Data Science Intensive

Joshua Cook

September 21, 2018

Contents

List of Figures

List of Tables

Chapter 1

Lectures

1.1 Vectors and Matrices

1.1.1 The Vector

A vector is a multi-dimensional element.

e.g. (1), $(1, 3, 5, 8)$, $(1, 2, \ldots, 50)$

As opposed to a scalar

e.g. $1, \sqrt{2}, e, \pi, 101010, 47$

1.1.2 Scalar Arithmetic

- add
- subtract
- multiply
- divide

1.1.3 Representing Data with Matrices

We will use n to represent the number of distinct data points, or observations, in our data set. We will let p denote the number of variables that are available for use in making predictions.

In general, we will let x_{ij} represent the value of the j^{th} variable for the i^{th} observation, where $i \in [1 : n]$ and $j \in [1 : p]$.

i will be used to index the samples or observations (from 1 to n) and j will be used to index the variables (from 1 to p). We let X denote a $n \times p$ matrix whose $(i, j)^{th}$ element is x_{ij}.

$$X = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{np} \end{pmatrix}$$

It is useful to visualize X as a spreadsheet of numbers with n rows and p columns.

1.1.4 Representing a Matrix in Python

In Python, we can represent a matrix using the `numpy` library and its `array` class.

Listing 1.1: Import `numpy`

```
In [1]: import numpy as np
```

Listing 1.2: Generate a list of random numbers

```
In [2]: import random

        n = 5
        p = 3
        list_of_numbers = []
        for _ in range(15):
            list_of_numbers.append(random.randint(1, 100))
```

Listing 1.3: Display the list of random numbers

```
In [3]: list_of_numbers

Out[3]: [37, 68, 61, 10, 10, 5, 67, 69, 97, 30, 67, 60, 60, 96,
        32]
```

Here, we turn our list of numbers into an `np.array` object, then reshape that object into an $n \times p$ matrix.

Listing 1.4: Reshape the list of random numbers into an $n \times p$ matrix

```
In [4]: X = np.array(list_of_numbers).reshape((n, p))
```

Listing 1.5: Display the new matrix, X

```
In [5]: X

Out[5]: array([[37, 68, 61],
               [10, 10,  5],
               [67, 69, 97],
               [30, 67, 60],
               [60, 96, 32]])
```

Listing 1.6: Display the shape of the new matrix, X

```
In [6]: X.shape

Out[6]: (5, 3)
```

1.1.5 The Elements of X

X is comprised of two different kinds of vectors:

- Each row of X is a vector of length p (contains p elements).
- Each row of X is an observation representing a single sample or point from our data set
- Each column of X is a vector of length n (contains n elements).
- Each column of X is all of the observations for a given variable over our entire data set

All of the rows of X are the elements of X considered as row vectors. This is probably how you are most comfortable thinking about X i.e. some data set.

At the same time, all of the column of X are the elements of X considered as column vectors.

Suppose we consider a column of X as

$$\mathbf{x}_j = \begin{pmatrix} x_{1j} \\ x_{2j} \\ \vdots \\ x_{nj} \end{pmatrix}$$

Note that this is taking into account that we always think of a single vector as a column.

We can then think of X as

$$X = \begin{pmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_p \end{pmatrix}$$

1.1.6 Column-Oriented Matrices in Python

In Python, it is easier to work with the columns of a matrix using the Pandas library, and its `DataFrame` class.

Listing 1.7: Import `pandas`

```
In [7]: import pandas as pd
```

Listing 1.8: Convert X into a `DataFrame`

```
In [8]: X = pd.DataFrame(X)
```

Listing 1.9: Display the matrix X, now a `DataFrame`

```
In [9]: X
```

Table 1.1: Display the matrix X, now a `DataFrame`

	0	1	2
0	37	68	61
1	10	10	5
2	67	69	97
3	30	67	60
4	60	96	32

Listing 1.10: Display the first column of X

```
In [10]: X[0]

Out[10]: 0    37
         1    10
         2    67
         3    30
         4    60
         Name: 0, dtype: int64
```

Listing 1.11: Display the first three columns of X

```
In [11]: X[[0,1,2]]
```

Table 1.2: Display the first three columns of X

	0	1	2
0	37	68	61
1	10	10	5
2	67	69	97
3	30	67	60
4	60	96	32

Listing 1.12: Display the first and third columns of X

```
In [12]: X[[0,2]]
```

Table 1.3: Display the first and third columns of X

	0	2
0	37	61
1	10	5
2	67	97
3	30	60
4	60	32

1.1.7 Referencing a Row in a `DataFrame`

It may still be necessary to reference the row of a matrix represented as a `DataFrame`. This can be done using the `.loc()` method. With `.loc()` rows can be referenced by their row index.

Listing 1.13: Display the second row of X

```
In [13]: X.loc[2]

Out[13]: 0    67
         1    69
         2    97
         Name: 2, dtype: int64
```

Note that even here, the row vector is represented as a column of numbers.

1.1.8 The Transpose

We obtain the transpose of a matrix by flipping it around its axis.

So that

$$X^T = \begin{pmatrix} x_{11} & x_{21} & \dots & x_{n1} \\ x_{12} & x_{22} & \dots & x_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ x_{1p} & x_{2p} & \dots & x_{np} \end{pmatrix}$$

Listing 1.14: Display the transpose of X

```
In [14]: X.T
```

Table 1.4: Display the transpose of X

	0	1	2	3	4
0	37	10	67	30	60
1	68	10	69	67	96
2	61	5	97	60	32

1.1.9 The Dot Product

It is possible to multiply two vectors of equal length together. This operation is called the **dot product** and yields a scalar value, that is, a single number.

This number is obtained by

1. multiplying each of the corresponding values from the two vectors
2. summing the results

Written in Python, the process would look as follows. Note that we are making use of a convention of writing vectors as a repeated lowercase letter.

Listing 1.15: Prepare a list of the multiples of two vectors

```
In [15]: aa = np.array([1,2,3])
         bb = np.array([-2,-1,0])

         multiples = []

         for a, b in zip(aa, bb):
             multiples.append(a*b)
```

Listing 1.16: Display the list `multiples`

```
In [16]: multiples

Out[16]: [-2, -2, 0]
```

Listing 1.17: Calculate the sum of the list `multiples` to perform the dot product

```
In [17]: dot_product = sum(multiples)
```

Listing 1.18: Display the dot product

```
In [18]: dot_product

Out[18]: -4
```

1.1.10 The Dot Product as a Row Vector Times a Column Vector

Strictly speaking, the dot product is the multiplication of a row vector times a column vector. So that if we have

$$\mathbf{a} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \text{ and } \mathbf{b} = \begin{pmatrix} -2 \\ -1 \\ 0 \end{pmatrix}$$

then the dot product we just performed would be written as

$$\mathbf{a}^T\mathbf{b} = \begin{pmatrix}1 & 2 & 3\end{pmatrix}\begin{pmatrix}-2\\-1\\0\end{pmatrix} = 1\cdot -2 + 2\cdot -1 + 3\cdot 0 - 4$$

or

Listing 1.19: Calculate the dot product using `.dot()`

```
In [19]: aa.dot(bb)

Out[19]: -4
```

1.1.11 Matrix Multiplication

At the heart of the vast majority of work that we will be doing is matrix multiplication. Let us consider two matrices, A and B.

$$A = \begin{pmatrix}1 & 2\\3 & 4\end{pmatrix} \text{ and } B = \begin{pmatrix}5 & 6\\7 & 8\end{pmatrix}$$

Then the product of A and B is denoted AB.

The $(i,j)^{th}$ element of AB is computed by computing the dot product of the ith row of A with the jth row of B.

Listing 1.20: Create two matrices, A and B, as `DataFrames`

```
In [20]: A = pd.DataFrame([[1,2],[3,4]])
         B = pd.DataFrame([[5,6],[7,8]])
```

So the (1,1)th element of AB should be $1\cdot 5 + 2\cdot 7 = 19$.

Listing 1.21: Perform the dot product of the first row of A with the first column of B

```
In [21]: A.loc[0].dot(B[0])

Out[21]: 19
```

We would repeat this process for every element of AB. As you can imagine this would be very tedious by hand.

Listing 1.22: Multiply A times B

```
In [22]: A.dot(B)
```

Table 1.5: Multiply A times B

	0	1
0	19	22
1	43	50

1.1.12 The importance of Matrices

Transformations

- Rotation Matrix (project data from two axes to one)
- Compute the difference vector

1.2 Statistical Learning

You may be more used to performing calculations where you have a known formula and you wish to use some **input** to calculate an **output**.

For example, you might be calculating the force of an object

$$f = ma$$

Listing 1.23: Define a function for calculating force

```
In [1]: def force(mass, acceleration):
            return mass * acceleration
```

Listing 1.24: Calculate the force on an object given mass and acceleration

```
In [2]: m = 10
        a = 9.8

        force(m,a)
```

Statistical learning is different. Here, we are given the inputs and outputs:

- X / input / features
- y / output / target

We make an assumption that there is some function f such that

$$y = f(X) + \varepsilon$$

Note:

1. X is often a vector
2. ε is the "error term" and represents both known and unknown sources of noise

From a known X and y we seek the formula or function, f, that best describes the relatioship between them.

This can be trivial.

Listing 1.25: Core imports for numerical computing in Python

```
In [3]: import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import seaborn as sns

        %matplotlib inline
```

Listing 1.26: Define x and y

```
In [4]: x = [-1,3]
        y = [3,5]
```

Here, we have two x values and two y values. It is trivial to find a formula describing the relationship between x and y.

Listing 1.27: Plot the line defining the relationship between x and y

```
In [5]: fig, ax = plt.subplots(1,1,figsize=(20,5))

        ax.plot(x[0], y[0], 'g^', ms=25)
        ax.plot(x[1], y[1], 'g^', ms=25)
        ax.plot((x[0], y[0]), (x[1], y[1]))
        plt.show()
```

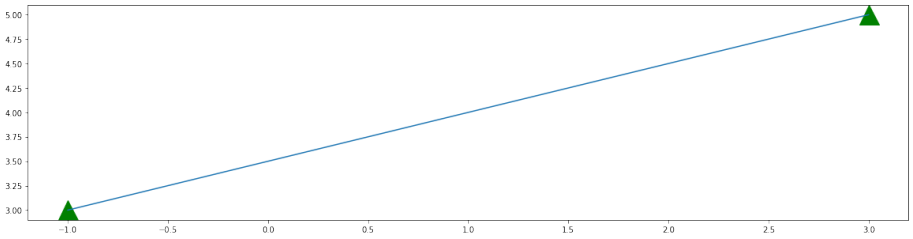

Figure 1.1: Plot the line defining the relationship between (x) and (y)

This is a perfectly determined system. We have two variables and two data points i.e. $n = p$.

The situation is more complicated when $n \neq p$.

Listing 1.28: Add more values to x and y

```
In [6]: x += [0,1,2]
        y += [3.25, 4, 4.75]

        points = [
            (x[0], y[0]),
            (x[1], y[1]),
            (x[2], y[2]),
            (x[3], y[3]),
            (x[4], y[4])
        ]
```

Listing 1.29: Plot new x and y

```
In [7]: fig, ax = plt.subplots(1,1,figsize=(20,5))

        for point in points:
            ax.plot(*point, 'g^', ms=25)

        plt.show()
```

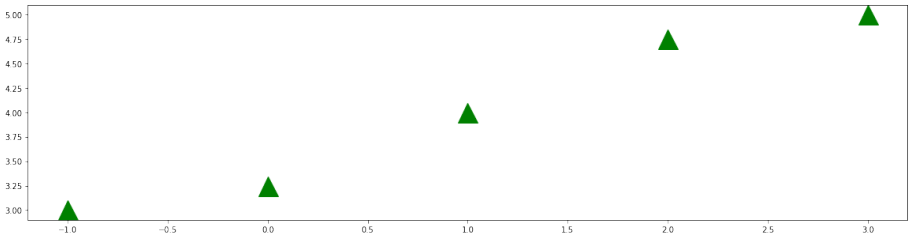

Figure 1.2: Plot new (x) and (y)

The line here does not fit perfectly. The system is **overdetermined** i.e. $n > p$.

1.2.1 The best f aka $\widehat{f}$

We seek "the best" f, called $\widehat{f}$ or "eff hat".

We will quantify what "best" mean in terms of error.

For $\widehat{y} = \widehat{f}(X)$, we consider the error over our data

$$\mathrm{E}\left[(y - \widehat{y})^2\right] = \left[f(X) - \widehat{f}(X)\right]^2 + \mathrm{Var}(\varepsilon)$$

1.2.2 Reasons to estimate f

1. Prediction

- Given some new value for X, what would we expect y to be?

2. Inference

- Which features $(X_1, X_2, \ldots)$ are associated with the target y?
- What is the relationship between the target y and each feature $(X_1, X_2, \ldots)$?

- Can the relationship between the target y and each feature be adequately summarized using a linear equation, or is the relationship more complicated?

Listing 1.30: Load Advertising data as a `DataFrame`

```
In [8]: Advertising = pd.read_csv('data/Advertising.csv',
                                  index_col=0)
```

1.2.3 Inference on Advertising Data

Here we create a series of **linear models** to help us make inferences. Linear models allow for relatively simple and interpretable inference, but may not yield as accurate predictions as some other approaches

Listing 1.31: Plot the Advertising Data

```
In [9]: fig, ax = plt.subplots(1,3,figsize=(20,5))

        advertising_types = ['TV', 'radio', 'newspaper']

        for i, feature in enumerate(advertising_types):

            sns.regplot(Advertising[feature],
                        Advertising['sales'], ax=ax[i])

        plt.show()

Out[9]: /Users/joshuacook/miniconda3/lib/python3.6/site-packages/
        scipy/stats/stats.py:1713: FutureWarning: Using a non-
        tuple sequence for multidimensional indexing is
        deprecated; use `arr[tuple(seq)]` instead of `arr[seq]`.
        In the future this will be interpreted as an array index,
         `arr[np.array(seq)]`, which will result either in an
        error or a different result.
          return np.add.reduce(sorted[indexer] * weights, axis=
        axis) / sumval
```

- Which media contribute to sales?
- Which media generate the biggest boost in sales?
- How much increase in sales is associated with a given increase in TV advertising?

1.2.4 Parametric versus Non-Parametric

Parametric Models

1. We make an assumption about the functional form, or shape, of f.
2. After a model has been selected, we need a procedure that uses the training data to fit or train the model.

The linear model is an important example of a parametric model. We consider

$$\widehat{f}(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$$

- A linear model is specified in terms of $p + 1$ parameters $\beta_0, \beta_1, \ldots, \beta_p$
- We estimate the parameters by fitting the model to training data.
- Although it is almost never correct, a linear model often serves as a good and interpretable approximation to the unknown true function $f(X)$.
- the most common method to fitting linear models is the ordinary least squares (OLS) method

Non-Parametric Models

- Non-parametric methods do not make explicit assumptions about the functional form of f.
- Any parametric approach brings with it the possibility that the functional form used to estimate f is very different from the true f, in which case the resulting model will not fit the data well.
- In contrast, non-parametric approaches completely avoid this danger, since essentially no assumption about the form of f is made.
- But non-parametric approaches do suffer from a major disadvantage: since they do not reduce the problem of estimating f to a small number of parameters, a very large number of observations (far more than is typically needed for a parametric approach) is required in order to obtain an accurate estimate for f ... **THE CURSE OF DIMENSIONALITY**.

K Nearest Neighbors is an important example of a non-parametric model.

1.2.5 Tradeoffs

The Trade-Off Between Overfitting and Underfitting

A potential disadvantage of a parametric approach is that the model we choose will usually not match the true unknown form of f.

If the chosen model is too far from the true f, then our estimate will be poor.

We can try to address this problem by choosing flexible models that can fit many different possible functional forms for f.

Listing 1.32: Define correlated data with random noise

```
In [10]: xx = np.linspace(-2,2,20)
         yy = xx**3 + 2*np.random.random(20) + 100
```

Listing 1.33: Import the Linear Regression model from Scikit-Learn

```
In [11]: from sklearn.linear_model import LinearRegression
```

Listing 1.34: Instantiate an instance of the Linear Regression model

```
In [12]: lr = LinearRegression()
```

Listing 1.35: Fit the new Linear Regression model

```
In [13]: lr.fit(xx.reshape(-1, 1), yy)

Out[13]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs
         =1, normalize=False)
```

Listing 1.36: Display the coefficients and intercept of the fit model

```
In [14]: lr.coef_, lr.intercept_

Out[14]: (array([2.83186003]), 100.95322073455777)
```

Listing 1.37: Plot the line of best fit

```
In [15]: fig, ax = plt.subplots(1,1,figsize=(20,5))

         ax.scatter(xx, yy)
         ax.plot(xx, xx*lr.coef_ + lr.intercept_)

         plt.show()
```

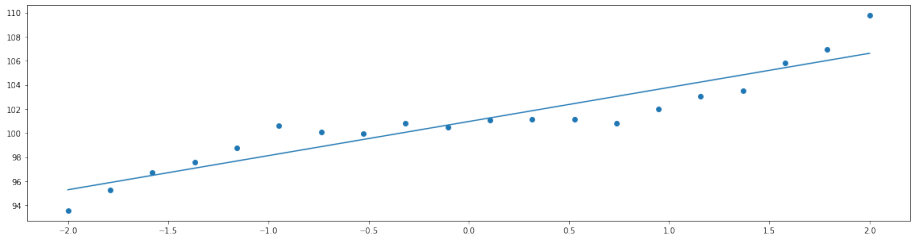

Figure 1.3: Plot the line of best fit

Listing 1.38: Import a helper function

```
In [16]: from lib.lec_3_helper import fit_and_plot_poly_to_degree
```

Listing 1.39: Plot lines of best fit for higher order functions

```
In [17]: fig, ax = plt.subplots(1,1,figsize=(20,5))

         ax.scatter(xx, yy)
         ax.plot(xx, xx*lr.coef_ + lr.intercept_)

         for i in [2,3,7,20]:
             fit_and_plot_poly_to_degree(xx, yy, i, ax)
         plt.legend()

         plt.show()
```

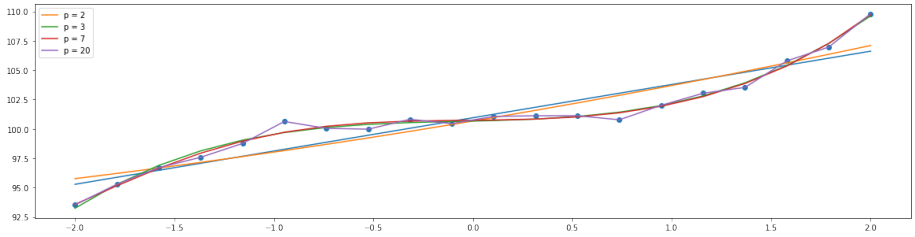

Figure 1.4: Plot lines of best fit for higher order functions

Listing 1.40: Instantiate and Fit Linear Regression on Sales using TV data

```
In [18]: lr = LinearRegression()
         lr.fit(Advertising[['TV']], Advertising.sales)
         lr.coef_, lr.intercept_

Out[18]: (array([0.04753664]), 7.032593549127695)
```

Listing 1.41: Plot line of best fit

```
In [19]: fig, ax = plt.subplots(1,1,figsize=(20,5))

         predictions = lr.predict(Advertising[['TV']])

         ax.scatter(Advertising.TV, Advertising.sales,
                    label='actual')

         ax.scatter(Advertising.TV, predictions,
                    label='prediction')

         plt.legend()

         plt.show()
```

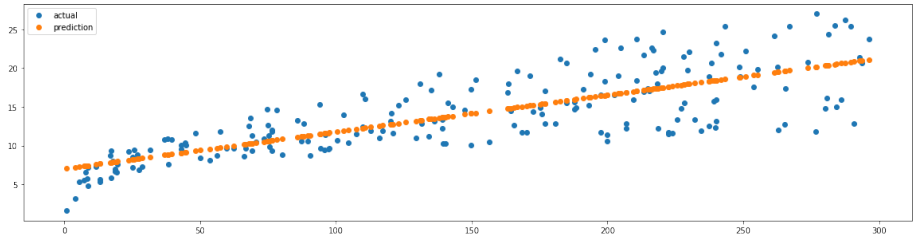

Figure 1.5: Plot line of best fit

Listing 1.42: Plot higher order lines of best fit

```
In [20]: fig, ax = plt.subplots(1,1,figsize=(20,5))

         for i in [1,2,3,20,30,50]:
             fit_and_plot_poly_to_degree(
                 Advertising.TV,
                 Advertising.sales,
                 i, ax, scatter=True)

         ax.scatter(Advertising.TV, Advertising.sales,
                    label='actual', s=100, alpha=0.6)

         plt.legend()

         plt.show()
```

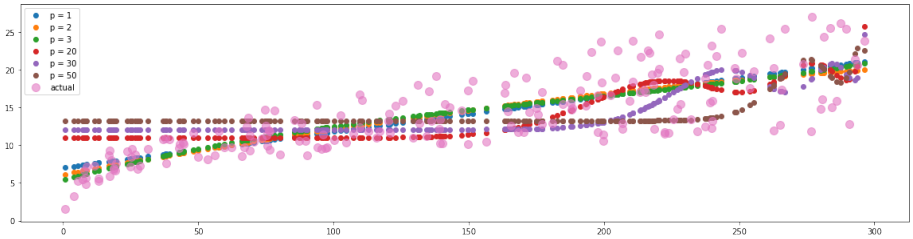

Figure 1.6: Plot higher order lines of best fit

In general, fitting a more flexible model requires estimating a greater number of parameters. These more complex models can lead to a phenomenon known as **overfitting** the data, which essentially means they follow the errors, or noise, too closely. We can generalize this in terms of model flexibility. In general, less flexible models may overfit, whereas more flexible models may overfit.

The Trade-Off Between Prediction Accuracy and Model Interpretability

We can also think about flexibility and accuracy in terms of our ability to interpret models.

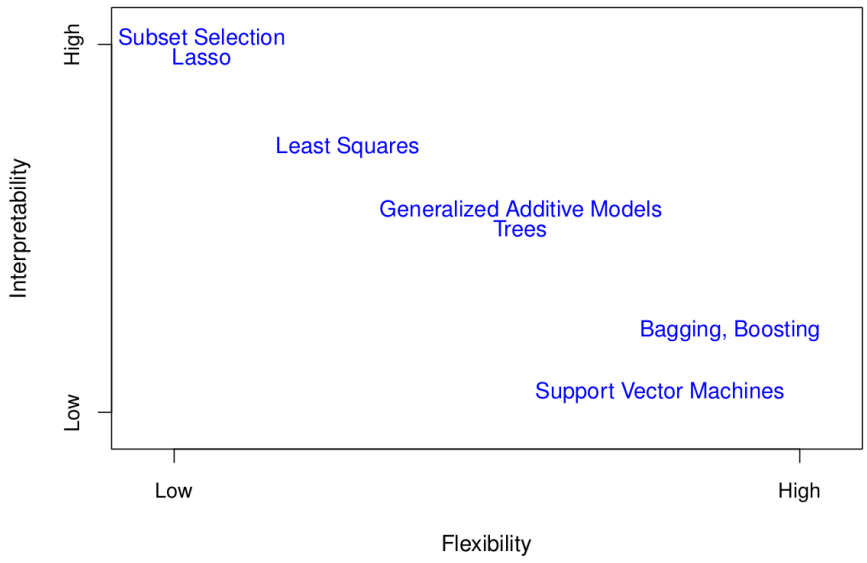

Figure 1.7: Model Flexibility

Models on the right can often be "black box" models that are difficult to interpret. Their increased flexibility may make them stronger at making accuracte predictions.

Models on the left will be much more simple, but they may also be less accurate.

1.2.6 Statistical Learning Domains

1. Supervised Learning
2. Unsupervised Learning
3. Regression
4. Classification
5. Cluster Analysis

Why is it necessary to introduce so many different statistical learning approaches, rather than just a single best method?

TANSTAAFL

1.3 Pythonic Sequences

1.3.1 Looping

Listing 1.43: A list of names

```
In [1]: names = ['Nellie', 'Ronald', 'Judith', 'Lavonda']
```

Write a `for`-loop that prints each name:

1. Using indexing
2. Using iteration on the list

Listing 1.44: A `for`-loop that iterates through this list

```
In [2]: n = len(names)
        for i in range(n):
            print(names[i])

Out[2]: Nellie
        Ronald
        Judith
        Lavonda
```

Listing 1.45: A more "pythonic" `for`-loop

```
In [3]: for name in names:
            print(name)

Out[3]: Nellie
        Ronald
        Judith
        Lavonda
```

Listing 1.46: A list of colors and a list of ratios

```
In [4]: colors = ["red", "green", "blue", "purple"]
        ratios = [0.2, 0.3, 0.1, 0.4]
```

Write a `for`-loop that prints

```
percentage color
```

e.g.

```
20.0% red
```

1. Using `enumerate`
2. Using `zip`

Listing 1.47: Using `enumerate` to match values in two lists

```
In [5]: for i, color in enumerate(colors):
            ratio = ratios[i]
            percentage = 100*ratio
            print("{}% {}".format(percentage, color))

Out[5]: 20.0% red
        30.0% green
        10.0% blue
        40.0% purple
```

Listing 1.48: Catching tuple outputs in multiple values

```
In [6]: ratio, color = (0.2, 'red')
```

Listing 1.49: Zipping two lists together into a list of matched tuples

```
In [7]: list(zip(ratios, colors))

Out[7]: [(0.2, 'red'), (0.3, 'green'), (0.1, 'blue'), (0.4, '
        purple')]
```

Listing 1.50: Using `zip` to match values in two lists

```
In [8]: for ratio, color in zip(ratios, colors):
            percentage = 100*ratio
            print("{}% {}".format(percentage, color))

Out[8]: 20.0% red
        30.0% green
        10.0% blue
        40.0% purple
```

Listing 1.51: Someone's favorite numbers

```
In [9]: my_favorite_numbers = [1, 1, 2, 3, 5, 8, 13]
```

Write a `for`-loop to create a list called `my_favs_doubled` that doubles each number in `my_favorite_numbers`.

Listing 1.52: Doubling the values in `my_favorite_numbers` with a `for`-loop

```
In [10]: my_favs_doubled = []

         for number in my_favorite_numbers:
             doubled = 2 * number
             my_favs_doubled.append(doubled)

         my_favs_doubled

Out[10]: [2, 2, 4, 6, 10, 16, 26]
```

1.3.2 Comprehensions

Write a list comprehension to create a list called `my_favs_doubled` that doubles each number in `my_favorite_numbers`.

Listing 1.53: Doubling the values in `my_favorite_numbers` with a list comprehension

```
In [11]: my_favs_doubled = [2 * number for number in
         my_favorite_numbers]
```

Write a list comprehension that creates a list of tuples of a number and its square e.g. `(1,1)`, `(2,4)` for the numbers from 1 to 10.

Listing 1.54: Creating a list of tuples of a number and its squares

```
In [12]: [(number, number ** 2) for number in range(1,11)]

Out[12]: [(1, 1),
          (2, 4),
          (3, 9),
          (4, 16),
          (5, 25),
          (6, 36),
          (7, 49),
          (8, 64),
          (9, 81),
          (10, 100)]
```

Exercise

Write a list comprehension called `COLORS` that converts the list `colors` to uppercase strings.

Listing 1.55: Insert your code here

```
In [13]:
```

Write a nested list comprehension to create the matrix

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

using nested lists.

Listing 1.56: Insert your code here

```
In [14]:
```

1.3.3 Iterators

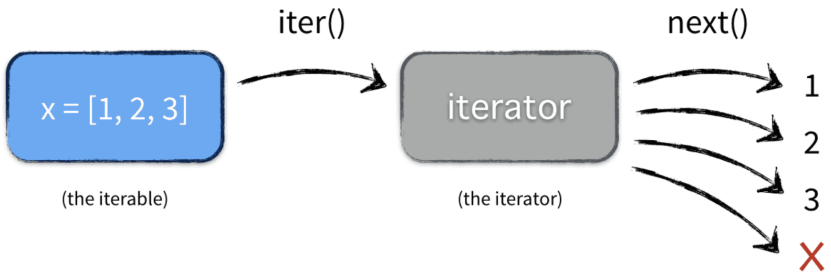

Figure 1.8: Iterators

Listing 1.57: Create a simple list

```
In [15]: x = [1, 2, 3]
```

Listing 1.58: Create two iterators named `y` and `z` on the list `x` using `iter()`

```
In [16]: y = iter(x)
         z = iter(x)
```

Listing 1.59: Use `next()` to display the next value of `y`

```
In [17]: next(y)

Out[17]: 1
```

Listing 1.60: Use `next()` to display the next value of `y`

```
In [18]: next(y)

Out[18]: 2
```

Listing 1.61: Use `next()` to display the next value of `z`

```
In [19]: next(z)

Out[19]: 1
```

Listing 1.62: Display the values of `x`

```
In [20]: x

Out[20]: [1, 2, 3]
```

type(x), type(y), type(z)

Exercise

Write a `for`-loop to print each value of `x`.

Listing 1.63: Insert your code here

```
In [22]:
```

1.3.4 `itertools`

Listing 1.64: Import `count` from `itertools`

```
In [23]: from itertools import count
```

Listing 1.65: Define a new object called `counter` that is `count` initialized with the value 12

```
In [24]: counter = count(12)
```

What are the `next` three values of `counter`?

Listing 1.66: Display the next three values of `counter`

```
In [25]: next(counter), next(counter), next(counter)

Out[25]: (12, 13, 14)
```

What is the type of `counter`?

Listing 1.67: Display the `type` of `counter`

```
In [26]: type(counter)

Out[26]: itertools.count
```

Exercise

Import `cycle` from `itertools`.

Listing 1.68: Insert your code here

```
In [27]:
```

Define a new object called `color_cycle` which is a `cycle` on the list `colors`.

Listing 1.69: Insert your code here

```
In [28]:
```

What is the type of `color_cycle`?

Listing 1.70: Insert your code here

```
In [29]:
```

What are the next 7 values of `color_cycle`?

Listing 1.71: Insert your code here

```
In [30]:
```

Import `islice` from `itertools`.

Listing 1.72: Insert your code here

```
In [31]:
```

Define a new object called `limited_color_cycle` which is an `islice` on `color_cycle` from 0 to 15.

Listing 1.73: Insert your code here

```
In [32]:
```

Write a `for`-loop to print all of the values in `limited_color_cycle`.

Listing 1.74: Insert your code here

```
In [33]:
```

1.3.5 Fibonacci

Define a new class called `Fib`.

1. Define an `__init__` function on `self` that sets two `self` variables, `prev` and `curr` and initialized them to 0 and 1, respectively.
2. Define an `__iter__` function on `self` that returns `self`.
3. Define a `__next__` function on `self` that
4. stores `self.curr` as a temporary value
5. adds `self.prev` to `self.curr`
6. assigns the temporary value to `self.prev`
7. returns the temporary value

Listing 1.75: Define the `class Fib`

```
In [34]: class Fib:
             def __init__(self):
                 self.prev = 0
                 self.curr = 1

             def __iter__(self):
                 return self

             def __next__(self):
                 tmp = self.curr
                 self.curr += self.prev
                 self.prev = tmp
                 return tmp
```

Listing 1.76: import `islice`

```
In [35]: from itertools import islice
```

Listing 1.77: Use `islice` to generate a list of the first ten Fibonacci numbers

```
In [36]: fib = Fib()
         list(islice(fib, 0, 10))

Out[36]: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Define a new function called `fib`.

1. Store `prev` and `curr` as 0 and 1, respectively.
2. While `True`
3. `yield` the value stored in `curr`
4. store `curr` as a temporary variable
5. add `prev` to `curr`
6. assign the temporary variable to `prev`

Exercise - Generator Expressions

Write a list comprehension called `squares` that is the squares of the first ten numbers.

Listing 1.78: Insert your code here

```
In [40]:
```

Write a set comprehension called `squares_set` that is the squares of the first ten numbers.

Listing 1.79: Insert your code here

```
In [41]:
```

Write a dictionary comprehension called `squares_dict` that is the squares of the first ten numbers using the number itself as the key and the square as the value.

Listing 1.80: Insert your code here

```
In [42]:
```

Write a generator expression called `lazy_squares` that is the squares of the first six numbers.

Listing 1.81: Insert your code here

```
In [43]:
```

Get the first two values in `lazy_squares`.

Listing 1.82: Insert your code here

```
In [44]:
```

Obtain the rest as a list.

Listing 1.83: Insert your code here

```
In [45]:
```

1.4 The Bias-Variance Tradeoff Revisited

Listing 1.84: Set up Numerical Python environment

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import seaborn as sns

        %matplotlib inline
        plt.rc('figure', figsize=(20, 6))
```

Listing 1.85: Load Iris dataset

```
In [2]: from sklearn.datasets import load_iris

        feature_names = [
            'sepal_length',
            'sepal_width',
            'petal_length',
            'petal_width'
        ]

        IRIS = load_iris()
        iris_df = pd.DataFrame(IRIS.data, columns=feature_names)
        labels = IRIS.target_names
```

In his 1996 paper, "The Lack of A Priori Distinctions between Learning Algorithms," David Wolpert asserts that "There is No Free Lunch in Machine Learning", essentially saying that the performance of on machine learning models are equivalent when averaged across all possible problems. In other words, there is no way to know which model is going to perform at best for a particular problem. In practice this means that we must have strong methods for assessing the performance of our models.

Thinking about model performance is complex. We cannot simply choose the model that performs best with the data that we have. The reason for this is that the data we have represents a sample of the actual data that we could possibly collect now or in the future. The "best model" is not necessarily one that performs best on the data that we have. The best model is a model that performs extremely well on the data that we have but it's also capable of generalizing to new data. In statistical learning, we have a framework for thinking about this called The Bias-Variance Tradeoff.

This framework is difficult to understand. Adding to the difficulty is the fact that both "bias" and "variance" are important concepts for working in applied

statistics **and** the meaning of these terms is difficult to reconcile with their meaning when thinking about the Bias-Variance Tradeoff.

Let us try to come to a high level understanding of these terms in this setting before looking at them in application. You might think of **bias** as the extent to which a particular model can learn to represent an underlying physical phenomenon. A low bias means that the model was able to learn the phenomenon well. For example, looking at the Iris data set, if we are building a regression model to predict the petal width then bias would be the degree to which our model was able to learn the relationship between petal length and petal width or sepal width and petal width.

Variance on the other hand is the extent to which a particular model would change if fit with different data. We previously looked at sampling our data set. We saw the measured means of value change with each sample. From this we can infer that a model predicting the meaning with only a few points of data has a high variance.

1.4.1 The Gory Details

If we have split our data into a training set and a testing set, then we can think of choosing the best model in terms of optimizing the expected test error, MSE_{test}

Let's consider sources of possible test error:

$$MSE_{test} = \mathbb{E}\left[(y - \widehat{y})^2\right] = \text{Var}\left(\widehat{y}\right) + \left(\text{Bias}\left(\widehat{y}\right)\right)^2 + \text{Var}\left(\epsilon\right)$$

This is intended to be a conceptual and not an actual calculation to be performed. Let's think about what each of these terms might represent. The variance is error introduced to the model by the specific choice of training data. Of course this isn't something that we choose, at least not with out using randomness, but the training data that is used will impact the model. By nature, variance is a squared value and that's always positive. Bias is introduced by choosing a specific model. Note that it is squared here and thus also always positive. The last term is the variance caused by noise in the system. We have no way of controlling this, nor of actually knowing what is truly noise and what is model variance or bias. Again this term is always positive.

The important thing is that all three of these terms are always positive. The impact of this is that one kind of error cannot be offset by another kind of error. A high variance cannot be offset by a low bias. **In order to choose the best model, we are going to need to simultaneously minimize both bias and variance.** The problem is changing an aspect of our model to decrease one Will typically increase the other – The Bias-Variance Tradeoff.

1.4.2 Model Assessment

Consider these eight models for predicting petal width x_{pw} on the Iris data set, using the other three features, petal length x_{pl}, sepal length x_{sl}, and sepal width x_{sw}

$$
\begin{aligned}
\widehat{f_0} &= \widehat{x}_{pw} = \beta_0 \\
\widehat{f_1} &= \widehat{x}_{pw} = \beta_0 + \beta_1 x_{pl} \\
\widehat{f_2} &= \widehat{x}_{pw} = \beta_0 + \beta_1 x_{sl} \\
\widehat{f_3} &= \widehat{x}_{pw} = \beta_0 + \beta_1 x_{sw} \\
\widehat{f_4} &= \widehat{x}_{pw} = \beta_0 + \beta_1 x_{pl} + \beta_1 x_{sl} \\
\widehat{f_5} &= \widehat{x}_{pw} = \beta_0 + \beta_1 x_{pl} + \beta_1 x_{sw} \\
\widehat{f_6} &= \widehat{x}_{pw} = \beta_0 + \beta_1 x_{sl} + \beta_1 x_{sw} \\
\widehat{f_7} &= \widehat{x}_{pw} = \beta_0 + \beta_1 x_{pl} + \beta_1 x_{sl} + \beta_1 x_{sw}
\end{aligned}
$$

We will split our dataset into ten randomly chosen subsets of 50 points each. We will assess the performance of each of these models. We will use the mean of the performance to represent bias and the standard deviation of the performance to represent variance.

Listing 1.86: Create 10 sample datasets from full data set

```
In [3]: samples = []

        for _ in range(10):
            samples.append(iris_df.sample(50))
```

Listing 1.87: Create 10 sample datasets from full data set with list comprehension

```
In [5]: samples = [iris_df.sample(50) for _ in range(10)]
```

Listing 1.88: Show length of each sample in the list `samples`

```
In [6]: [len(sample_set) for sample_set in samples]

Out[6]: [50, 50, 50, 50, 50, 50, 50, 50, 50, 50]
```

Listing 1.89: Import the design matrices library, `dmatrices` from `patsy`

```
In [7]: from patsy import dmatrices
```

Listing 1.90: Import the Linear Regression Model from Sklearn

```
In [8]: from sklearn.linear_model import LinearRegression
```

Listing 1.91: Simple example of computing a list of squared differences

```
In [9]: a = np.array((1,2,3))
        b = np.array((4,5,6))

        (b - a)**2

Out[9]: array([9, 9, 9])
```

Listing 1.92: Define a Mean Squared Error function

```
In [10]: def MSE(actual, predicted):
             return sum((actual - predicted)**2)/len(actual)
```

Listing 1.93: Define a function for running our tests

```
In [11]: def test_func(model_description):

             test = dict()

             test['samples'] = [
                 dmatrices(model_description, sample)
                 for sample in samples
             ]

             test['models']  = [
                 LinearRegression(fit_intercept=False)
                 for _ in range(10)
             ]

             test['scores']  = []

             models_and_samples = zip(test['models'],
                                      test['samples'])

             for model, sample in models_and_samples:
                 target = sample[0]
                 features = sample[1]
                 model.fit(features, target)

                 mse = MSE(model.predict(features), target)

                 test['scores'].append(mse)

             test['scores'] = np.array(test['scores'])

             results = { 'description' : model_description }

             results['bias'] = test['scores'].mean()
             results['variance'] = test['scores'].std()

             return test, results
```

Listing 1.94: Get the results of running the test on the minimal model

```
In [12]: model_1 = "petal_width ~ 1"
         test_1, results_1 = test_func(model_1)
```

Listing 1.95: Display the results of the MSE scores on the minimal model

```
In [13]: test_1['scores']

Out[13]: array([[0.490656],
                [0.674404],
                [0.513156],
                [0.5229  ],
                [0.648164],
                [0.514644],
                [0.656464],
                [0.679104],
                [0.556804],
                [0.634244]])
```

Listing 1.96: Show how we are collecting results

```
In [14]: test_1['scores'].mean(), test_1['scores'].std()

Out[14]: (0.5890540000000001, 0.07202068898865105)
```

Listing 1.97: Display the results returned by the test

```
In [15]: results_1

Out[15]: {'description': 'petal_width ~ 1',
          'bias': 0.5890540000000001,
          'variance': 0.07202068898865105}
```

Listing 1.98: Define all models

```
In [16]: model_2 = "petal_width ~ 1 + petal_length"
         model_3 = "petal_width ~ 1 + sepal_length"
         model_4 = "petal_width ~ 1 + sepal_width"
         model_5 = "petal_width ~ 1 + petal_length + sepal_width"
         model_6 = "petal_width ~ 1 + petal_length + sepal_length"
         model_7 = "petal_width ~ 1 + sepal_length + sepal_width"
         model_8 = "petal_width ~ 1 + petal_length + sepal_length
         + sepal_width"
```

Listing 1.99: Execute the test on the other seven model and collect the results

```
In [17]: test_2, results_2 = test_func(model_2)
         test_3, results_3 = test_func(model_3)
         test_4, results_4 = test_func(model_4)
         test_5, results_5 = test_func(model_5)
         test_6, results_6 = test_func(model_6)
         test_7, results_7 = test_func(model_7)
         test_8, results_8 = test_func(model_8)
         results = [
             results_1,
             results_2,
             results_3,
             results_4,
             results_5,
             results_6,
             results_7,
             results_8
         ]
```

Listing 1.100: Turn the results into a `DataFrame` for ease of display

```
In [18]: results = pd.DataFrame(results)
```

Listing 1.101: Add colors to results

```
In [19]: results['color'] = ['red','orange', 'yellow', 'green',
                             'blue', 'cyan', 'purple' ,'black']
```

Listing 1.102: Display results

```
In [20]: results
```

Table 1.6: Display results

	bias	description	variance	color
0	0.589054	petal_width ~1	0.072021	red
1	0.037983	petal_width ~1 + petal_length	0.008209	orange
2	0.191311	petal_width ~1 + sepal_length	0.025984	yellow
3	0.482659	petal_width ~1 + sepal_width	0.091107	green
4	0.036313	petal_width ~1 + petal_length + sepal_width	0.007740	blue
5	0.036575	petal_width ~1 + petal_length + sepal_length	0.008074	cyan
6	0.141968	petal_width ~1 + sepal_length + sepal_width	0.028287	purple
7	0.032446	petal_width ~1 + petal_length + sepal_length ...	0.006924	black

Listing 1.103: Plot the bias versus the variance

```
In [21]: plt.figure(figsize=(10,5))

         for i in results.index:
             plt.scatter(
                 results.loc[i].bias,
                 results.loc[i].variance,
                 c=results.loc[i].color,
                 label=results.loc[i].description
             )

         plt.xlabel('Bias')
         plt.ylabel('Variance')

         plt.legend()

         plt.show()
```

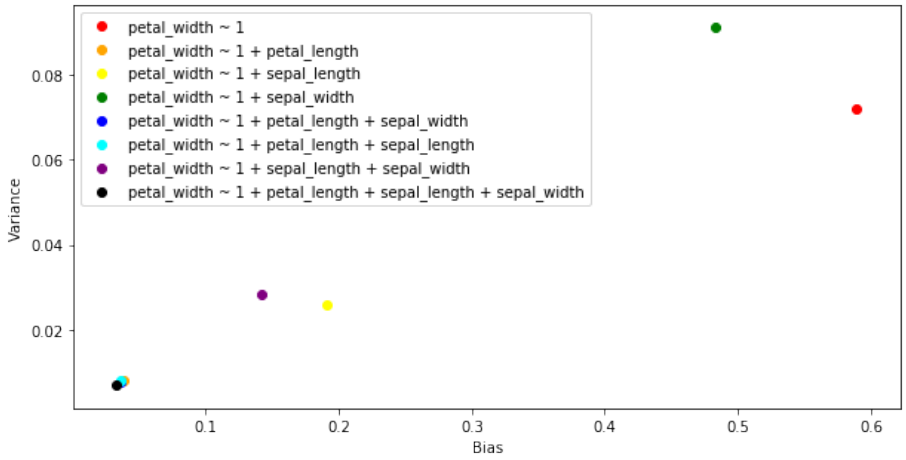

Figure 1.9: Plot the bias versus the variance

1.5 Probability

We are all familiar with the phrase "the probability that a coin will land heads is 0.5". But what does this mean? There are actually at least two different interpretations of probability. One is called the **frequentist** interpretation. In this view, probabilities represent long run frequencies of events. For example, the above statement means that, if we flip the coin many times, we expect it to land heads about half the time.

The other interpretation is called the **Bayesian** interpretation of probability. In this view, probability is used to quantify our uncertainty about something; hence it is fundamentally related to information rather than repeated trials. In the Bayesian view, the above statement means we believe the coin is equally likely to land heads or tails on the next toss.

One big advantage of the Bayesian interpretation is that it can be used to model our uncertainty about events that do not have long term frequencies. For example, we might want to compute the probability that the polar ice cap will melt by 2020 CE. This event will happen zero or one times, but cannot happen repeatedly. Nevertheless, we ought to be able to quantify our uncertainty about this event; based on how probable we think this event is, we will (hopefully!) take appropriate actions. To give some more machine learning oriented examples, we might have received a specific email message, and want to compute the probability it is spam. Or we might have observed a "blip" on our radar screen, and want to compute the probability distribution over the location of the corresponding target (be it a bird, plane, or missile). In all these cases, the

idea of repeated trials does not make sense, but the Bayesian interpretation is valid and indeed quite natural.

The basic rules of probability theory are the same, no matter which interpretation is adopted.

1.5.1 Basic Probability

The expression $p(A)$ denotes the probability that the event A is true. For example, A might be the logical expression "it will rain tomorrow".

We have:

- $0 \leq p(A) \leq 1$
- $p(A) = 0$ means the event definitely will not happen
- $p(A) = 1$ means the event definitely will happen
- $p(\neg A)$ denotes the probability of the event not A, that is that A will not occur
- $p(\neg A) = 1 - p(A)$
- We will often write $A = 1$ to mean the event A is true, and $A = 0$ to mean the event A is false

1.5.2 Discrete random variables

A **discrete random variable** X is a set of possible observed events. For example, we might have that X is the integer age of of the students in our class.

We can intuit that certainly $X \in [0, 100]$ (X is *in* the set of integers from 0 to 100). We might take a sample from X and this will signify the age of one member of the class. In terms of probability, we might think of the event $P(X = x)$, the probability that our sample is some number x. We can also call this simply $p(x)$. Assuming that no one in the class has the same integer age, we have an equal chance of sampling every student, and there are n students in the class, we could say $p(x) = \frac{1}{n}$. As with all probability, $0 \leq p(x) \leq 1$.

ADVANCED NOTE $p(x)$ is called a **probability mass function** or pmf.

1.5.3 Probability of Two Events Occurring

Given two events, A and B, we define the probability of A or B as follows:

$$p(A \vee B) = p(A) + p(B) - p(A \wedge B)$$

$$p(A \vee B) = p(A) + p(B) \text{if } A \text{ and } B \text{ are mutually exclusive}$$

1.5.4 Joint Probability

Joint probability refers to two events co-occurring.

$$p(A, B) = p(A \wedge B) = p(A|B)p(B) = p(B|A)p(A)$$

This is sometimes called **the product rule**. Note that in both $p(A|B)p(B)$ and $p(B|A)p(A)$, *both events are occurring.* You should read $p(A|B)p(B)$ as "the probability of of A given B times the probability of B".

1.5.5 Conditional Probability

$$p(A|B) = \frac{p(A, B)}{p(B)} \text{ if } p(B) > 0$$

1.5.6 Bayes Rule

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)} \text{ if } p(B) > 0$$

1.5.7 An Example: A Cancer Detection Test

Suppose a medical institution has developed a test for assessing whether or not a patient has cancer. The test has been around for a long time (meaning we can use frequentist statistics to measure its success) and we know that it is 98% successful in identifying cancer when a patient has cancer and 99% successful in returning a negative when a patient does not have cancer. We can rewrite each of these as a conditional probability:

$$p(\text{positive test}|\text{cancer}) = 0.99$$
$$p(\text{negative test}|\text{no cancer}) = 0.97$$

We also know that cancer in the American population is extremely rare. Approximately 0.4% of people develop cancer. We can thus say $p(\text{cancer}) = 0.004$. *NOTE: these numbers are made up for demonstration.*

1.5.8 Probability that a patient has cancer

What we wish to know is, given a positive test, what is the probability that the patient has cancer? What is $p(\text{cancer}|\text{positive test})$?

1.5.9 Bayes Rule

We can find this probability by calculating

$$p(\text{cancer}|\text{positive test}) = \frac{p(\text{positive test}|\text{cancer})p(\text{cancer})}{p(\text{positive test})}$$

The calculation is pretty straightforward, except for the calculation of $p(\text{positive test})$. This calculation must include all of the ways in which we can obtain a positive test. We have to include the false positives in the calculation. The false positive rate is $1 - p(\text{negative test}|\text{no cancer}) = 0.03$

$$\begin{aligned} p(\text{positive test}) &= p(\text{positive test}|\text{cancer})p(\text{cancer}) + p(\text{positive test}|\text{no cancer})p(\text{no cancer}) \\ &= 0.99 \cdot 0.004 + 0.03 * 0.999 \\ &= 0.03384 \end{aligned}$$

Then,

$$\begin{aligned} p(\text{cancer}|\text{positive test}) &= \frac{p(\text{positive test}|\text{cancer})p(\text{cancer})}{p(\text{positive test})} \\ &= \frac{0.99 \cdot 0.004}{0.03384} \\ &= 0.11702 \end{aligned}$$

1.5.10 The Counter-intutition of False Positives

Below we visualize a population of 1000 patients to whom this test has been administered. In 1000 patients, we would expect 996 of them to be cancer-free. But according to the test, with 996 patients, we would expect 30 **false positives**. In the same population, we would expect 4 patients to actually have cancer. Luckily, we would expect the test to correctly identify all four of these patients. This would be a total of 34 positive tests, the vast majority of these being false positives.

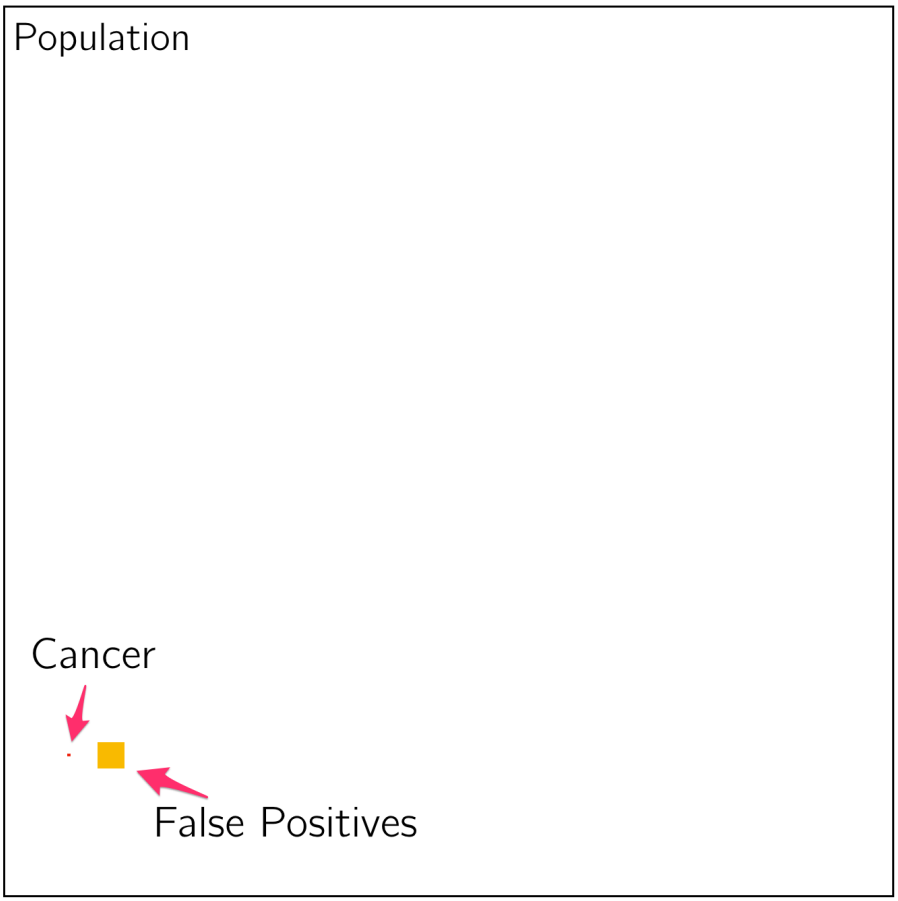

Figure 1.10: A population of 1000 patients

We might also look at these results using a **confusion matrix**

	True Positive	True Negative
Predicted Positive	4	30
Predicted Negative	0	968

In this particular case, this result may be preferable to lowering the sensitivity of our test to lower the false positive rate, but at the expense of missing true positives. One can imagine a situation in which the opposite were true so that we would want to lower the false positive rate at the expense of missing true positives. For example, we might consider a test to see if a patient is a match for a certain kind of organ donation. In this case, it is preferable that every

positive match is a true positive at the expense of possibly missing one.

1.6 Probabilistic Model Selection / In-Sample Model Selection

From a probabilistic perspective we seek the model estimate $\widehat{f}$ that is most probable given the data. Let H_f be the hypothesis that a specific model is the correct model and $p(D)$ represent the probability of the data, then we seek $\widehat{f}$ that maximizes, that is we seek

$$\widehat{f} = \underset{f}{\operatorname{argmax}} \left(p(H_f|D) \right)$$

We can use Bayes' Rule to invert this probability so that

$$\widehat{f} = \underset{f}{\operatorname{argmax}} \left(\frac{p(D|H_f)p(H_f)}{p(D)} \right)$$

If we consider that each model is equally likely and that the probability of the data $p(D)$ is a constant applied to every calculation, then without loss of generality, we can say

$$\widehat{f} = \underset{f}{\operatorname{argmax}} \left(p(D|H_f) \right)$$

In the Regression setting, it can be shown that maximizing this equation is equivalent to minimizing the residual sum of squares,

$$\text{RSS} = \sum (y_i - \widehat{y})^2 = (\mathbf{y} - \widehat{\mathbf{y}})^T (\mathbf{y} - \widehat{\mathbf{y}})$$

1.6.1 Generating Model Hypotheses

For this particular task we will consider the hypothesis space to be all of the linear models possible given our data sets. There will be eight possible models:

$$
\begin{aligned}
\widehat{f}_1 &= \beta_0 \\
\widehat{f}_2 &= \beta_0 + \beta_1 x_1 \\
\widehat{f}_3 &= \beta_0 + \beta_1 x_2 \\
\widehat{f}_4 &= \beta_0 + \beta_1 x_3 \\
\widehat{f}_5 &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 \\
\widehat{f}_6 &= \beta_0 + \beta_1 x_1 + \beta_2 x_3 \\
\widehat{f}_7 &= \beta_0 + \beta_1 x_2 + \beta_2 x_3 \\
\widehat{f}_8 &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3
\end{aligned}
$$

Essentially we are forming what is known as a Power Set of possible feature combinations. The number of elements in a Power Set is given by 2^p where p is the number of elements in the set.

For $p = 2$, this is four models. For $p = 3$, this is eight models.

For $p = 100$, this is a hypothesis space With a dimension of 1.27×10^{30}. If we trained one model per second, it would take us 4.02×10^{22} years to search the entire hypothesis space. For perspective, physicists estimate that the universe is approximately 13.8×10^9 years old.

In other words, we will rarely be able to exhaustively search a hypothesis space.

Listing 1.104: Load Iris Dataset

```
In [1]: iris.data = read.csv("data/iris.csv", row.names='X')
```

Listing 1.105: Display the first n rows of the `DataFrame`

```
In [2]: head(iris.data)
```

Table 1.8: Display the first n rows of the `DataFrame`

	sepal_length	sepal_width	petal_length	petal_width	label
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0
5	5.4	3.9	1.7	0.4	0

Listing 1.106: Install the Package `GGally`

```
In [3]: install.packages("GGally")

Out[3]: Updating HTML index of packages in '.Library'
        Making 'packages.html' ... done
```

Listing 1.107: Load the library `GGally`

```
In [4]: library(GGally)

Out[4]: Loading required package: ggplot2
```

Listing 1.108: Display `ggpairs` for the Iris Dataset

```
In [5]: ggpairs(iris.data)
```

Listing 1.109: Perform a Logistic Regression on the Species Label

```
In [6]: iris.glm = glm("label ~ 1 +
                                sepal_length +
                                sepal_width +
                                petal_length +
                                petal_width", data = iris.data)
```

Listing 1.110: Display the `summary` of the Logistic Regression

```
In [7]: summary(iris.glm)

Out[7]:
        Call:
        glm(formula = "label ~ 1 + sepal_length + sepal_width +
        petal_length + petal_width",
            data = iris.data)

        Deviance Residuals:
             Min        1Q    Median        3Q       Max
        -0.59046  -0.15230   0.01338   0.10332   0.55061

        Coefficients:
                     Estimate Std. Error t value Pr(>|t|)
        (Intercept)   0.19208    0.20470   0.938 0.349611
        sepal_length -0.10974    0.05776  -1.900 0.059418 .
        sepal_width  -0.04424    0.05996  -0.738 0.461832
        petal_length  0.22700    0.05699   3.983 0.000107 ***
        petal_width   0.60989    0.09447   6.456 1.52e-09 ***
        ---
        Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1
        ' ' 1

        (Dispersion parameter for gaussian family taken to be
        0.04798457)

            Null deviance: 100.0000  on 149  degrees of freedom
        Residual deviance:   6.9578  on 145  degrees of freedom
        AIC: -22.935

        Number of Fisher Scoring iterations: 2
```

1.6.2 The Log-Likelihood

Without going too far into the math, we can think of the log-likelihood as a **likelihood function** telling us how likely a model is given the data.

This value is not human interpretable but is useful as a comparison.

Listing 1.111: Calculate the Log-Likelihood of the Logistic Regression

```
In [8]: logLik(iris.glm)

Out[8]: 'log Lik.' 17.46751 (df=6)
```

"All models are wrong, but some are useful." - George Box

William of Occam

We might be concerned with one additional property - the **complexity** of the model. **Occam's razor** is the problem-solving principle that, when presented with competing hypothetical answers to a problem, one should select the one that makes the fewest assumptions.

Figure 1.11: William of Occam

We can represent this idea of complexity in terms of both the number of features we use and the amount of data.

1.6.3 Bayesian Information Criterion

The BIC is formally defined as

$$\text{BIC} = \ln(n)k - 2\ln(\widehat{L}).$$

where

- $\widehat{L}$ = the maximized value of the likelihood function of the model M
- x = the observed data
- n = the number of data points in x, the number of observations, or equivalently, the sample size;
- k = the number of parameters estimated by the model. For example, in multiple linear regression, the estimated parameters are the intercept, the q slope parameters, and the constant variance of the errors; thus, $k = q+2$.

It might help us to think of it as

$$\text{BIC} = \text{complexity} - \text{likelihood}$$

1.6.4 Akaike Information Criterion

The AIC is formally defined as

$$\text{AIC} = 2k - 2\ln(\widehat{L}).$$

where

- $\widehat{L}$ = the maximized value of the likelihood function of the model M
- x = the observed data
- n = the number of data points in x, the number of observations, or equivalently, the sample size;
- k = the number of parameters estimated by the model. For example, in multiple linear regression, the estimated parameters are the intercept, the q slope parameters, and the constant variance of the errors; thus, $k = q+2$.

It might help us to think of it as

$$\text{AIC} = \text{complexity} - \text{likelihood}$$

We can think of - likelihood as bias and complexity as variance.

Listing 1.112: Calculate the BIC using R

```
In [9]: BIC(iris.glm)

Out[9]: [1] -4.871215
```

Listing 1.113: Calculate the BIC manually

```
In [10]: n = length(iris.glm$fitted.values)
         p = length(coefficients(iris.glm))

         likelihood = 2 * logLik(iris.glm)
         complexity = log(n)*(p+1)

         bic = complexity - likelihood
         bic

Out[10]: 'log Lik.' -4.871215 (df=6)
```

Listing 1.114: Define a function to calcuate the BIC

```
In [11]: BIC_of_model = function (model) {
             n = length(model$fitted.values)
             p = length(coefficients(model))

             likelihood = 2 * logLik(model)
             complexity = log(n)*(p+1)

             bic = complexity - likelihood
             return(bic)
         }
```

Listing 1.115: Calculate the BIC with function

```
In [12]: BIC_of_model(iris.glm)

Out[12]: 'log Lik.' -4.871215 (df=6)
```

1.6.5 Model Selection

Here, we choose the optimal model by removing features one by one.

Listing 1.116: Define full model

```
In [13]: model_1  = "label ~ 1 +
                              sepal_length +
                              sepal_width +
                              petal_length +
                              petal_width"
```

Listing 1.117: Define four models each with one feature removed

```
In [14]: model_2a = gsub('\\+ petal_width','', model_1)
         model_2b = gsub('\\+ petal_length', '', model_1)
         model_2c = gsub('\\+ sepal_width', '', model_1)
         model_2d = gsub('\\+ sepal_length', '', model_1)
```

model_2a

Listing 1.118: Perform Logistic Regression on these models

```
In [16]: iris.glm.1 = glm(model_1, data=iris.data)
         iris.glm.2a = glm(model_2a, data=iris.data)
         iris.glm.2b = glm(model_2b, data=iris.data)
         iris.glm.2c = glm(model_2c, data=iris.data)
         iris.glm.2d = glm(model_2d, data=iris.data)
```

Listing 1.119: Display the results of the BIC on these models

```
In [17]: print(c('model_1', BIC_of_model(iris.glm.1)))
         print(c('model_2a', BIC_of_model(iris.glm.2a )))
         print(c('model_2b', BIC_of_model(iris.glm.2b )))
         print(c('model_2c', BIC_of_model(iris.glm.2c )))
         print(c('model_2d', BIC_of_model(iris.glm.2d )))

Out[17]: [1] "model_1"           "-4.87121487462612"
         [1] "model_2a"          "28.0137935908893"
         [1] "model_2b"          "5.69337438932066"
         [1] "model_2c"           "-9.31979403027607"
         [1] "model_2d"          "-6.1930960954627"
```

Listing 1.120: Display the results of the BIC on these models using R

```
In [18]: print(c('model_1', BIC(iris.glm.1)))
         print(c('model_2a', BIC(iris.glm.2a )))
         print(c('model_2b', BIC(iris.glm.2b )))
         print(c('model_2c', BIC(iris.glm.2c )))
         print(c('model_2d', BIC(iris.glm.2d )))

Out[18]: [1] "model_1"           "-4.87121487462612"
         [1] "model_2a"          "28.0137935908893"
         [1] "model_2b"          "5.69337438932066"
         [1] "model_2c"           "-9.31979403027607"
         [1] "model_2d"          "-6.1930960954627"
```

Listing 1.121: Define three models with two features removed

```
In [19]: model_3a = gsub('\\+ petal_width','', model_2c)
         model_3b = gsub('\\+ petal_length','', model_2c)
         model_3c = gsub('\\+ sepal_length','', model_2c)
```

Listing 1.122: Perform Logistic Regressions

```
In [20]: iris.glm.3a = glm(model_3a, data=iris.data)
         iris.glm.3b = glm(model_3b, data=iris.data)
         iris.glm.3c = glm(model_3c, data=iris.data)
```

Listing 1.123: Display the results of the BIC using R

```
In [21]: print(c('model_1', BIC(iris.glm.1)))
         print(c('model_2c', BIC(iris.glm.2c )))
         print(c('model_3a', BIC(iris.glm.3a )))
         print(c('model_3b', BIC(iris.glm.3b )))
         print(c('model_3c', BIC(iris.glm.3c )))

Out[21]: [1] "model_1"            "-4.87121487462612"
         [1] "model_2c"           "-9.31979403027607"
         [1] "model_3a"          "25.3174210943167"
         [1] "model_3b"          "15.4504250116728"
         [1] "model_3c"          "-5.0467304546584"
```

Listing 1.124: Display best three feature model

```
In [22]: model_2c

Out[22]: [1] "label ~ 1 + sepal_length  + petal_length +
         petal_width"
```

Listing 1.125: Display best two feature model

```
In [23]: model_3c

Out[23]: [1] "label ~ 1   + petal_length + petal_width"
```

Listing 1.126: Display Log-Likelihood's of Models

```
In [24]: print(c('model_1', logLik(iris.glm.1)))
         print(c('model_2c', logLik(iris.glm.2c )))
         print(c('model_3a', logLik(iris.glm.3a )))
         print(c('model_3b', logLik(iris.glm.3b )))
         print(c('model_3c', logLik(iris.glm.3c )))

Out[24]: [1] "model_1"           "17.4675133196018"
         [1] "model_2c"          "17.1864852503787"
         [1] "model_3a"           "-2.63743995896586"
         [1] "model_3b"          "2.29605808235611"
         [1] "model_3c"          "12.5446358155217"
```

Listing 1.127: Display the results of the AIC using R

```
In [25]: print(c('model_1', AIC(iris.glm.1)))
         print(c('model_2a', AIC(iris.glm.2a )))
         print(c('model_2b', AIC(iris.glm.2b )))
         print(c('model_2c', AIC(iris.glm.2c )))
         print(c('model_2d', AIC(iris.glm.2d )))
         print(c('model_3a', AIC(iris.glm.3a )))
         print(c('model_3b', AIC(iris.glm.3b )))
         print(c('model_3c', AIC(iris.glm.3c )))

Out[25]: [1] "model_1"            "-22.9350266392037"
         [1] "model_2a"          "12.960617120408"
         [1] "model_2b"            "-9.35980208116062"
         [1] "model_2c"            "-24.3729705007573"
         [1] "model_2d"           "-21.246272565944"
         [1] "model_3a"           "13.2748799179317"
         [1] "model_3b"           "3.40788383528778"
         [1] "model_3c"            "-17.0892716310434"
```

1.7 Using Masks to Create a Vector

1.7.1 A Vectorized Solution To `fizzbuzz`

`fizzbuzz` is a canonical "coding interview" problem. You might want to read Joel Grus' humorous attempt to use tensorflow to solve `fizzbuzz`[1]. The challenge is to iterate over the numbers from 1 to 100, printing "fizz" if the number is divisible by 3, "buzz" if the number is divisible by 5, "fizzbuzz" if the number is divisible by 15, and the number itself otherwise. Typically this problem is solved using for-loops and if-else statements and is used as a basic assessment of programming ability. Such a solution might look like this

[1] http://joelgrus.com/2016/05/23/fizz-buzz-in-tensorflow/

Listing 1.128: a first attempt at `fizzbuzz`

```
In [1]: fizzbuzz = function (n) {
            for (i in 1:n) {
                if (i %% 15 == 0) {
                    cat("fizzbuzz", sep="\n")
                }
                else if (i %% 5 == 0) {
                    cat("buzz", sep="\n")
                }
                else if (i %% 3 == 0) {
                    cat("fizz", sep="\n")
                }
                else {
                    cat(i, sep="\n")
                }
            }
        }
```

Listing 1.129: display `fizzbuzz` for $n = 15$

```
In [2]: fizzbuzz(15)

Out[2]: 1
        2
        fizz
        4
        buzz
        fizz
        7
        8
        fizz
        buzz
        11
        fizz
        13
        14
        fizzbuzz
```

It may be a bit much to come up with a solution to this problem using tensorflow. It is, however, very useful to think about solving this problem using masks and filters. Suppose we begin with a simple solution vector as follows

Listing 1.130: start the `solution` vector

```
In [3]: solution = 1:15
        solution

Out[3]:  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

The challenge is to replace the values we don't need with the correct strings. Sure we can iterate over this list check the value to see if it's divisible by three or five but using a vectorized solution we can do it all at once.

The Steps to doing this are as follows:

1. Create a mask for a certain condition we might wish to check
2. Use that mask to restrict the values of the original `solution` we are looking at
3. Replace to values of the restricted vector with the appropriate string

First, we create a mask called `mod15_mask`. Note, that when we display it there is only a single `TRUE` value, in the position where the value is divisible by 15 (and in this case is actually 15).

Listing 1.131: display a mod 15 mask

```
In [4]: solution %% 15 == 0

Out[4]:  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
         FALSE FALSE FALSE FALSE
         [13] FALSE FALSE  TRUE
```

Listing 1.132: create the mod 15 mask

```
In [5]: mod15_mask = solution %% 15 == 0
         mod15_mask

Out[5]:  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
         FALSE FALSE FALSE FALSE
         [13] FALSE FALSE  TRUE
```

Next, we filter the `solution` using the `mod15_mask`.

Listing 1.133: filter `solution` using the mod 15 mask

```
In [6]: solution[mod15_mask]

Out[6]: [1] 15
```

Finally, we assign the filtered values the string `"fizzbuzz"`

Listing 1.134: assign values to the filtered `solution` vector

```
In [7]: solution[mod15_mask] = "fizzbuzz"
```

Let's have a look at the current value of our solution.

Listing 1.135: display `solution` vector

```
In [8]: solution

Out[8]:  [1] "1"         "2"         "3"         "4"         "5"
           "6"
         [7] "7"         "8"         "9"         "10"        "11"
           "12"
        [13] "13"        "14"        "fizzbuzz"
```

We can repeat this technique to build an entire solution to the problem.

Listing 1.136: a vectorized `fizzbuzz`

```
In [9]: fizzbuzz = function (n) {
            solution = 1:n
            mod3_mask = (solution %% 3 == 0)
            mod5_mask = (solution %% 5 == 0)
            mod15_mask = (solution %% 15 == 0)

            solution[mod3_mask] = "fizz"
            solution[mod5_mask] = "buzz"
            solution[mod15_mask] = "fizzbuzz"

            cat(solution,sep="\n")
        }

        fizzbuzz(15)

Out[9]: 1
        2
        fizz
        4
        buzz
        fizz
        7
        8
        fizz
        buzz
        11
        fizz
        13
        14
        fizzbuzz
```

In terms of the why of doing a vectorized approach, there are tremendous speed gains to be had implementing your algorithms using vectors rather than loops[2].

1.8 The Curse of Dimensionality

Listing 1.137: Define Numerical Python environment

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        %matplotlib inline
```

1.8.1 The difference between `np.linspace` and `np.logspace`

Listing 1.138: Generate an array from 1 to 4 with 10 values

```
In [2]: np.linspace(1,4,10)

Out[2]: array([1.        , 1.33333333, 1.66666667, 2.        ,
        2.33333333,
               2.66666667, 3.        , 3.33333333, 3.66666667, 4.
                ])
```

Listing 1.139: Generate an array from 10^1 to 10^4 with 10 values

```
In [3]: np.logspace(1,4,10)

Out[3]: array([   10.        ,    21.5443469 ,    46.41588834,
        100.        ,
                 215.443469  ,   464.15888336,  1000.        ,
        2154.43469003,
                4641.58883361, 10000.        ])
```

[2]http://www.noamross.net/blog/2014/4/16/vectorization-in-r–why.html

Listing 1.140: Define `datasets` of varying size using `make_classification`

```
In [4]: from sklearn.datasets import make_classification

        n_vals = np.logspace(1, 4, 20)

        datasets = {
            int(n): make_classification(int(n))
            for n in n_vals
        }
```

Listing 1.141: Load models from Scikit-Learn

```
In [5]: from sklearn.neighbors import KNeighborsClassifier
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.linear_model import LogisticRegression
        from sklearn.svm import SVC
```

Listing 1.142: Define dictionary of models

```
In [6]: models = {
            'knn' : KNeighborsClassifier(),
            'lr' : LogisticRegression(),
            'dt' : DecisionTreeClassifier(),
            'svm' : SVC(),
        }
```

Listing 1.143: import `time` library

```
In [7]: from time import time
```

https://stackoverflow.com/questions/2322355/proper-name-for-python-operator

Listing 1.144: Define functions to time fit and prediction

```
In [8]: def time_fit(data, model):
            start = time()
            model.fit(*data)
            fit_time = time() - start
            return fit_time

        def time_predict(data, model):
            start = time()
            model.predict(data[0])
            predict_time = time() - start
            return predict_time
```

Listing 1.145: Define function to test all models on a dataset of size n

```
In [9]: def fit_predict_model_on_n(model_name, n):

            model = models[model_name]
            dataset = datasets[n]

            fit_time = time_fit(dataset, model)

            predict_time = time_predict(dataset, model)

            return {
                'model_name' : model_name,
                'n' : n,
                'fit_time' : fit_time,
                'predict_time' : predict_time
            }
```

Listing 1.146: Run all models against all datasets

```
In [10]: results = []
         for n in datasets.keys():
             for model in models.keys():
                 results.append(fit_predict_model_on_n(model, n))
```

Listing 1.147: Run all models against all datasets as a list comprehension

```
In [11]: results = [
             fit_predict_model_on_n(model, n)
             for n in datasets.keys()
             for model in models.keys()
         ]
```

Listing 1.148: Collect results in a `DataFrame`

```
In [12]: results_df = pd.DataFrame(results)
```

Listing 1.149: Display a sample of the results

```
In [13]: results_df.sample(5)
```

Table 1.9: Display a sample of the results

	fit_time	model_name	n	predict_time
12	0.000189	knn	29	0.000335
63	0.085971	svm	2335	0.060358
17	0.000497	lr	42	0.000056
3	0.000429	svm	10	0.000079
13	0.000418	lr	29	0.000051

Listing 1.150: Plot Fit Time and Prediction Time

```
In [14]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,5))

         for model_name in results_df.model_name.unique():

             model_mask = (results_df.model_name == model_name)
             model_results_df = results_df[model_mask]

             ax1.set_title('Fit Time')
             ax1.plot(model_results_df.n,
                      model_results_df.fit_time,
                      label=model_name)
             ax1.set_ylim((0,5))

             ax1.legend()

             ax2.set_title('Prediction Time')
             ax2.set_ylim((0, 5))
             ax2.plot(model_results_df.n,
                      model_results_df.predict_time,
                      label=model_name)

             ax2.legend()

         plt.show()
```

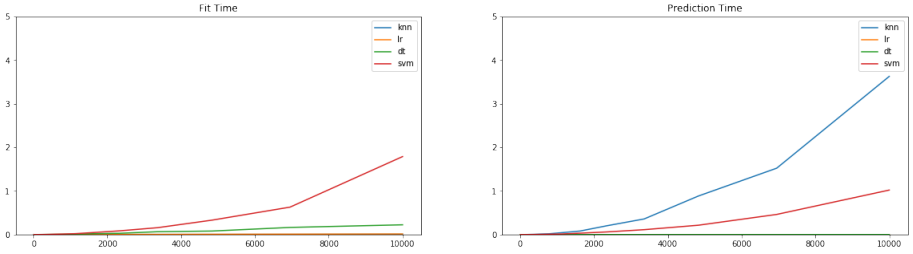

Figure 1.12: Plot Fit Time and Prediction Time

Listing 1.151: Plot Fit Time and Prediction Time with Logarithmic Scale for n

```
In [15]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,5))

         for model_name in results_df.model_name.unique():

             model_mask = (results_df.model_name == model_name)
             model_results_df = results_df[model_mask]

             ax1.set_title('Fit Time')
             ax1.plot(model_results_df.n,
                      model_results_df.fit_time,
                      label=model_name)
             ax1.set_xscale('log')
             ax1.set_ylim((0, 5))

             ax1.legend()

             ax2.set_title('Prediction Time')
             ax2.plot(model_results_df.n,
                      model_results_df.predict_time,
                      label=model_name)
             ax2.set_xscale('log')
             ax2.set_ylim((0, 5))

             ax2.legend()

         plt.show()
```

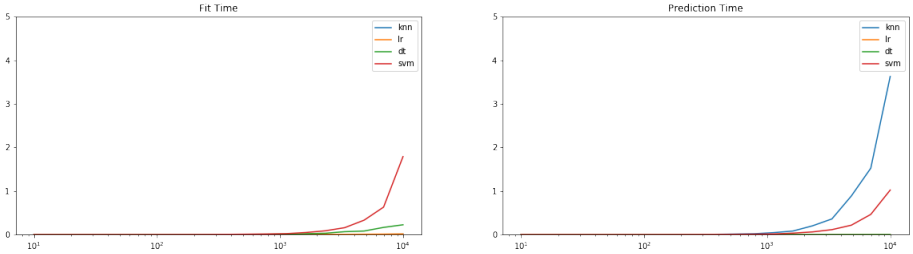

Figure 1.13: Plot Fit Time and Prediction Time with Logarithmic Scale for (n)

1.9 Revisting the bias-variance tradeoff

The aim of this lecture is to visually explore the concept of bias and variance in modeling and the tradeoff between the two.

Below we will be fitting models predicting student morale from the day number in the course.

Let's assume that the Bruin gods model student morale as such according to a cubic polynomial interpolated from the following data, one data point for

each week:

Listing 1.152: True measurements of morale

```
In [1]: moralepoints = [
            20, 30, 35, 18, 3, 12,
            35, 44, 53, 62, 73
        ]
```

Listing 1.153: Define Numerical Python environment

```
In [2]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        from scipy.interpolate import interp1d
        from sklearn.linear_model import LinearRegression
        from sklearn.preprocessing import PolynomialFeatures
        from sklearn.pipeline import Pipeline

        np.random.seed(42)

        %matplotlib inline
```

Listing 1.154: Define vector of days

```
In [3]: weeks = 10
        days = np.arange(weeks*7+1)
```

1.9.1 Actual Data According to the Universe

Listing 1.155: Create tuples of weekpoints and measurements of morale

```
In [4]: weekpoints = 7*np.arange(weeks+1)

        list(zip(weekpoints, moralepoints))

Out[4]: [(0, 20),
         (7, 30),
         (14, 35),
         (21, 18),
         (28, 3),
         (35, 12),
         (42, 35),
         (49, 44),
         (56, 53),
         (63, 62),
         (70, 73)]
```

1.9.2 Morale function as Interpolated by the Bruin Gods

The `interp1d` function creates an interpolation function for us between the week numbers and morale points. The `kind='cubic'` parameter indicates the smoothing of the interpolation.

Listing 1.156: Build interpolation function using weekpoints and measurements of morale

```
In [5]: morale_func = interp1d(weekpoints,
                               moralepoints,
                               kind='cubic')
```

We can plot out the "true function" of days predicting morale below:

Listing 1.157: Compute true values associated with days

```
In [6]: morale_true = morale_func(days)
```

Listing 1.158: Plot true morale over the span of a course

```
In [7]: plt.figure(figsize=(20, 5))

        plt.plot(days, morale_true,
                 linewidth=5, color='gold',
                 alpha=0.5, label='true function')

        plt.xlabel('days', fontsize=16)
        plt.ylabel('morale', fontsize=16)
        plt.title('Morale over time', fontsize=20)

        plt.legend(loc='upper left')

        plt.show()
```

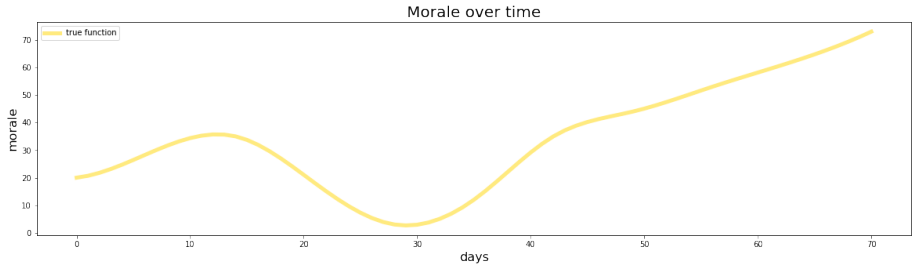

Figure 1.14: Plot true morale over the span of a course

Important Points

1. We can never know this function.
2. We can only model this function.
3. In building a model, we will introduce error.
4. A model will have three kinds of error:

$$\text{Error} = \text{Error from Bias} + \text{Error from Variance} + \text{Unknown Error}$$

5. All three of these are strictly non-negative.

The goal when building a model is to minimize the error from bias *and* the error from variance.

The perfect model would have zero bias and zero variance. In reality, the bias or the variance (typically) increases while the other decreases.

We can interpret this "true function" in different ways:

1. **As morale without individual variance:** the true function is the baseline morale for each time point that all students vary around to some degree. Students' morale at any given time point is this baseline morale plus or minus some individual deviation. When coding this in python, for each "student" we take the *true* measure of morale from the morale function, and then to create the individual variance we will add random noise to that starting point.
2. **As morale without measurement error:** maybe all student morale at every timepoint is the same, it's just that our way of measuring morale is unreliable. The true function represents the morale at any timepoint for any student without measurement error. If our measurement tool was perfect (a perfect morale survey) we would measure the same morale for every student at each time point. But if the survey form is broken and randomly changes answers by students, for example, then we are adding noise to the measured morale for each observation.
3. **As an average morale across infinite students:** our measurements of morale vary at each time point for each student, but if we had an infinite number of students and averaged all of their morale measurements across all time points, we would end up with the true function of `morale(time)`.

In each case, we are describing morale as a function of time without error, where each is describing a different source of error:

1. Error resulting from an imperfect relationship between time and morale.

2. Error resulting from an imperfect ability to measure morale.
3. Error resulting from an insufficient amount of data to quantify the relationship correctly.

We will see that each of these sources of error combined make up the full error in any model we build. Respectively:

1. The **bias**.
2. The **variance**.
3. The **irreducible error**.

You will always have error in your models, it just depends how much and what proportion of each type. We will formalize these components of error in a model further down.

1.9.3 Generate a sample of students

Say we have four students: student A, B, C, and D.

Each student has had their morale checked 12 different times throughout the course, but not necessarily at the same times.

Below is a function that will generate the days and morale for each student as a dictionary object.

Note: here we are using the morale function to get the true morale for a given day, then adding the individual variance and/or measurement error from a normal distribution.

Listing 1.159: Define a dictionary of measured morale for four students

```
In [8]: students = {}
        for student in ['A','B','C','D']:

            daysamp  = np.random.choice(days, 12)
            morales  = morale_func(daysamp)
            morales += np.random.normal(0, 13, 12)

            students[student] = {
                'days' : daysamp,
                'morale': morales
            }
```

1.9.4 Student A's morale over time

Below we can plot student A's morale at each day. The true function is also plotted in yellow.

Listing 1.160: Plot Measured Values for Student A versus True Morale

```
In [9]: plt.figure(figsize=(20, 5))

        plt.plot(days, morale_true,
                 linewidth=5, color='gold',
                 alpha=0.5, label='true function')

        plt.scatter(students['A']['days'],
                    students['A']['morale'],
                    s=70, c='darkred',
                    label='student A', alpha=0.7)

        plt.xlabel('days', fontsize=16)
        plt.ylabel('morale', fontsize=16)
        plt.title('Morale over time\n', fontsize=20)
        plt.xlim([0, 72])

        plt.legend(loc='upper left')

        plt.show()
```

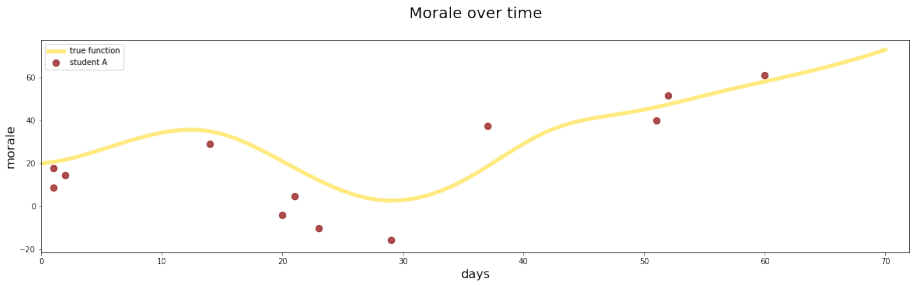

Figure 1.15: Plot Measured Values for Student A versus True Morale

1.9.5 Build a model for days predicting morale using student A's data

With this student's data, I decide to model the relationship between days and morale with a linear regression.

My model is:

$$\hat{morale} = \beta_0 + \beta_1 days$$

Construct the model:

Listing 1.161: Construct a Linear Regression model for Student A

```
In [10]: studA_days = students['A']['days']
         studA_mor = students['A']['morale']

         Amod = LinearRegression()
         Amod.fit(studA_days[:, np.newaxis], studA_mor)

Out[10]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs
         =1, normalize=False)
```

Listing 1.162: Plot the modeled relationship between days and morale

```
In [11]: plt.figure(figsize=(20, 5))

         plt.plot(days, morale_true,
                  linewidth=5, color='gold',
                  alpha=0.5, label='true function')

         plt.plot(days, Amod.predict(days[:, np.newaxis]),
                  lw=7., c='darkred',
                  alpha=0.5, label='model')

         plt.scatter(students['A']['days'],
                     students['A']['morale'],
                     s=70, c='darkred',
                     label='student A', alpha=0.7)

         plt.xlabel('days', fontsize=16)
         plt.ylabel('morale', fontsize=16)
         plt.title('Morale over time\n', fontsize=20)
         plt.xlim([0, 72])

         plt.legend(loc='upper left')

         plt.show()
```

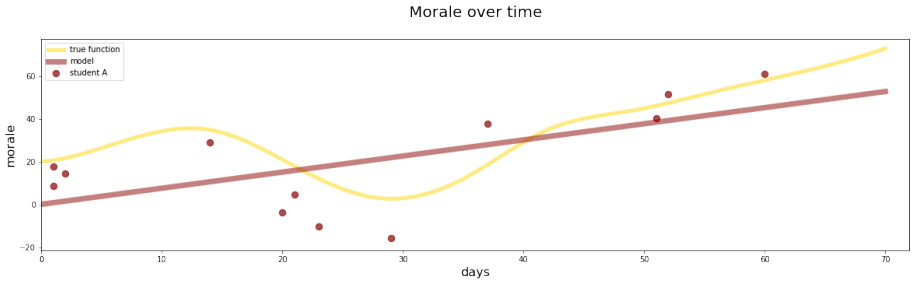

Figure 1.16: Plot the modeled relationship between days and morale

1.9.6 The total error of a model

As you can see above, our regression line is an imperfect representation of the true function. Furthermore, the regression model fails to perfectly capture the variance of the morale in our sample data.

The model is our blueprint for the estimation of morale. We have chosen to estimate morale from simply the number of days that have elapsed in the course. In doing so, we have made an assumption: morale is a linear function of days.

When we talk about the bias-variance tradeoff in modeling, and the bias and variance components of error, it is important to think about this in the context of building our model on many samples of the data. For example, we take our model parameterization `morale ~ days` and build this on the data for student A, then student B, then student C, then student D, and so on. Think of the "students" as random samples of morale and days elapsed from the overall population.

There are three sources of error in a model:

$$\text{Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

$$E\Big[\big(\hat{f}(x) - f(x)\big)^2\Big] = \big(\,\text{E}[\,\hat{f}(x)\,] - f(x)\,\big)^2 + \text{E}\big[\,\big(\,\hat{f}(x) - \text{E}[\,\hat{f}(x)\,]\big)^2\,\big] + E\Big[\big(y - f(x)\big)^2\Big]$$

Where:

- $f(x)$ is the true function of y given the predictors.
- $\hat{f}(x)$ is the estimate of y with the model fit on a random sample of the predictors.
- $E\Big[\big(\hat{f}(x) - f(x)\big)^2\Big]$ is the average squared error across multiple models fit on different random samples between the model and the true function.
- $\text{E}[\,\hat{f}(x)\,]$ is the average of estimates for given predictors across multiple models fit on different random samples.
- $E\Big[\big(y - f(x)\big)^2\Big]$ is the average squared error between the true values and the predictions from the true function of the predictors. This is the **irreducible error**.
- $\big(\,\text{E}[\,\hat{f}(x)\,] - f(x)\,\big)^2$ is the squared error between the average predictions across multiple models fit on different random samples and the prediction of the true function. This is the **bias** (squared).

- $\mathrm{E}\big[\ (\ \hat{f}(x) - \mathrm{E}[\ \hat{f}(x)\ ])^2\ \big]$ is the average squared distance between individual model predictions and the average prediction of models across multiple random samples. This is the **variance**.

The irreducible error is "noise" – error in the measurement of our target that cannot be accounted for by our predictors.

- The true function represents the most perfect relationship between predictors and target, but that does not mean that our variables can perfectly predict the target.
- The irreducible error can be thought of as the measurement error: variation in the target that we cannot represent.

We will go into the bias and variance components individually, in more detail.

1.9.7 Bias

The $bias^2$ is the source of error in our model that represents how *oversimplified* our model is. In our example, we have built a function to predict the morale of students as a linear function of days elapsed. However, we can see from the true function that the relationship between days and morale is not a line. This error is encapsulated in the bias.

$$Bias^2 = \big(\ \mathrm{E}[\ \hat{f}(x)\ ] - f(x)\ \big)^2$$

Remember: we have to think about measuring the bias and variance using different fits of our model across multiple random samples! The bias represents the deviation of the *average predictions across models* from the true function.

What does having a high vs. low bias mean?

- If our models are consistently wrong, then the bias will be large.
- Alternatively, if our models are consistently correct then the bias will be small.
- Bias will be small if the errors across our models built on random samples and tested using the same predictors are incorrect in different directions that average out close to 0.

Linear methods like regression tend to have a high bias because we construct a simplification of the true function.

Below we can plot the error differences between the true function and our model built on student A's data.

No matter how many lines we fit on different students, those regression lines are never going to average together into the nonlinear true function! Our average estimate across models will always deviate from the true function somewhere across the days.

Listing 1.163: Morale over time (difference from true function)

```
In [12]: plt.figure(figsize=(20, 5))

         plt.plot(days, morale_true,
                  linewidth=5, color='gold',
                  alpha=0.5, label='true function')

         predictions = Amod.predict(days[:, np.newaxis])

         plt.plot(days, predictions,
                  lw=7., c='darkred',
                  alpha=0.5, label='model')

         plt.scatter(students['A']['days'],
                     students['A']['morale'],
                     s=70, c='darkred',
                     label='student A', alpha=0.7)

         for d in students['A']['days']:
             p = Amod.predict(d)
             plt.plot([d, d],
                      [p, morale_func(d)],
                      c='black', lw=3.)

         plt.xlabel('days', fontsize=16)
         plt.ylabel('morale', fontsize=16)
         plt.xlim([0, 72])

         plt.legend(loc='upper left')

         plt.show()
```

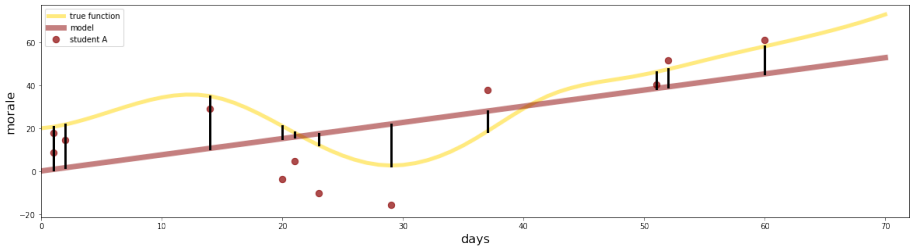

Figure 1.17: Morale over time (difference from true function)

1.9.8 Variance

The second component of error in the model is the variance of our predictions.

Variance describes the extent to which the individual predictions from models built on different samples (students) deviate from the mean of all the model predictions.

$$\text{Variance} = \text{E}\big[\ (\ \hat{f}(x) - \text{E}[\ \hat{f}(x)\ ])^2\ \big]$$

High vs. low variance

- The variance will be large if, for the same observation, models built on different random samples of the data will produce very different predictions.
- Variance is a measure of how *consistent* our model's predictions will be if it were fit on another sample of data.
- Variance is low if the data we train the model on has very little effect on the predictions.

Note that variance is not a measure of how correct or incorrect the predictions are. It is a measure of how variable they are!

1.9.9 Measuring more students

To better visualise the concept of model variance, lets say we measure a second student, student "B":

Listing 1.164: Plot Measured Values for Students A and B versus True Morale

```
In [13]: plt.figure(figsize=(20, 5))

         plt.plot(days, morale_true,
                  linewidth=5, color='gold',
                  alpha=0.5, label='true function')

         plt.scatter(students['A']['days'],
                     students['A']['morale'],
                     s=70, c='darkred',
                     label='student A', alpha=0.7)

         plt.scatter(students['B']['days'],
                     students['B']['morale'],
                     s=70, c='steelblue',
                     label='student B', alpha=0.7)

         plt.xlabel('days', fontsize=16)
         plt.ylabel('morale', fontsize=16)
         plt.title('Morale over time\n', fontsize=20)
         plt.xlim([0, 72])

         plt.legend(loc='upper left')

         plt.show()
```

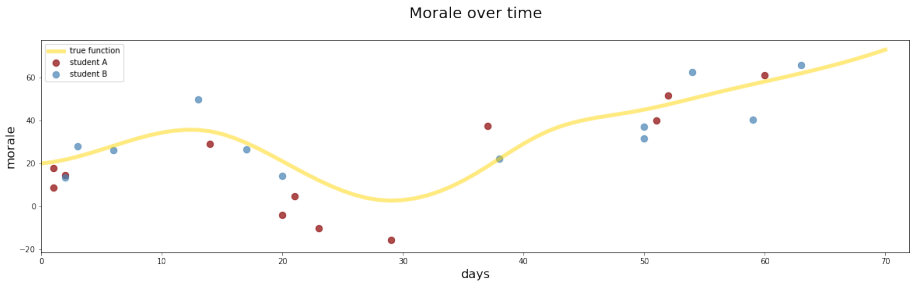

Figure 1.18: Plot Measured Values for Students A and B versus True Morale

Listing 1.165: Fit model for Student B

```
In [14]: studB_days = students['B']['days']
         studB_mor = students['B']['morale']

         Bmod = LinearRegression()
         Bmod.fit(studB_days[:, np.newaxis], studB_mor)

Out[14]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs
         =1, normalize=False)
```

Listing 1.166: Morale over time (difference between estimates)

```
In [15]: plt.figure(figsize=(20, 5))

         plt.plot(days, morale_true,
                  linewidth=5, color='gold',
                  alpha=0.5, label='true function')

         Apred = Amod.predict(days[:, np.newaxis])

         plt.plot(days, Apred,
                  lw=7., c='darkred',
                  alpha=0.5, label='A model')

         Bpred = Bmod.predict(days[:, np.newaxis])

         plt.plot(days, Bpred,
                  lw=7., c='steelblue',
                  alpha=0.5, label='B model')

         plt.scatter(students['A']['days'],
                     students['A']['morale'],
                     s=70, c='darkred',
                     label='student A', alpha=0.7)

         plt.scatter(students['B']['days'],
                     students['B']['morale'],
                     s=70, c='steelblue',
                     label='student B', alpha=0.7)

         plt.fill_between(days, Apred, Bpred,
                          color='grey', hatch='//',
                          edgecolor=None)

         plt.xlabel('days', fontsize=16)
         plt.ylabel('morale', fontsize=16)
         plt.xlim([0, 72])

         plt.legend(loc='upper left')

         plt.show()
```

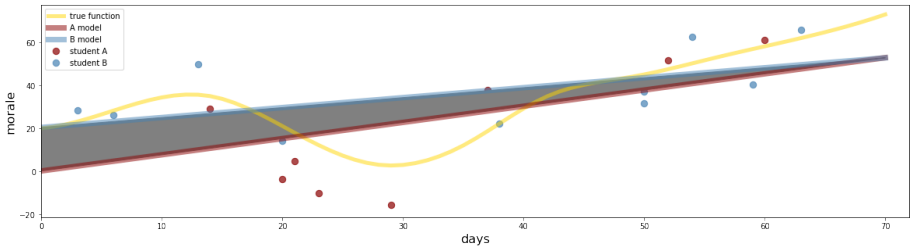

Figure 1.19: Morale over time (difference between estimates)

1.9.10 Bias and variance using 3 models

Below we will assess the morale of 3 different students over the days in the course at different times.

We can build these simple `morale ~ time` models for each and plot the regression lines.

These models are **high bias and low variance**. This is because there is a considerable amount of difference between the average of the model predictions and the true function, but not a lot of variation in predictions at time points across our models for the 3 students.

Listing 1.167: Morale over Time

```
In [16]: plt.figure(figsize=(20, 5))

         plt.plot(days, morale_true,
                  linewidth=5, color='gold',
                  alpha=0.5, label='true function')

         plt.scatter(students['A']['days'],
                     students['A']['morale'],
                     s=70, c='darkred',
                     label='student A', alpha=0.7)

         plt.scatter(students['B']['days'],
                     students['B']['morale'],
                     s=70, c='steelblue',
                     label='student B', alpha=0.7)

         plt.scatter(students['C']['days'],
                     students['C']['morale'],
                     s=70, c='darkgreen',
                     label='student C', alpha=0.7)

         plt.xlabel('days', fontsize=16)
         plt.ylabel('morale', fontsize=16)
         plt.xlim([0, 72])

         plt.legend(loc='upper left')

         plt.show()
```

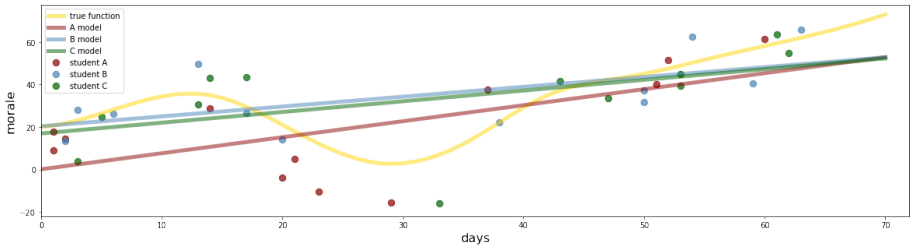

Figure 1.20: Morale over Time

Listing 1.168: Fit model for Student B

```
In [17]: studC_days = students['C']['days']
         studC_mor = students['C']['morale']

         Cmod = LinearRegression()
         Cmod.fit(studC_days[:, np.newaxis], studC_mor)

Out[17]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs
         =1, normalize=False)
```

Listing 1.169: Morale over Time

```
In [18]: plt.figure(figsize=(20, 5))

         plt.plot(days, morale_true,
                  linewidth=5, color='gold',
                  alpha=0.5, label='true function')

         Apred = Amod.predict(days[:, np.newaxis])

         plt.plot(days, Apred,
                  lw=5., c='darkred',
                  alpha=0.5, label='A model')

         Bpred = Bmod.predict(days[:, np.newaxis])

         plt.plot(days, Bpred,
                  lw=5., c='steelblue',
                  alpha=0.5, label='B model')

         Cpred = Cmod.predict(days[:, np.newaxis])

         plt.plot(days, Cpred,
                  lw=5., c='darkgreen',
                  alpha=0.5, label='C model')

         plt.scatter(students['A']['days'],
                     students['A']['morale'],
                     s=70, c='darkred',
                     label='student A', alpha=0.7)

         plt.scatter(students['B']['days'],
                     students['B']['morale'],
                     s=70, c='steelblue',
                     label='student B', alpha=0.7)

         plt.scatter(students['C']['days'],
                     students['C']['morale'],
                     s=70, c='darkgreen',
                     label='student C', alpha=0.7)

         plt.xlabel('days', fontsize=16)
         plt.ylabel('morale', fontsize=16)
         plt.xlim([0, 72])

         plt.legend(loc='upper left')

         plt.show()
```

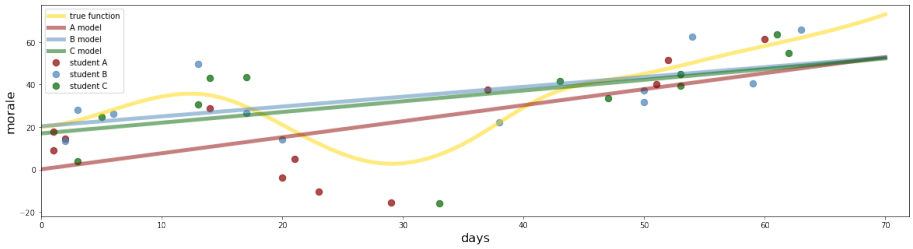

Figure 1.21: Morale over Time

1.9.11 Increasing complexity to try and capture the true function

Currently our models predict that morale simply increases over time.

Looking at the true function, we can see that there is an increase, then a decrease, and then an increase again.

Just modeling with a linear effect of time alone can't fit a curve; there is only one coefficient being multiplied by time to make our morale prediction. We could, however, add more variables created from time such as $time^2$, $time^3$, etc.

$$\hat{morale} = \beta_0 + \beta_1 t + \beta_2 t^2$$

$$\hat{morale} = \beta_0 + \beta_1 t + \beta_2 t^2 + \beta_3 t^3 + \beta_4 t^4$$

$$\hat{morale} = \beta_0 + \beta_1 t + ... + \beta_{16} t^{16}$$

The plots below show the difference in the fit when you add different numbers of "polynomial" time variables:

Listing 1.170: Define a function to prepare a Polynomial Regression function

```
In [19]: def polynomial_modeler(X, y, degrees):

             poly_feat = PolynomialFeatures(degree=degrees,
                                            include_bias=False)

             linear_regression = LinearRegression()

             pipeline = Pipeline([
                 ("polynomial_features", poly_feat),
                 ("linear_regression", linear_regression)
             ])

             pipeline.fit(X[:, np.newaxis], y)

             return pipeline
```

Listing 1.171: Define a function to prepare a Plot of Polynomial Regressions for various powers

```
In [20]: def plot_polyfit(X, y, truefunc, degrees=[1,2,4,16],
                          student_color='darkred', name='A'):

             # set the plot size
             plt.subplots(figsize=(20,2))

             # create a plot for each polynomial degree plotted
             for i in range(len(degrees)):
                 ax = plt.subplot(1, len(degrees), i + 1)

                 poly_model = polynomial_modeler(X, y, degrees[i])

                 X_test = np.linspace(1, 70, 200)
                 predictions = poly_model.predict(
                     X_test[:, np.newaxis]
                 )

                 plt.plot(X_test, predictions,
                          lw=5., c=student_color,
                          label="model", alpha=0.6)

                 plt.plot(days, morale_true,
                          linewidth=5, color='gold',
                          alpha=0.5, label='true function')

                 plt.scatter(X, y,
                             label="Student observations",
                             c=student_color, s=40)

                 plt.xlabel("days")
                 plt.ylabel("morale")
                 plt.xlim((0, 72))
                 plt.ylim((-20, 100))
                 plt.legend(loc="best")

                 plt.title('Student ' +
                           name +
                           f" (degree {degrees[i]})")
```

Listing 1.172: Run the Polynomial Plot for Student A

```
In [21]: plot_polyfit(students['A']['days'],
                      students['A']['morale'],
                      morale_func,
                      student_color='darkred',
                      name='A')
```

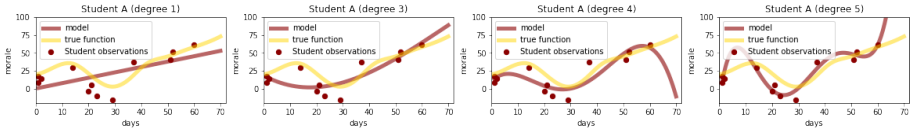

Figure 1.22: Run the Polynomial Plot for Student A

Listing 1.173: Run the Polynomial Plot for Student A

```
In [22]: plot_polyfit(students['A']['days'],
                      students['A']['morale'],
                      morale_func,
                      student_color='darkred',
                      name='A',
                      degrees=[1,3,4,5])
```

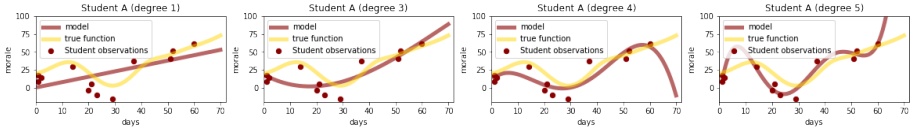

Figure 1.23: Run the Polynomial Plot for Student A

Listing 1.174: Run the Polynomial Plot for Student B

```
In [23]: plot_polyfit(students['B']['days'],
                      students['B']['morale'],
                      morale_func,
                      student_color='steelblue',
                      name='B')
```

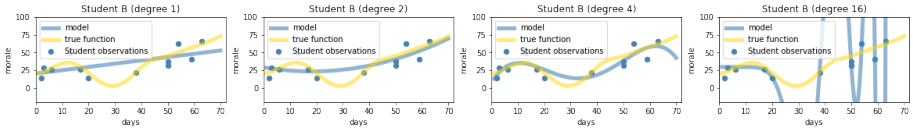

Figure 1.24: Run the Polynomial Plot for Student B

Listing 1.175: Run the Polynomial Plot for Student C

```
In [24]: plot_polyfit(students['C']['days'],
                      students['C']['morale'],
                      morale_func,
                      student_color='darkgreen',
                      name='C')
```

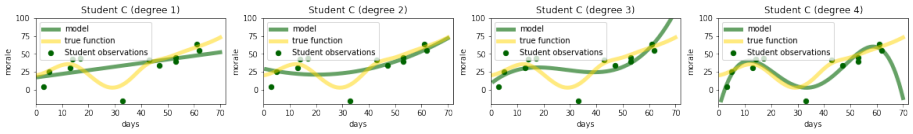

Figure 1.25: Run the Polynomial Plot for Student C

Listing 1.176: Run the Polynomial Plot for Student C

```
In [25]: plot_polyfit(students['C']['days'],
                      students['C']['morale'],
                      morale_func,
                      student_color='darkgreen',
                      name='C',
                      degrees=[1,2,3,4])
```

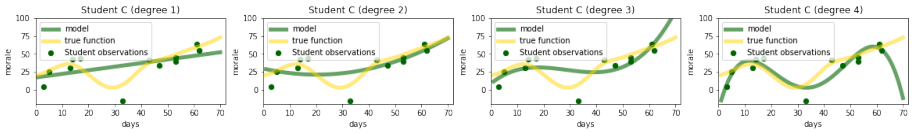

Figure 1.26: Run the Polynomial Plot for Student C

1.9.12 Higher complexity means higher variance (and lower bias)

The variance of predictions across our models goes up as we increase the model complexity. This is equivalent to saying that the variance of our model is increasing.

Increasing the complexity of the model at the expense of good future predictions is known as "overfitting" the data. High variance and overfitting are intrinsically related: if your predictions are inconsistent across samples, you are more likely to make the wrong predictions on future data.

Likewise, high bias and underfitting are related. If your model is too basic, it may give very consistent predictions but at the cost of oversimplifying the relationship between the target and predictors.

Below are student A and student B fit with the 16-polynomial time model and the area showing the difference in predictions at time points between them. Compare this to the area we saw earlier with the single time term.

Listing 1.177: Plot 16^{th}-order Regression Models for Students A and B versus True Morale

```
In [26]: plt.figure(figsize=(20, 5))

         plt.plot(days, morale_true,
                  linewidth=5, color='gold',
                  alpha=0.5, label='true function')

         Amod_complex = polynomial_modeler(
             students['A']['days'],
             students['A']['morale'],
             16
         )

         Bmod_complex = polynomial_modeler(
             students['B']['days'],
             students['B']['morale'],
             16
         )


         Apred = Amod_complex.predict(days[:, np.newaxis])

         plt.plot(days, Apred,
                  lw=7., c='darkred',
                  alpha=0.5, label='A model')

         Bpred = Bmod_complex.predict(days[:, np.newaxis])

         plt.plot(days, Bpred,
                  lw=7., c='steelblue',
                  alpha=0.5, label='B model')

         plt.scatter(students['A']['days'],
                     students['A']['morale'],
                     s=70, c='darkred',
                     label='student A', alpha=0.7)

         plt.scatter(students['B']['days'],
                     students['B']['morale'],
                     s=70, c='steelblue',
                     label='student B', alpha=0.7)

         plt.fill_between(days, Apred, Bpred,
                          color='grey', hatch='//',
                          edgecolor=None)

         plt.xlabel('days', fontsize=16)
         plt.ylabel('morale', fontsize=16)
         plt.title('Morale over time (16th polynomial)\n',
         fontsize=20)
         plt.xlim([0, 72])
         plt.ylim([-2500,2500])

         plt.legend(loc='upper left')

         plt.show()
```

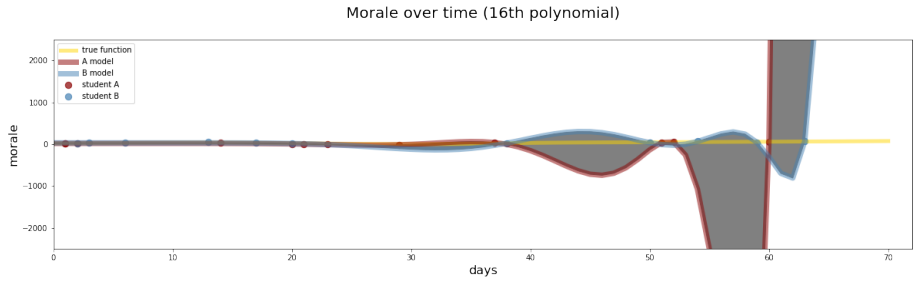

Figure 1.27: Plot (16ˆ{th})-order Regression Models for Students A and B versus True Morale

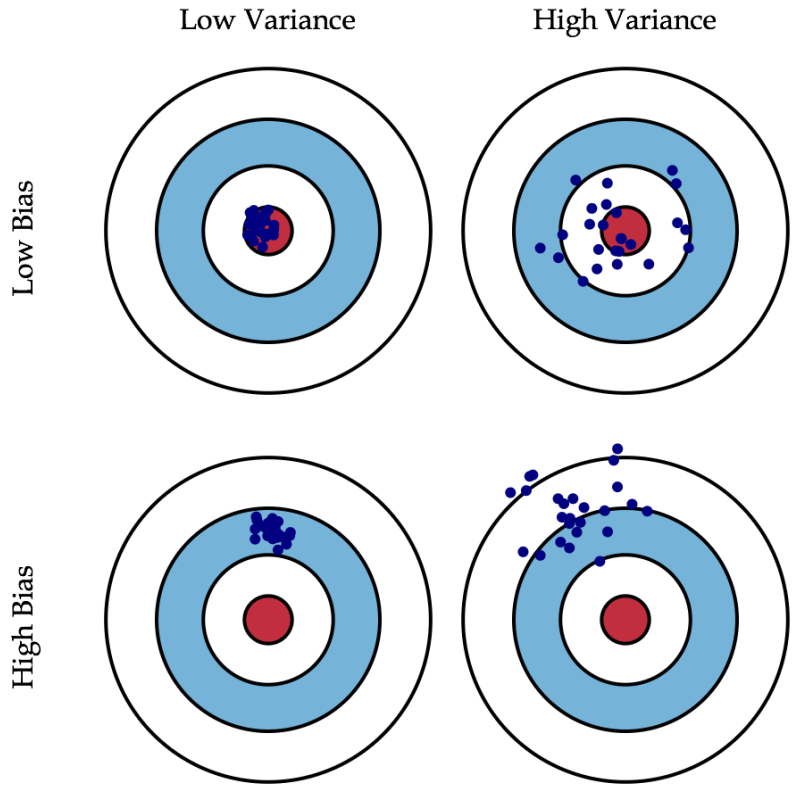

Figure 1.28: Illustrating the Bias-Variance Tradeoff

1.9.13 An Illustrative Example: Voting Intentions

Imagine we conduct a phone poll to see how voters in a town will vote in the next election and get these results:

Voting Republican	Voting Democratic	Non-Respondent	Total
13	16	21	50

Listing 1.178: Create Poll Simulation

```
In [27]: poll_results = {'voting_republican': 13,
                         'voting_democratic': 16,
                         'non-respondent': 21}
         total = sum([val for key,val in poll_results.items()])

         total_voting = sum([
             val for key,val in poll_results.items()
             if key != 'non-respondent'
         ])
```

Listing 1.179: Probability of Voting Republican According to Poll

```
In [28]: probability_of_voting_R = (poll_results['
         voting_republican']/
                                     float(total_voting))
         probability_of_voting_R

Out[28]: 0.4482758620689655
```

We put out our press release that the Democrats are going to win by over 10 points; but, when the election comes around, it turns out they actually lose by 10 points.

1.10 Pipelines

Figure 1.29: This is not a pipe

So far, we've established a general workflow where we:

1. **Clean the data Fill/impute/drop `NaN` values**

- One-hot encode categorical variables
- Label-encode target if categorical
- Check for skew / deskew

1. **Preprocess the data**

- Feature selection (`SelectKBest, SelectFromModel, SelectPercentile, RFE`, etc.)
- Scaling (`StandardScaler, MinMaxScaler`)

1. **Modeling**

- Classification (`KNeighborsClassifier, LogisticRegression`, etc.)
- Regression (`Lasso, Ridge, ElasticNet`, etc.)

For every dataset, we've done some version of all of these. Pipelines give us a convenient way to chain these tasks together. As a result, we can feed cleaned data into a pipeline and a trained model a the end!

Listing 1.180: Import the Python Numerical Stack

```
In [1]: import pandas as pd
        import numpy as np
```

Listing 1.181: Load bike sharing data

```
In [2]: bike_data = pd.read_csv('data/bike_sharing.csv',index_col
        =0)
```

Listing 1.182: Display a few columns from the head of `DataFrame`

```
In [3]: bike_data[['datetime', 'season', 'holiday', 'workingday',
         'temp']].head()
```

Table 1.11: Display a few columns from the head of `DataFrame`

	datetime	season	holiday	workingday	temp
0	2011-01-01 00:00:00	1	0	0	9.84
1	2011-01-01 01:00:00	1	0	0	9.02
2	2011-01-01 02:00:00	1	0	0	9.02
3	2011-01-01 03:00:00	1	0	0	9.84
4	2011-01-01 04:00:00	1	0	0	9.84

1.10.1 Clean the data

Listing 1.183: Convert the datetime colum to datetime using pd.to_datetime()

```
In [4]: bike_data['datetime'] = pd.to_datetime(bike_data['
        datetime'])
```

Listing 1.184: Make a feature for the day of week

```
In [5]: bike_data['dayofweek'] = bike_data['datetime'].apply(
        lambda x: x.dayofweek)
```

Listing 1.185: Make a feature for month

```
In [6]: bike_data['month'] = bike_data['datetime'].apply(lambda x
        : x.month)
```

Listing 1.186: Make a feature for hour

```
In [7]: bike_data['hour'] = bike_data['datetime'].apply(lambda x:
         x.hour)
```

Listing 1.187: Drop the datetime column

```
In [8]: bike_data.drop('datetime', axis=1, inplace=True)
```

Listing 1.188: Split up our features and target into features and target

```
In [9]: features = bike_data.drop('count',axis=1)

        target = bike_data['count']
```

Listing 1.189: Get dummies of categorical columns

```
In [10]: num_cols = ['temp','atemp','humidity','windspeed']
         cat_cols = [i for i in features.columns if i not in
         num_cols]

         features_dummies = pd.get_dummies(features, columns=
         cat_cols)
```

1.10.2 `Pipeline`

`Pipeline` is a class in `sklearn` that allows us to chain steps together.

We add steps to the pipeline using a list of tuples of the form `[('step name', sklearn object)...]`

Let's make a `Pipeline` that scales the data and fits a `RandomForestRegressor` model.

Listing 1.190: Load necessary classes and functions from Scikit-Learn

```
In [11]: from sklearn.pipeline import Pipeline
         from sklearn.ensemble import RandomForestRegressor
         from sklearn.preprocessing import (StandardScaler,
                                            MinMaxScaler)
         from sklearn.linear_model import Lasso, Ridge
         from sklearn.model_selection import train_test_split
```

Listing 1.191: Train-test split your data

```
In [12]: split_data = train_test_split(
             features_dummies,
             target,
             random_state = 42
         )

         X_train, X_test, y_train, y_test = split_data
```

Listing 1.192: Instantiate your pipeline

```
In [13]: simple_pipe = Pipeline([
             ('scaler',StandardScaler()),
             ('lasso', Lasso())
         ])
```

Listing 1.193: Fit the pipeline to your training features and target

```
In [14]: simple_pipe.fit(X_train, y_train)

Out[14]: Pipeline(memory=None,
              steps=[('scaler', StandardScaler(copy=True,
          with_mean=True, with_std=True)), ('lasso', Lasso(alpha
          =1.0, copy_X=True, fit_intercept=True, max_iter=1000,
             normalize=False, positive=False, precompute=False,
          random_state=None,
             selection='cyclic', tol=0.0001, warm_start=False))])
```

Listing 1.194: What's your train r2 score?

```
In [15]: simple_pipe.score(X_train, y_train)

Out[15]: 0.6388499611022919
```

Listing 1.195: What's your test r2 score?

```
In [16]: simple_pipe.score(X_test, y_test)

Out[16]: 0.6274260376321519
```

We now have a fit `Pipeline` object that scores just like any other model. This consists of a `StandardScaler` and a `Lasso`. What properties does this `Pipeline` have?

- `.steps` gives you a list of tuples containing the names of your steps and the fit object of the step itself.
- `.named_steps` gives you a dictionary with your pipeline objects where the keys are the names and the values are the fit sklearn object.

Listing 1.196: Look at the steps

```
In [17]: simple_pipe.steps

Out[17]: [('scaler', StandardScaler(copy=True, with_mean=True,
         with_std=True)),
          ('lasso', Lasso(alpha=1.0, copy_X=True, fit_intercept=
         True, max_iter=1000,
               normalize=False, positive=False, precompute=False,
         random_state=None,
               selection='cyclic', tol=0.0001, warm_start=False))]
```

Listing 1.197: Look at the named steps

```
In [18]: simple_pipe.named_steps

Out[18]: {'scaler': StandardScaler(copy=True, with_mean=True,
         with_std=True),
          'lasso': Lasso(alpha=1.0, copy_X=True, fit_intercept=
         True, max_iter=1000,
             normalize=False, positive=False, precompute=False,
         random_state=None,
             selection='cyclic', tol=0.0001, warm_start=False)}
```

We can access each step and use it if we'd like. Let's look at the mean and standard deviation of our data from the scaler object.

Listing 1.198: Display means from `StandardScaler`

```
In [19]: simple_pipe.steps[0][1].mean_

Out[19]: array([2.02449804e+01, 2.36701605e+01, 6.18351298e+01,
         1.27913341e+01,
                2.45835375e-01, 2.51224890e-01, 2.49755022e-01,
         2.53184713e-01,
                9.70602646e-01, 2.93973542e-02, 3.16756492e-01,
         6.83243508e-01,
                6.62175404e-01, 2.60289074e-01, 7.74130328e-02,
         1.22488976e-04,
                1.41474767e-01, 1.42822146e-01, 1.43312102e-01,
         1.43434591e-01,
                1.41597256e-01, 1.44169525e-01, 1.43189613e-01,
         7.96178344e-02,
                8.30475257e-02, 8.31700147e-02, 8.14551690e-02,
         8.56197942e-02,
                8.41499265e-02, 8.31700147e-02, 8.47623714e-02,
         8.18226360e-02,
                8.51298383e-02, 8.51298383e-02, 8.29250367e-02,
         4.28711416e-02,
                4.10338070e-02, 4.22586967e-02, 4.00538951e-02,
         4.12787849e-02,
                4.20137188e-02, 4.16462518e-02, 4.12787849e-02,
         4.21362077e-02,
                4.12787849e-02, 4.18912298e-02, 4.06663400e-02,
         4.11562959e-02,
                4.04213621e-02, 4.10338070e-02, 4.22586967e-02,
         4.18912298e-02,
                4.31161195e-02, 4.37285644e-02, 4.17687408e-02,
         4.15237629e-02,
                4.02988731e-02, 4.14012739e-02, 4.29936306e-02])
```

Listing 1.199: .std_ is deprecated, use .scale_

```
In [20]: simple_pipe.named_steps['scaler'].scale_

Out[20]: array([7.79554011e+00, 8.46820631e+00, 1.92546121e+01,
         8.21341117e+00,
                4.30581401e-01, 4.33717586e-01, 4.32871171e-01,
         4.34835848e-01,
                1.68917583e-01, 1.68917583e-01, 4.65211583e-01,
         4.65211583e-01,
                4.72968433e-01, 4.38792288e-01, 2.67245683e-01,
         1.10667959e-02,
                3.48510628e-01, 3.49891384e-01, 3.50390844e-01,
         3.50515490e-01,
                3.48636592e-01, 3.51261545e-01, 3.50266110e-01,
         2.70700637e-01,
                2.75954044e-01, 2.76139029e-01, 2.73532858e-01,
         2.79801796e-01,
                2.77612529e-01, 2.76139029e-01, 2.78527758e-01,
         2.74094313e-01,
                2.79074809e-01, 2.79074809e-01, 2.75768880e-01,
         2.02566549e-01,
                1.98368429e-01, 2.01178774e-01, 1.96085646e-01,
         1.98934278e-01,
                2.00620453e-01, 1.99779482e-01, 1.98934278e-01,
         2.00899845e-01,
                1.98934278e-01, 2.00340597e-01, 1.97516047e-01,
         1.98651593e-01,
                1.96945362e-01, 1.98368429e-01, 2.01178774e-01,
         2.00340597e-01,
                2.03118487e-01, 2.04490531e-01, 2.00060274e-01,
         1.99498220e-01,
                1.96659284e-01, 1.99216486e-01, 2.02842743e-01])
```

Listing 1.200: Display β_i for Lasso Model

```
In [21]: simple_pipe.named_steps['lasso'].coef_

Out[21]: array([ 37.81956127,  15.07856264, -20.97367854,
         -4.80325321,
                 -8.67445325,   0.        ,  -0.        ,
         16.55810977,
                  0.        ,  -0.        ,  -0.        ,   0.
             ,
                  1.69301497,  -0.        , -13.9527773 ,
         -0.16920321,
                 -0.97024358,  -0.45100847,   0.        ,   0.
             ,
                  0.24940573,   1.67373388,  -3.22686251,
         -3.55528056,
                 -0.        ,   0.        ,  -0.        ,
         5.86413596,
                  0.        ,  -7.73988209,  -3.77614438,
         3.19136327,
                  1.1122235 ,   0.        ,   0.        ,
         -27.62730192,
                -30.25156015, -31.97763363, -33.90388915,
         -33.8726006 ,
                -31.07937045, -18.90032333,   4.83038122,
         34.76910242,
                  3.98765501,  -6.1951688 ,  -1.00767564,
         3.73829618,
                  3.83990337,   0.        ,   1.67571803,
         13.79605162,
                 44.87442287,  41.18548178,  19.23742649,
         2.42356056,
                 -6.19354438, -12.34345661, -20.45458108])
```

1.10.3 The `make_pipeline` helper function

While `Pipeline` gives us the ability to explicitly name our steps, this can be cumbersome, especially when we may not care what are steps are named. If this is the case, we use `make_pipeline`.

Let's execute the same pipeline, except this time we'll use `make_pipeline`.

Listing 1.201: Import `make_pipeline` helper function

```
In [22]: from sklearn.pipeline import make_pipeline
```

Listing 1.202: Define a second `Pipeline` using `make_pipeline`

```
In [23]: another_pipe = make_pipeline(
             StandardScaler(),
             Lasso()
         )
```

Listing 1.203: Fit second `Pipeline`

```
In [24]: another_pipe.fit(X_train, y_train)

Out[24]: Pipeline(memory=None,
              steps=[('standardscaler', StandardScaler(copy=True,
         with_mean=True, with_std=True)), ('lasso', Lasso(alpha
         =1.0, copy_X=True, fit_intercept=True, max_iter=1000,
            normalize=False, positive=False, precompute=False,
         random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False))])
```

Listing 1.204: Display training score

```
In [25]: another_pipe.score(X_train, y_train)

Out[25]: 0.6388499611022919
```

Listing 1.205: Display testing score

```
In [26]: another_pipe.score(X_test, y_test)

Out[26]: 0.6274260376321519
```

Even though we don't name them, `make_pipeline` still has a `.named_steps` attribute. It automatically assigns names to each step and we can access them similarly to how we did before.

Listing 1.206: Display named steps for `Pipeline`

```
In [27]: another_pipe.named_steps

Out[27]: {'standardscaler': StandardScaler(copy=True, with_mean=
         True, with_std=True),
          'lasso': Lasso(alpha=1.0, copy_X=True, fit_intercept=
         True, max_iter=1000,
             normalize=False, positive=False, precompute=False,
         random_state=None,
             selection='cyclic', tol=0.0001, warm_start=False)}
```

Listing 1.207: Display steps for `Pipeline`

```
In [28]: another_pipe.steps

Out[28]: [('standardscaler', StandardScaler(copy=True, with_mean=
         True, with_std=True)),
          ('lasso', Lasso(alpha=1.0, copy_X=True, fit_intercept=
         True, max_iter=1000,
               normalize=False, positive=False, precompute=False,
         random_state=None,
               selection='cyclic', tol=0.0001, warm_start=False))]
```

1.10.4 Aside: Transformation pipelines

Although it's standard to have a pipeline end in a model, it's also possible to have a pipeline just for transformers, as shown below:

Listing 1.208: Import transformers from Scikit-Learn

```
In [29]: from sklearn.feature_selection import (SelectFromModel,
                                                 SelectKBest,
                                                 f_regression)
```

Listing 1.209: Make transformation pipeline

```
In [30]: transformer_pipe = make_pipeline(
             SelectKBest(score_func=f_regression, k=40),
             StandardScaler(),
             SelectFromModel(Lasso())
         )
```

Listing 1.210: Fit transformation pipeline

```
In [31]: transformer_pipe.fit(X_train, y_train)

Out[31]: Pipeline(memory=None,
              steps=[('selectkbest', SelectKBest(k=40, score_func
         =<function f_regression at 0x1167a8158>)), ('
         standardscaler', StandardScaler(copy=True, with_mean=True
         , with_std=True)), ('selectfrommodel', SelectFromModel(
         estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=
         True, max_iter=1000,
            normalize=False, positive=False, precompute=False,
         random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False),
                 norm_order=1, prefit=False, threshold=None))])
```

Listing 1.211: Use transformation pipeline to transform data

```
In [32]: features_skb_scaled_sfm = transformer_pipe.transform(
         X_train)
```

Listing 1.212: Show shape of training set

```
In [33]: X_train.shape

Out[33]: (8164, 59)
```

Listing 1.213: Show shape of transformed data

```
In [34]: features_skb_scaled_sfm.shape

Out[34]: (8164, 36)
```

Listing 1.214: Show steps in transformation pipeline

```
In [35]: transformer_pipe.steps

Out[35]: [('selectkbest',
           SelectKBest(k=40, score_func=<function f_regression at
          0x1167a8158>)),
           ('standardscaler', StandardScaler(copy=True, with_mean=
          True, with_std=True)),
           ('selectfrommodel',
            SelectFromModel(estimator=Lasso(alpha=1.0, copy_X=True,
           fit_intercept=True, max_iter=1000,
               normalize=False, positive=False, precompute=False,
          random_state=None,
               selection='cyclic', tol=0.0001, warm_start=False),
                    norm_order=1, prefit=False, threshold=None))]
```

from sklearn.preprocessing import FunctionTransformer

Listing 1.215: Make transformation pipeline

```
In [37]: transformer_pipe = make_pipeline(
             SelectKBest(score_func=f_regression, k=40),
             FunctionTransformer(lambda x: x + 1),
             FunctionTransformer(np.log),
             StandardScaler(),
             SelectFromModel(Lasso())
         )
```

transformer_pipe.fit(X_train, y_train)

1.10.5 Using Pipelines with `GridSearchCV`

So far, we've only chained transformers and models together in a `pipeline`. What if we want to use GridSearch to tune our model in the `pipeline`?

Since we have to refer to our steps by name, let's use `Pipeline` instead of `make_pipeline`.

Let's make a pipeline with the following steps:

```
('skb', SelectKBest(score_func=f_regression, k=40)),
('scaler', StandardScaler()),
('sfm', SelectFromModel(Lasso())),
('regr', ElasticNet())
```

Listing 1.216: Load necessary models

```
In [39]: from sklearn.model_selection import (GridSearchCV,
                                                ShuffleSplit)
         from sklearn.linear_model import ElasticNet
```

Listing 1.217: Create model pipeline

```
In [40]: pipe_for_gs = Pipeline([
             ('skb', SelectKBest(score_func=f_regression, k=40)),
             ('scaler', StandardScaler()),
             ('sfm', SelectFromModel(Lasso())),
             ('regr', ElasticNet())
         ])
```

pipe_for_gs.steps

Next, let's make our parameter grid. When using a `Pipeline`, we need to specify which step our params are for. To do that, we use the name we gave the step (in this case `'rf'` for `RandomForestRegressor`), with a **dunder** to reference a parameter for that model.

As an example, if we wanted to tune `ElasticNet`'s `l1_ratio` parameter, we use `regr__l1_ratio:[.1,.5,.9]`.

Let's fill out the params below to tune `alpha` and `l1_ratio`:

Listing 1.218: Define parameter grid

```
In [42]: params = {
             'regr__l1_ratio':[.1,.3,.5,.7,.9],
             'regr__alpha':np.logspace(-3,3,7)
         }
```

Now pass your pipeline into `GridSearchCV` with your parameters, using `ShuffleSplit`

Listing 1.219: Create grid search model pipeline

```
In [43]: gspipe = GridSearchCV(
             pipe_for_gs,
             param_grid=params,
             cv=ShuffleSplit(n_splits=5, random_state=42),
             n_jobs=-1
         )
```

Listing 1.220: Fit the grid search model pipeline

```
In [44]: gspipe.fit(X_train, y_train.ravel())

Out[44]: GridSearchCV(cv=ShuffleSplit(n_splits=5, random_state=42,
          test_size='default',
                train_size=None),
                error_score='raise',
                estimator=Pipeline(memory=None,
              steps=[('skb', SelectKBest(k=40, score_func=<
         function f_regression at 0x1167a8158>)), ('scaler',
         StandardScaler(copy=True, with_mean=True, with_std=True))
         , ('sfm', SelectFromModel(estimator=Lasso(alpha=1.0,
         copy_X=True, fit_intercept=True, max_iter=1000,
            normalize=False, positive=False, precomput...alse,
         precompute=False,
               random_state=None, selection='cyclic', tol=0.0001,
         warm_start=False))]),
                fit_params=None, iid=True, n_jobs=-1,
                param_grid={'regr__l1_ratio': [0.1, 0.3, 0.5, 0.7,
          0.9], 'regr__alpha': array([1.e-03, 1.e-02, 1.e-01, 1.e
         +00, 1.e+01, 1.e+02, 1.e+03])},
                pre_dispatch='2*n_jobs', refit=True,
         return_train_score='warn',
                scoring=None, verbose=0)
```

Listing 1.221: Display best score for grid search model pipeline

```
In [45]: gspipe.best_score_

Out[45]: 0.6441703040776783
```

To get the `.steps` or `.named_steps`, we need to access `GridSearchCV`'s `.best_estimator_` parameter, which contains our `Pipeline`. How do we access our model? Our scaler?

Listing 1.222: Display named steps for best estimator

```
In [46]: gspipe.best_estimator_.named_steps

Out[46]: {'skb': SelectKBest(k=40, score_func=<function
         f_regression at 0x1167a8158>),
          'scaler': StandardScaler(copy=True, with_mean=True,
         with_std=True),
          'sfm': SelectFromModel(estimator=Lasso(alpha=1.0, copy_X
         =True, fit_intercept=True, max_iter=1000,
             normalize=False, positive=False, precompute=False,
         random_state=None,
             selection='cyclic', tol=0.0001, warm_start=False),
                   norm_order=1, prefit=False, threshold=None),
          'regr': ElasticNet(alpha=0.001, copy_X=True,
         fit_intercept=True, l1_ratio=0.9,
                max_iter=1000, normalize=False, positive=False,
         precompute=False,
                random_state=None, selection='cyclic', tol=0.0001,
          warm_start=False)}
```

Listing 1.223: Display regression model

```
In [47]: best_enet = gspipe.best_estimator_.named_steps['regr']
```

best_enet.coef_

Listing 1.224: Display `SelectFromModel` transformer

```
In [49]: gspipe.best_estimator_.named_steps['sfm']

Out[49]: SelectFromModel(estimator=Lasso(alpha=1.0, copy_X=True,
         fit_intercept=True, max_iter=1000,
            normalize=False, positive=False, precompute=False,
         random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False),
                 norm_order=1, prefit=False, threshold=None)
```

Chapter 2

Case Study: Linear Modeling

Listing 2.1: Instantiate Environment

```
In [1]: %run preamble.py
        %matplotlib inline
```

2.1 Vectorized Functions

2.1.1 Back to Algebra I

The slope-intercept equation:

$$y = mx + b$$

Here, we are defining a relationship between two variables, x and y.

You should think of this as a **mapping** $x \mapsto y$ or the **function** $f(x) = y$.

We can use R to model a linear function of two variables.

The word "model" here is important. A mathematical function in a computer is by necessity a model. We typically think of mathematical functions as continuous. In between two points on a function curve, we can always find a third point.

Computationally i.e. in a computer, we can not do this. We *must* work with a finite set of values.

2.1.2 A Note on Markdown and Latex in Jupyter

Jupyter markdown cells can take markdown, html, or latex to render text. Latex (pronouced "lay-teck") is a programming language for designed for rendering technical writing, especially math.

For example, if we write

```
$$\frac{1}{2}$$
```

it will appear as

$$\frac{1}{2}$$

2.1.3 A Vector of x values

2.1.4 A Note on Nomenclature

We will use `xx` for the name for a vectors of x values and `yy` for a vector of y values.

Listing 2.2: Create an array of ten values from 1 to 10

```
In [2]: xx = np.arange(1,11)
```

`xx` is a vector of values from 1 to 10 incremented by 1.

Listing 2.3: Display the vector `xx`

```
In [3]: xx

Out[3]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
```

2.1.5 A Vector of y values

If we consider the slope-intercept equation:

$$y = 0.1x + 1$$

We can generate a `yy` vector of values by performing simple vector arithmetic.

Listing 2.4: Create a vector of values from 1.1 to 2 using vector arithmetic

```
In [4]: yy = 0.1*xx + 1
```

Listing 2.5: Display the vector `yy`

```
In [5]: yy

Out[5]: array([1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. ])
```

Here, each value in `xx` gets multiplied by 0.1 and incremented by 1.

Listing 2.6: Plot the `yy` versus `xx`

```
In [6]: initialize_2d_plot(-2,11,-1,5)
        plt.scatter(xx, yy)

Out[6]: <matplotlib.collections.PathCollection at 0x11d22b7f0>
```

Listing 2.7: Instantiate Environment

```
In [1]: %run preamble.py
        %matplotlib inline
```

2.2 A System of Linear Equations

$$f(x) = 3x - 2$$

$$f(x) = -5x + 6$$

Listing 2.8: Create an array of values

```
In [2]: xx = np.arange(-1, 2, .1)
```

Listing 2.9: Display the vector `xx`

```
In [3]: xx

Out[3]: array([-1.00000000e+00, -9.00000000e-01, -8.00000000e-01,
         -7.00000000e-01,
               -6.00000000e-01, -5.00000000e-01, -4.00000000e-01,
         -3.00000000e-01,
               -2.00000000e-01, -1.00000000e-01, -2.22044605e-16,
          1.00000000e-01,
                2.00000000e-01,  3.00000000e-01,  4.00000000e-01,
          5.00000000e-01,
                6.00000000e-01,  7.00000000e-01,  8.00000000e-01,
          9.00000000e-01,
                1.00000000e+00,  1.10000000e+00,  1.20000000e+00,
          1.30000000e+00,
                1.40000000e+00,  1.50000000e+00,  1.60000000e+00,
          1.70000000e+00,
                1.80000000e+00,  1.90000000e+00])
```

Listing 2.10: Create vectors representing the system

```
In [4]: yy1 = 3*xx - 2
        yy2 = -5*xx + 6
```

2.2.1 Plot System of Equations in Python

Listing 2.11: Plot the system

```
In [5]: fig = initialize_2d_plot(-1,2,-7,15)

        plt.scatter(xx, yy1, label='f1')
        plt.scatter(xx, yy2, label='f2')

        plt.legend()

Out[5]: <matplotlib.legend.Legend at 0x11e040c88>
```

2.2.2 The Solution

The solution to this system of equations is a hyperplane in one less dimension than the two equations. Each equation is a line, a 1-D hyperplane. The hyperplane describing their intersection is a point a 0-D hyperplane.

2.2.3 Find the Solution Analytically

$$y = 3x - 2 \tag{2.1}$$
$$y = -5x + 6 \tag{2.2}$$
$$3x - 2 = -5x + 6 \quad \text{(both are equal to } y\text{)}$$
$$8x = 8 \tag{2.3}$$
$$x = 1 \tag{2.4}$$
$$y = 3 \cdot 1 - 2 = 1 \quad \text{(plug } x\text{=1 into the first equation)}$$
$$y = -5 \cdot 1 + 6 = 1 \quad \text{(plug } x\text{=1 into the second equation)}$$

Both equations yield $y = 1$ implying that our solution is $(1, 1)$.

Listing 2.12: Plot the system with solution

```
In [6]: fig = initialize_2d_plot(-1,2,-7,15)
        plt.scatter(xx, yy1, label='f1')
        plt.scatter(xx, yy2, label='f2')
        plt.plot(1,1,'o',c='red', markersize=15)
        plt.legend()

Out[6]: <matplotlib.legend.Legend at 0x11e264128>
```

Listing 2.13: Instantiate Environment

```
In [1]: %run preamble.py
        %matplotlib inline
```

2.3 The Linear Combination

The basis of linear algebra is the **vector** and the **linear combination**.

2.3.1 The Vector

A vector is a multi-dimensional element.
e.g. (1), $(1, 3, 5, 8)$, $(1, 2, \ldots, 50)$
As opposed to a **scalar**
e.g. $1, \sqrt{2}, e, \pi, 101010, 47$

2.3.2 Scalar Arithmetic

- add
- subtract
- multiply
- divide

2.3.3 Vector Arithmetic

Vector Arithmetic has two defined operations, *vector addition* and *scalar multiplication*. That is, provided that we have vectors of equal length, we can add two vectors together, and we can multiply a vector by a scalar value.

2.3.4 Vector Addition

Here, we add two vectors. This is done by adding corresponding elements of the vector. Note the result is a vector of the same shape, signifying that it is a member of the same vector space.

$$(-1, 2) + (3, 5) = (2, 7)$$

Note that all three of these vectors are elements of the two-dimensional real-valued vector space, $\mathbb{R}^2$. We use the symbol $\in$ to mean "an element of" so that we can write

$$(-1, 2), (3, 5), (2, 7) \in \mathbb{R}^2$$

which means that these three vectors are elements of $\mathbb{R}^2$.

Listing 2.14: Perform vector addition with `numpy`

```
In [2]: a = np.array((-1,2))
        b = np.array((3,5))
        c = a + b
        c

Out[2]: array([2, 7])
```

Listing 2.15: Assert that vector addition is commutative

```
In [3]: a + b == b + a

Out[3]: array([ True,  True])
```

Listing 2.16: Plot vector addition

```
In [4]: initialize_2d_plot(-2,4,-2,10)
        draw_vector(b)
        draw_vector(a, tail=b)
        draw_vector(c, kwargs={'color':'green', 'linewidth':3})
        draw_vector(a)
        draw_vector(b, tail=a)
```

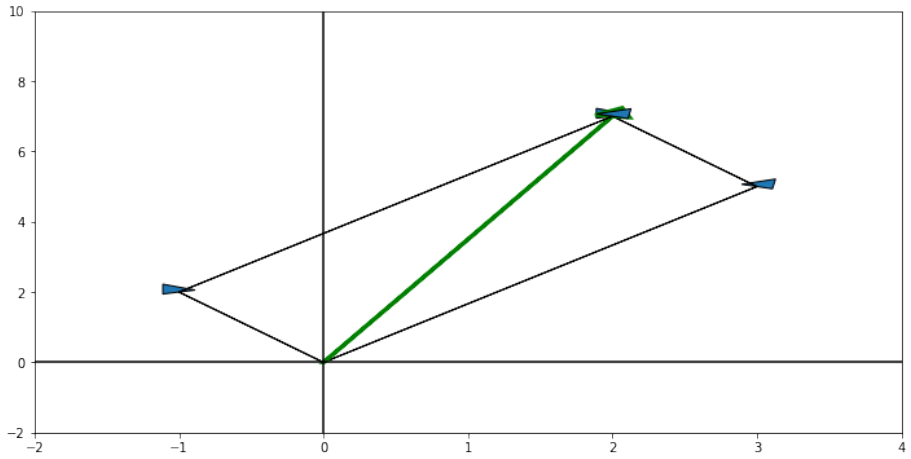

Figure 2.1: Plot vector addition

2.3.5 Vector Addition in General

Generally, where vector $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^p$, where $\mathbb{R}^p$ is a p-dimensional real-valued vector space

$$\mathbf{u} + \mathbf{v} = (u_1, \ldots, u_p) + (v_1, \ldots, v_p) = (u_1 + v_1, \ldots, u_p + v_p) = (w_1, \ldots, w_p) = \mathbf{w}$$

2.3.6 Scalar Multiplication

Here, we add multiply a vector by a scalar value. This is done by multiplying each element of the vector by the scalar. Note the result is a vector of the same shape, signifying that it is a member of the same vector space.

$$3 \cdot (-1.5, 1.3) = (-4.5, 3.9)$$

$$(-1.5, 1.3), (-4.5, 3.9) \in \mathbb{R}^2$$

Listing 2.17: Perform scalar multiplication with `numpy`

```
In [5]: d = np.array((-1.5,1.3))
        e = 3*d
        e

Out[5]: array([-4.5,  3.9])
```

Listing 2.18: Plot scalar multiplication

```
In [6]: initialize_2d_plot(-5,1,-1,5)
        draw_vector(e, kwargs={'color':'green', 'linewidth':6})
        draw_vector(d)
        draw_vector(d, tail=d)
        draw_vector(d, tail=d+d)
```

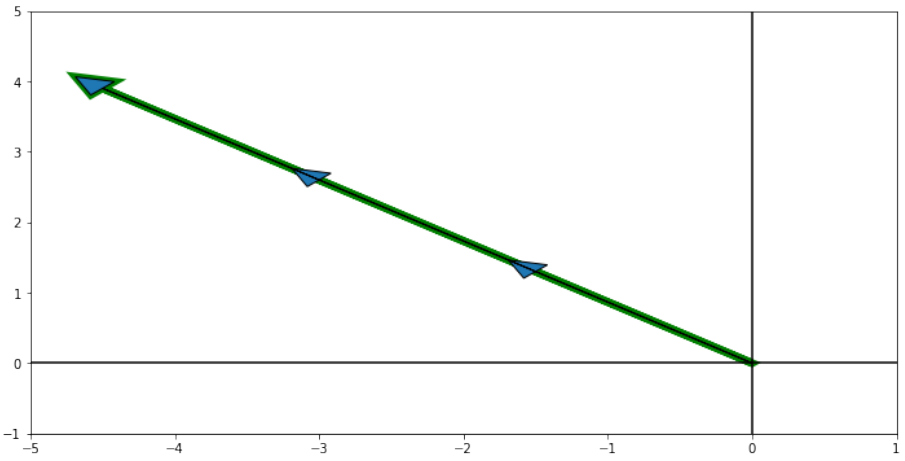

Figure 2.2: Plot scalar multiplication

2.3.7 Scalar Multiplication in General

Generally, with vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^p$, where $\mathbb{R}^p$ is a p-dimensional real-valued vector space and scalar $\beta \in \mathbb{R}$

$$\beta\mathbf{u} = \beta(u_1, \ldots, u_p) = (\beta u_1, \ldots, \beta u_p) = (v_1, \ldots, v_p) = \mathbf{v}$$

2.3.8 The Linear Combination

A linear combination is the vector result of a vector addition and scalar multiplication of vectors.

Listing 2.19: Perform linear combination with `numpy`

```
In [7]: f = np.array((-1,1))
        g = np.array((2,-1))
        h = 2*f + 3*g
```

Listing 2.20: Plot linear combination

```
In [8]: initialize_2d_plot(-3,5,-2,3)
        draw_vector(h, kwargs={'color':'green', 'linewidth':6})
        draw_vector(2*f)
        draw_vector(3*g, tail=2*f)
```

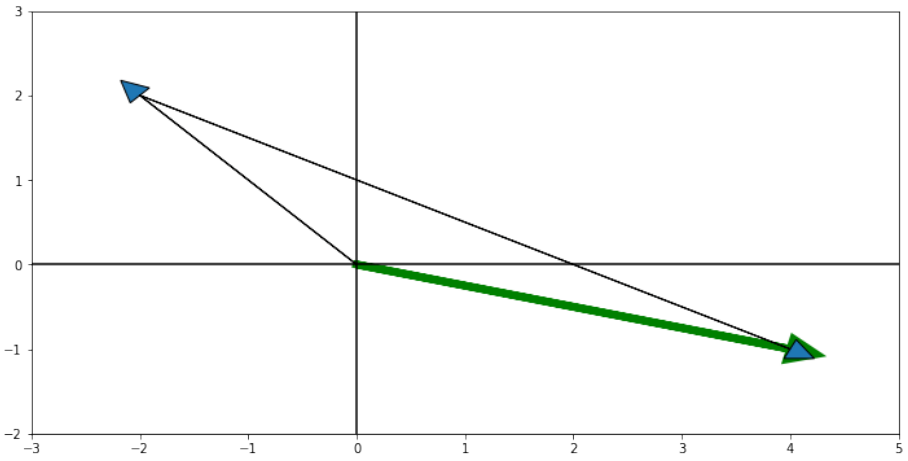

Figure 2.3: Plot linear combination

2.3.9 Linear Combination in General

Generally, with vectors $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{R}^p$, where $\mathbb{R}^p$ is a p-dimensional real-valued vector space and scalars $\beta, \gamma \in \mathbb{R}$

$$\beta\mathbf{u}+\gamma\mathbf{v} = \beta(u_1, \ldots, u_p)+\gamma(v_1, \ldots, v_p) = (\beta u_1+\gamma v_1, \ldots, \beta u_p+\gamma v_p) = (w_1, \ldots, w_p) = \mathbf{w}$$

Listing 2.21: Instantiate Environment

```
In [1]: %run preamble.py
        %matplotlib inline
```

2.4 The Dot Product

2.4.1 The length of a vector

The length of a vector is the number of elements in that vector.

Listing 2.22: Show the length of `a`

```
In [2]: len(a)

Out[2]: 2
```

`a` and `h` have the same length.

Listing 2.23: Show that `a` and `h` have the same length

```
In [3]: len(a) == len(h)

Out[3]: True
```

2.4.2 The magnitude of a vector

The magnitude of a vector can be calculated with the `numpy.linalg.norm` function.

Listing 2.24: Calculate the magnitude of `a`

```
In [4]: np.linalg.norm(a)

Out[4]: 2.23606797749979
```

`a` and `h` do not have the same magnitude.

Listing 2.25: Show that `a` and `h` do not have the same magnitude

```
In [5]: np.linalg.norm(a) == np.linalg.norm(h)

Out[5]: False
```

2.4.3 Magnitude as Euclidean Distance

The magnitude of a 2-d vector is something you have certainly seen before.

Create component vectors along the x and y axes.

Listing 2.26: Plot the individual components of `a`

```
In [6]: initialize_2d_plot(-2,3,-2,3)

        draw_vector(a)

        draw_vector((-1,0))
        draw_vector((0,2),tail=(-1,0))
```

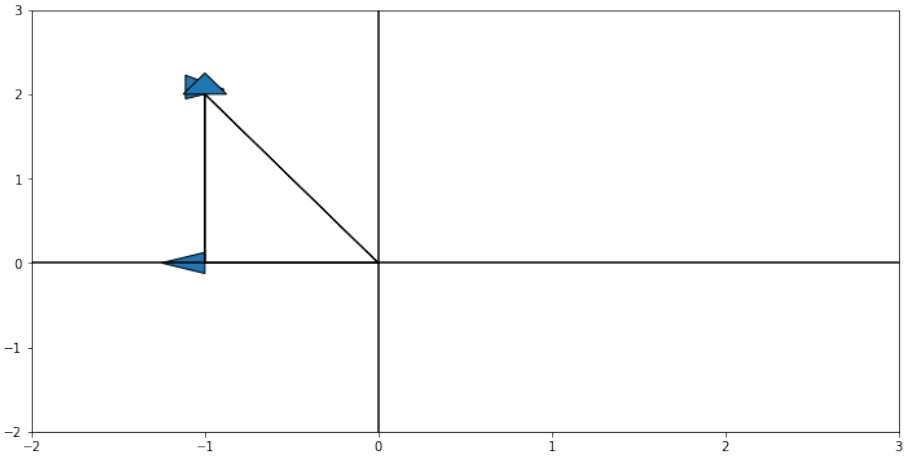

Figure 2.4: Plot the individual components of `a`

2.4.4 Euclidean Distance

The magnitude of a is

$$||a|| = \sqrt{a_x^2 + a_y^2}$$

which should be all to familiar to you as the Pythagorean Theorem.
Incredibly, this generalizes to p-dimensional vectors.

$$||a_p|| = \sqrt{a_1^2 + a_2^2 \cdots + a_p^2}$$

2.4.5 The ℓ_2-Norm

This computation has a fancy name, the ℓ_2-norm.
We just saw this above with the function `np.linalg.norm`.

Listing 2.27: Manually calculate the magnitude of `a`

```
In [7]: np.sqrt(a[0]**2 + a[1]**2)

Out[7]: 2.23606797749979
```

Listing 2.28: Show that the manual calculation is equivalent to `np.linalg.norm`

```
In [8]: np.sqrt(a[0]**2 + a[1]**2) == np.linalg.norm(a)

Out[8]: True
```

We will see our friend, the ℓ_2-norm again.

For now, let's consider the first computation.

$$||a|| = \sqrt{a_x^2 + a_y^2}$$

We can rewrite this as

$$||a|| = \sqrt{a_x a_x + a_y a_y}$$

Let's consider just the computation under the radical

$$a_x a_x + a_y a_y$$

This is actually a special computation, the **dot product**.

2.4.6 Definition of the Dot Product

The dot product, also known as the inner product, is an operation defined over a vector space that yields a scalar.

We can think of is as a mapping of two vectors to a scalar value

$$f : \mathbf{u}, \mathbf{v} \mapsto \mathbb{R}$$

or the function

$$f(\mathbf{u}, \mathbf{v}) = \alpha$$

where $\alpha \in \mathbb{R}$.

2.4.7 Example

Given

$$\mathbf{u} = (1, 0, -1)$$

$$\mathbf{v} = (-3, 3, -2)$$

The dot product of $\mathbf{u}$ and $\mathbf{v}$ is $\langle \mathbf{u}, \mathbf{v} \rangle$ where

$$\langle \mathbf{u}, \mathbf{v} \rangle = 1 \cdot (-3) + 0 \cdot 3 + (-1) \cdot (-2) = -1$$

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum u_i \cdot v_i$$

Listing 2.29: Compute the dot product of `u` and `v`

```
In [9]: u = np.array((1,0,-1))
        v = np.array((-3,3,-2))
        u.dot(v)

Out[9]: -1
```

NOTE:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{v}, \mathbf{u} \rangle$$

for all $\mathbf{u}$, $\mathbf{v}$

We will need this later.

2.4.8 The Magnitude is the Square Root of the Dot Product of a Vector with Itself

Knowing this, it is easy to see that

$$a_x a_x + a_y a_y$$

is the dot product of $\mathbf{a}$ with itself.
Then

$$||a|| = \sqrt{\langle \mathbf{a}, \mathbf{a} \rangle}$$

Listing 2.30: Show that the above is true

```
In [10]: a.dot(a) == a[0]**2 + a[1]**2

Out[10]: True
```

Listing 2.31: Show another way to compute the magnitude

```
In [11]: np.sqrt(a.dot(a)) == np.linalg.norm(a)

Out[11]: True
```

2.4.9 Geometric Interpretation of the Dot Product

Geometrically, the dot product is the magnitude of the project of one vector onto another vector. the green vector is $proj_{ab}$, the **projection** of **a** onto **b** - it is the part of **a** that is in the same direction as **b**. The red vector is the **error vector**, the part of **a** that is *not* in the same direction as **b**.

Listing 2.32: Compute the projection

```
In [12]: b_norm = np.linalg.norm(b)
         b_hat = b/b_norm
         a_dot_b = a.dot(b)
         proj_ab = b_hat*(a_dot_b/b_norm)
         err = a - proj_ab
```

Listing 2.33: Plot the projection

```
In [13]: initialize_2d_plot(x_min=-6, x_max=8, y_min=-1, y_max=6)
         draw_vector(a)
         draw_vector(proj_ab, kwargs={'color':'green', 'linewidth
         ':6, 'alpha':0.3})
         draw_vector(b)
         draw_vector(err,proj_ab, kwargs={'head_width':.1, '
         head_length':.25, 'color':'red', 'linewidth':2, 'alpha
         ':0.3})
```

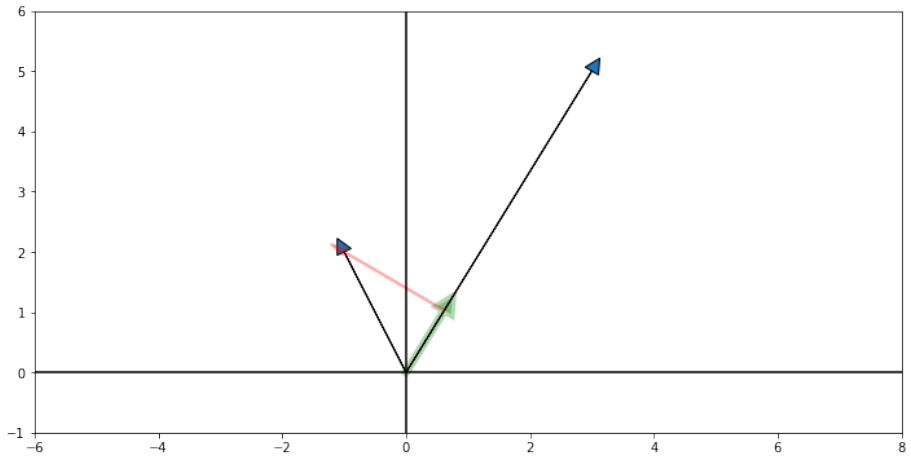

Figure 2.5: Plot the projection

2.4.10 The projection of a vector onto itself is the vector itself!!!

Thus, the dot product is the magnitude of the vector!

2.4.11 A Note on Writing Vectors

These two forms of vector representation are equivalent

$$(a, b, c, d) = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

This form is known as the **column vector** form.

A vector written as boldface later e.g. $\mathbf{u}$ is typically considered to be in column vector form.

2.4.12 A Note on Writing Vectors

A vector written as

$$\begin{pmatrix} e & f & g & h \end{pmatrix}$$

is considered to be written in the **row vector** form.

2.4.13 Changing Vectors Forms

Vectors can be transformed from one form to the other via the transpose operation.

$$\mathbf{m} = (1, 2, 3, 4)$$

then

$$\mathbf{m}^T = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}$$

2.4.14 `.T` as the transpose

We can achieve the transpose in `numpy` with the `.T` attribute.

Note that the transpose on a vector is weird. This is because `numpy` doesn't really draw clear distinctions between row vectors and column vectors.

Listing 2.34: Show something that is mathematically untrue, but computationally true

```
In [14]: a.T == a

Out[14]: array([ True,  True])
```

2.4.15 The importance of row and column vectors

This is important because we think of the dot product as **a row vector times a column vector**

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum u_i \cdot v_i = \mathbf{u}^T \mathbf{v} = \begin{pmatrix} 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} -3 \\ 3 \\ -2 \end{pmatrix} = 1 \cdot (-3) + 0 \cdot + (-1) \cdot (-2) = 1$$

Linear Algebraic Solutions
Solving Systems via Linear Algebra
Inverting the Problem
An Overdetermined System
Statistical Learning
Minimizing the Loss
The Normal Equations

Chapter 3

Case Study: Seeds

This will be a study of the Seeds data set available through the UCI Machine Learning Repository[1], where additional information on the data set can be found[2]. The data set is very similar to Fisher's Iris data set. It is used here as an alternative to Iris, as a simple and straightforward data set that is useful for introducing the main concepts of statistical learning and data science, especially Supervised and Unsupervised Learning and Classficication, Regression, and Clustering.

The UCI ML Repository describes the data as

> The data collected here is measurements of geometrical properties of kernels belonging to three different varieties of wheat. A soft X-ray technique and GRAINS package were used to construct all seven, real-valued attributes.

The **target** (also called labels, response, or dependent variable) fall into one of three classes just as in the Iris data set. The n, the number of observations (or points or rows), in the data set is very small. Iris contains 150 rows, whereas Seeds contains 210. The p, the number of salient **features** (also called predictors, input, or independent variables) is a bit larger than in Iris. Iris has four features, whereas Seeds has seven. In both data sets, all of the features are numerical and continuous.

3.1 Structure of the Case Study

The Case Study consists of three notebooks, a `README.md` file, and a Python file, `visualization.py`, arranged as follows[3]:

[1] https://archive.ics.uci.edu/ml/index.php
[2] https://archive.ics.uci.edu/ml/datasets/seeds
[3] The 'tree' tool is a useful command line tool that is available via 'apt', 'brew', and 'yum'.

Listing 3.1: The Structure of the Case Study

```
$ tree
.
|-- 00-introduction.ipynb
|-- 01-load-and-verify-data.ipynb
|-- 02-exploratory-data-analysis.ipynb
+-- visualization.py
```

The `00-introduction.ipynb` notebook contains information describing the data set and the structure of the project. The `01-load-and-verify-data.ipynb` notebook is the first phase of the Case Study, the **Loading and Verification of Data** and contains all steps required for obtaining, loading, preprocessing, verifying, and saving locally the data prior to any analysis work. The `02-exploratory-data-analysis.ipynb` notebook is the second phase of the Case Study and contains the **Exploratory Data Analysis** (EDA) work. The `visualization.py` file contains two Python functions that generate advanced visualizations.

3.1.1 Programming Method: Working in Multiple Notebooks

The reader will note that the work in this Case Study has been split across three notebooks. While it is possible to do the work in this Case Study in a single notebook, we will consider it a best practice to use multiple notebooks and to do so to help to structure the work. Each notebook will represent a major phase of the work, while minor phases of work will be designated with subheadings within a specific notebook. This helps with computing resources, especially the execution of JavaScript required to render a Jupyter Notebook, but also with referring to earlier work in later phases of a project.

3.1.2 Use of Statistical Learning Models for Exploratory Data Analysis

The EDA work done in this Case Study uses several statistical learning models inluding:

1. Linear Regression (a supervised regression model)
2. Logistic Regression (a supervised classification model)
3. K-Nearest Neighbors Classifier (a supervised classification model)
4. T-SNE (an unsupervised dimensionality reduction model)
5. K-Means (an unsupervised clustering model)

Each of these models is used with a specific EDA-related purpose. Statistical learning models, especially supervised learning models, are often used for building predictive models whose efficacy is evaluated accordingly. It is of note that these models were used here for the purposes of EDA, and as such, have not been subjected to the rigorous model assessment associated with building a predictive model.

3.2 Load and Verify Data

3.2.1 Load the Seeds Data Set from a URL

You will begin the Case Study by loading the data directly from a URL. The data is hosted in a whitespace-separated file on the UCI Machine Learning Repository.

3.2.2 Programming Method: Importing a Python Module as Alias

The first Python code you will run in this Case Study is code importing libraries necessary for the data science worked performed in this case study. Each of the `import` statements you will run is of the following format:

Listing 3.2: Importing a Python Module as Alias

```
import full_library_name as short_library_name
```

This has the effect of importing the entire library to be used under an alias. For example, you will import `numpy` using the import statement

Listing 3.3: Importing `numpy` with an alias

```
import numpy as np
```

This imports the entire `numpy` library, but the library but the library will only be available under the shortened alias `np`. This is a common technique in Python programming and useful when we will be making frequent reference to the library.

Import the Python Numerical Stack

The entire Case Study is performed using the Python Numerical Stack. You first load the libraries that comprise the Python numerical stack, including

- `matplotlib`
- `numpy`
- `pandas`
- `seaborn`.

Listing 3.4: Import the Python Numerical Stack

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import seaborn as sns
```

Run the IPython Magic Command `%matplotlib inline`

You make use of the IPython Magic[4] command `matplotlib` to configure the notebook for interactive `matplotlib` work. The `inline` argument specifies that we wish to work interactively with a Jupyter Notebook. Note that as with other IPython Magic commands, we begin this line of code with the `%` symbol. The result is that any images generated by our Python code will be rendered within the notebook while you work.

Listing 3.5: Render images inline with IPython magic

```
In [2]: %matplotlib inline
```

Define the Dataset URL

Next, you define a variable to hold the URL of the data set as a string. Note that you give the variable an all caps name. While this has no semantic meaning in Python, you do this to signify that this string should be thought of as a constant, that is that it is a value to be stored by the notebook that will never change.

Listing 3.6: Define the Dataset URL

```
In [3]: SEEDS_URL = ('https://archive.ics.uci.edu/' +
                     'ml/machine-learning-databases/' +
                     '00236/seeds_dataset.txt')
```

You display the string stored as `SEEDS_URL` to verify that it matches the expected value.

[4] https://ipython.readthedocs.io/en/stable/interactive/magics.html

Listing 3.7: Display the Dataset URL

```
In [4]: SEEDS_URL

Out[4]: 'https://archive.ics.uci.edu/ml/machine-learning-
        databases/00236/seeds_dataset.txt'
```

Display a sample of the Data Set URL

It is worth entering the `SEEDS_URL` into a browser, both to verify that the URL matches the correct URL, but also to get a preliminary sense of what the data should look like. Copy the value stored as `SEEDS_URL` and paste it into the address bar of any browser. If the URL is correct, you will see the data stored at the UCI Machine Learning Repository in your browser window as plain text. The first five rows of the data are as follows:

Listing 3.8: A Sample of the Seeds dataset text file

```
15.26    14.84    0.871    5.763    3.312    2.221    5.22     1
14.88    14.57    0.8811   5.554    3.333    1.018    4.956    1
14.29    14.09    0.905    5.291    3.337    2.699    4.825    1
13.84    13.94    0.8955   5.324    3.379    2.259    4.805    1
16.14    14.99    0.9034   5.658    3.562    1.355    5.175    1
...
```

Use `pd.read_csv()` to load data as `DataFrame`

Next, use the `pd.read_csv()` method available as part of the Pandas library to read the text file stored at that URL into a `DataFrame`.

Listing 3.9: Use `pd.read_csv()` to load data as `DataFrame`

```
In [5]: seeds_df = pd.read_csv(SEEDS_URL, header=None, sep='\s+')
```

A bit of special handling is required here. First, it is necessary to specify on load that the file has no header row. This is done using the argument `header=None`. Second, it is necessary to specify that this data is whitespace-separated. In other words, whitespace is used to separate the values in a row of data rather than the more conventional commas. This is done using the regular expression `\s+`. This argument signifies that one or more whitespace character(s) should be used as separator.

Manually Define Column Names

Because no header row is specified in the text file, it is necessary to manually specify the names of each column in your new `DataFrame`. These names were

obtained from the attribute information section at the UCI Machine Learning Repository. Note that you make all of the names computer-friendly by replacing whitespace with underscores.

The names of the columns of a `DataFrame` are stored as the attribute `df.columns`. You specify these names manually by assigning a list of the correct length to this attribute.

Listing 3.10: Manually Define Column Names

```
In [6]: seeds_df.columns = [
            "area",
            "perimeter",
            "compactness",
            "length_of_kernel",
            "width_of_kernel",
            "asymmetry_coefficient ",
            "length_of_kernel_groove",
            "seed_class"
        ]
```

Basic Data Verification

As a means of verifying that the data is properly loaded, you display both the type and the shape of the `DataFrame` and the first five rows. This is done using the `type()` function, the `.shape` attribute, and the `.head()` method, respectively. It is worth noting that `.shape` is an attribute of and `.head()` is a method of the `Pandas.DataFrame` class, while `type()` is a Python built-in function.

Listing 3.11: Show the type of `seeds_df`

```
In [7]: type(seeds_df)

Out[7]: pandas.core.frame.DataFrame
```

Listing 3.12: Show the shape of `seeds_df`

```
In [8]: seeds_df.shape

Out[8]: (210, 8)
```

As expected, `seeds_df` is a Pandas `DataFrame`. It has a shape of `(210, 8)`. This corresponds to a $n = 210$ and $p = 7$, plus 1 target column.

Listing 3.13: Show the `.head()` for a few columns of `seeds_df`

```
In [9]: seeds_df[
            ['area', 'perimeter',
             'compactness',
             'length_of_kernel']
        ].head()
```

Table 3.1: Show the `.head()` for a few columns of `seeds_df`

	area	perimeter	compactness	length_of_kernel
0	15.26	14.84	0.8710	5.763
1	14.88	14.57	0.8811	5.554
2	14.29	14.09	0.9050	5.291
3	13.84	13.94	0.8955	5.324
4	16.14	14.99	0.9034	5.658

After running `seeds_df.head()` make sure to compare the first five rows of the loaded `DataFrame` to the first five rows of the plain text file in your browser.

3.2.3 Verify Type Casting

During the loading process Pandas will explicitly cast each column into the most appropriate data type[5]. Based on information available at the UCI machine learning repository page for this data set, it is expected that the first seven columns to be floating-point, that is type `float`, and the eighth target column, `seed_class` to be categorical.

Display `DataFrame.dtypes`

Next, you use the `.dtypes` attribute to display the data types of your `seeds_df` `DataFrame`.

[5]https://rushter.com/blog/pandas-data-type-inference/

Listing 3.14: Display `DataFrame.dtypes`

```
In [10]: seeds_df.dtypes

Out[10]: area                        float64
         perimeter                   float64
         compactness                 float64
         length_of_kernel            float64
         width_of_kernel             float64
         asymmetry_coefficient       float64
         length_of_kernel_groove     float64
         seed_class                    int64
         dtype: object
```

Note that the first seven columns have been correctly cast, but `seed_class` has been incorrectly designated as type `int`. This is no doubt due to the fact that the `seed_class` categories are encoded numerically.

Display Unique Values for the `seed_class` Column

Using the `.unique()` method, you display the unique values of the `seed_class` column.

Listing 3.15: Display Unique Values for the `seed_class` Column

```
In [11]: seeds_df.seed_class.unique()

Out[11]: array([1, 2, 3])
```

Indeed, the seed classes, Kama, Rosa and Canadian, have been encoded as the numbers 1, 2, and 3. As all of the values are integers, Pandas incorrectly intuited that the column should be of type `int`.

Programming Method: Transform and Reassign a Variable

The programming pattern being used here is one that is fairly common. Here we take some variable, transform that variable, and then reassign the result to the original variable name. In the simplest sense, you might think about doing this with an integer

Listing 3.16: Transform and Reassign `my_int`

```
my_int = 4
my_int = my_int + 5
```

Of course, this pattern is so common that it has a shortcut that works in most popular programming languages

Listing 3.17: Simple Incremenation of `my_int`

```
my_int = 4
my_int += 5
```

In general, where it is not necessary to keep incremental results of a variable, we use a pattern like this one. For example, you might imagine recording the time that a certain process takes using the `time.time()` function.

Listing 3.18: Time a Process

```
import time
start = time.time()
## execute some process
elapsed_time = time.time() - start
```

Note that `elapsed_time` is a number of seconds stored as a `float`, while you may only be interested in the number of seconds as an `int`. A wasteful method would be

Listing 3.19: Cast `elapsed_time` as an integer

```
elapsed_time_int = int(elapsed_time)
```

Using this, you have actually stored two variables `elapsed_time` and `elapsed_time_int`. When transforming and reassigning, you simply overwrite the first variable with the change you wish to make

Listing 3.20: Cast `elapsed_time` as an integer and reassign to `my_int`

```
elapsed_time = int(elapsed_time)
```

Here you have made use of the transformation and reassignment programming pattern. We cast the time `elapsed_time` as an `int` and reassign this to the original `elapsed_time`, thus only ever storing a single value.

□ **Note**: It is worth noting that this is not always seen as a best practice, and in many cases it makes more sense to not transform your variables after you have assigned them. The practice is so common in imperative programming (as opposed to functional), that it is worth familiarizing yourself with it, even if you do not intend to handle your variables in this way.

3.2.4 Fix Typecasting Error

You will need to correct the incorrect typecasting manually. You will do this using the `.astype()` method. This method, applied to a column, returns a copy of that column cast into the requested data type. To complete the casting, you save this typecast copy back as the original column, using the transformation and reassignment programming pattern. This pattern is:

Listing 3.21: Recast the type of a Pandas Series

```
df.my_column = df.my_column.astype(<REQUESTEDTYPE>)
```

Cast `seed_class` as a Pandas Column of type `category`

Here, you cast `seed_class` as the Pandas data type, `category`[6].

Listing 3.22: Cast `seed_class` as a Pandas Column of type `category`

```
In [12]: seeds_df.seed_class = seeds_df.seed_class.astype('
         category')
```

Once more, you display the data types of your `seeds_df DataFrame`.

You now see that the target column, `seed_class` is now correctly encoded as categorical.

Listing 3.23: Display `DataFrame.dtypes`

```
In [13]: seeds_df.dtypes

Out[13]: area                        float64
         perimeter                   float64
         compactness                 float64
         length_of_kernel            float64
         width_of_kernel             float64
         asymmetry_coefficient       float64
         length_of_kernel_groove     float64
         seed_class                 category
         dtype: object
```

You now see that the target column, `seed_class` is correctly encoded as a `category`.

[6] https://pandas.pydata.org/pandas-docs/stable/categorical.html

3.2.5 Save Data Set as Local Files

So that you do not need to retrieve your data set from remote URL each time you wish work with it, you next use the Pandas `DataFrame` methods, `.to_csv()` and `.to_pickle()`, to store local versions the downloaded file. Each of these takes a string as its argument. This string will be used as the name of the file to be saved locally.

Programming Method: The Python Pickle Format

While you are no doubt familiar with the CSV format, this may be the first time you are learning about the Pickle format[7]. The Pickle format is a Python specific way to save information. When you write a `DataFrame` to disk as a Pickle you can load it in later, *exactly* as it was when you saved it. Furthermore, just like Pandas has a `pd.read_csv()` method, it also includes the method `pd.read_pickle()` to help you load a Pickled `DataFrame` at some later point.

Listing 3.24: Save the `DataFrame` as a CSV

```
In [14]: seeds_df.to_csv('seeds.csv')
```

Listing 3.25: Save the `DataFrame` as a pickle

```
In [15]: seeds_df.to_pickle('seeds.p')
```

3.3 Exploratory Data Analysis

Next, we perform some exploratory data and analysis (EDA) on our data set. This will be the terminal phase of this Case Study. In general, EDA is performed for the purposes of making inferences over a data set. In a project with a goal of developing a predictive model over the data set, EDA is useful for helping to determine which family of predictive models is most appropriate.

3.3.1 Import the Python Numerical Stack

To begin, you import the Python Numerical Stack. Note that this was done in a previous notebook, but it must be done for each notebook.

[7] https://docs.python.org/3/library/pickle.html

Listing 3.26: Import the Python Numerical Stack

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import seaborn as sns

        %matplotlib inline
```

Load Saved Data Set

In `01-load-and-verify-data.ipynb`, you retrieved the data set with which you are working from a remote URL and performed the basic preprocessing step of making sure that all columns are stored as the correct data type. After this, you saved the processed `DataFrame` to a local Pickle file called `seeds.p`. Here, you load the saved `DataFrame` for use in this notebook.

Listing 3.27: Load data from pickle file

```
In [2]: seeds_df = pd.read_pickle('seeds.p')
```

3.3.2 Visualization with a Pair Plot

Because of its simplicty, the first step in EDA is often to prepare a Pair Plot of the data set. This

Visualization: Prepare a Pair Plot

A simple yet powerful technique for beginning EDA is the Seaborn function `pairplot()`. Seabornsb is a Python statistical data visualization library build on top of the popular visualization library `matplotlib`. Essentially, it gives us a simple API for rendering many common statistical plots. `pairplot` takes a `DataFrame` as argument and returns a `PairGrid` plot of the pairwise relationships between each of the variables in the passed `DataFrame`.

Note that the central diagonal axis is treated differently than others. It shows the univariate distribution plot of each feature.

Note that it the plots above the central diagnoal axis are symmetric to the plot below the central diagonal axis and it is only necessary to examine one set of plots.

sup: https://seaborn.pydata.org

Listing 3.28: Pair plot of Seeds Dataset

```
In [3]: sns.pairplot(seeds_df);
```

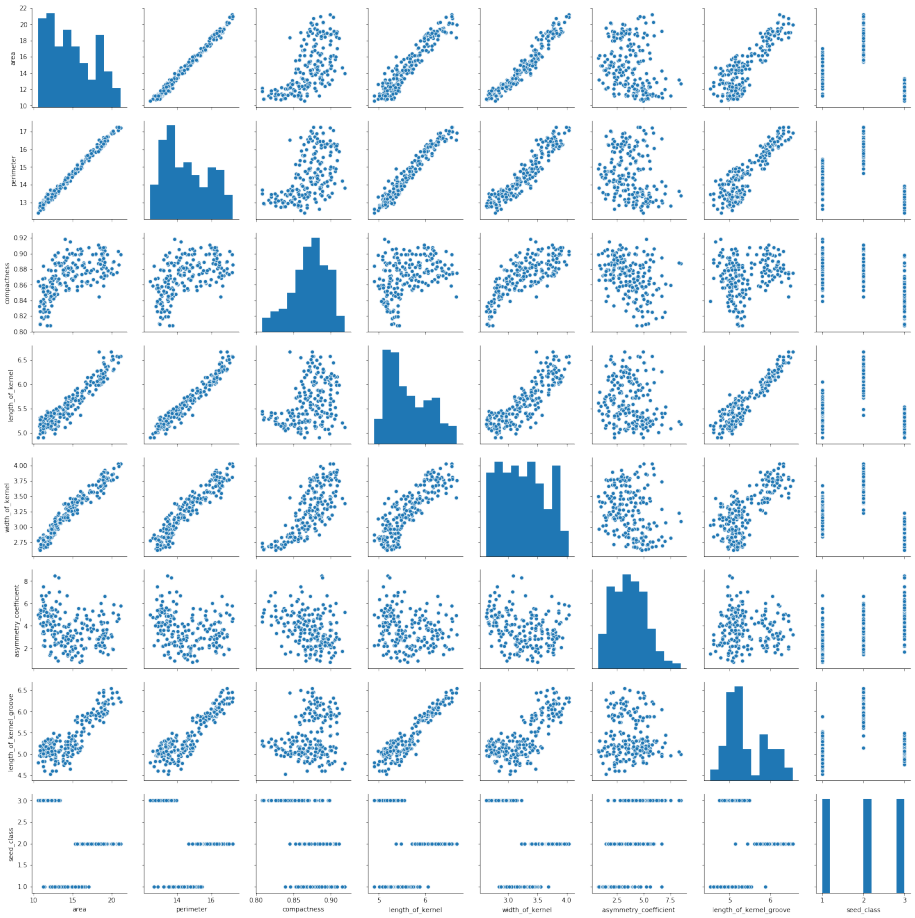

Figure 3.1: Pair plot of Seeds Dataset

Discussion: Interpretation of the `PairGrid` Visualization

We make note of a few things in the `PairGrid`.

There are several highly linear relationships in this dataset, for example, `area-perimeter`, as shown here:

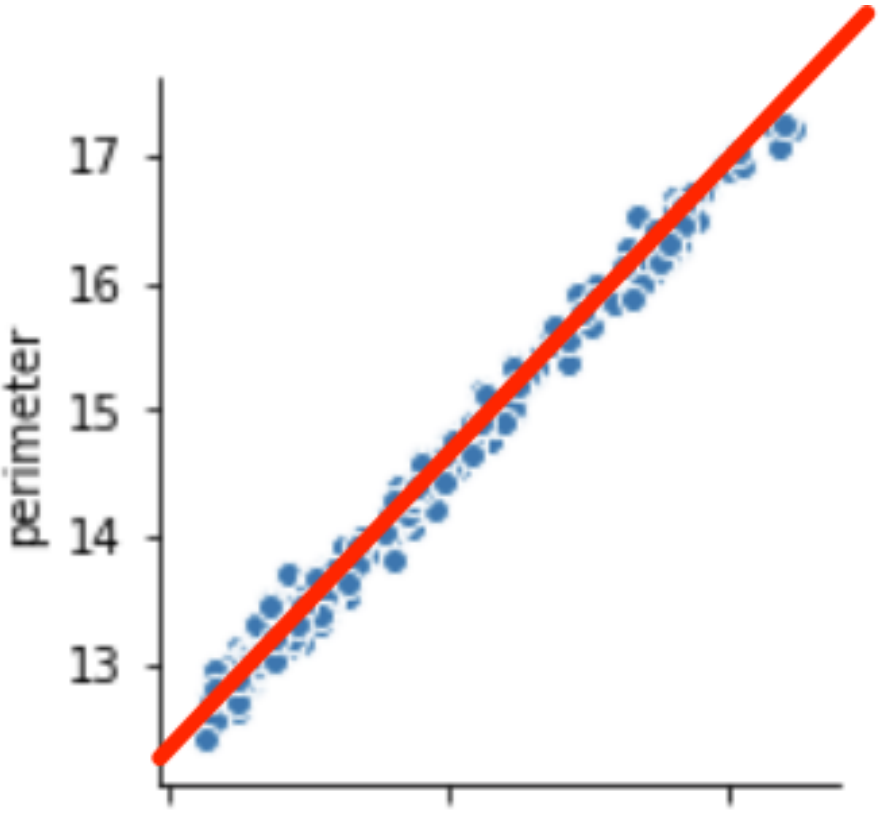

Figure 3.2: Relationship between Area and Perimeter

Note that there are other strongly linear relationships that are not as linear as `area-perimeter`, but are still strongly linear, e.g. `length_of_kernel-width_of_kernel`, as shown here:

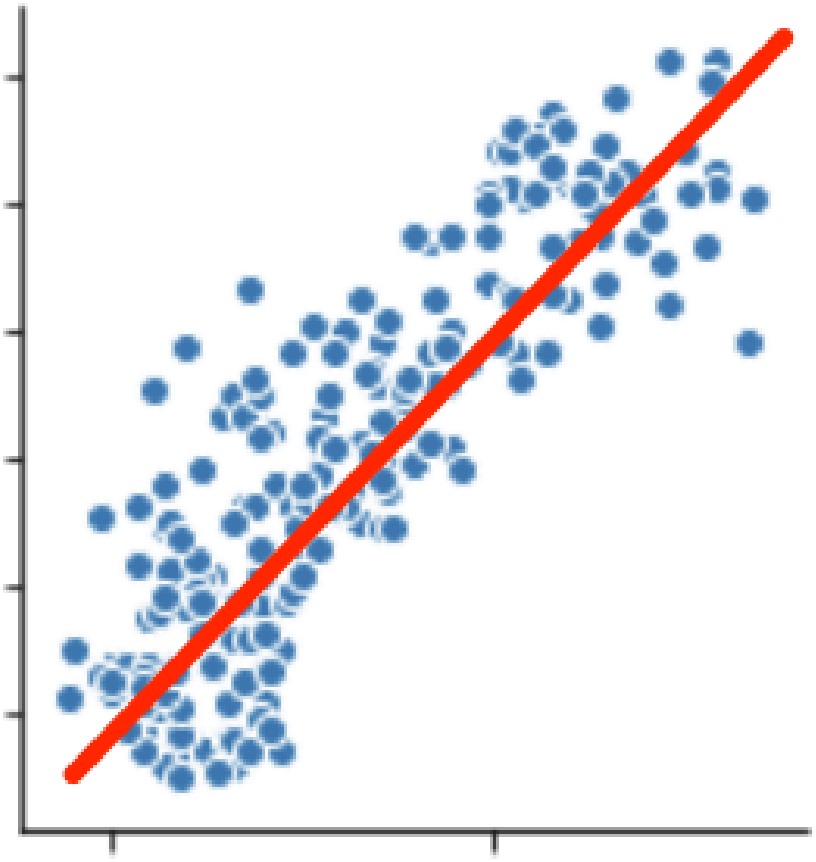

Figure 3.3: Relationship between Length and Width of Kernel

While, there does not appear to be a clear linear separation between the classes there is some indication that we will be able to separate the classes by feature variables. Here, we can see that `area` may be a feature we can use to separate the classes:

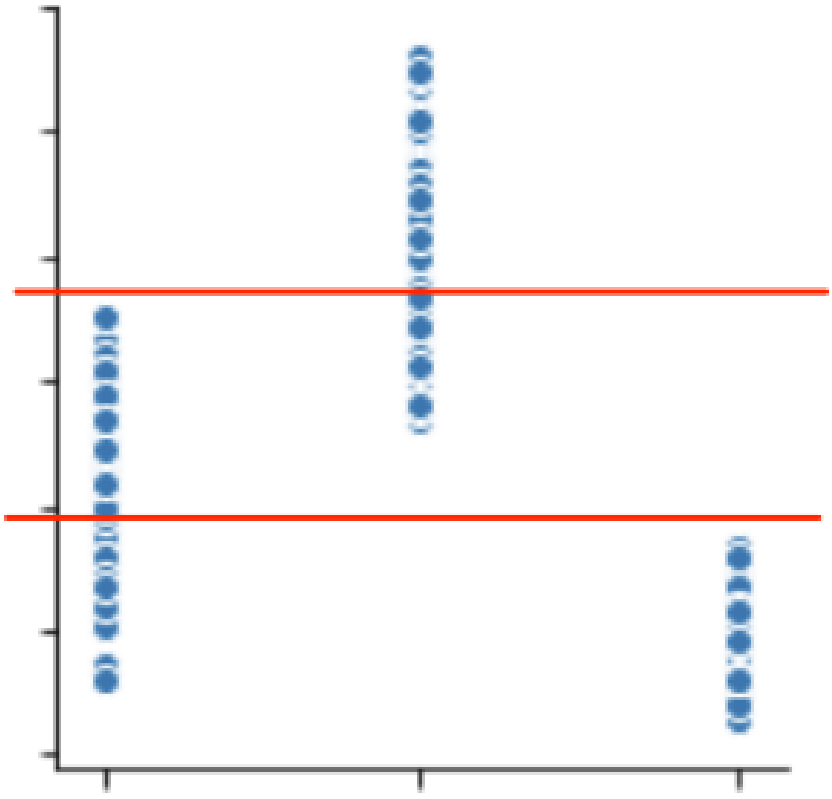

Figure 3.4: Class Separation

Discuss that these linearities can help us to choose a model.

3.3.3 Modeling for Inference

As you continue the exploration of this data set, you will use a machine learning model for the purposes of EDA, in this case a simple single variable classification model using Logistic Regression. In *Introduction to Statistical Learning with R*, James et al. propose that an inferential investigation of a data set may be interested in the following:

- Which predictors are associated with the response?
- What is the relationship between the response and each predictor?

- Can the relationship between Y and each predictor be adequately summarized using a linear equation, or is the relationship more complicated?

Here, you use single-variable logistic regression to assess the performance of each feature individually as a predictor of `seed_class`. This is a simple technique for making inferences about the strength of associations between each feature and the target class.

You will use Scikit-Learn for a simple implementation of the complex mathematical model, Logistic Regression. Logistic Regression is a regression model that creates the best linear separator of classes in a model. In other words, the model you build will find the line or lines that best separates each target class from the others.

Create `features` and `target` Subsets of Data Set

First, you create two subsets of the complete data set:

1. `features`, a `DataFrame` that includes everything but the target column
2. `target`, a `Series` or Pandas Column, that is just the target column

Listing 3.29: Drop target feature for analysis

```
In [4]: features = seeds_df.drop('seed_class', axis=1)
        target = seeds_df.seed_class
```

Programming Method: Classes and Objects

For those who are not familiar with object-oriented programming (OOP), this next step may be a bit of a challenge. Using the OOP paradigm, classes of objects are defined in code. To use a class, we must instantiate it as an object. To make this a bit more concrete, you might think of the class, Toyota Prius. This class is a kind of car, the definition of which exists in blueprints stored by the Toyota Corporation. The Prius that you drive, however, is not the class, but an object of the class Toyota Prius. It conforms to the definition as defined in the blueprints, but it is a specific instance of that class that you are driving around.

You do the same thing here with `lr_model` and `LogisticRegression()`. `LogisticRegression` is a class defined in the `sklearn.linear_model` library. `lr_model` is an object of class `LogisticRegression` that we will use to do our work.

Import the `LogisticRegression` Class and instantiate a `LogisticRegression` Model

We import the `LogisticRegression` class from the `sklearn.linear_model` library and instantiate the `lr_model` object.

Listing 3.30: Import the `LogisticRegression` Class and instantiate a `LogisticRegression` Model

```
In [5]: from sklearn.linear_model import LogisticRegression

        lr_model = LogisticRegression()
```

Programming Method: `for`-loop Over Features

Using a `for`-loop you iterate through each feature and use it to fit a Logistic Regression model on the data. You then score each of these models using Accuracy (the default scoring method for Scikit-Learn's classification models). Finally, each result is appended to a list called `lm_scores`. After running this cell, `lm_scores` will be a list of dictionaries containing the results of each test.

Programming Method: Fit and Score

With `lr_model` you are fitting and scoring a model for each step in the `for`-loop. This is a common technique with Scikit-Learn predictive models. You perform the `model.fit()` function on the data. This will execute the particular learning algorithm for your chosen model and store the fit model. You can then use the `model.score()` to score a particular fit model.

For a Supervised Learning model, both fitting and scoring take both features and target as an argument e.g. `model.fit(X, y)` and `model.score(X, y)`. Here, `X` is the features and `y` is the target. Note that any combination of features and target can be used. Care must be taken to make sure that the indices of `X` and `y` match up e.g. that the values of the *ith* row of `X` match the target in the *ith* row of `y`. Additionally, it is required that `X` and `y` have the same number of rows.

During scoring, the interim step of generating predictions is performed. In order to assess the model performance, a vector of `yhat` values is generated internall and compared to the actual values, i.e. the ground truth, stored in the `y` vector.

Listing 3.31: Fit and score individual features

```
In [6]: lm_scores = []
        for feat in features.columns:
            features_subset = features[[feat]]
            lr_model.fit(features_subset, target)
            lm_scores.append({
                'feature' : feat,
                'score' : lr_model.score(features_subset, target)
            })
```

Programming Method: Manually Build a `DataFrame`

A list of dictionaries is a useful structure for storing information, but it is not an ideal way for displaying information. This can be seen by displaying `lm_scores`.

Listing 3.32: Display list of dictionaries, `lm_scores`

```
In [7]: lm_scores

Out[7]: [{'feature': 'area', 'score': 0.8047619047619048},
         {'feature': 'perimeter', 'score': 0.719047619047619},
         {'feature': 'compactness', 'score': 0.5095238095238095},
         {'feature': 'length_of_kernel', 'score':
        0.6666666666666666},
         {'feature': 'width_of_kernel', 'score':
        0.7285714285714285},
         {'feature': 'asymmetry_coefficient ', 'score':
        0.5714285714285714},
         {'feature': 'length_of_kernel_groove', 'score':
        0.6047619047619047}]
```

Even this simple set of results becomes unwieldy.

Fortunately, it is trivial to convert a list of dictionaries with the same keys to a pandas `DataFrame`. We know that all of the dictionaries in `lm_scores` have the same keys, but we can confirm this with a simple Python list comprehension.

Listing 3.33: Show keys in each dictionary in `lm_scores`

```
In [8]: [d.keys() for d in lm_scores]

Out[8]: [dict_keys(['feature', 'score']),
         dict_keys(['feature', 'score']),
         dict_keys(['feature', 'score']),
         dict_keys(['feature', 'score']),
         dict_keys(['feature', 'score']),
         dict_keys(['feature', 'score']),
         dict_keys(['feature', 'score'])]
```

Provided that each dictionary has the same keys, each key will become the name of a column in the created `DataFrame` with the values of each dictionary forming a row.

Listing 3.34: Define and display `Dataframe lm_scores_df`

```
In [9]: lm_scores_df = pd.DataFrame(lm_scores)
        lm_scores_df
```

Table 3.2: Define and display `Dataframe lm_scores_df`

	feature	score
0	area	0.804762
1	perimeter	0.719048
2	compactness	0.509524
3	length_of_kernel	0.666667
4	width_of_kernel	0.728571
5	asymmetry_coefficient	0.571429
6	length_of_kernel_groove	0.604762

To make interpretation even easier, you perfom two transformations on the `DataFrame`. First, you assign the `feature` column to the `Index`, or row names, of the `DataFrame` using the `lm_scores_df.set_index()` function. The `inplace=True` argument specifies that this changes is to be written to the `DataFrame` stored in memory. Second, you sort the `score` column in descending order using the `lm_scores_df.sort_values(ascending=Fale)` function. Again, the `inplace=True` argument specifies that this changes is to be written to the `DataFrame` stored in memory.

Listing 3.35: Make `feature` the Index of `lm_scores_df`

```
In [10]: lm_scores_df.set_index('feature', inplace=True)
```

Listing 3.36: Sort `lm_scores_df` by Score

```
In [11]: lm_scores_df.sort_values('score', ascending=False,
         inplace=True)
```

Listing 3.37: Pickle scores for later use

```
In [12]: lm_scores_df.to_pickle('lm_scores')
```

Discussion: Interpreting Coefficients of Single Variable Logistic Regression

From these results, you can see that `area` is the strongest predictor as a single variable, followed by `width_of_kernel`. It is worth noting that these scores are using the features as a single variable predictor. The order of feature strength may be significantly different if using a multi-variable model.

3.3.4 T-SNE Model for Visualization of High-Dimensional Data

Given a data set with $p = 7$, you can imagine that it will be difficult to visualize this data sets using conventional means. Humans are accustomed to looking at data in only two dimensions. We are able to visualize in three dimensions, but three-dimensional visualization is already significantly more difficult then two dimensional. And beyond that, we are out of luck. Luckily for you, there is a fairly simple, at least in implementation, method for visualizing high-dimensional data using a model called T-SNE (often pronounced "tease knee"). The math behind this technique[8] can be fairly involved, but for your purposes it suffices to understand that you are creating a two-dimensional representation of our seven-dimensional features.

You will be building a T-SNE model over your data set simply for helping to prepare a simple visualization of our data. In other words, you are again building a machine learning model solely to help with the EDA process.

The implementation of takes only a few lines of code using the Python library, Scikit-Learn[9].

Import the `TSNE` Class

Import the model that you need from Scikit-Learn. The `TSNE` model is in the `sklearn.manifold` library.

Listing 3.38: Import the `TSNE` Class

```
In [13]: from sklearn.manifold import TSNE
```

Instantiate a `TSNE` Model

Next, you create a `TSNE()` object named `two_dim_model`. Here, you instantiate a `TSNE` object with 2 components.

[8] http://colah.github.io/posts/2014-10-Visualizing-MNIST/
[9] http://scikit-learn.org/stable/

Listing 3.39: Instantiate a `TSNE` Model

```
In [14]: two_dim_model = TSNE(n_components=2)
```

Note, that if you request the type of the `two_dim_model` object it tells you that it is a `TSNE` object from the `sklearn.manifold` library.

Listing 3.40: Display the `type` of `two_dim_model`

```
In [15]: type(two_dim_model)

Out[15]: sklearn.manifold.t_sne.TSNE
```

Fit the Model and Transform the Data

Next, you use the `two_dim_model` object to transform your data. The transformation that you are performing is to project the original data from seven dimensions to two dimensions. Note that in this case you are not interested in overwriting the original data with this transformation. For this reason, you save the transformed data as a new variable, `features_2d`.

Programming Method: Fit and Transform

With `two_dim_model` you performed the "fit" and the "transformation" in one step using the method `.fit_transform()`. In this, `two_dim_model` learned the nature of the data, that is, it was "fit" to the data. Then we used the fit model to "transform" the data, resulting in the two-dimensional `np.array`, `featured_2d`.

Listing 3.41: Fit and transform `features`

```
In [16]: features_2d = two_dim_model.fit_transform(features)
```

If you display the shape of each of the objects, indeed we can see that `features` has $p = 7$, while `features_2d` has $p = 2$.

Listing 3.42: Display shape of Features and shape of transformed Features

```
In [17]: features.shape, features_2d.shape

Out[17]: ((210, 7), (210, 2))
```

Scikit-Learn Transformation objects always return `np.array` objects. It is best to continue to work with a Pandas `DataFrame`. For this reason, you turn `features_2d` into a `DataFrame`. You give the columns of the new `DataFrame` the names `Component_1` and `Component_2`. This new `DataFrame` will be used in the next step to prepare a two-dimensional scatter plot by class.

Listing 3.43: Convert transformed features into `DataFrame`

```
In [18]: features_2d = pd.DataFrame(features_2d)
         features_2d.columns = ['Component_1', 'Component_2']
```

3.3.5 Visualization: Scatter Plot by Class

The file `visualization.py` includes a helper function `scatter_plot_by_class`. As the name suggests, this function returns a scatter plot of our data with each point labeled by it's class using color.

Programming Method: Import Functions From File

You have previously imported functions and classes from various libraries available to you using your local Python installation. Here, for the first time, you import a function from a project file, `visualization.py`. This can be done by simply creating a file with the `.py` suffix in the current working directory. In this way, the Python interpreter will be able to easily locate the file and is able to import any functions or classes from that file as if they were part of the installed and configured Python libraries.

Import `scatter_plot_by_class`

Listing 3.44: Import `scatter_plot_by_class`

```
In [19]: from visualization import scatter_plot_by_class
```

Programming Method: A Simple Method for Controlling Plots

For most of the plots you will be generating, you will begin the cell with the line

```
fig, ax = plt.subplots(NROWS, NCOLS, figsize=(XDIM,YDIM))
```

This gives a simple and easily repeatable pattern for controlling the appearance of rendered plots within a notebook. Most often, we are interested in controlling two aspects of our plots: 1) the dimension of the plot as rendered and 2) how many and the arrange of subplots to include. Here, you use this pattern in its most trivial sense creating a "set of subplots" with one row and

one column. This is a bit of overkill for this simple usage. The benefit is in using a single uniform and easily remembered pattern for working with plots and subplots.

When using this pattern, you will most often work with the `ax` object returned by the function. The `ax` object contains references to the individual subplots in a figure. In the case where a figure is created with 1 row and 1 column of plots i.e. a single plot, the `ax` object is simply a reference to that plot.

Listing 3.45: Prepare scatter plot by class

```
In [20]: fig, ax = plt.subplots(1,1,figsize=(20,5))
         scatter_plot_by_class(ax, features_2d, target)
```

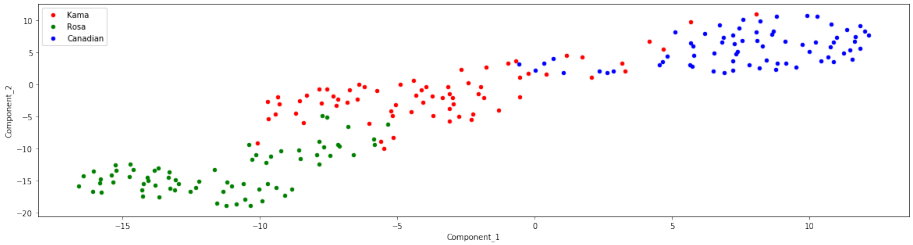

Figure 3.5: Prepare scatter plot by class

Discussion: Two-Dimensional Scatter Plot by Class

Note: TSNE is a randomized-process and your results may vary slightly from the results shown below.

The two-dimensional representation of the data suggests that you may be able to easily classify this data. You can see clear linear separations in the groupings of each of the three seed types. These linear separations indicate that a linear model, such as Logistic Regression, may perform well as a classification model on this data set.

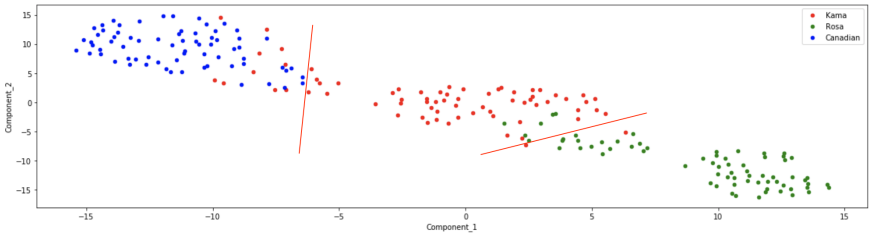

Figure 3.6: Scatter plot and class separation

You can also note clear shapes in the data and that class relationships appear to have some locality. This indicates that an instance-based model like K-nearest Neighbors may perform well as a classification model on this data set.

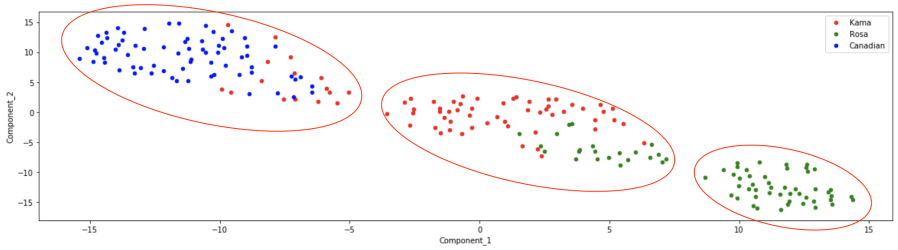

Figure 3.7: Scatter plot with possible clusters

3.3.6 Visualization: Cluster Model

To continue your exploration of this data set, you next prepare a cluster model. If you are able to create a strong cluster model, this will reinforce the idea that K-Nearest Neighbors is a good classification model for this data set.

To do this you follow a similar pattern as before using the Scikit-Learn library.

You can obtain the cluster model, `KMeans` from the `sklearn.cluster` library. KMeans[10] is amongst the most straightforward models in terms of interpretation. Using this model, you seek k clusters based upon the notion of the most unique *centroids* or mean values over the dataset, thus KMeans. As with TSNE, the implementation of this can be fairly complex, and you will simply rely on Scikit-Learn to implement this model.

Import the `KMeans` Class

Listing 3.46: Import the `KMeans` Class

```
In [21]: from sklearn.cluster import KMeans
```

Instantiate a `KMeans` Model

After importing the model, you instantiate `cluster_model`, an object of class `KMeans` with `n_clusters` set to 3.

[10] http://stanford.edu/ cpiech/cs221/handouts/kmeans.html

Listing 3.47: Instantiate a `KMeans` Model

```
In [22]: cluster_model = KMeans(n_clusters=3)
```

Fit the Model

With `cluster_model` you will not actually transform the data. Rather you will have the model learn from the data and then use the information learned. `cluster_model` will learn the three most likely cluster centroids over the data set and then use these to create new labels for the data. You will then compare these new labels to the original labels obtained from the UCI Machine Learning Repository stored as `target`.

Listing 3.48: Fit the Model

```
In [23]: cluster_model.fit(features)

Out[23]: KMeans(algorithm='auto', copy_x=True, init='k-means++',
         max_iter=300,
             n_clusters=3, n_init=10, n_jobs=1,
         precompute_distances='auto',
             random_state=None, tol=0.0001, verbose=0)
```

Access Cluster Labels from the Fit Model

Having fit the model you access the labels created during the fit using the model attribute `.labels_`.

Listing 3.49: Access Cluster Labels from the Fit Model

```
In [24]: cluster_labels = cluster_model.labels_
```

Finally, you prepare a plot showing both the original levels and the new labels generated by the cluster model. As expected, the cluster model does a decent job of labeling the data however as expected its struggles some of the boundaries between the different clusters.

Visualization: Scatter Plots by Class and Cluster

Note that here you use the figure preparation code `fig, ax = plt.subplots(2,1,figsize=(20,10))`. This time, you specify that the figure should have two rows and one column of subplots. As a result, the `ax` object is a list containing two references, one to each of the two subplots in the figure. Note that you refer to these subplots using `ax[0]` and `ax[1]`.

Listing 3.50: Visualization: Scatter Plots by Class and Cluster

```
In [25]: fig, ax = plt.subplots(2,1,figsize=(20,10))

         scatter_plot_by_class(ax[0], features_2d, target)
         scatter_plot_by_class(ax[1], features_2d, cluster_labels)

         ax[0].set_title('Actual Labels')
         ax[1].set_title('Cluster Model Labels')

         plt.show()
```

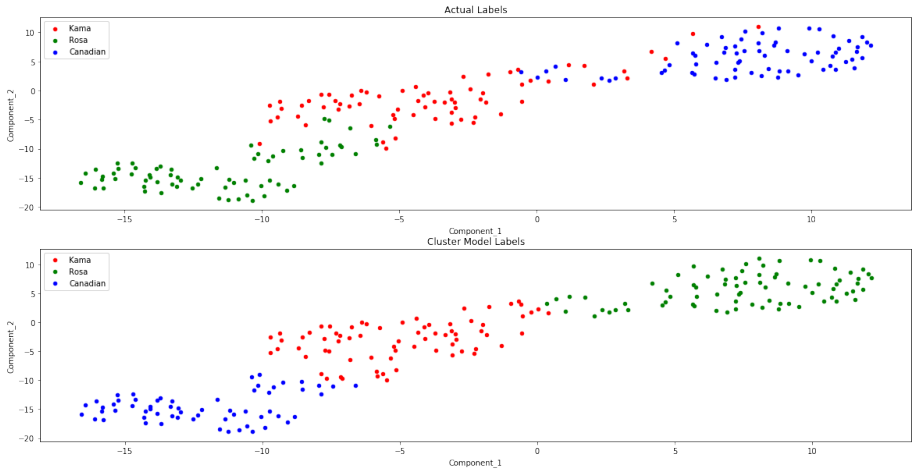

Figure 3.8: Visualization: Scatter Plots by Class and Cluster

Discussion: Scatter Plots by Class and Cluster

A visual inspection of the labels generated by the cluster model as compared to the original labels shows that the cluster model performs very strongly. The cluster model makes some error particularly at separation boundaries, but in general is able to identify the proper clustering of points.

It is of note that the cluster labels themselves are simply numerical references to the stored centroids. In other words, a label of `0` signifies that the point should be associated with the first stored mean. The cluster labels should not be taken to mean an association with a particular species of wheat. The cluster model is an unsupervised model and has no notion of the actual labels. Recall that during the fitting process

```
cluster_model.fit(features)
```

the model was not passed the labels at all.

It is also of note that while `features_2d` is used for plotting the labels returned by the cluster model, the model itself was fit using all seven features.

3.3.7 Plotting the Decision Boundary

During the **Modeling for Inference** phase you noted that the two strongest predictive features when used individually were `area` and `width_of_kernel`. Ypu use this knowledge to prepare the final visualization of this Case Study, a two-dimension scatter plot with a decision boundary. This plot takes a predictive model and uses the model to prepare a colored areas representing partitions of the feature space that should be considered indicative of a given class.

Import `scatter_plot_with_decision_boundary`

As before you import `scatter_plot_with_decision_boundary` from the local Python file `visualization.py`.

Listing 3.51: Import `scatter_plot_with_decision_boundary`

```
In [26]: from visualization import
         scatter_plot_with_decision_boundary
```

Import the `KNeighborsClassifier` Class and instantiate a `KNeighborsClassifier` Model

You will be preparing two decision boundary plots, one for a Logistic Regression model and the other for a K-Nearest Neighbors model. It is necessary to import `KNeighborsClassifier` from the `sklearn.neighbors` module.

Listing 3.52: Import the `KNeighborsClassifier` Class and instantiate a `KNeighborsClassifier` Model

```
In [27]: from sklearn.neighbors import KNeighborsClassifier

         knn_model = KNeighborsClassifier(n_neighbors=4)
```

Visualization: Scatter Plots with Decision Boundary

Listing 3.53: Visualization: Scatter Plots with Decision Boundary

```
In [28]: fig, ax = plt.subplots(2,1,figsize=(20,15))

         scatter_plot_with_decision_boundary(
             ax[0], features, target,
             'area', 'width_of_kernel',
             lr_model
         )

         scatter_plot_with_decision_boundary(
             ax[1], features, target,
             'area', 'width_of_kernel',
             knn_model
         )

         ax[0].set_title("Decision Boundary Logistic Regression
         Area - Width of Kernel")

         ax[1].set_title("Decision Boundary K Nearest Neighbors
         Area - Width of Kernel")

         plt.show()
```

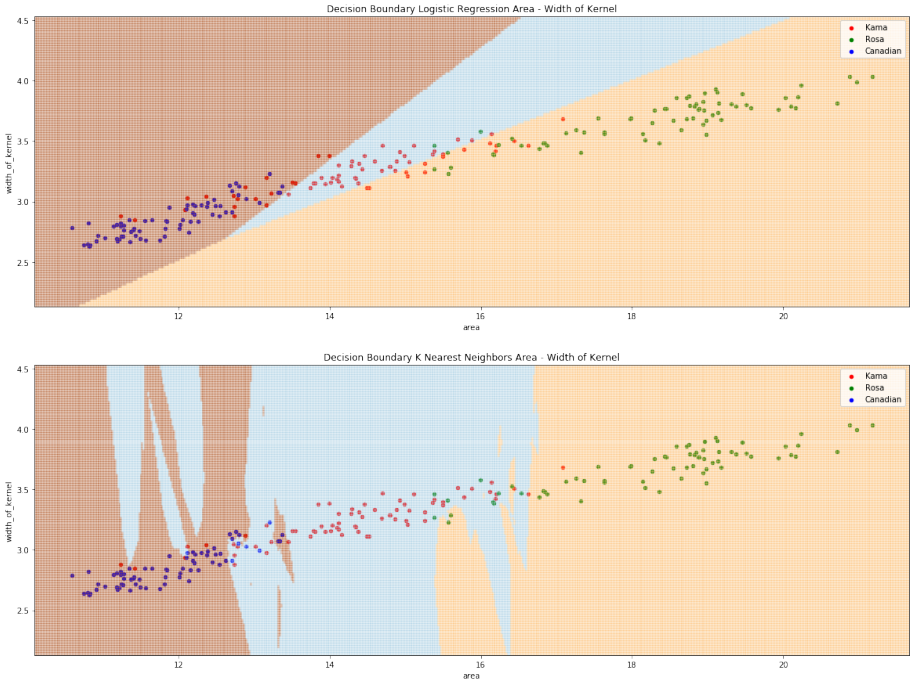

Figure 3.9: Visualization: Scatter Plots with Decision Boundary

Discussion: Scatter Plots with Decision Boundary

The scatter plots with decision boundary show that both models, each using only `area` and `width_of_kernel` to a reasonably good job of capturing the class relationship in the data. You noted some linear separation in the data class relationships and indeed the model is able to encode this reasonably well. It is of note that this linear separation breaks down in a few notable locations. Either of these linear separations, for example, would have returned fewer errors:

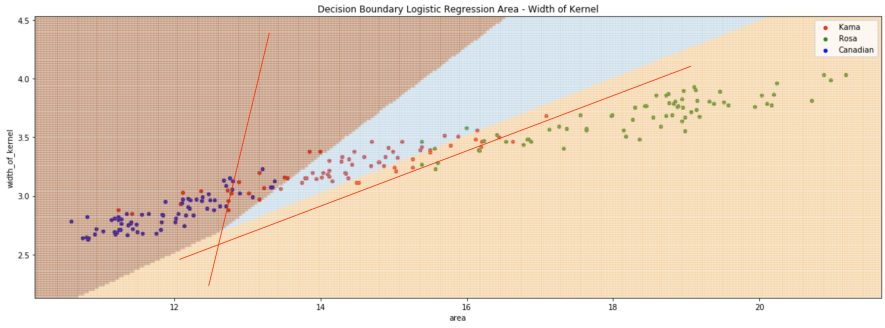

Figure 3.10: Scatter plot with decision boundary

3.4 Building a Predictive Model

Listing 3.54: Import the Python Numerical Stack

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import seaborn as sns

        %matplotlib inline
```

Listing 3.55: Load data from pickle file

```
In [2]: seeds_df = pd.read_pickle('seeds.p')
```

Listing 3.56: Import necessary models and functions from Scikit-Learn

```
In [3]: from sklearn.linear_model import LogisticRegression
        from sklearn.metrics import accuracy_score
        from sklearn.neighbors import KNeighborsClassifier
        from sklearn.model_selection import train_test_split
```

Listing 3.57: Instantiate a KNN Model

```
In [4]: knc = KNeighborsClassifier()
```

Listing 3.58: Prepare data for modeling

```
In [5]: features = seeds_df.drop('seed_class', axis=1)
        target = seeds_df.seed_class

        split_data = train_test_split(
            features,
            target,
            random_state = 10
        )

        X_train, X_test, y_train, y_test = split_data
```

Listing 3.59: Score Models for $k \in [2, 20)$

```
In [6]: scores = []

        for i in range(2,20):
            knc = KNeighborsClassifier(i)
            knc.fit(X_train, y_train)

            train_score = knc.score(X_train, y_train)
            test_score = knc.score(X_test, y_test)

            scores.append({
                'k' : i,
                'train_score' : train_score,
                'test_score' : test_score
            })
```

Listing 3.60: Display results from model fitting in a `DataFrame`

```
In [7]: scores_df = pd.DataFrame(scores)
        scores_df.set_index('k', inplace=True)
        scores_df
```

Table 3.3: Display results from model fitting in a `DataFrame`

_level_0 k	test_score _level_1	train_score _level_1
2	0.943396	0.961783
3	0.886792	0.961783
4	0.905660	0.936306
5	0.905660	0.917197
6	0.924528	0.917197
7	0.905660	0.910828
8	0.924528	0.910828
9	0.924528	0.917197
10	0.905660	0.917197
11	0.924528	0.910828
12	0.905660	0.917197
13	0.905660	0.898089
14	0.905660	0.898089
15	0.905660	0.904459
16	0.905660	0.898089
17	0.924528	0.904459
18	0.924528	0.904459
19	0.924528	0.910828

Listing 3.61: Display a learning curve for $k \in [2, 20)$

```
In [8]: scores_df.plot()

Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x11b010828>
```

Listing 3.62: Score Models for $k \in [2, 4, \ldots, 20)$

```
In [9]: scores = []
        for i in range(1,10):
            knc = KNeighborsClassifier(i*2)
            knc.fit(X_train, y_train)

            train_score = knc.score(X_train, y_train)
            test_score = knc.score(X_test, y_test)

            scores.append({
                'k' : i*2,
                'train_score' : train_score,
                'test_score' : test_score
            })
```

Listing 3.63: Display results from model fitting in a `DataFrame`

```
In [10]: scores_df = pd.DataFrame(scores)
         scores_df.set_index('k', inplace=True)
         scores_df
```

Table 3.4: Display results from model fitting in a `DataFrame`

_level_0 k	test_score _level_1	train_score _level_1
2	0.943396	0.961783
4	0.905660	0.936306
6	0.924528	0.917197
8	0.924528	0.910828
10	0.905660	0.917197
12	0.905660	0.917197
14	0.905660	0.898089
16	0.905660	0.898089
18	0.924528	0.904459

Listing 3.64: Display a learning curve for $k \in [2, 4, \ldots, 20)$

```
In [11]: scores_df.plot()

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x11d0c3588>
```

Listing 3.65: Load sorted features from pickle

```
In [12]: lm_scores_df = pd.read_pickle('lm_scores')

         sorted_features = list(lm_scores_df.index)
```

Listing 3.66: Instantiate two models

```
In [13]: lr = LogisticRegression()
         knc = KNeighborsClassifier(2)
```

Listing 3.67: Convert features to `Dataframes`

```
In [14]: X_train_df = pd.DataFrame(X_train)
         X_test_df = pd.DataFrame(X_test)

         X_train_df.columns = features.columns
         X_test_df.columns = features.columns
```

Listing 3.68: Score Models for subsets of data sorted by best feature

```
In [15]: final_performance_scores = []

         for i in range(len(sorted_features)):
             feats = sorted_features[:i+1]

             lr.fit(X_train_df[feats], y_train)
             knc.fit(X_train_df[feats], y_train)

             final_performance_scores.append({
                 'features' : ' '.join(feats),
                 'feature_added' : sorted_features[i],
                 'lr_train_score' : lr.score(X_train_df[feats],
         y_train),
                 'lr_test_score' : lr.score(X_test_df[feats],
         y_test),
                 'knc_train_score' : knc.score(X_train_df[feats],
         y_train),
                 'knc_test_score' : knc.score(X_test_df[feats],
         y_test)
             })
```

Listing 3.69: Display results from model fitting in a `DataFrame`

```
In [16]: final_performance_scores_df = pd.DataFrame(
         final_performance_scores)
         final_performance_scores_df.set_index('feature_added',
         inplace=True)
         final_performance_scores_df
```

Table 3.5: Display results from model fitting in a `DataFrame`

_level_0 feature_added	features _level_1	knc_test_score _level_1	knc_train _l
area	area	0.886792	0.8
width_of_kernel	area width_of_kernel	0.886792	0.9
perimeter	area width_of_kernel perimeter	0.886792	0.9
length_of_kernel	area width_of_kernel perimeter length_of_kernel	0.886792	0.9
length_of_kernel_groove	area width_of_kernel perimeter length_of_kerne...	0.943396	0.9
asymmetry_coefficient	area width_of_kernel perimeter length_of_kerne...	0.943396	0.9
compactness	area width_of_kernel perimeter length_of_kerne...	0.943396	0.9

Listing 3.70: Display learning curves

```
In [17]: final_performance_scores_df.plot(rot=45, figsize=(20,7))

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x11d1f56d8>
```

Chapter 4

Case Study: Ames

4.1 Grid Search with Cross-Validation

Listing 4.1: Load the preprocessed Datasets

```
In [2]: %run src/preprocessing-final.py
```

Listing 4.2: Display Dataframes currently loaded

```
In [2]: %whos DataFrame

Out[2]: Variable    Type          Data/Info
        ----------------------------------
        dataset_1   DataFrame           MSSubClass_20  MSSu
        <...>[1444 rows x 382 columns]
        dataset_2   DataFrame           MSSubClass_20  MSSu
        <...>[1444 rows x 390 columns]
        dataset_3   DataFrame           LotFrontage   LotAr
        <...>[1444 rows x 382 columns]
        dataset_4   DataFrame           LotFrontage   LotAr
        <...>[1444 rows x 390 columns]
```

Listing 4.3: Import the Python Numerical Stack

```
In [3]: import matplotlib.pyplot as plt
        import numpy as np
        import seaborn as sns
```

Listing 4.4: Import metrics from Scikit-Learn

```
In [4]: from sklearn.metrics import r2_score, mean_squared_error,
         mean_absolute_error
```

Listing 4.5: Suppress Warnings

```
In [4]: import warnings
        warnings.filterwarnings('ignore')
```

Listing 4.6: Load Grid Search Cross-Validator

```
In [6]: from sklearn.model_selection import GridSearchCV
```

Listing 4.7: Import Linear Models

```
In [11]: from sklearn.linear_model import Lasso, Ridge,
         SGDRegressor
         from sklearn.svm import LinearSVR, SVR, SVC
```

http://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

4.1.1 Most Appropriate Estimators

Listing 4.8: Define Grid Search parameters for four models

```
In [8]: gs_param_lasso = {
            'alpha' : np.logspace(-1,5,7)
        }

        gs_param_ridge = {
            'alpha' : np.logspace(-1,5,7)
        }

        gs_param_sgd = {
            'penalty' : ['l1', 'l2'],
            'alpha' : np.logspace(-1,5,7)
        }

        gs_param_linear_svr = {
            'C' : np.logspace(-5, 5, 7)

        }
```

Listing 4.9: Define Grid Search models

```
In [9]: lasso_grid_search = GridSearchCV(
            Lasso(),
            param_grid=gs_param_lasso,
            n_jobs=-1
        )

        ridge_grid_search = GridSearchCV(
            Ridge(),
            param_grid=gs_param_ridge,
            n_jobs=-1
        )

        sgd_grid_search = GridSearchCV(
            SGDRegressor(),
            param_grid=gs_param_ridge,
            n_jobs=-1
        )

        linearsvr_grid_search = GridSearchCV(
            LinearSVR(),
            param_grid=gs_param_linear_svr,
            n_jobs=-1
        )
```

Listing 4.10: Perform fit on Dataset 2

```
In [10]: lasso_grid_search.fit(dataset_2, target_2)
         ridge_grid_search.fit(dataset_2, target_2)
         sgd_grid_search.fit(dataset_2, target_2)
         linearsvr_grid_search.fit(dataset_2, target_2)

Out[10]: GridSearchCV(cv=None, error_score='raise',
                estimator=LinearSVR(C=1.0, dual=True, epsilon=0.0,
          fit_intercept=True,
               intercept_scaling=1.0, loss='epsilon_insensitive',
         max_iter=1000,
               random_state=None, tol=0.0001, verbose=0),
                fit_params=None, iid=True, n_jobs=-1,
                param_grid={'C': array([  1.00000e-05,   4.64159e
         -04,   2.15443e-02,   1.00000e+00,
                  4.64159e+01,   2.15443e+03,   1.00000e+05])},
                pre_dispatch='2*n_jobs', refit=True,
         return_train_score='warn',
                scoring=None, verbose=0)
```

Listing 4.11: Retrieve Best Estimator from Grid Search models

```
In [12]: best_lasso = lasso_grid_search.best_estimator_
         best_ridge = ridge_grid_search.best_estimator_
         best_sgd = sgd_grid_search.best_estimator_
         best_linearsvr = linearsvr_grid_search.best_estimator_
```

best_ridge

best_lasso

best_linearsvr

best_sgd

Listing 4.12: Display Best Scores from Grid Search models

```
In [17]: (lasso_grid_search.best_score_,
          ridge_grid_search.best_score_,
          sgd_grid_search.best_score_,
          linearsvr_grid_search.best_score_)

Out[17]: (0.89403721260739477,
          0.88839030757234749,
          0.85018593475934334,
          0.8802845170990975)
```

Listing 4.13: Display best model

```
In [18]: best_lasso

Out[18]: Lasso(alpha=100.0, copy_X=True, fit_intercept=True,
         max_iter=1000,
            normalize=False, positive=False, precompute=False,
         random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False)
```

4.1.2 Next Level

Listing 4.14: Import kernel enabled support vector machine

```
In [19]: from sklearn.svm import SVR
```

Listing 4.15: Import ensemble models

```
In [5]: from sklearn.ensemble import (AdaBoostRegressor,
                                      GradientBoostingRegressor,
                                      RandomForestRegressor)
```

Listing 4.16: Define Grid Search parameters for four models

```
In [6]: gs_param_svr = {
            'kernel' : ['rbf'],
            'C' : np.logspace(-5, 5, 7)
        }

        gs_param_adaboost = {

        }

        gs_param_gradboost = {
            'max_depth' : [1,2,3,4,5],
            'max_features' : ['sqrt', 'auto', 'log2']
        }

        gs_param_random_forest = {
            'n_estimators' : [10,20,50,100],
            'max_features' : ['sqrt', 'auto', 'log2']
        }
```

Listing 4.17: Define Grid Search models

```
In [22]: svr_grid_search = GridSearchCV(
             SVR(),
             param_grid=gs_param_svr,
             n_jobs=-1
         )

         adaboost_grid_search = GridSearchCV(
             AdaBoostRegressor(),
             param_grid=gs_param_adaboost,
             n_jobs=-1
         )

         gradboost_grid_search = GridSearchCV(
             GradientBoostingRegressor(),
             param_grid=gs_param_gradboost,
             n_jobs=-1
         )

         random_forest_grid_search = GridSearchCV(
             RandomForestRegressor(),
             param_grid=gs_param_random_forest,
             n_jobs=-1
         )
```

Listing 4.18: Perform fit on Dataset 2

```
In [23]: svr_grid_search.fit(dataset_2, target_2)
         adaboost_grid_search.fit(dataset_2, target_2)
         gradboost_grid_search.fit(dataset_2, target_2)
         random_forest_grid_search.fit(dataset_2, target_2)

Out[23]: GridSearchCV(cv=None, error_score='raise',
                estimator=RandomForestRegressor(bootstrap=True,
         criterion='mse', max_depth=None,
                    max_features='auto', max_leaf_nodes=None,
                    min_impurity_decrease=0.0, min_impurity_split=
         None,
                    min_samples_leaf=1, min_samples_split=2,
                    min_weight_fraction_leaf=0.0, n_estimators=10,
          n_jobs=1,
                    oob_score=False, random_state=None, verbose=0,
          warm_start=False),
                fit_params=None, iid=True, n_jobs=-1,
                param_grid={'n_estimators': [10, 20, 50, 100], '
         max_features': ['sqrt', 'auto', 'log2']},
                pre_dispatch='2*n_jobs', refit=True,
         return_train_score='warn',
                scoring=None, verbose=0)
```

Listing 4.19: Display score for gradient boosted model

```
In [24]: gradboost_grid_search.best_score_

Out[24]: 0.89917980524703134
```

Listing 4.20: Import pickle library

```
In [25]: import pickle
```

Listing 4.21: Import `joblib` from Scikit-Learn

```
In [26]: from sklearn.externals import joblib
```

Listing 4.22: Use `joblib` to export best Lasso model

```
In [27]: joblib.dump(best_lasso, 'best_lasso.p')

Out[27]: ['best_lasso.p']
```

Listing 4.23: Use `joblib` to load best Lasso model

```
In [28]: lasso_model = joblib.load('best_lasso.p')
```

Listing 4.24: Retrieve Best Estimator from Grid Search models

```
In [29]: best_svr = svr_grid_search.best_estimator_
         best_adaboost = adaboost_grid_search.best_estimator_
         best_gradboost = gradboost_grid_search.best_estimator_
         best_random_forest = random_forest_grid_search.
         best_estimator_
```

Listing 4.25: Display Best Scores from Grid Search models

```
In [30]: (svr_grid_search.best_score_,
          adaboost_grid_search.best_score_,
          gradboost_grid_search.best_score_,
          random_forest_grid_search.best_score_)

Out[30]: (0.85965192186105299,
          0.81063365281284472,
          0.89917980524703134,
          0.86191183641570013)
```

Listing 4.26: Display best model scores

```
In [26]: lasso_grid_search.best_score_, gradboost_grid_search.
         best_score_

Out[26]: (0.89400639458492726, 0.89929462910866398)
```

gradboost_grid_search.fit(dataset_4, target_4)

gradboost_grid_search = GridSearchCV(GradientBoostingRegressor(), cv=3, param_grid=gs_param_gradboost, n_jobs=-1) gradboost_grid_search.fit(dataset_4, target_4) gradboost_grid_search.best_score_

results = pd.DataFrame(gradboost_grid_search.cv_results_)

results[‘colors’] = pd.Series([‘red’, ‘moccasin’, ‘gold’, ‘blue’, ‘yellow’, ‘green’, ‘black’, ‘purple’, ‘cyan’, ‘teal’, ‘magenta’, ‘lavenderblush’, ‘deepskyblue’, ‘brown’, ‘coral’])

plt.figure(figsize=(12,5))
plt.scatter(results[‘mean_test_score’], results[‘std_test_score’], c=results[‘colors’]) plt.legend()

4.1.3 Neural Network

Listing 4.27: Import multilayer perceptron model

```
In [27]: from sklearn.neural_network import MLPRegressor
```

Listing 4.28: Define grid search parameters

```
In [28]: gs_param_nn = {
             'hidden_layer_sizes' : [
                 (8, ), (4,4), (2,2,2)
             ],
             'alpha' : np.logspace(-3,3,7)
         }
```

Listing 4.29: Define grid search model

```
In [29]: nn_grid_search = GridSearchCV(MLPRegressor(), param_grid=
         gs_param_nn, n_jobs=-1)
```

Listing 4.30: Fit model

```
In [30]: nn_grid_search.fit(dataset_2, target_2)

Out[30]: GridSearchCV(cv=None, error_score='raise',
                estimator=MLPRegressor(activation='relu', alpha
         =0.0001, batch_size='auto', beta_1=0.9,
                beta_2=0.999, early_stopping=False, epsilon=1e-08,
                hidden_layer_sizes=(100,), learning_rate='constant
         ',
                learning_rate_init=0.001, max_iter=200, momentum
         =0.9,
                nesterovs_momentum=True, power_t=0.5, random_state
         =None,
                shuffle=True, solver='adam', tol=0.0001,
         validation_fraction=0.1,
                verbose=False, warm_start=False),
                fit_params=None, iid=True, n_jobs=-1,
                param_grid={'hidden_layer_sizes': [(8,), (4, 4),
         (2, 2, 2)], 'alpha': array([  1.00000e-03,   1.00000e-02,
            1.00000e-01,   1.00000e+00,
                  1.00000e+01,   1.00000e+02,   1.00000e+03])},
                pre_dispatch='2*n_jobs', refit=True,
         return_train_score='warn',
                scoring=None, verbose=0)
```

Listing 4.31: Display best neural network score

```
In [31]: ###### Display best neural network score
         nn_grid_search.best_score_

Out[31]: -4.9890829237619414
```

Chapter 5

Configuring AWS

In this first chapter, you will configure your local system and create an AWS instance in order to do data science work. To do this work, you will use the containerization technology Docker in order to streamline the work of configuring and provisioning your data science platform. You will use Docker Compose to manage the platform and its two containers and rely on Amazon Web Services (AWS) to manage your hardware.

If you have your own Jupyter Notebook setup previously installed, feel free to use it, in which case, it is not necessary to work through this first chapter.

5.1 Amazon Web Services

If you have not already done so, set up an AWS account[1].

You will be using AWS to manage the hardware upon which your data science platform will run. We will leave the details of what exactly "hardware" means to AWS. This is to say that AWS may be allocating resources as a virtual machine, but for your purposes, the experience will be as if you are using a physical system across the room from you.

The most popular service offered by Amazon Web Services is the Elastic Compute Cloud (EC2), "a web service that provides secure, resizable compute capacity in the cloud"[2]. For our purposes, compute capacity means a cloud-based computer you will use to run your platform.

[1] As of 2018/08/25, detailed instructions for doing this can be obtained here: https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/.

[2] https://aws.amazon.com/ec2/

Readers new to AWS will be able to work through this text using the AWS Free Tier[3]. For the first 12 months following sign up, new users receive 750 Hours per month of EC2 time. This amounts to 31.25 days of availability and, provided that readers keep only one server running at a time, ensures that readers can work through this text at no cost.

5.2 Configure your Local System

In this first chapter, with the exception of some AWS system configuration in the browser, all of the work that you will be doing will take place at the command line. Work at the command line is done via a special type of application called a shell. A shell is a user interface that provides access to the operating system of a computer. We prefer the shell to other modes of working with computers because of its simplicity.

The most popular shell is the Bourne Again Shell or Bash. If you are using a Mac OS X system or a Linux system, you will already have Bash available to you in an application called Terminal. If you are using a Windows system, we recommend the use of Git-Bash[4]. We have no preference as to the settings used when configuring Git-Bash. The default settings are fine when installing the program. If you are using a Chromebook, we recommend the use of Termius, available from the Chrome Web Store.

5.2.1 SSH Keys

All of the work that you will be doing will take place remotely. As such, there is very little configuration to be done for the local system. The one thing that you will need to do is configure a set of SSH Keys to enable secure connection to the remote system you bring online.

[3] https://aws.amazon.com/free/
[4] https://gitforwindows.org

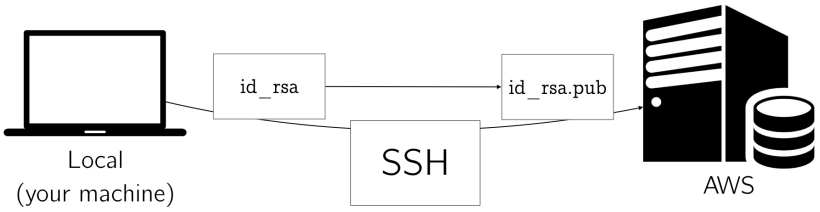

Figure 5.1: Connecting with SSH Keys

An SSH Key is a password-less method of authenticating to a remote system using public-key cryptography. Authentication is done using a key pair consisting of a public key, which can be shared publicly, and a private key, which is known only to the user (See Figure 5.1). One might think of the public key as the lock on your front door, accessible to anyone, and the private key as the key in your pocket so that only you are able to open your door and gain access to your home.

You will generate this key pair on our local system and then provide the public key to AWS so that it can be added to any system you wish to launch. You will keep the private key on our local system and use it whenever you wish to gain access.

5.3 Create a New Key Pair

You will use the Bash tool `ssh-keygen` to create a new key pair. To begin open a new terminal session[5], where you will examine whether or not you already have a key pair. Launching a new Bash session will put you in your home directory.

The canonical location for storing SSH Keys is in a folder called `~/.ssh` in our home directory[6]. Note that this directory begins with a `.` which makes it a

[5]On a Mac or Linux system, simply open the Terminal application. On Windows, open Git-Bash. On Chromebook, open Termius.

[6]On every system that we will use, the ∼ symbol is used as an alias for your home directory. The location of the actual home directory will vary by system. Ubuntu users' home directory will be '/home/username'. Mac OS X users' home directory will be '/Users/username'. Windows/Git-Bash users' home directory will be '/c/Users/username'.

hidden directory. In Listing 5.1, you use `cd` to navigate to our home directory and `ls -la` to display all of the contents of our home directory in a list.

home: On every system that we will use, the ~ symbol is used as an alias for your home directory. The location of the actual home directory will vary by system. Ubuntu users' home directory will be `/home/username`. Mac OS X users' home directory will be `/Users/username`. Windows/Git-Bash users' home directory will be `/c/Users/username`.

□ **Note:** Occasionally in code listings, I will truncate the output. If you see `...` in the listing, this should be taken to mean that there is additional output generated that is not important for the discussion.

Listing 5.1: List the contents of our home directory

```
$ cd ~
$ ls -la
...
drwxr-xr-x   21 joshuacook  staff    714 Jul 31  2017 .pylint.d
drwx------   11 joshuacook  staff    374 Jan 28 18:49 .ssh
drwxr-xr-x    6 joshuacook  staff    204 Feb  2 22:01 .vim
-rw-------    1 joshuacook  staff  20788 Feb 10 08:54 .viminfo
-rw-r--r--@   1 joshuacook  staff   1263 Jul 26  2017 .vimrc
drwx------@   5 joshuacook  staff    170 Aug 26 09:12 Applications
drwx------+  19 joshuacook  staff    646 Feb 11 09:32 Desktop
drwx------+   6 joshuacook  staff    204 Feb  4 12:18 Documents
...
```

My local system is running Mac OS X and has the `.ssh` folder already. As the directory already exists, in Listing 5.2, I list the contents of my `.ssh` directory.

Listing 5.2: Display contents of the `.ssh` directory

```
$ ls -la .ssh
total 96
drwx------  11 joshuacook  staff    374 Jan 28 18:49 .
drwxr-xr-x+ 74 joshuacook  staff   2516 Feb 10 08:54 ..
-rw-------   1 joshuacook  staff   1679 Jan 11 16:21 id_rsa
-rw-r--r--   1 joshuacook  staff    418 Jan 11 16:21 id_rsa.pub
```

As can be seen, I already have an SSH Keypair named `id_rsa` and `id_rsa.pub`. If this is true for you, as well, you should skip the next step and not create new SSH Keys (See Listing 5.3).

If when listing the home directory, you do not see a folder called `.ssh` or when displaying the contents of `.ssh` you do not see an SSH Keypair named `id_rsa` and `id_rsa.pub`, a new SSH Keypair will need to be created. In Listing 5.3, you create a new SSH Keypair using the `ssh-keygen` command line utility.

During the creation of the SSH Keypair, you will be prompted three times. The first asks where you should save the SSH Keypair, defaulting to the `.ssh/id_rsa` in our home directory. In Listing 5.3, you see that this is being done at `/Users/joshuacook/.ssh/id_rsa` on my local system where my username is `joshuacook`. The second and third prompts will ask for a passphrase to be added to the key. For our purposes, leaving this passphrase empty will be fine. In other words, the default options are preferable and you may simply hit <ENTER> three times.

Listing 5.3: Create a new SSH Keypair

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/joshuacook/.ssh/id_rsa)
          :
Created directory '/Users/joshuacook/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/joshuacook/.ssh/id_rsa
          .
Your public key has been saved in /Users/joshuacook/.ssh/id_rsa.pub
          .
The key fingerprint is:
SHA256:MkCnhaAzjcjRHCUc/pdUsxrk7be6+gNXTrEchJyFOqs joshuacook@LOCAL
The key's randomart image is:
+---[RSA 2048]-----+
| .===oo..o*o      |
|o+o=o+o o=oo      |
|*...o  +.o. +     |
| o  ...o=  =      |
|     .o+S.+.      |
|      .= ....     |
|      . o  .      |
|     E   ..       |
|       .o+o       |
+----[SHA256]-----+
```

You can verify the SSH Keypair you just created by displaying the Public Key in our shell (Listing 5.4). Here, you use the `cat` command, which concatenates the contents of `id_rsa.pub` to the shell output.

Listing 5.4: Display Public SSH Key

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDdnHPEiq1a4OsDDY+g9luWQS8pCjBmR
64MmsrQ9MaIaE5shIcFB1Kg3pGwJpypiZjoSh9pS55S9LckNsBfn8Ff42ALLj
R8y+WlJKVk/ODvDXgGVcCc0t/uTvxVx0bRruYxLW167J89UnxnJuRZDLeY9fD
OfIzSR5eglhCWVqiOzB+OsLqR1W04Xz1oStID78UiY5msW+EFg25Hg1wepYMC
JG/Zr43ByOYPGseUrbCqFBS1K1QnzfWRfEKHZbtEe6HbWwz1UDL2NrdFXxZAI
XYYoCVtl4WXd/WjDwSjbMmtf3BqenVKZcP2DQ9/W+geIGGjvOTfUdsCHennYI
EUfEEP joshuacook@LOCAL
```

5.4 Configure your AWS Account

That is the sum of the local configuration you will need to do in order to get started. The next thing you will need to do is configure our AWS Account. To do this, you will need to configure a Key Pair corresponding to the SSH Keypair on your local system. The AWS Key Pair is slightly misnamed as it is not in fact a pair, but rather is simply the public portion of the SSH Keypair you have on our local system.

To begin, log in to your AWS control panel and navigate to the EC2 Dashboard (Figure 5.2). First, access "Services" (Figure 5.2, #1) then access "EC2" (Figure 5.2, #2). The Services link can be accessed from any page in the AWS website.

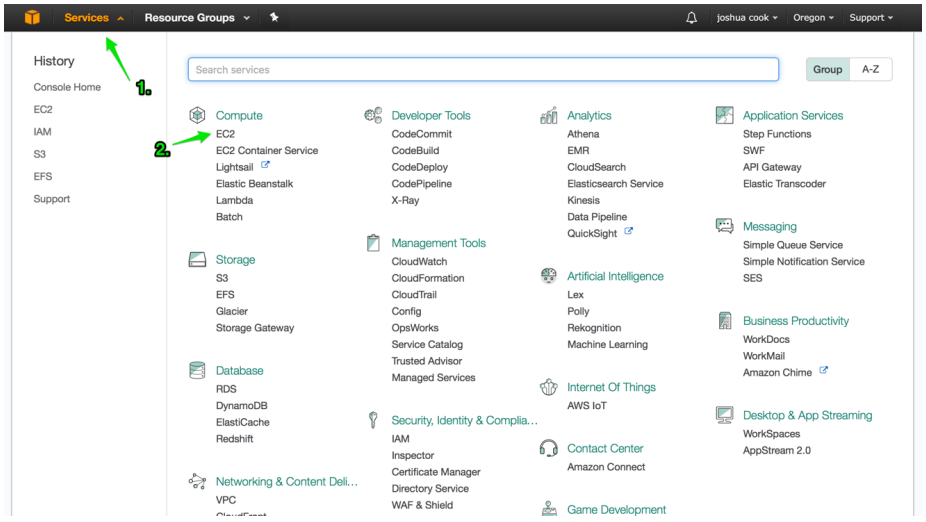

Figure 5.2: Access EC2 Dashboard

5.4.1 Add the Public Key to AWS

Once at the EC2 control panel, access the Key Pairs pane using either link (Figure 5.3).

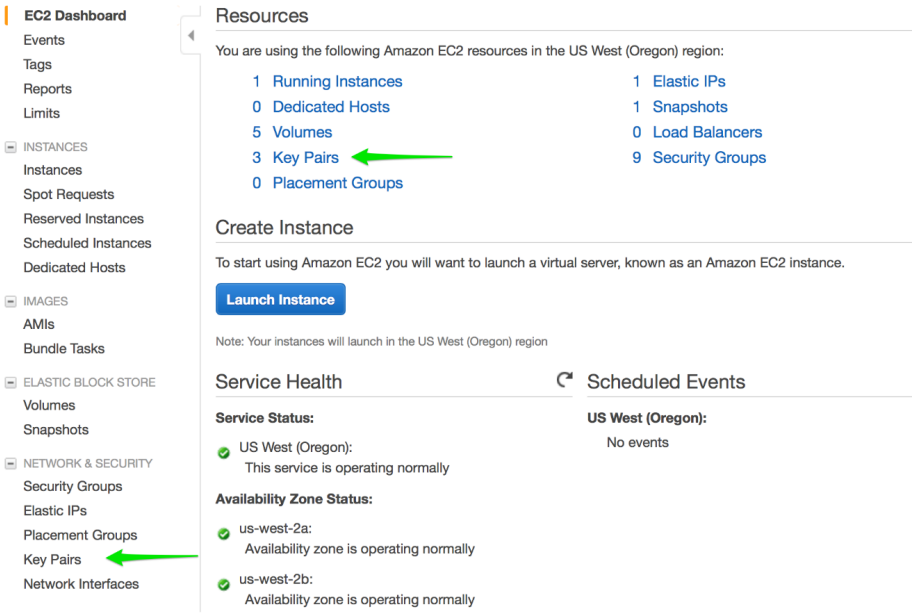

Figure 5.3: Access Key Pairs in the EC2 Dashboard

From the Key Pairs pane, choose "Import Key Pair." This will activate a modal that you can use to create a new key pair associated with a region on your AWS account. Make sure to give the key pair a computer-friendly name, like `from-MacBook-2018`. Paste the contents of your public key (`id_rsa.pub`) into the public key contents. Prior to clicking Import, your key should appear as in Figure 5.4. Click Import to create the new key.

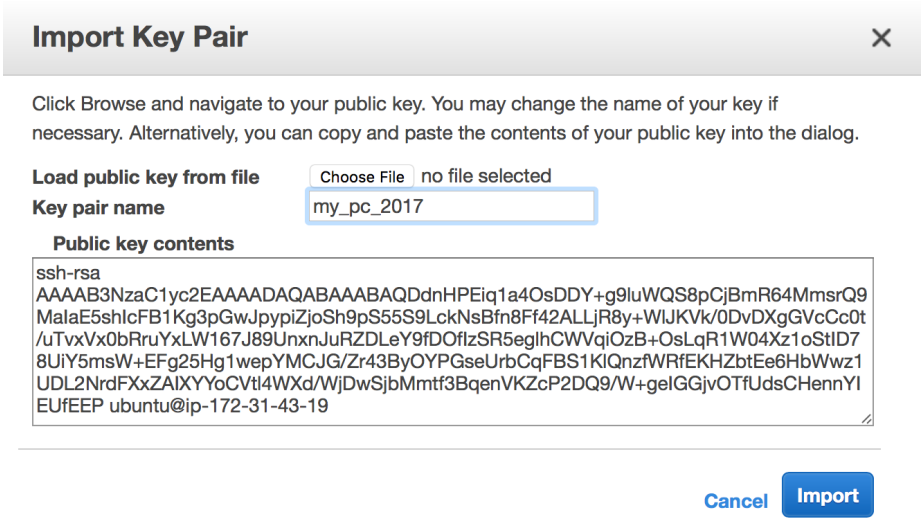

Figure 5.4: Import a New Public Key

You have created a key pair between AWS and your local system. When you create a new instance, you will instruct AWS to provision the instance with this public key and thus you will be able to access the cloud-based system from your local system using your private key.

5.5 Launch a New EC2 Instance

To create a new instance, start from the EC2 Dashboard and click the Launch Instance button (Figure 5.5).

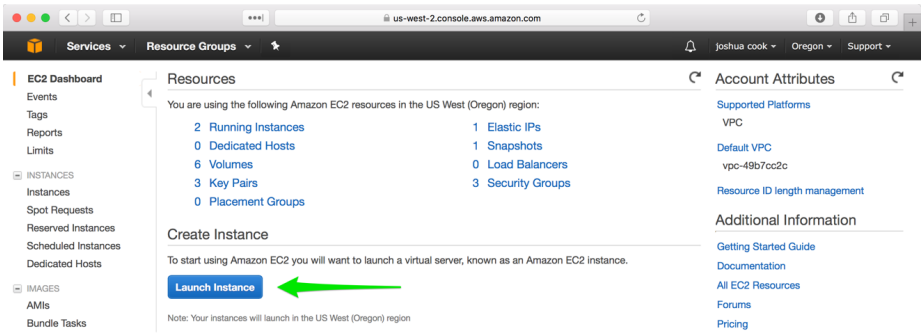

Figure 5.5: Begin the launch process for a new instance

5.5.1 Step 1: Choose an Amazon Machine Image (AMI)

The launching of a new instance is a multi-step process that walks the user through all configurations necessary. The **first tab** is "Choose AMI." An AMI is an Amazon Machine Image[7]. and contains the software you will need to run your sandbox machine. I recommend choosing the latest stable Ubuntu Server release that is free-tier eligible. At the time of writing, this was ami-efd0428f, Ubuntu Server 16.04 LTS (HVM), SSD Volume Type (Figure 5.6).

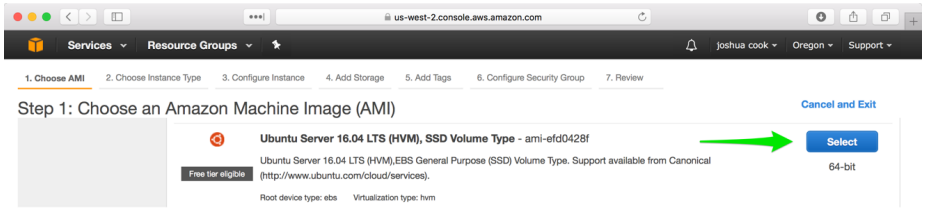

Figure 5.6: Choose the latest stable Ubuntu Server release as AMI

5.5.2 Step 2: Choose Instance Type

The **second tab** is "Choose Instance Type." In practice, I have found that the free tier, `t2.micro` (Figure 5.7), is sufficient for many applications. Furthermore, the instance type may always be changed later should the need present itself.

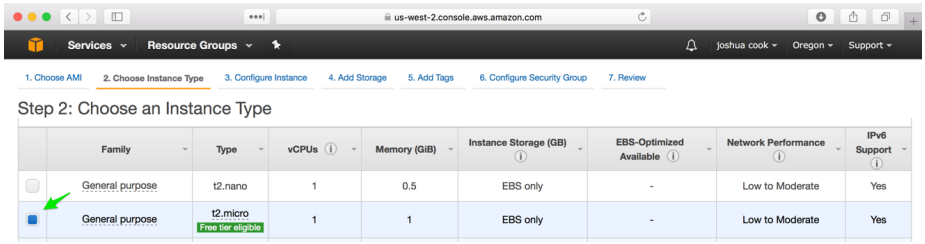

Figure 5.7: Choose `t2.micro` for Instance Type

5.5.3 Step 3: Configure Instance Details

The **third tab**, "Configure Instance," can be safely ignored.

5.5.4 Step 4: Add Storage

The **fourth tab** is "Add Storage." This option is also specific to intended usage. It should be noted that Jupyter Docker images can take up more than

[7] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

5GB of disk space in the local image cache. For this reason, it is recommended to raise the value from the default 8GB to 30GB. Furthermore, as noted on this tab:

```
Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnet
```

5.5.5 Step 5: Add Tags

The fifth tab, "Add Tags," can be safely ignored.

5.5.6 Step 6: Configure Security Group

The sixth tab, "Configure Security Group," is critical for the proper functioning of your systems. By default this tab will be set up to "Create a **new** security group". This will not work for us! Ultimately, we will be accessing our system via a web browser which we require at a minimum that port 80 is open. We recommend simply using the default group which will open our system on all ports. If greater security is required for your specific application a more restrictive security group may be defined and used.

Select the "default" security group (Figure 5.8).

Figure 5.8: Choose the latest stable Ubuntu Server release as AMI

☐ **Note:** You may receive a Warning stating, "Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only." This is expected and is okay.

5.5.7 Step 7: Review Instance Launch

Finally, click "Review and Launch." Here, you see the specific configuration of the EC2 instance you will be creating. Verify that you are creating a `t2.micro` (Figure 5.9, #2)running the latest free tier-eligible version of Ubuntu Server (Figure 5.9, #1)and that it is available to all traffic (Figure 5.9, #3), and then click the Launch button (Figure 5.9, #4).

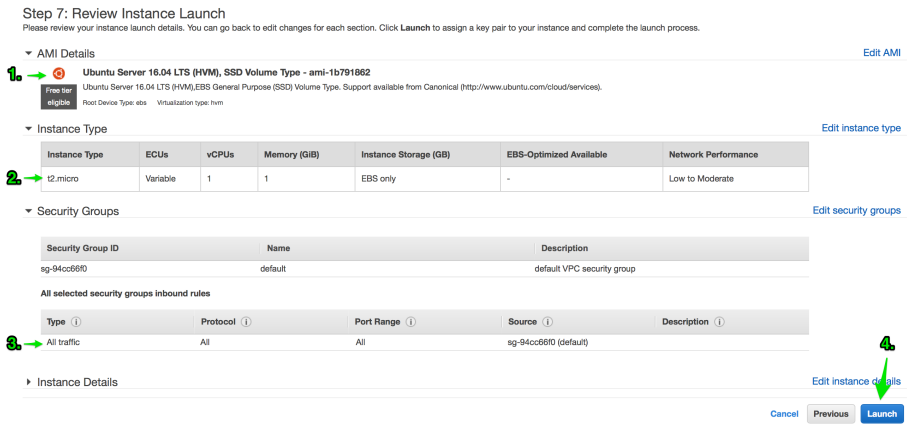

Figure 5.9: Review and launch the new instance

5.5.8 Add an SSH Key

In a final confirmation step, you will see a modal titled "Select an existing key pair or create a new key pair." Select the key pair you previously created. Check the box acknowledging access to that key pair and launch the instance (Figure 5.10).

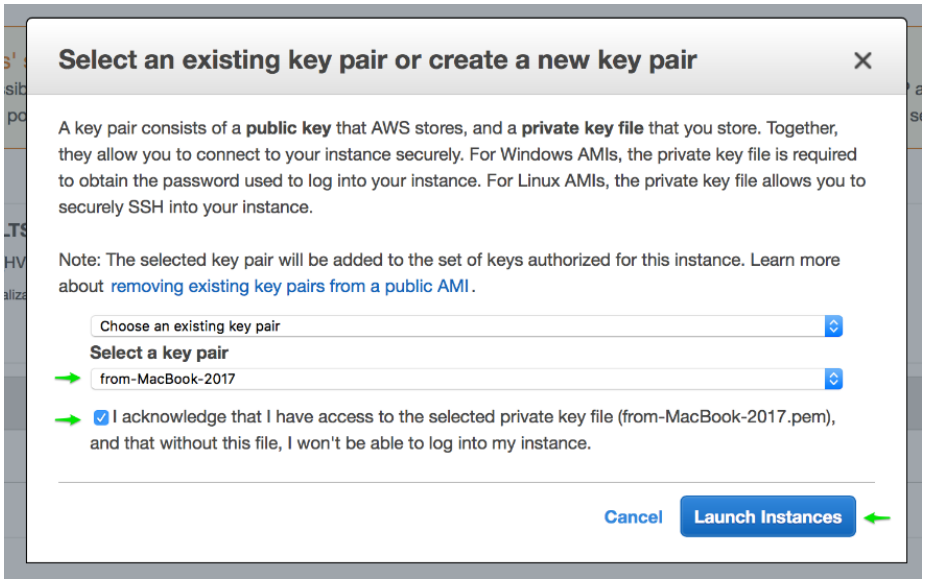

Figure 5.10: Add a key pair to the instance

☐ **Note:** If this step is not done correctly, that is, if the correct key pair is not added to the launching instance, the instance will need to be terminated and a new instance will need to be launched. There is now way to add a key pair to a running instance.

You should see a notification that the instance is now running. Click the View Instances tab in the lower right corner to be taken to the EC2 Dashboard Instances pane, where you should see your new instance running.

5.5.9 Examing the Newly Launched Instance

Make note of the IP address of the new instance (Figure 5.11).

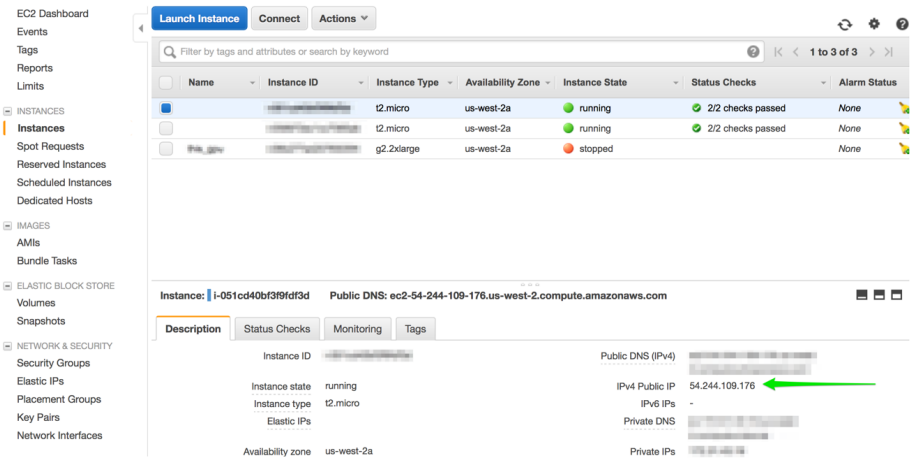

Figure 5.11: Note the IP address of the new instance

5.6 Git and Github

As you work through this text, you will be developing a series of data science projects. Tracking software development work is typically done using version control software. One of the most popular version control tools is `git`. Additonally, it can be useful to use a version control hosting service as a remote backup for work being tracked using `git`. The remote service we will use is Github.com. In my experience, learners who are new to version control often confuse `git` and Github, so it bares repeating – we will use `git` to track changes we make to our code and Github as a remote backup for these changes.

5.6.1 Configuring Github

We will assume that you have a Github account. Once this has been done, you will need to configue an SSH connection between AWS and Github. This

next part may potentially create a confusion. We are actually going to need a new SSH Keypair, this one associated with our AWS instance. This is because it is our AWS instance that will be connecting to Github, not our local machine (Figure See 5.12).

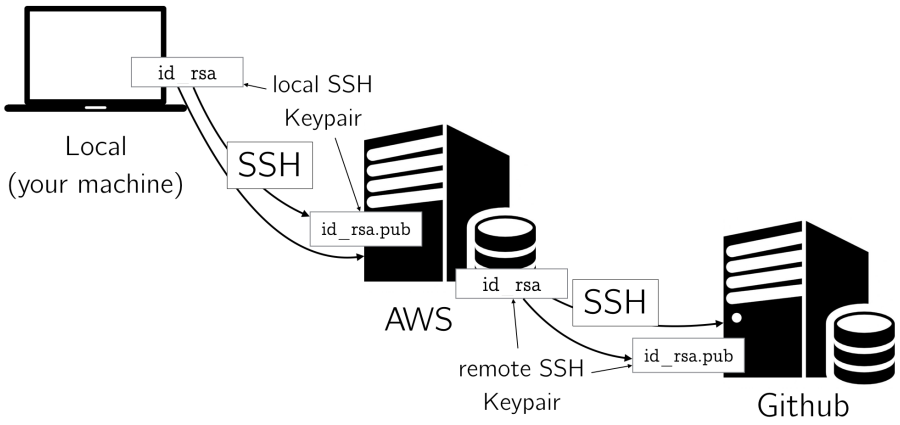

Figure 5.12: SSH Connections

5.6.2 Create a New Key Pair

In Listing 5.6, you create a new key pair on your remote AWS instance. In Listing 5.5, you connect to your new AWS instance. To do this we will use the IP address we made note of in Figure 5.11. We use SSH to connect to our remote AWS instance. Note that we use the username, `ubuntu`, the default username for the Ubuntu 16 AMI provided by AWS.

Listing 5.5: SSH into New Instance

```
$ ssh ubuntu@54.244.109.176
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-64-generic x86_64)
```

□ **Note:** The first time you access your EC2 instance, you should see the following message: `The authenticity of host '54.244.109.176 (54.244.109.176)' can't be established ... Are you sure you want to continue connecting (yes/no)? This is expected. You should hit <ENTER> to accept or type yes and hit <ENTER>.`

In Listing 5.6, you create a new SSH Keypair on our remote AWS instance. Again, during the creation of the SSH Keypair, you will be prompted three times. The first asks where you should save the SSH Keypair, defaulting to

the `.ssh/id_rsa` in our home directory. In Listing 5.6, you see that this is being done at `/home/ubuntu/.ssh/id_rsa`[8]. The second and third prompts will ask for a passphrase to be added to the key. For our purposes, leaving this passphrase empty will be fine. In other words, the default options are preferable and you may simply hit <ENTER> three times.

Listing 5.6: Create a new SSH Keypair

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
Created directory '/home/ubuntu/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_rsa.
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:ZSpFpgSRgRqlQom8yVBG2dZo1tgkPQdrmUGgMXGDtRY
ubuntu@ip-172-31-43-19
The key's randomart image is:
+---[RSA 2048]----+
|o=XBE/*.o        |
|==+=O**O.        |
|=o++o *o. o      |
|o+ . . . +       |
|       . S       |
|        .        |
|                 |
|                 |
|                 |
+----[SHA256]-----+
```

As before, you can verify the SSH Keypair you just created by displaying the Public Key in your shell (Listing 5.7). Again, you use the `cat` command, which concatenates the contents of `id_rsa.pub` to the shell output.

Listing 5.7: Display Public SSH Key

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDQ896GUMgCMAIW79gwF3ojRjcUYCKUKc8b+q
iQlah2jtr7s0K4WRGjktOy3lCCHO+1UK/GrzY1Y4VxCKoKJDH3G9N5UzyGhlxa/2Ah
kKxzHht1knyh/mkVGqYUhuHpXfxUQAstCFrIdp3G0MDPiko2qeJcBF7JSv1lLMbIuM
XuVU/Mzq6BU+tEogScYytmLckyEe1j8RJ+e5nBURwmkgj3UAN1DzmU/lVwLlltEpmC
DlOel4yEXAw8yBwM3GwjahfiBThvBHpsc43HxWrkM8Yi/kdDnvsDZYxU4zhXZPsPab
UY/LfxEod9c6Sui5W8GtAfdi6krnqbzxrKt81Mradh ubuntu@ip-172-31-43-19
```

[8]This should be the same for everyone now, as you should be working on an AWS `t2.micro` running an Ubuntu system where the user's name is `ubuntu`

5.6.3 Add the Public Key to Github

Previously, you added your local SSH public key to your AWS account. Now, you will add your AWS SSH public key to your Github account[9]. First, access the **Settings** for your account by clicking the profile photo in the upper-right corner of any page on Github. Next, in the user settings sidebar, select **SSH and GPG keys**. On the SSH and GPG Keys page, click **New SSH key**. On the next page, give your key a descriptive title e.g. "AWS Feb 2018" and then paste your AWS public key in to the "Key" field . Finally, click **Add SSH key** and confirm your Github password , if prompted.

5.7 Learning to read the Bash Prompt

During your work you will no doubt notice that an idle SSH connection may become disconnected and/or unresponsive. Should this happen, simply close the terminal session, launch a new one, and reconnect to the remote instance.

The most important thing is that you are aware of which system your current shell session is connected to. Shell prompts are designed to relay this information to you immediately. If you are new to working with Bash, you may need to train yourself to being aware of the prompt when typing. Listing 5.8 shows the default AWS Bash prompt. The information contained is the username, `ubuntu`, and the private IP address of the AWS instance. **This is not the public address you use to connect**. What is useful about this, is that we can immediately see that the user is `ubuntu`. This tells us we are connected to AWS.

Listing 5.8: The default AWS Bash prompt

```
ubuntu@ip-172-31-21-89:~$
```

Your local system will no doubt display something different (See Listing 5.9). Again, the important thing is to take note of what is displayed by the prompt and to learn to associate that prompt with the correct system. As you become a more advanced Bash user, you may wish to personalize your prompt, but for now it is imperative that you learn to read the prompt in order to always know to which system you are connected.

Listing 5.9: A local Bash prompt

```
joshuas-macbook-pro:~$
```

[9] https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/

5.8 Test your SSH Connection to Github

Having added you AWS Public Key to your Github account, you should verify your SSH connection from your AWS instance. In Listing 5.10, we attempt to connect to Github via SSH. As before, we receive a message about the authenticity of the connection. Again, type `yes`, and continue. If successful, you will see a message telling you have successfully authenticated but that Github does not provide shell access.

Listing 5.10: Verify Github SSH Key

```
ubuntu@ip-172-31-21-89:~$ ssh -T git@github.com
The authenticity of host 'github.com (IP ADDRESS)' can't be
established.
RSA key fingerprint is
16:27:ac:a5:76:28:2d:36:63:1b:56:4d:eb:df:a6:48.
Are you sure you want to continue connecting (yes/no)? yes
Hi username! You've successfully authenticated, but GitHub does
not provide shell access.
```

5.9 Docker

Having configured our SSH connections and provisioned a new AWS EC2 instance, it is time to get to the business of building your data science platform. To do this you will use the containerization platform Docker and its Docker Compose tool. While Docker is very easy to use, it can be difficult to understand for the uninitiated. In an earlier work, *Docker for Data Science*dockerfordatascience, I wrote:

dockerfordatascience: https://www.apress.com/us/book/9781484230114

> [Using Docker] we add a layer of complexity to our software, but in doing so gain the advantage of ensuring that our local development environment will be identical to any possible environment into which we would deploy the application.

It may be simpler, however, to simply think about using Docker as a way to manage a running process. Your system will be running two processes: an IPython shell and a PostgreSQL server. Were you to not use Docker, you would need to ensure that the AWS instance had all of the libraries required to run both of those processes (and keep those libraries up to date).

Instead, you will let Docker manage the processes using a container for each process. Each respective container will be run using a predefined image built using best practices and ready to run their respective process. The exchange

is this: you will take on the congitive burden of *understanding* what Docker is doing and Docker (and the Docker community) will take over the burden of making sure that your processes run.

5.9.1 Docker Compose

Docker Compose is a tool built for managing an application consisting of multiple containers. Using Docker Compose, it is possible to completely define an application using a simple text file. To make this conversation less abstract, let's have a look at the `docker-compose.yml` file you will use to define your first application (See Listing 5.11).

Listing 5.11: Your Data Science Application

```
version: "3"
services:
  ipython_shell:
    image: jupyter/scipy-notebook
  database:
    image: postgres
    volumes:
      - postgres_data:/var/lib/postgresql/data
volumes:
  postgres_data
```

That's it. This simple file completely defines a fully-functioning Data Science Application. In it, we define the two services we need: `ipython_shell` and `database`. These two services are defined using the `jupyter/scipy-notebook` and `postgres` images. When we launch the application, the images will be pulled from Docker Hub into our local memory and then launched. The one other thing we do is create a data volume `postgres_data`. We will use this as the data volume for our database server so that if for some reason we have to shut our system down, we do not lose our data. The data will exist on this volume independent of the services.

□ **Note:** Throughout this text, when discussing infrastructure, I may casually refer to containers, services, and processes. At the risk of annoying your local site reliability engineer, you may treat these as terms as synonomous. Care should be taken, however, not to confuse services/containers/processes and images. An image defines a service, but a service should be thought of as a living and active thing. You may loosely compare the service-image relationship to the object-class relationship in Object-Oriented Programming. A service is a running container defined by an image, just like an object is an instance of a class that exists in memory.

5.9.2 Installing and Configuring Docker

Installing Docker on your AWS instance is a downright trivial process. It consists of running an install script that can be obtained from Docker and then adding your user to the Docker group. In Listing 5.12, we run these two commands. First, we download the install script from https://get.docker.com, then immediately pipe the script into the shell (| `sh`).

□ **Note:** It is generally considered to be a significant security vulnerability to execute arbitrary code obtained from an unknown, or untrusted source. For our purposes, the source (https://get.docker/com) is considered trustworthy, we are using SSL to perform the curl, and in practice this is the method I use to install Docker. Still, it may make the security minded more comfortable to `curl` the script, inspect, and then run it.

Listing 5.12: Install Docker via a Shell Script

```
$ curl -sSL https://get.docker.com/ | sh
# Executing docker install script, commit: 1d31602
+ sudo -E sh -c apt-get update -qq >/dev/null
...

Client:
 Version:   18.02.0-ce
 API version:   1.36
 Go version:    go1.9.3
 Git commit:    fc4de44
 Built: Wed Feb  7 21:16:33 2018
 OS/Arch:   linux/amd64
 Experimental:  false
 Orchestrator:  swarm

Server:
 Engine:
  Version:  18.02.0-ce
  API version:  1.36 (minimum version 1.12)
  Go version:   go1.9.3
  Git commit:   fc4de44
  Built:    Wed Feb  7 21:15:05 2018
  OS/Arch:  linux/amd64
  Experimental: false

...
```

When the script completes there is one last thing to be done. In Listing 5.13, you add the `ubuntu` user to the `docker` group. By default, the command line docker client will require sudo access in order to issue commands to the docker daemon. You can add the `ubuntu` user to the `docker` group in order to allow the `ubuntu` user to issue commands to docker without sudo.

Listing 5.13: Add the Ubuntu User to the Docker Group

```
$ sudo usermod -aG docker ubuntu
```

Finally, in order to force the changes to take effect, you should disconnect and reconnect to their remote system. You can achieve this by typing `exit` or `ctrl-d` and then reconnecting via ssh to your EC2 instance.

5.9.3 Installing and Configuring Docker

Recall that regardless of your local operating system, you are working on an AWS EC2 Instance running the Linux variant, Ubuntu. As such, `docker-compose`can be installed using the instructions provided here: https://github.com/docker/compose/releases, which are written specifically for Linux machines. As of the writing of this book, this consists of two steps.

In Listing 5.14, you use `curl` to retrieve the `docker-compose` binary from Github. As of the writing of this book. the latest version of `docker-compose` was `1.19.0`. You should retrieve the latest version from the above url.

Listing 5.14: Retrieve `docker-compose` binary from Github

```
$ sudo curl -L https://github.com/docker/compose/releases/download
          /1.19.0/docker-compose-`uname -s`-`uname -m` -o /usr/
          local/bin/docker-compose
```

In Listing 5.15, we use the `chmod`chmod utility to allow `docker-compose` to be executed (`+x`).

chmod: The unix "change mode" utility. I pronounce it "shmod".

Listing 5.15: Enable Docker Compose to be Executed

```
$ sudo chmod +x /usr/local/bin/docker-compose
```

Finally, in Listing 5.16, we check the version of `docker-compose` against what we expect to have installed.

Listing 5.16: Check Docker-Compose Version

```
$ docker-compose -v
docker-compose version 1.19.0, build 9e633ef
```