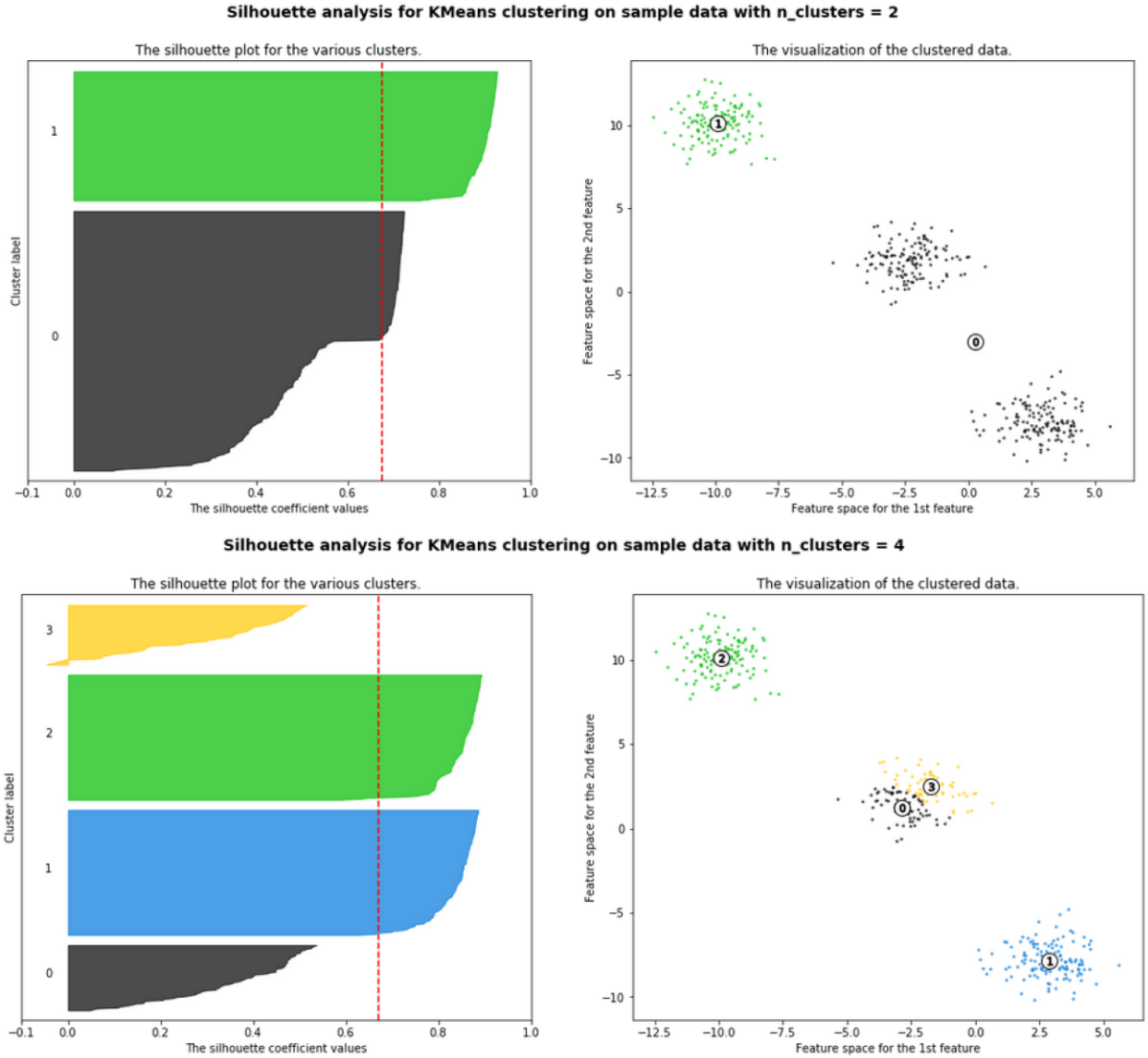
- Once you have figured out approximately how many clusters you have, you should run the analysis a few times with different cluster numbers
- A silhouette chart helps to visualize how well the clusters are at grouping similar items together
- Higher silhouette score (i.e., closer to 1) means in general the cluster does a good job at grouping similar items together
- Graphing how similar each item is to its neighbors helps to visualize how good the cluster is also
- Ideally, you want to settle upon a number of clusters that has a good mix of:
  - A high silhouette value
  - Few members that are far off from the average silhouette value
  - A number of clusters that are reasonably similar in size
  - A number of clusters that makes business sense of what you're trying to describe

Silhouette Charts (continued)

```
For n_clusters = 2 The average silhouette_score is : 0.6756049213871368
For n_clusters = 3 The average silhouette_score is : 0.8378250424949772
For n_clusters = 4 The average silhouette_score is : 0.6699001879846088
For n_clusters = 5 The average silhouette_score is : 0.5071441264659202
For n_clusters = 6 The average silhouette_score is : 0.3347353201539845
```



Silhouette Charts (continued)

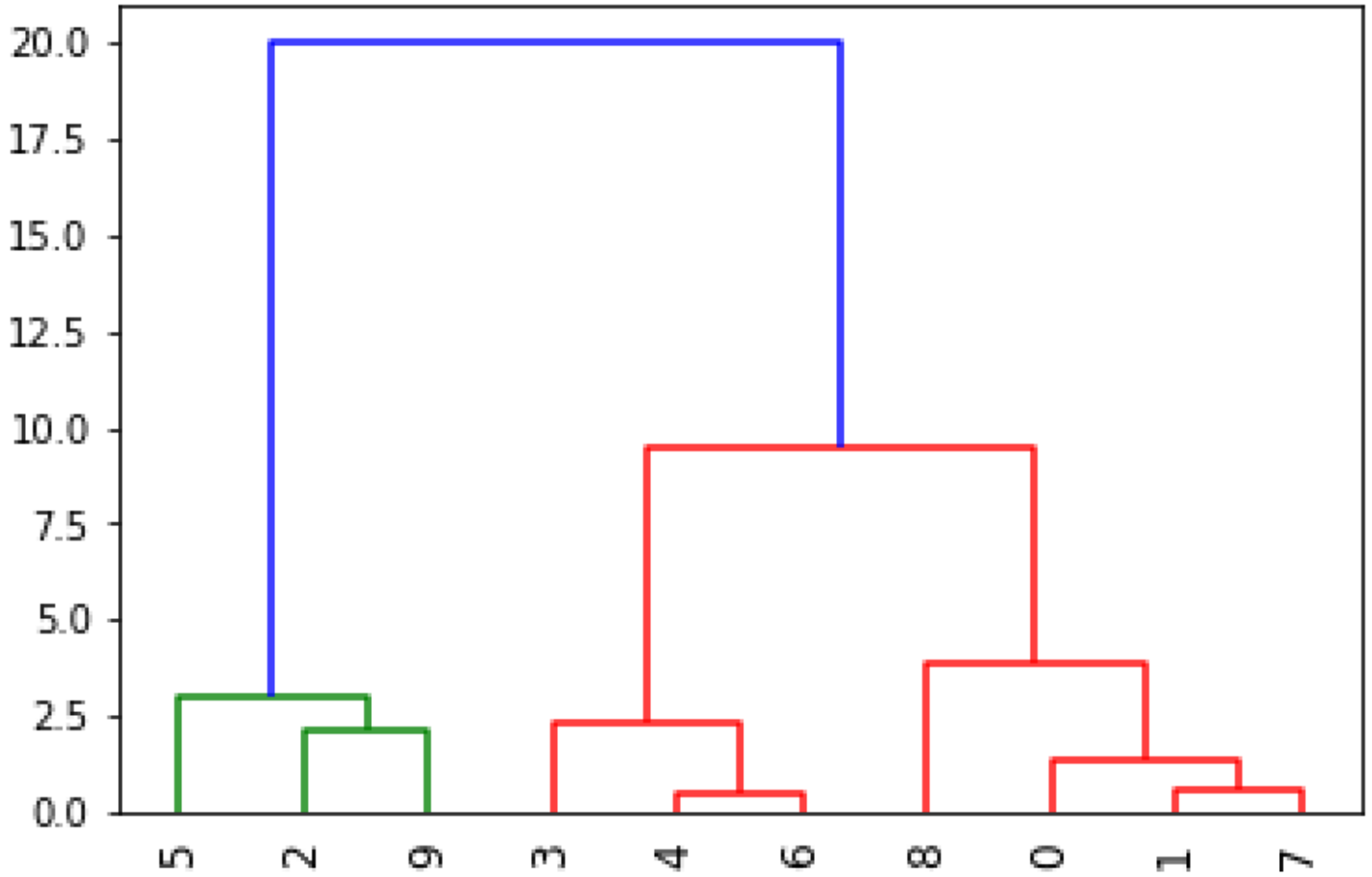


Hierarchical Clustering

- Often called bottom-up
- Finds two clusters closest to one another and merges them and keeps doing it until there is one big cluster
  - Uses distance of the features to determine closeness
- Creates a graph called a dendrogram which helps visualize the clusters and how similar they are
- Usually a good first step to take before K-Means to get a feel for how many clusters you should start with

```
x, y = make_blobs(n_samples=10, n_features=2, centers=3)
print (x)
print (y)
from scipy.cluster.hierarchy import dendrogram, linkage
z = linkage(x, 'ward')
dendrogram(z, leaf_rotation = 90, leaf_font_size=12)
```

Hierarchical Clustering (continued)



Chapter Concepts

Cluster Analysis

Algorithms

Chapter Summary

Next Steps

- The unsupervised model of clustering doesn't make predictions so much as it helps understand the data
- Another unsupervised model to explore is association rules
  - Used to describe patterns like "people who like X also like Y"

Chapter Summary

In this chapter, we have:

- Explored Cluster Analysis
- Compared two algorithms
  - K-Means
  - Hierarchical

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

CHAPTER 7: CLASSIFICATION MODELS

Chapter Objectives

In this chapter, we will:

- Understand the use cases for Classification models
- Discuss and compare various algorithms
  - Naive Bayes
  - Decision Tree
  - Logistic Regression
  - Neural Network

Chapter Concepts

Classification Model

Algorithms

Chapter Summary

Classification

- There are many business use cases for wanting to predict whether a record will fall into one category or another
  - Is a credit card swipe fraudulent or not?
  - Does a patient have a disease?
  - Will an applicant be a profitable customer?
- Classification can make such predictions by using historical data to train the model to look for patterns
- To see how good a job the model does, you test it with another set of data that was not used to train the model
  - By comparing the known values to the predicted ones, you can judge how well the model performs
- If the model guesses better than a coin flip or random guess, it may not be perfect, but it's better than nothing and could be used to create actionable business decisions with a best guess
- There are many different algorithms that can do classification, so it's best to run the same data through many different models to see which works best for your data

Steps to Classification

- There are some consistent steps you need to do regardless of which algorithm you use, although some models have different requirements
- Usually categorical data needs to be re-encoded as either a series of numbers or as dummy encoded data
  - There's really no consistency among different algorithms
  - Learn from experience and examples which algorithms require which style of data
- All models should take the data set and split it into a training and testing set
- First, you fit the model with the training set
  - Can take some time for large datasets
- Once the model is trained, you can see how good it is at making predictions by using a predict function and comparing those values to known values for the variable you are trying to predict
- After you've picked the model that does the best job, you can use it to predict values either individually or in a batch for new data as it comes in

Notes on Classification

- Often you are trying to predict an either/or value
  - Is a card swipe fraudulent or legitimate?
  - Does a patient have cancer or not?
- Not limited to just two choices, you could predict whether a record falls into a category with many different values
  - It just gets trickier sometimes to interpret the results
- The math and techniques behind the scenes can get complicated, but you don't really need to know any of it to use the algorithms
- Just knowing how to identify that you need a classification model and how to prep the data and interpret the results is often good enough to get started
- As you get more sophisticated, you can learn the math behind the scenes to tweak the results and try to get better results

Applying Classification

- Let's look at some data first, and use this with several different algorithms

```
import pandas as pd
df = pd.read_csv('CreditCardFraud.csv')
print (df.shape, df.columns)
print (df.isFraud.value_counts())
print (df.type.value_counts())

(6362620, 11) Index(['step', 'type', 'amount', 'nameOrig',
'oldbalanceOrg', 'newbalanceOrig',       'nameDest', 'oldbalanceDest',
'newbalanceDest', 'isFraud',       'isFlaggedFraud'],
dtype='object')

0    63544071       8213
Name: isFraud, dtype: int64
CASH_OUT    2237500
PAYMENT     2151495
CASH_IN     1399284
TRANSFER     532909
DEBIT         41432
Name: type, dtype: int64
```

Change Categorical Column

- Change the type categorical column to codes and keep certain columns

```
columns = ['type', 'amount', 'oldbalanceOrg', 'newbalanceOrig',
           'oldbalanceDest', 'newbalanceDest', 'isFlaggedFraud',
           'isFraud']
df = df[columns]
df.type = pd.Categorical(df.type).codes
```

- Split the data into training and testing sets
  - Check the ratios of the two sets are about the same

```
from sklearn.model_selection import train_test_split
from sklearn import preprocessing as pp
trainX, testX, trainY, testY = train_test_split \
        (df[df.columns[:-1]], df.isFraud, \
        train_size = train_size, test_size = test_size)
print (testY.value_counts())
print (trainY.value_counts()/trainY.count())
print (testY.value_counts()/testY.count())
print (trainX[:10])
```

Chapter Concepts

Classification Model

Algorithms

Chapter Summary

Naive Bayes

- Based on Bayes theorem and involves calculating lots of probabilities of events occurring based on prior observations of conditions related to that event
- Pros
  - It's easy and fast to train often outperforming more complex algorithms
  - Performs well with categorical data
  - Works well for multi-class predictions
- Cons
  - Doesn't do well if a category is found in the test set that is missing from the training set
  - Not useful for tweaking the false negatives (FN)/false positives (FP)

Apply Naive Bayes

- Load the module, create a model and train it

```
from sklearn.naive_bayes import GaussianNB
modelNB = GaussianNB()
modelNB.fit(trainX, trainY)
```

- Run a prediction on the reserved testing set and compare the predicted values to the known values

```
predY = modelNB.predict(testX)
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(testY, predY)
print (cm)
def cm_percent(cm, length):
    import numpy as np
    return np.ndarray(shape = (2,2), buffer = \
    np.array([100 *(cm[0][0] + cm[1][1])/length, \
    100 * cm[0][1]/length, 100 * cm[1][0]/length, \
    100 * (cm[1][0] + cm[0][1])/length]))
print (cm_percent(cm, len(testY)))
print (testY.value_counts())
print (len(testY))
```

Interpret the Results

- The confusion matrix shows/compares the predictions to the known values
  - 1,263,180 were predicted to be good charges and were indeed good
  - 285 were predicted to be fraudulent and were indeed fraudulent
  - 7667 were predicted to be good but were in fact fraudulent (false positive or Type I error)
    - 1,263,180 + 7667 = 1,270,847
  - 1392 were predicted to be fraudulent but were in fact good (false negative or Type II error)
    - 1392 + 285 = 1677

```
[[1263180    7667]
[   1392     285]]

[[99.28810773  0.60250337]
 [ 0.1093889   0.71189227]]

0   1270847
1   1677

1272524
```

- Results as percentages
  - 99.28% correct
  - .71% incorrect
  - .60% false positive
  - .10% false negative
- Overall pretty good at predicting

Save and Load Model

- To save the results of a lengthy training process
  - `pip install joblib`

```
from joblib import dump, load
dump(modelNB, 'modelNB.joblib')
```

- To reload a saved model

```
modelNB2 = load('modelNB.joblib')
predY = modelNB2.predict(testX)
cm = confusion_matrix(testY, predY)
print (cm)
cmp = cm_percent(cm, len(testY))
print (cmp)
```

Decision Trees

- Data is split up based on some factor among the independent variables, then evaluated for how good a job it did
- Recursively keeps applying this algorithm over and over until it comes up with a good set of rules
- Pros
  - It's easy
  - Performs well with categorical data and continuous data
  - Transparency lets you see how it made its choices
- Cons
  - Calculations take a lot longer as you add more and more columns
  - Becomes difficult to understand the decision tree as it gets larger

Apply Decision Tree

- Load the module, create a model and train it

```
from sklearn.tree import DecisionTreeClassifier
modelDT = DecisionTreeClassifier()
modelDT.fit(trainX, trainY)
```

- Run a prediction on the reserved testing set and compare the predicted values to the known values

```
def important_features(model, columns):
   return pd.DataFrame(model.feature_importances_, \
         columns=['Importance'], \
         index = columns).sort_values(['Importance'], \
         ascending = False)

predY = modelDT.predict(testX)
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(testY, predY)
print (cm)
print (cm_percent(cm, len(testY)))
print (testY.value_counts(), len(testY))
print (important_features(modelDT, trainX.columns))
```

Interpret the Results

- Results of the confusion matrix are interpreted exactly as before
  - The numbers different so see which did a better job
- Decision Trees have additional information about what factors contributed to its decisions
  - Calculated by number of samples that reach the node divided by the total number of samples
  - The higher the number, the more important the feature

	Importance
oldbalanceOrg	0.432194
newbalanceDest	0.228966
amount	0.161474
oldbalanceDest	0.087661
newbalanceOrig	0.069822
type	0.019578
isFlaggedFraud	0.000305

- Based on these results, we can see the most important factors as to how it decided a transaction was legitimate or fraudulent where `oldbalanceOrg` and `newbalanceDest` is followed by the amount
  - The remaining columns were of less importance

Random Forest

- Creates Decision Trees on randomly selected samples of the training set
- Performs multiple iterations and gets prediction results
- Votes on the best random sample
- Pros
  - Often highly accurate due to the strength of multiple predictions
  - Usually does not suffer from overfitting
  - Can see the relative feature importance which is useful in revising the model
- Cons
  - Slow to generate because of multiple iterations
  - Compared to a Decision Tree you cannot really see the path of the tree

Apply Random Forest

- Load the module, create a model and train it

```
from sklearn.ensemble import RandomForestClassifier
modelRF = RandomForestClassifier(n_estimators=10)
modelRF.fit(trainX, trainY)
```

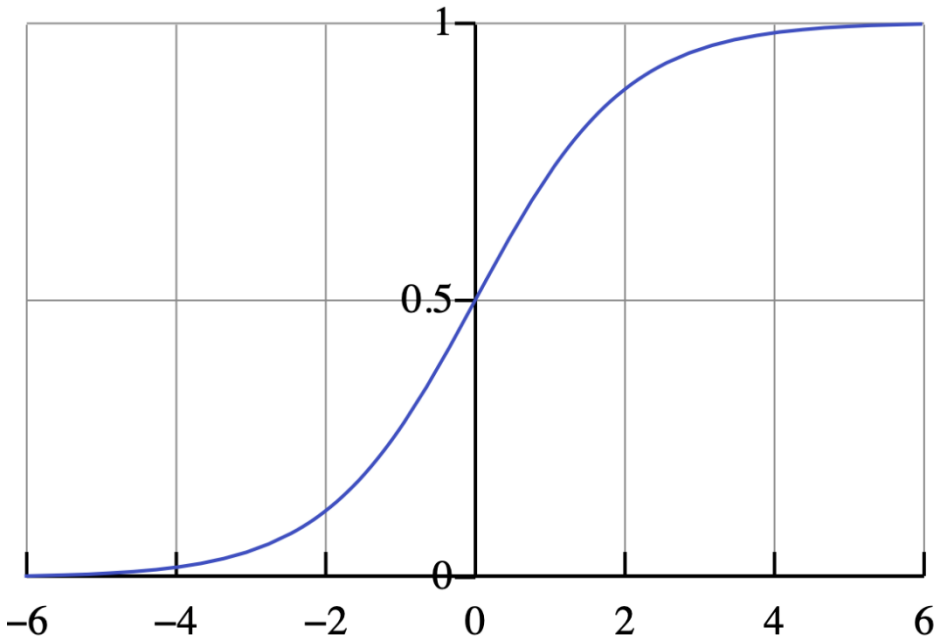
- Run a prediction on the reserved testing set and compare the predicted values to the known values

```
predY = modelRF.predict(testX)
from sklearn import metrics
print ("Accuracy:",metrics.accuracy_score(testY, predY))
cm = confusion_matrix(testY, predY)
print (cm)

import pandas as pd
feature_imp = pd.Series(modelRF.feature_importances_,\
    index=trainX.columns).sort_values(ascending=False)
print (feature_imp)
```

Logistic Regression

- Yet another alternative to categorizing, but with a twist
  - Not only predicts a value but predicts the probability of its occurrence
  - Allows you to change a probability threshold to favor false positives or false negatives
- The math behind it involves logarithms and finding coefficients of the independent variables

$$\ell = \log \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$



Logistic Regression (continued)

- Data needs to be dummy encoded skipping one value as a reference value
- Pros
  - Works better in cases with low signal to noise ratio
  - Allows for tweaking of false positives and false negatives
  - Transparency lets you see how it made its choices
- Cons
  - Does not perform well with too many features (independent variables)
  - Not good with large number of categorical values within a feature because of dummy encoding

Apply Logistic Regression

- Need to dummy encode categorical features
  - Use this helper function to make that easier

```
from sklearn.model_selection
import train_test_splitfrom sklearn
import preprocessing as pp
def dummy_code(data, columns, drop_first = True):
   for c in columns:
      dummies = pd.get_dummies(data[c], prefix = c, \
      drop_first = drop_first)
      i = list(data.columns).index(c)
      data = pd.concat([data.iloc[:,:i], dummies, \
             data.iloc[:,i+1:]], axis = 1)
return data

df2 = dummy_code(df, ['type'], drop_first = True)
trainX, testX, trainY, testY = train_test_split(df2.iloc[:,
df2.columns != 'isFraud'], df2.isFraud, train_size = train_size,
test_size = test_size)
print (testX.columns)
print (testX.head())
```

Apply Logistic Regression (continued)

- Create and train the logistic model

```
from sklearn.linear_model import LogisticRegression
modelLR = LogisticRegression(multi_class='auto', solver='lbfgs')
modelLR.fit(trainX, trainY)
print (modelLR.coef_)

[[-2.34708708e-08 -2.08083063e-09 -3.31118886e-07
  -8.31500683e-10  -8.26614837e-04  8.30307380e-04
  -9.09162361e-04  7.04021006e-07  -2.41542762e-06
  1.05305835e-11]]
```

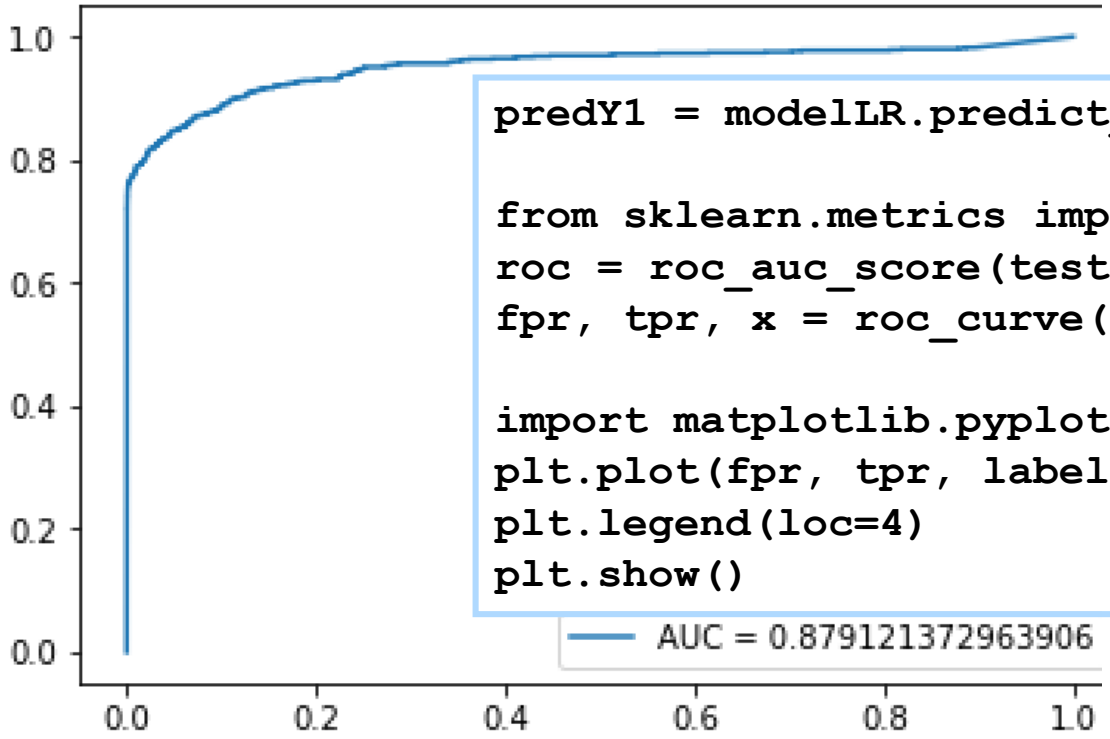
Interpret the Results

- The coefficients show how the math formula is trained
  - Multiply each feature by its coefficient and sum them up to get the probability
  - ln(p/(1-p)) = β + β1X1 + β2X2 + β3X3 + … + βnXn
- Confusion matrix is interpreted just like all the other models

```
0.998001609399901 3 0.0019983906000987013
[[1268705    2162]
 [    381    1276]]
PC FP
FN PW
[[9.98001609e+01 1.69898564e-01]
 [2.99404962e-02 1.99839060e-01]]
```

ROC Curve

- Another way to look at how good a model is at making predictions is the Receiver Operating Characteristic curve and the AUC (Area Under Curve)
- Plots the True Positive Rate (TPR) vs the False Positive Rate (FPR)
- A good model gives a value close to 1




```
predY1 = modelLR.predict_proba(testX)

from sklearn.metrics import roc_auc_score, roc_curve
roc = roc_auc_score(testY, predY)
fpr, tpr, x = roc_curve(testY, predY1[:,1])

import matplotlib.pyplot as plt
plt.plot(fpr, tpr, label = 'AUC = ' + str(roc))
plt.legend(loc=4)
plt.show()
```

Different Thresholds

- Additionally, you can do the prediction probabilities
  - Set a threshold probability to determine whether the prediction is positive or negative
  - Allows you to tweak the accuracy, false positives and false negatives
- Use the `predict_proba` function instead

```
predY = modelLR.predict_proba(testX)
print (predY[:10])

[9.99999999e-001 1.72171774e-113]    # 99% likely positive
[8.26513185e-001 1.73486815e-001]    # 82% likely positive
[6.28732003e-178 1.00000000e+000]    # 100% likely negative
[5.20000000e-001 4.80000000e-001]    # 52% likely positive
```

Tweak the Results

- Run the prediction on the same trained model several times with different probability thresholds and compare the accuracy and FP/FN

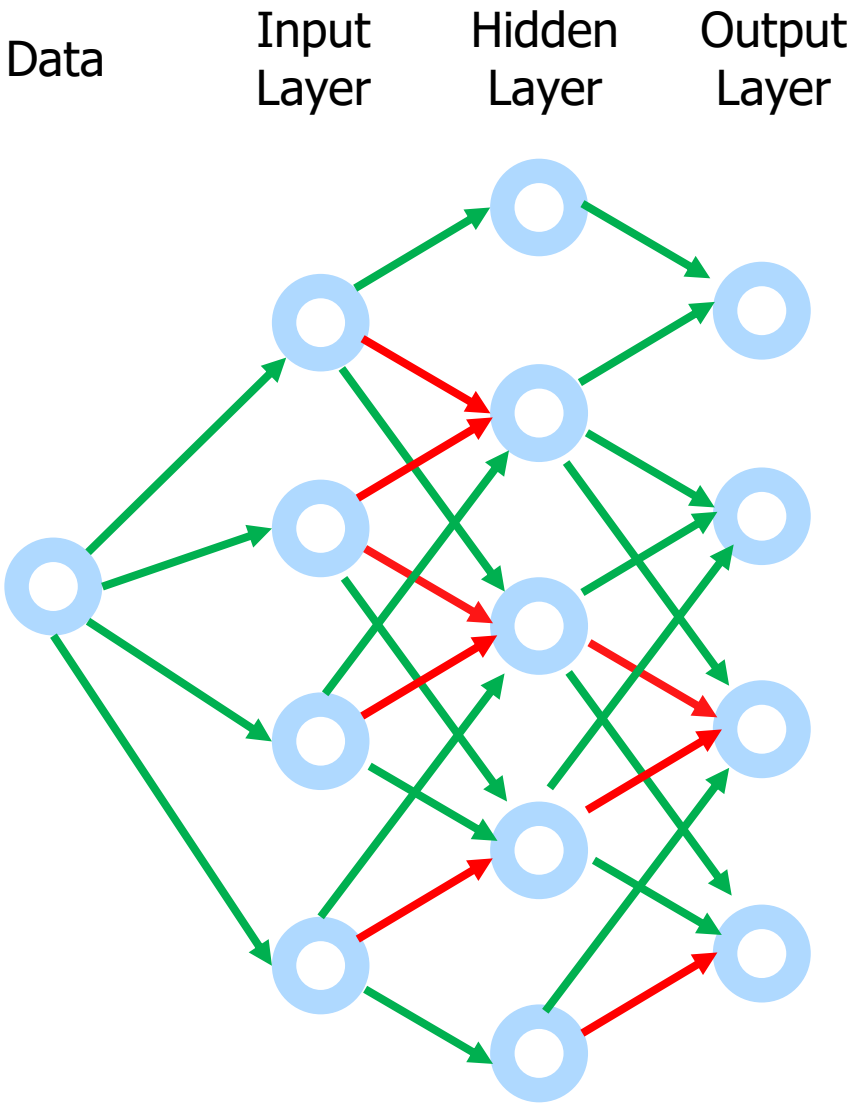
```
predY = modelLR.predict_proba(testX)
print (predY[:10])
print (modelLR.score(testX, testY))
for threshold in range(30, 91, 10):
    predY1 = np.where(predY[:,1] >= threshold/100, 1, 0)
    mse = np.mean((predY1 - testY)**2)
    cm = confusion_matrix(testY, predY1)
    print ('\nTHRESHOLD', threshold, 'MSE', mse)
    print (cm)
    print (cm_percent(cm, len(testY), legend = False))
```

- Highest accuracy turned out to be about 70 in this case
- Lower threshold yielded more false positives, fewer false negatives
  - Tune the threshold to fit the business case of whether you favor FP or FN

Neural Networks

- Simulates the way the human brain solves
  - Uses a perceptron, algorithm, or function run in multiple layers
- Not only predicts a value but predicts the probability of its occurrence
  - Allows you to change a probability threshold to favor false positives or false negatives
- Pros
  - Often perform better than others which can be important where accuracy is desired (predicting cancer)
  - Good for unusual data like image, video, audio
- Cons
  - Black box, you don't know how it made its decision
  - Not appropriate in cases where transparency is important
  - Require a lot more data to train than other models
  - Computationally expensive
- Cool visualization of Neural Network from Google
  - https://playground.tensorflow.org/

Neural Network Visualized



Apply Neural Network

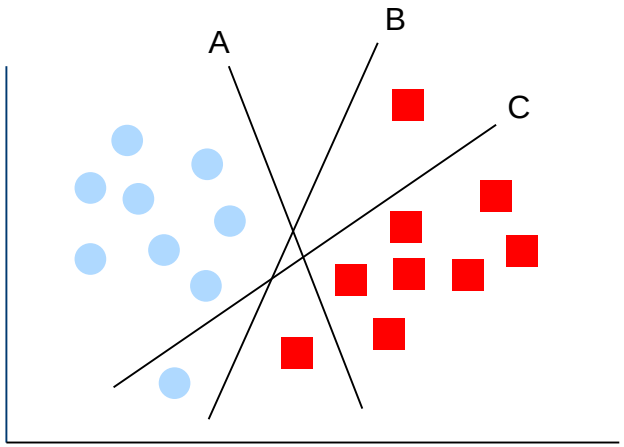
- Need to dummy encode categorical features
  - This time keep the first element
- Also works best if you rescale the numeric values
  - Rescale the whole dataset before splitting it or else the results differ

```
from sklearn.model_selection import train_test_split
from sklearn import preprocessing as pp
# rescale the data
df2 = dummy_code(df, ['type'], drop_first = False)
print (df2.columns)
df2[['amount',  'oldbalanceOrg', 'newbalanceOrig',
'oldbalanceDest', 'newbalanceDest']] /= df2[['amount',
'oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest',
'newbalanceDest']].max()
trainX, testX, trainY, testY =
train_test_split(df2.iloc[:,df2.columns != 'isFraud'],
df2.isFraud, train_size = train_size, test_size = test_size)
```

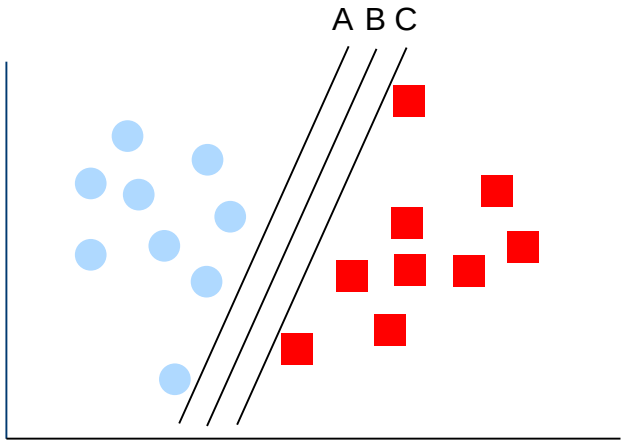
- Look at the results just like all the other classification models

Support Vector Machine

- Support Vector Machine attempts to find the hyperplane that separates the two classes better—in this case, B clearly does that

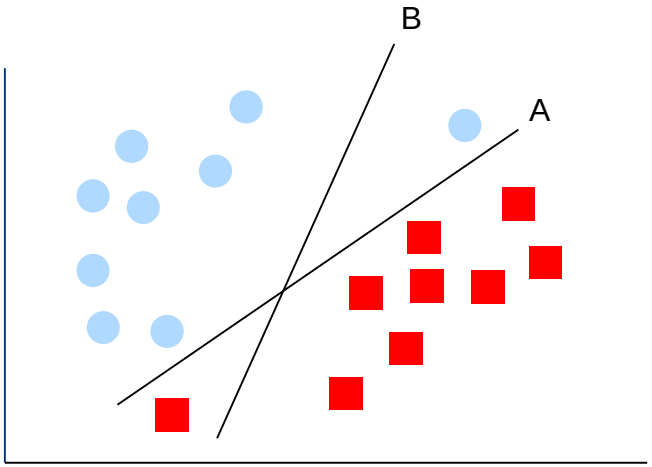


- Next, it tries to find the right hyperplane that maximizes the distances from that plane to the nearest data points in either group—again, it is B in this case
  - This is called a margin

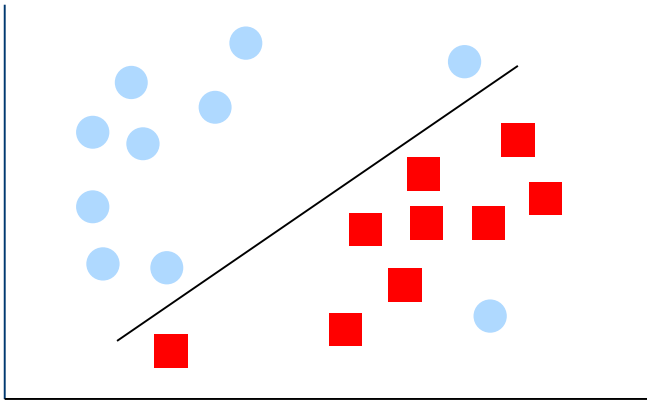


Support Vector Machine (continued)

- In this case, B has a bigger margin but A is the plane it determined did a better job of classifying them so it would choose A

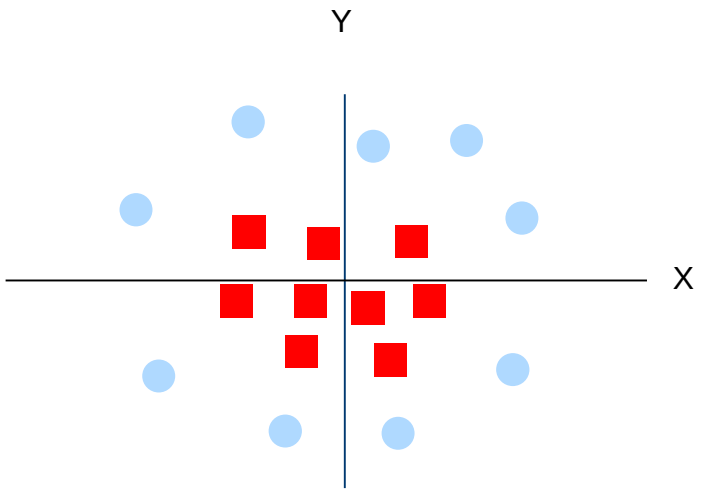


- Here we have a blue outlier, and SVM has a feature that allows such outliers to be ignored

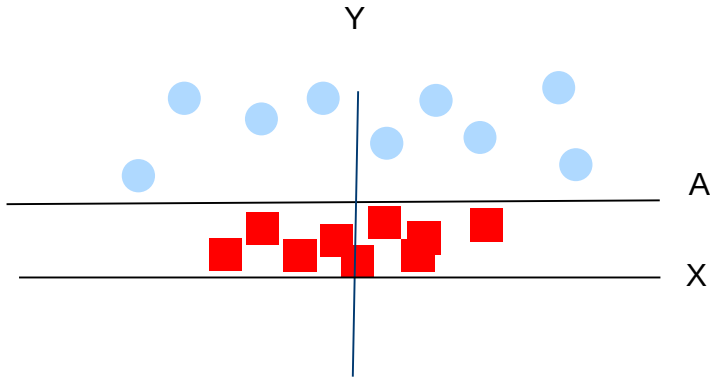


Support Vector Machine (continued)

- Here it would be impossible to draw a straight line that separates the groups, but we could translate this into a new feature z = x2 + y2



- The new feature 'z' could then be used to find the hyperplane using a built-in feature called kernel trick
  - Effectively it can solve for a non-linear feature



Apply SVM

- SVM looks like most other models and has lots of parameters to play with
  - Run it through several combinations to compare the results

```
from sklearn import svm
train_size = .03; test_size = .01
trainX, testX, trainY, testY = train_test_split( \
   df2.iloc[:,df2.columns != 'isFraud'], df2.isFraud, \
   train_size = train_size, test_size = test_size)

def do_SVM(kernel, gamma):
    print ("\nKernel:", kernel, "Gamma:", gamma)
    modelSVM = svm.SVC(gamma = gamma,  kernel = kernel)
    modelSVM.fit(trainX, trainY)
    print (modelSVM.score(testX, testY))
    predY = modelSVM.predict(testX)
    print (confusion_matrix(testY, predY))

do_SVM('linear', gamma='auto')
for kernel in ['rbf', 'poly', 'sigmoid']:
    for gamma in ['auto', 10, 100]:
        if not (kernel == 'poly' and gamma == 100):
           do_SVM(kernel, gamma)
```

Chapter Concepts

Classification Model

Algorithms

Chapter Summary

Classification Review

- Classification is one of the most widely used models
- It is supervised
- Good at predicting either/or or multiple-choice categories
- Lots of algorithms
- No one algorithm is best for all situations so often it involves running many of them, documenting the results, and choosing the best for your data and business case
- Can save the results of a lengthy training to a file and reload it for use with the predict function when needed

Next Steps

- Classification is one of the most useful models
- We have explored several different algorithms here and there are tons more, each with its own strengths and weaknesses
- Some other algorithms to explore:
  - Support Vector Machines
  - Boosted Trees
  - Random Forests
  - K-Nearest Neighbor
  - Stochastic Gradient Descent

Chapter Summary

In this chapter, we have:

- Understood the use cases for Classification models
- Discussed and compared various algorithms
  - Naive Bayes
  - Decision Tree
  - Logistic Regression
  - Neural Network

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

CHAPTER 8: REGRESSION ANALYSIS

Chapter Objectives

In this chapter, we will:

- Introduce Linear Regression
- Compare two algorithms
  - Scikit-learn
  - Statsmodel

Chapter Concepts

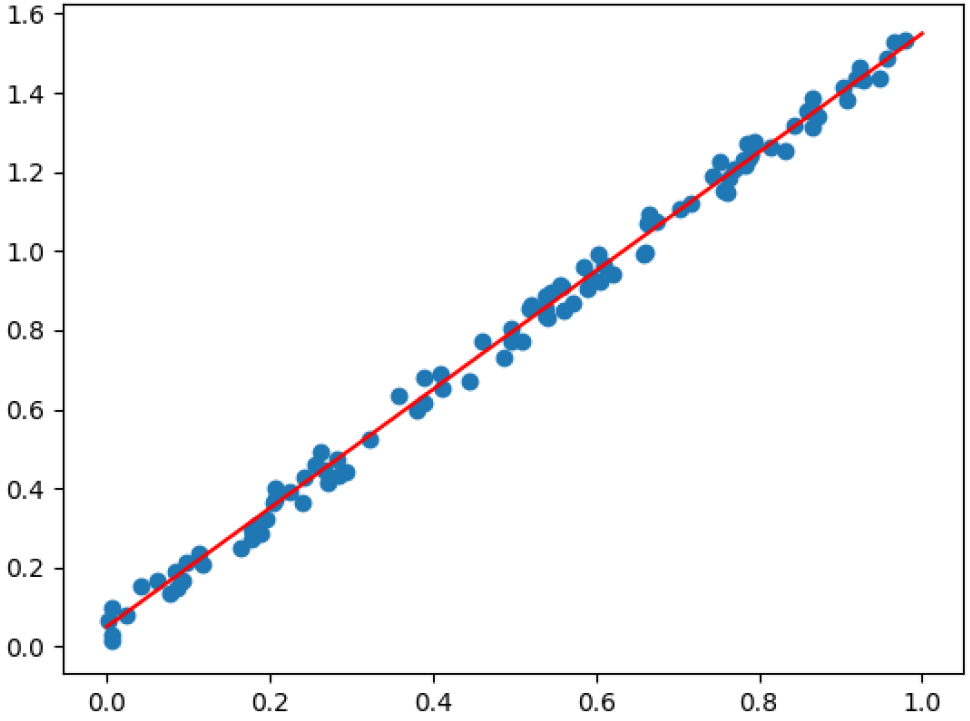
Regression Analysis

Algorithms

Chapter Summary

Linear Regression

- Given a collection of X, Y points, you could easily see there is a pattern
- If you remember enough algebra, you could describe the pattern of dots as roughly following the red line, which could be described with the formula y = 1.5x + .01



Linear Regression (continued)

- The idea is that the line that best describes the pattern of dots is the one that has the least distances of the dots from the line
- The formula that describes the line could then be used to predict a value that we have not observed
  - The better the line and formula are at describing that pattern of dots, the more accurate that prediction should be
- Extrapolate this idea onto more than just two axes and instead try to find a line that goes through many different dimensions and you have the idea of multiple linear regression
  - $y = \alpha + \beta 1 x1 + \beta 2 x2 + \ldots + \beta i xi + \varepsilon$
- Has many use cases
  - Predicting a stock or commodity price
  - Predicting election results
  - Predicting crime rate

Linear Regression (continued)

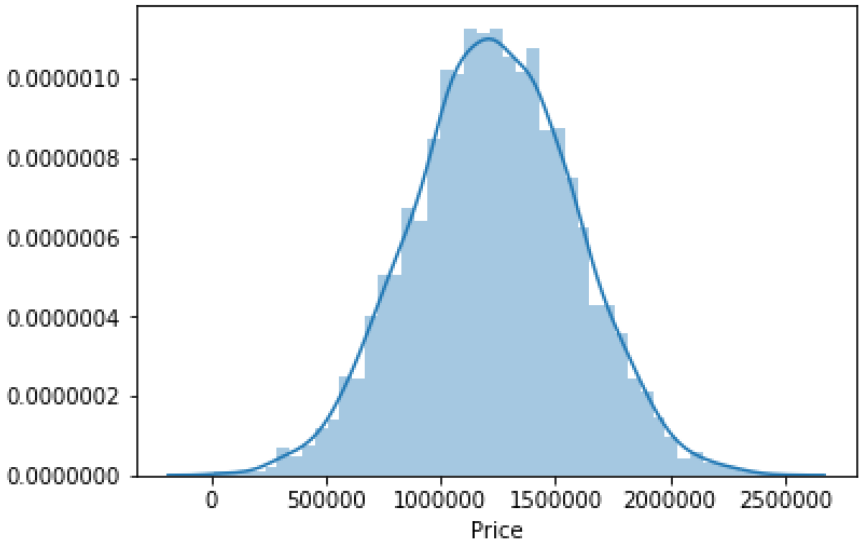
- Is a supervised model that requires training from a known set of data and testing to see how good it is at predicting before using it for real predictions
- Only works with numeric values
  - Categorical data needs to be dummy encoded
- Does not deal well with missing data, so must be fixed by removing or replacing with central tendency
- There are many algorithms to do this, each with its own pros and cons

Dataset

- For our examples, let's use a public data set of housing data

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
USAhousing = pd.read_csv('USA_Housing.csv')
print (USAhousing.columns)
print (USAhousing.head())
sns.distplot(USAhousing['Price'])
```

- The data has no categorical columns but does have an address we will ignore
- Plotting the distribution of Prices shows that they are normally distributed



Chapter Concepts

Regression Analysis

Algorithms

Chapter Summary

Create a Scikit Model

- Prep the data and fit the model on the training set

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
x = USAhousing[['Avg. Area Income', 'Avg. Area House Age',
'Avg. Area Number of Rooms', 'Avg. Area Number of Bedrooms',
'Area Population']]
y = USAhousing['Price']
trainX, testX, trainY, testY = train_test_split(x, y, test_size
= 0.4, random_state = 101)

from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(trainX, trainY)
```

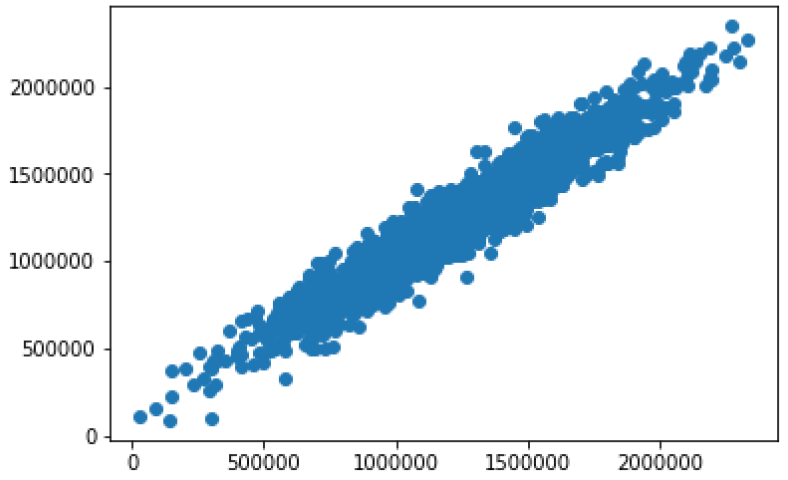
View the Model

- Run predictions on the test set and compare them to the real values to see how well the model did at predicting

```
predictions = lm.predict(testX)
plt.scatter(testY, predictions)
print ("Mean squared error: %.2f" % mean_squared_error(testY,
predictions))
print ('Variance score: %.2f' % r2_score(testY, predictions))
```

- The variance score, also called R-Squared, indicates how well in general the model fits
  - Ranges from 0 – 1, the closer to 1 the better
  - .92 means this model fit reasonably well

```
Mean squared error: 10460958907.21
Variance score: 0.92
```



Create a Stats Model

- The stats library offers a different version of the algorithm
- Provides a little more information about the accuracy of the model

```
import statsmodels.api as sm
model = sm.OLS(trainY, trainX).fit()
print (model.summary())
predictions = model.predict(testX)
plt.scatter(testY, predictions)
plt.show()
```

Interpret the Model

- You automatically get the R-squared from the summary function
  - Additionally, the Adjusted R-squared is helpful because it helps to compare models with different numbers of predictor variable
- The P factors also identify which features are most significant in influencing the value you are trying to predict

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Price   R-squared:                       0.965
Model:                            OLS   Adj. R-squared:                  0.965
Method:                 Least Squares   F-statistic:                 1.633e+04
Date:                Fri, 17 May 2019   Prob (F-statistic):               0.00
Time:                        23:08:27   Log-Likelihood:                -41426.
No. Observations:                3000   AIC:                          8.286e+04
Df Residuals:                    2995   BIC:                          8.289e+04
Df Model:                           5
Covariance Type:            nonrobust
================================================================================================
                                   coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------------------------
Avg. Area Income               10.1001      0.346     29.176      0.000       9.421      10.779
Avg. Area House Age          4.972e+04   3870.040     12.846      0.000    4.21e+04    5.73e+04
Avg. Area Number of Rooms   -9135.0559   4226.074     -2.162      0.031   -1.74e+04    -848.754
Avg. Area Number of Bedrooms 4272.2896   4029.066      1.060      0.289   -3627.728    1.22e+04
Area Population                8.4544      0.419     20.171      0.000       7.633       9.276
==============================================================================
Omnibus:                        0.002   Durbin-Watson:                   1.999
Prob(Omnibus):                  0.999   Jarque-Bera (JB):                0.000
Skew:                          -0.000   Prob(JB):                         1.00
Kurtosis:                       2.998   Cond. No.                     9.34e+04
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 9.34e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
Mean squared error: 59842619587.25
Variance score: 0.53
```

Chapter Concepts

Regression Analysis

Algorithms

Chapter Summary

Next Steps

- Regression has a lot more complexity to it once you master the basics
- Some subjects to explore in this area:
  - Under- and over-fitting a model
  - Correlation between the independent variables
  - Non-linear regression

Chapter Objectives

In this chapter, we have:

- Introduced Linear Regression
- Compared two algorithms
  - Scikit-learn
  - Statsmodel

ROITRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

CHAPTER 9: GRAPH DATABASE

Chapter Objectives

In this chapter, we will:

- Introduce the concept of graph databases
- Explore Python's NetworkX package

Chapter Concepts

Graph Databases

NetworkX

Chapter Summary

Graph Database

- Relational databases are useful for storing fields in a tabular format
- To show one record is connected to another in some way, you would build a related table and create a foreign key between them
- This can get messy to encode complex data with many relationships
- Graph databases encode the data in a different manner which places equal weight to the data of the record and the relationship it has with other records
- Once data is encoded in a graph database format, it is able to be queried in ways that would be difficult and time consuming in a relational database
- Graph databases have many use cases
  - Network and social analysis
  - Fraud detection
  - Supply chain transparency
  - Infrastructure monitoring

Terminology

- Graph databases are composed of nodes and edges instead of tables made up of rows and columns
- A node is like a record in that it stores data, but it is much less structured and more free form
  - Can store any data you like as a key-value pair
  - Much like JSON or Python dictionaries
  - Not all nodes have to have the same data
- Edges are connections between nodes and are like a relationship between objects
  - Edges themselves can have additional data, just like a node
  - A common data element found in an edge is weight, indicating how strong a relationship there is between the two nodes
  - Edges can be directed or undirected, meaning they flow one way or both ways

Common Queries

- There are two categories of queries you can run on graph databases
  - Egocentric answers questions about particular nodes
    - How many other nodes does it connect to?
    - How important is the node?
    - Is it centrally located or near the borders?
    - What other types of nodes is it connected to?
    - How far is it from one node to another?
    - What is the shortest path between two particular nodes?
  - Sociocentric answers questions about the network as a whole
    - Are there particular combinations of nodes that are more important?
    - Are there natural clusters or communities within the network?
    - Are there bottlenecks that could break down the network if they fail?
    - Where could we add additional connections to facilitate a more robust network or better connectivity between nodes?
    - Do nodes tend to connect to other similar nodes?

Chapter Concepts

Graph Databases

NetworkX

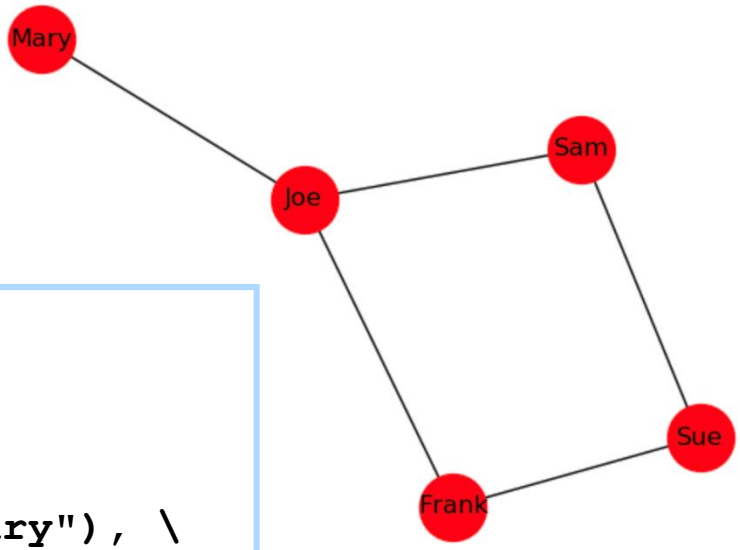
Chapter Summary

Tools

- There are many tools for graph databases
  - Neo4J is a full-fledged database cluster for storing and manipulating large graph databases
  - NetworkX is a Python library that has a lot of features and is suitable for running queries on data that may be stored in regular places, such as text files
  - GraphX is a Spark library that can be used with Python, Scala, and Java and works on a cluster to handle larger datasets
- All use the same concepts and terminologies and can solve the same problems—they just use different syntax to do it

NetworkX

- Easy to use Python package that supports creating in-memory graph databases that can be analyzed
- Integrates nicely with Python plotting packages to make it easy to display graph data
- The circles represent nodes
- The lines between the nodes are edges

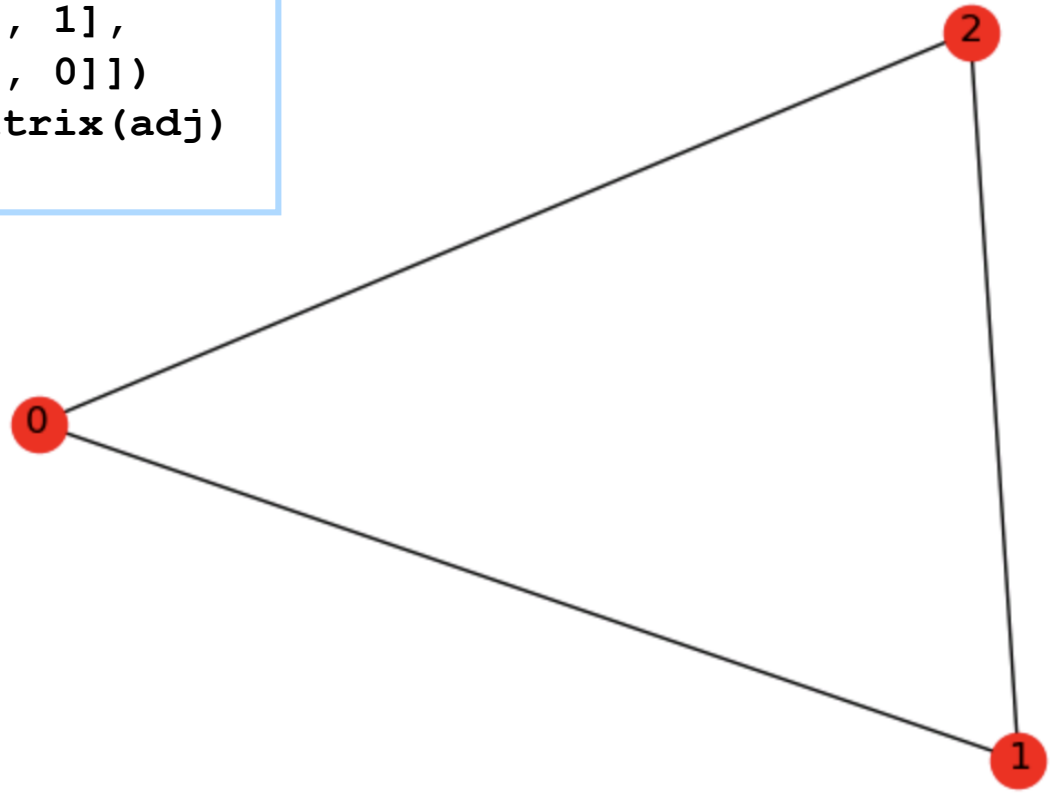



```
pip install networkx

import networkx as nx
G_names = nx.Graph()
G_names.add_edges_from([("Joe", "Mary"), \
("Joe", "Frank"), ("Sue", "Frank"), \
("Sam", "Joe"), ("Sue","Sam")])
nx.draw_networkx(G_names, node_size = 1000)
```

Reading Data from NumPy

- Data can be easily read from NumPy arrays

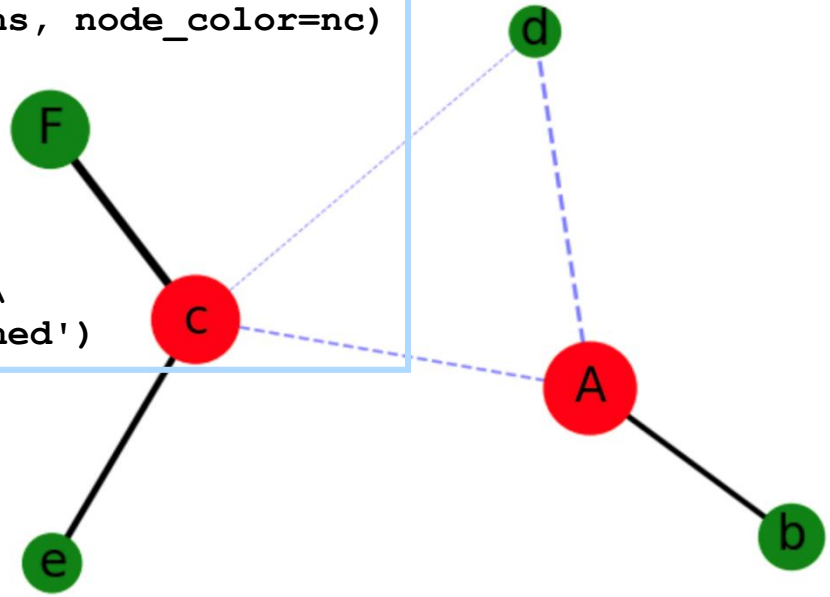
```
import numpy as np
adj = np.array([[0, 1, 1],
                [1, 0, 1],
                [1, 1, 0]])
G2 = nx.from_numpy_matrix(adj)
nx.draw_networkx(G2)
```



Changing Plot Attributes

- Using additional data in the nodes and edges, the plot can customize features such as size, shape, color, line thickness, and style to create a visual representation of the features that will help identify meaningful information

```
eigen = nx.eigenvector_centrality(G)
ns = [15 * x['size'] for _,x in G.nodes(data=True)]
nc = ['r' if x >= .5 else 'g' for x in eigen.values()]
nx.draw_networkx_nodes(G, pos, node_size=ns, node_color=nc)

# edges
nx.draw_networkx_edges(G, pos, \
    edgelist=elarge, width=elargeweight)
nx.draw_networkx_edges(G, pos, \
    edgelist=esmall, width=esmallweight, \
    alpha=0.5, edge_color='b', style='dashed')
```



Common Functions

- Once you have data encoded as nodes and edges, there are tons of functions that can be run on the data to find interesting features that can be used to find particular data or display it by changing various visual attributes
- Some common egocentric functions:
  - **`degree_centrality`** – number of links in or out
  - **`closeness_centrality`** – how quickly info can pass to other from here
  - **`eigenvector_centrality`** – how well connected to important nodes
  - **`betweenness_centrality`** – how likely to be in communication path
  - **`shortest_path`** – shortest route between two nodes
  - **`diameter`** – shortest distance between two furthest nodes
- Some common sociocentric functions:
  - **`degree_assortativity_coefficient`** – useful in finding how homogeneous the data is
  - **`find_cliques`** – find sections of graph where all nodes are connected to each other
  - **`density`** – describes how many potential connections actually exist

Chapter Concepts

Graph Databases

NetworkX

Chapter Summary

Next Steps

- We only had time to explore the tip of the iceberg for graph databases
- Learn about use cases and solutions that graph databases can solve and see how companies like Facebook, LinkedIn, and more have applied these techniques
- Explore technologies such as:
  - Neo4J
  - NetworkX
  - GraphX for Spark

Chapter Summary

In this chapter, we have:

- Introduced the concept of graph databases
- Explored Python's NetworkX package

ROI TRAINING

MAXIMIZE YOUR TRAINING INVESTMENT

Python for Data Scientists

COURSE SUMMARY

Course Summary

In this course, we have:

- Learned how to use Python for data science and machine learning
- Explored the most common packages such as NumPy, Pandas, and Sklearn
- Learned how to transform data into the correct shape for analysis
- Explored supervised and unsupervised models including Cluster, Classification, and Regression
- Become familiar with graph database concept using network package

ROI's Training Curricula

Agile Development	.NET and Visual Studio
Amazon Web Services (AWS)	Networking and IPv6
Azure	Oracle and SQL Server Databases
Big Data and Data Analytics	OpenStack and Docker
Business Analysis	Project Management
Cloud Computing and Virtualization	Python and Perl Programming
Excel and VBA	Security
Google Cloud Platform (GCP)	SharePoint
ITIL® and IT Service Management	Software Analysis and Design
Java	Software Engineering
Leadership and Management Skills	UNIX and Linux
Machine Learning and Neural Networks	Web and Mobile Apps
Microsoft Exchange	Windows and Windows Server

Please visit our website at www.ROItraining.com for a complete list of offerings

Ways to Stay in Touch

ROITraining
Maximize Your Training Investment

ROITraining.com

Follow us!
@ROITraining

Connect with us!
/company/roi-training

Like us!
/ROITraining

Follow us!
/roitraining

The management and staff of **ROI Training** would like to thank you for your commitment to continuing education.

Our mission is to deliver customized technology and management training solutions to large corporations and government agencies around the world. We strive to provide business professionals with the skills and knowledge necessary to increase work performance. We hope the training facilitated by your ROI instructor enables you to successfully apply what you have learned to your work.

We wish you continued success in your career and hope to see you again in the near future.

Best regards,

Brian Reimer and *David Carey*
Co-Founders