Introduction to Data Science

Joshua Cook

May 5, 2018

Contents

Chapter 1

The Seeds Dataset

Welcome to data science! What follows is a rapid fire survey of some of the things that you will learn how to do on your path to becoming a data scientist. This text is being written as a series of interactive Jupyter notebooks. If you have the notebooks you can read them at the same time as you read the text and run the code interactively. I won't go into the details of setting up your Jupyter notebook server in this text, but if you would like to do this you can find instructions here: http://jupyter.readthedocs.io/en/latest/install.html

We will be working in a Jupyter notebook and running Python code interactively to study some data. First, we will load in some libraries that we need to do this work.

1.1 Interactive Programming

> Interactive computing is a dialog between people and machines. — Beki Grinter

1.1.1 IPython

IPython is short for interactive Python and is an highly-evolved Python REPL (read-eval-print loop) with a set of tools for interacting with any and all Python libraries.

Note Be careful not to confuse IPython, the command line REPL, and IPython Notebook, the legacy notebook server that has evolved into Jupyter.

When an IPython session is terminated all interactions are lost.

1.1.2 Jupyter

Jupyter is:

- a web-based interactive application
- an interactive code interpreter
- a presentation environment
- a new paradigm in programming
- a way to save complex terminal sessions.

Jupyter is fundamentally changing the way we write code.
Jupyter replaces if `__name__ == "__main__"`:.

1.1.3 Jupyter as Persistent Interactive Computing

- Jupyter Notebooks are the evolution of IPython.
- Jupyter allows users to combine live code, markdown and latex-rich text, images, plots, and more in a single document.

- Jupyter is the successor to the IPython notebook, Jupyter was renamed as the platform began to support other software kernels, in particular **Ju**lia, **Pyt**hon, and **R**.

Jupyter notebooks are saved as JSON files and at their most basic level are IPython sessions that can be repeatedly run.

The output of the last line in a Jupyter cell will be implicitly displayed by the Jupyter Notebook. Try the following (Type the strings as you see them, one per line) in a Jupyter Notebook cell:

Jupyter Cell 1 Hello, World!

```
In[1]: "Hello, World!"

       Out[1]: 'Hello, World!'
```

Hit `SHIFT+Enter` to execute the cell.

The Jupyter notebook has implicitly rendered the string that appeared on the last line of the cell. To look at this a bit more, we import the `display` function that is being used by the notebook.

Jupyter Cell 2 import the explicit display function

```
In[2]: from IPython.display import display
```

Next, we type three strings, each on a different line. Note that only the last string is displayed. Again, the Jupyter notebook has implicitly rendered the string that appeared on the last line of the cell.

Jupyter Cell 3 implicit call to display

```
In[3]: "Hello, my baby!"
       "Hello, my honey!"
       "Hello, my ragtime gal!"

       Out[3]: 'Hello, my ragtime gal!'
```

Finally, we explicitly display all of the strings by calling the `display` function ourselves.

Jupyter Cell 4 explicit call to display

```
In[4]: display("Hello, my baby!")
       display("Hello, my honey!")
       display("Hello, my ragtime gal!")

       'Hello, my baby!'
       'Hello, my honey!'
       'Hello, my ragtime gal!'
```

1.1.4 How to Program Interactively

Define a variable `my_integer` that is equal to 5.

Jupyter Cell 5 []

```
In[5]: my_integer = 5
```

Note that in defining the variable, the value was not actually displayed. To reiterate, the only thing that will be implicictly displayed is the last value in a cell.

Jupyter Cell 6 []

```
In[6]: my_integer

       Out[6]: 5
```

Redefine `my_integer` to be equal to 23.

Jupyter Cell 7 []

```
In[7]: my_integer = 23
```

Once more display the result.

Jupyter Cell 8 []

```
In[9]: my_integer

       Out[9]: 23
```

1.1.5 Plotting a Strange Attractor

This is not a very interesting use of interactive programming.
https://en.wikipedia.org/wiki/Attractor#Strange_attractor

Jupyter Cell 9 []

```
In[None]: from scipy.integrate import odeint as odeint
          import matplotlib.pyplot as plt
          import numpy as np
          from mpl_toolkits.mplot3d import Axes3D
```

Jupyter Cell 10 []

In[None]:
```
def create_attractor(sigma, beta, rho):
    r0 = [0.,1.,0.]
    t = np.linspace(0,50,50000)

    def lorenz_oscillator(r0, sigma=sigma, beta=beta, rho=rho):
        x, y, z = r0
        return [sigma*(y - x), rho*x - x*z - y, x*y - beta*z]

    return odeint(lorenz_oscillator, r0, t)
```

Jupyter Cell 11 []

In[None]:
```
fig = plt.figure(figsize=(20,6))
fig.gca(projection='3d')

X = create_attractor(10.0, 8./3., 5.0)
plt.plot(X[:,0],X[:,1], X[:,2], label='lorenz oscillator')
plt.legend()
```

Jupyter Cell 12 []

In[None]:
```
fig = plt.figure(figsize=(20,6))
fig.gca(projection='3d')

X = create_attractor(10.0, 8./3., 10.0)
plt.plot(X[:,0],X[:,1], X[:,2], label='lorenz oscillator')
plt.legend()
```

Jupyter Cell 13 []

In[None]:
```
fig = plt.figure(figsize=(20,6))
fig.gca(projection='3d')

X = create_attractor(10.0, 8./3., 15.0)
plt.plot(X[:,0],X[:,1], X[:,2], label='lorenz oscillator')
plt.legend()
```

Jupyter Cell 14 []

In[None]:
```
fig = plt.figure(figsize=(20,6))
fig.gca(projection='3d')

X = create_attractor(10.0, 8./3., 32.0)
plt.plot(X[:,0],X[:,1], X[:,2], label='lorenz oscillator')
plt.legend()
```

What do you think the `rho` parameter does?

1.2 Python

IPython magic are special commands that can be used to interact with your System.
Enter the following into an IPython session:

Jupyter Cell 15 []

```
In[None]: (an_integer,
           a_list,
           a_dictionary,
           a_set,
           a_tuple) = 1, [1,2,3], {'k': 1}, {1,2,2,3}, (1,2)
```

1.2.1 Writing a Function

Functions in python are written using the keyword `def`.

```
def function_name(arg1, arg2):
    output_1 = do_something_with(arg1)
    output_2 = do_something_with(arg2)
    return
```

Jupyter Cell 16 []

```
In[None]: from sys import getsizeof

          def sizeof_and_value(variable):
              return (getsizeof(variable), variable)
```

Jupyter Cell 17 []

```
In[None]: sizeof_and_value(an_integer)
```

Jupyter Cell 18 []

```
In[None]: sizeof_and_value(a_tuple)
```

Write a Function

Write a function named `type_and_value` that returns the value and the type of a variable that is passed to it.
Use the block below to define the function.

Jupyter Cell 19 []

In[None]:
```
def type_and_value(var):
    """return the type and value of a variable.
    """
    ### BEGIN SOLUTION
    return type(var), var
    ### END SOLUTION
```

Make sure that your function can pass this test:

Jupyter Cell 20 []

In[None]:
```
assert type_and_value(an_integer) == (int, 1)

### BEGIN HIDDEN TESTS
assert type_and_value(an_integer) == (int, 1)
assert type_and_value(a_list) == (list,[1,2,3])
assert type_and_value(a_dictionary) == (dict,{'k': 1})
assert type_and_value(a_set) == (set,{1,2,2,3})
assert type_and_value(a_tuple) == (tuple,(1,2))
### END HIDDEN TESTS
```

1.2.2 IPython Magic

1.2.3 %whos

Prints a table with some basic details about each identifier you have defined interactively.

Jupyter Cell 21 []

In[None]:
```
%whos
```

1.2.4 Bash in IPython

Bash is a command line language used to interactive with an operating system.
Some simple Bash commands can be run in IPython/Jupyter, including

- `pwd`
- `cd`
- `ls`

`pwd` - print working directory

Jupyter Cell 22 []

In[None]:
```
%pwd
```

`cd` **- change directory**

Jupyter Cell 23 []

```
In[None]: %cd src/
```

Jupyter Cell 24 []

```
In[None]: %pwd
```

`ls` **- list files**

Jupyter Cell 25 []

```
In[None]: %ls
```

Jupyter Cell 26 []

```
In[None]: # HIDDEN TEST

          ### BEGIN HIDDEN TESTS
          import os
          assert os.getcwd().split('/')[-1] == 'src'
          ### END HIDDEN TESTS
```

1.2.5 IPython Magic commands

There are many IPython magic commands, but some of the more useful are

- `run`
- `matplotlib inline`
- `whos`

Jupyter Cell 27 []

```
In[None]: %run a_simple_script.py
```

Jupyter Cell 28 []

```
In[None]: %matplotlib inline
```

Jupyter Cell 29 []

```
In[None]: %run a_simple_script.py
```

Why didn't the image show up the first time?

1.3 The Python Numerical Stack

Consists of:

- numpy/scipy (vectors and computational mathematics)
- pandas (dataframes)
- matplotlib (plotting)
- seaborn (statistical plotting)
- scikit-learn (machine learning)

Jupyter Cell 30 []

```
In[None]: import matplotlib.pyplot as plt
          import numpy as np
          import pandas as pd
          import seaborn as sns

          %matplotlib inline
```

Note We typically only import what we need from scikit-learn e.g.

Jupyter Cell 31 []

```
In[None]: from sklearn.linear_model import LinearRegression
```

Data Set Information:

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. A few of the images can be found at [Web Link]

Separating plane described above was obtained using Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree Construction Via Linear Programming." Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society, pp. 97-101, 1992], a classification method which uses linear programming to construct a decision tree. Relevant features were selected using an exhaustive search in the space of 1-4 features and 1-3 separating planes.

The actual linear program used to obtain the separating plane in the 3-dimensional space is that described in: [K. P. Bennett and O. L. Mangasarian: "Robust Linear Programming Discrimination of Two Linearly Inseparable Sets", Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server: ftp ftp.cs.wisc.edu cd math-prog/cpo-dataset/machine-learn/WDBC/

> The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by the British statistician and biologist Ronald Fisher in his 1936 paper The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis. — Wikipedia

Jupyter Cell 32 []

In[None]:
```
seeds_data = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/00236/seeds_dataset.
                         header=None, sep="\s+")
seeds_data.columns = [
"area",
"perimeter",
"compactness",
"length of kernel",
"width of kernel ",
"asymmetry coefficient ",
"length of kernel groove",
"Class"
]
```

Jupyter Cell 33 []

In[None]:
```
seeds_data.shape
```

What does `.shape` do?

1.3.1 Dataframes

We will have loaded the breast cancer data into a dataframe for ease of manipulation.

Jupyter Cell 34 []

In[None]:
```
seeds_data.head()
```

1.3.2 Pair Plot

We will use Searborn to prepare a **Pair Plot** of the Iris dataset. A Pair Plot is an array of scatter plots, one for each pair of features in the data. Rather than plotting a feature against itself, the diagonal is rendered as a **probability distribution** of the given feature.

Jupyter Cell 35 []

In[None]:
```
sns.pairplot(seeds_data)
```

1.3.3 List Comprehension

We will use a **list comprehension** to remove the units and white space from the feature names to make them more "computer-friendly".

In general, list comprehensions have this form:

```
lc = [do_something_to(var) for var in some_other_list]
```

Jupyter Cell 36 []

In[None]:
```
def square_number(x):
    return x**2
```

Jupyter Cell 37 []

In[None]:
```
[square_number(i) for i in (1,2,3,4,5)]
```

Write your own list comprehension

Write a function that uses a list comprehension to change this list

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

into this list

`[2,3,4,5,6]`

Jupyter Cell 38 []

In[None]:
```
def incr_list_by_1(lst):
    """returns a list where each value in the list has been incremented by one"""

    ### BEGIN SOLUTION
    return [i+1 for i in lst]
    ### END SOLUTUON
```

Jupyter Cell 39 []

In[None]:
```
assert incr_list_by_1([1,2,3,4,5]) == [2,3,4,5,6]

### BEGIN HIDDEN TESTS
assert incr_list_by_1([1,2,3,4,5,1,2,3,4,5]) == [2,3,4,5,6,2,3,4,5,6]
### END HIDDEN TESTS
```

1.3.4 Remove Unit and White Space from Feature Name

Here we use a list comprehension to change the feature names:

Jupyter Cell 40 []

In[None]:
```
seeds_data.columns
```

Jupyter Cell 41 []

In[None]:
```
def remove_unit_and_white_space(feature_name):
    feature_name = feature_name.replace(' (cm)','')
    feature_name = feature_name.replace(' ', '_')
    return feature_name
```

Jupyter Cell 42 []

In[None]:
```
seeds_data_features_names = [remove_unit_and_white_space(name) for name in seeds_data.columns]
```

Jupyter Cell 43 []

In[None]:
```
seeds_data_features_names
```

Jupyter Cell 44 []

In[None]:
```
seeds_data.columns = seeds_data_features_names
seeds_data.head()
```

1.3.5 Export to CSV

Ultimately, we will export a CSV of the dataframe to disk. This will make it easy to access the same data from both Python and R.

Jupyter Cell 45 []

In[None]:
```
%ls
```

Jupyter Cell 46 []

In[None]:
```
%mkdir -p data
```

Jupyter Cell 47 []

In[None]:
```
%ls
```

Jupyter Cell 48 []

```
In[None]: bc_data.to_csv('data/seeds_data.csv')
```

1.4 Prediction

1.4.1 Why estimate f?

We can think of a given dataset upon which we are working as a representation of some actual phenomenon. We can imagine there to be some sort of "universal" function, f, that was used to generate the data, one that we can never truly know.

As data scientists, we will seek to estimate this function. We will call our estimate $\hat{f}$ ("eff hat").

There are two main reasons we might want to estimate f with $\hat{f}$:

- prediction
  - given some set of known inputs and known outputs, we may wish to create some function that can take a new set of inputs and predict what the output would be for these inputs
- inference
  - given some set of known inputs and (optionally) known outputs, we may wish to understand how the inputs (and outputs) interact with each other

Jupyter Cell 49 []

```
In[None]: %pwd
```

What does `pwd` tell us? What does this mean in the context of a Jupyter Notebook? Why would it be important to think about this before we load a csv file?

Jupyter Cell 50 []

```
In[None]: %ls
```

Jupyter Cell 51 []

```
In[None]: seeds_data.describe()
```

Jupyter Cell 52 []

```
In[None]: plt.figure(1, (20,10))

          sns.pairplot(seeds_data)
```

Having a look at the pair plot, we might say that we are able to the uniformity of cell shape using the uniformity of cell size.

Jupyter Cell 53 []

```
In[None]: plt.figure(1, (20,5))

          sns.regplot('area','perimeter', data=seeds_data)
```

1.4.2 Linear Regression

We might build a **simple regression model** to do this for us using scikit-learn. Here, the **input variable** would be `petal length` and the **output variable** would be `petal width`.

We will usually refer to our input variable(s) as **feature(s)** and our output variable as the **target**.

1.4.3 Build a Simple Regression Model

Jupyter Cell 54 []

```
In[None]: from patsy import dmatrices

          target, features = dmatrices("perimeter ~ area", seeds_data)
```

Jupyter Cell 55 []

```
In[None]: from sklearn.linear_model import LinearRegression
```

Jupyter Cell 56 []

```
In[None]: linear_regression_model = LinearRegression()
          linear_regression_model.fit(features, target)
```

1.4.4 Plot the Results

Having prepared the regression model, we use it to make predictions.

We then plot the predictions versus the actual values.

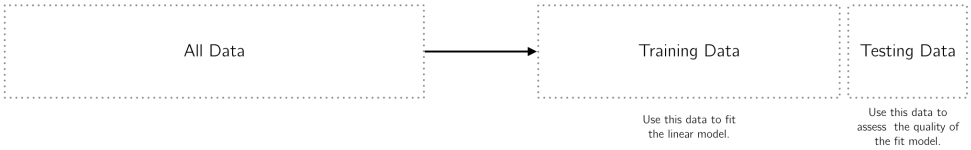
Jupyter Cell 57 []

```
In[None]: plt.figure(1, (20,5))

          sns.regplot('area','perimeter', data=seeds_data)

          predictions = linear_regression_model.predict(features)
          plt.scatter(seeds_data.area, predictions, marker='x', color='red')
```

What does this plot show us?

1.5 The Train-Test Split

What if we wish to know how well petal width can be predicted for unseen data?



1.5.1 Overfitting and Underfitting

When fitting a model for making predictions, a model is only as good as its ability to work on unseen data. A model that does not learn the underlying patterns in the data is said to be **underfit**. A model that learns that underlying patterns in the data too well is said to be **overfit**.

1.5.2 Learning Too Well is a Problem!?

It may seem odd to think of a model that has learned to well as being bad in some way, but recall that we are looking to make predictions with new input data. A model that is overfit will have learned the patterns in its **training** data, but will also have learned the noise inherent to this data. New input data will have completely different noise *by definition*. A model that is overfit will be poor at generalization and will not perform well on data it has never seen.

1.5.3 The Train-Test Split

Of course, we will not have access to the new data we will use at the time of fitting the model. We will have to simulate new data in some way. We do this, by creating **test** data using some fraction of the original data we started with.

Jupyter Cell 58 []

```
In[None]: from sklearn.model_selection import train_test_split
```

Jupyter Cell 59 []

```
In[None]: target, features = dmatrices("perimeter ~ area", seeds_data)
```

Of course, we will not have access to the new data we will use at the time of fitting the model. We will have to simulate new data in some way. We do this, by creating **test** data using some fraction of the original data we started with.

Jupyter Cell 60 []

```
In[None]: (features_train,
           features_test,
           target_train,
           target_test) = train_test_split(features, target, random_state=42)
```

Jupyter Cell 61 []

In[None]:
```
(features_train.shape,
 target_train.shape,
 features_test.shape,
 target_test.shape)
```

Jupyter Cell 62 []

In[None]:
```
features_test[:5]
```

Jupyter Cell 63 []

In[None]:
```
linear_regression_model = LinearRegression(fit_intercept=False)

linear_regression_model.fit(features_train, target_train)

petal_width_prediction_1_var = (linear_regression_model
                                .predict(features_test))
```

1.6 Inference

1.6.1 Why estimate f?

Note that the next few cells are executed using R.

There are two mean reasons we might want to estimate f with $\hat{f}$:

- prediction
  - given some set of known inputs and known outputs, we may wish to create some function that can take a new set of inputs and predict what the output would be for these inputs
- inference
  - given some set of known inputs and (optionally) known outputs, we may wish to understand how the inputs (and outputs) interact with each other

Jupyter Cell 64 []

In[None]:
```
plt.figure(1, (20,5))

plt.scatter(features_test[:, 1], target_test,
            marker='o', color='blue', alpha=0.5,
            label='actual test values')
plt.scatter(features_test[:, 1], petal_width_prediction_1_var,
            marker='x', color='red', alpha=0.5,
            label='predicted test values - 1 variable')
plt.legend()
```

Explain why we use the train-test split in the context of overfitting and underfitting.

Jupyter Cell 65 []

```
In[None]: seeds.data = read.csv('data/seeds_data.csv', row.names=1)
```

Sanity Check

Jupyter Cell 66 []

```
In[None]: head(seeds.data)
```

Jupyter Cell 67 []

```
In[None]: summary(seeds.data)
```

Jupyter Cell 68 []

```
In[None]: library(repr)
```

Jupyter Cell 69 []

```
In[None]: options(repr.plot.width=20, repr.plot.height=10)
```

Jupyter Cell 70 []

```
In[None]: pairs(seeds.data)
```

Jupyter Cell 71 []

```
In[None]: library(ggplot2)
```

Jupyter Cell 72 []

In[None]:
```
options(repr.plot.width=20, repr.plot.height=5)

ggplot(seeds.data, aes(length.of.kernel, length.of.kernel.groove)) +
  geom_point() +
  geom_smooth(method='lm')
```

1.6.2 Build a Simple Regression Model

Armed with this information we might say that we are able to predict petal width if we know petal length. We might build a **simple regression model** to do this for us using scikit-learn. Here, the **input variable** would be `petal length` and the **output variable** would be `petal width`.

We will usually refer to our input variable(s) as **feature(s)** and our output variable as the **target**.

Jupyter Cell 73 []

In[None]:
```
lm_1_var = lm('length.of.kernel.groove ~ length.of.kernel', seeds.data)
lm_1_var
```

Jupyter Cell 74 []

In[None]:
```
lm('length.of.kernel.groove ~ .', seeds.data)
```

What can be inferred from the coefficients?

- 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?

Chapter 2

The Iris Data Set

2.1 The Most Famous Dataset

> This is perhaps the best known database to be found in the pattern recognition literature. Fisher's paper is a classic in the field and is referenced frequently to this day. — UCI Machine Learning Repository

Jupyter Cell 75 []

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

%matplotlib inline
from sklearn.datasets import load_iris
from sklearn.linear_model import LinearRegression
```

> The Iris flower data set or Fisher's Iris data set is a multivariate data set introduced by the British statistician and biologist Ronald Fisher in his 1936 paper The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis. — Wikipedia

Jupyter Cell 76 []

In[2]:
```
IRIS = load_iris()
```

Jupyter Cell 77 []

In[3]:
```
type(IRIS.data)
```

Out[3]: numpy.ndarray

Jupyter Cell 78 []

```
In[4]: IRIS.data.shape

Out[4]: (150, 4)
```

What does `.shape` do?

2.1.1 Dataframes

We will load the Iris data into a dataframe for ease of manipulation.

Jupyter Cell 79 []

```
In[5]: iris_df = pd.DataFrame(IRIS.data, columns=IRIS.feature_names)
```

Jupyter Cell 80 []

```
In[6]: iris_df.head()
```

Out[6]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

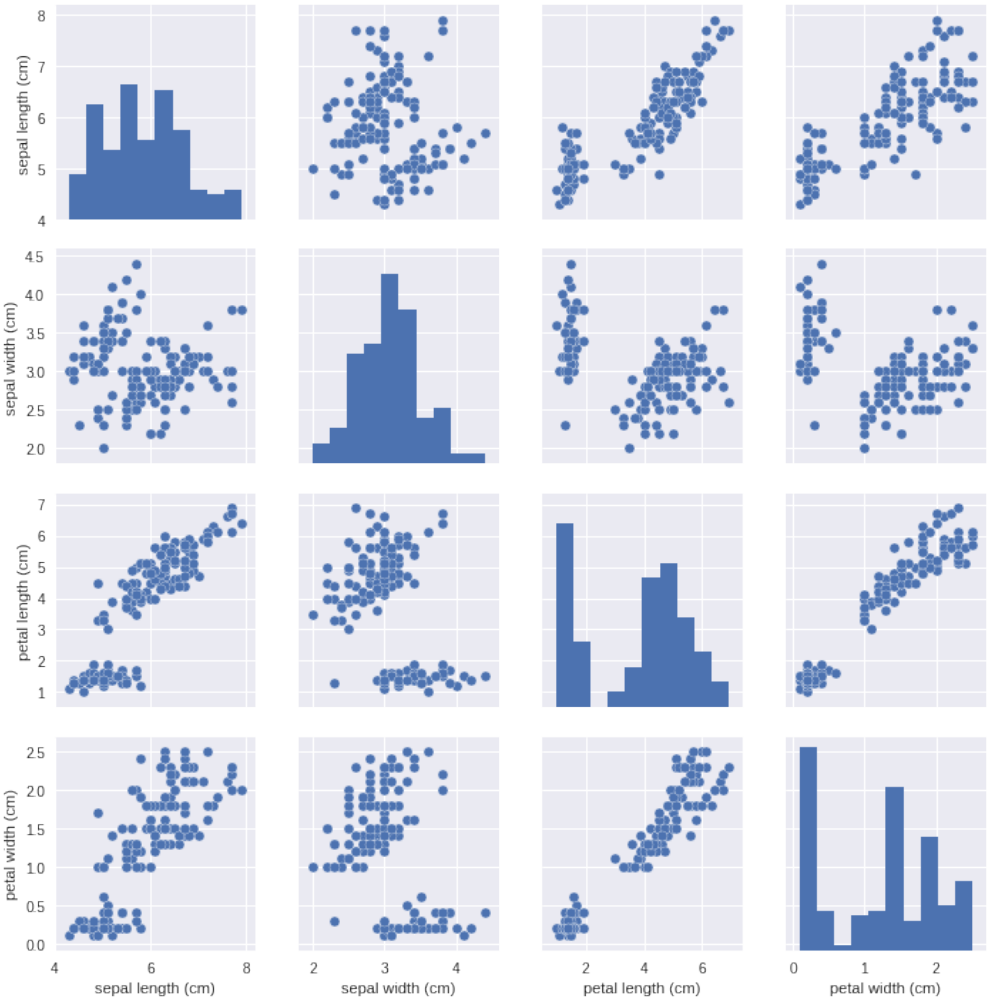
2.1.2 Pair Plot

We will use Searborn to prepare a **Pair Plot** of the Iris dataset. A Pair Plot is an array of scatter plots, one for each pair of features in the data. Rather than plotting a feature against itself, the diagonal is rendered as a **probability distribution** of the given feature.

Jupyter Cell 81 []

```
In[7]: sns.pairplot(iris_df)
```

```
Out[7]: <seaborn.axisgrid.PairGrid at 0x7f7d0e866cf8>
```



2.1.3 Remove Unit and White Space from Feature Name

Here we use a list comprehension to change the feature names:

Jupyter Cell 82 []

```
In[8]: IRIS.feature_names
```

```
Out[8]: ['sepal length (cm)',
         'sepal width (cm)',
         'petal length (cm)',
         'petal width (cm)']
```

Jupyter Cell 83 []

In[9]:
```
iris_features_names = IRIS.feature_names
iris_features_names
```

Out[9]:
```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

Jupyter Cell 84 []

In[10]:
```
def remove_unit_and_white_space(feature_name):
    feature_name = feature_name.replace(' (cm)','')
    feature_name = feature_name.replace(' ', '_')
    return feature_name
```

Jupyter Cell 85 []

In[11]:
```
iris_features_names = [remove_unit_and_white_space(name) for name in iris_features_names]
```

Jupyter Cell 86 []

In[12]:
```
iris_features_names
```

Out[12]:
```
['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
```

Jupyter Cell 87 []

In[13]:
```
iris_df.columns = iris_features_names
iris_df.head()
```

Out[13]:

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

2.1.4 Export to CSV

Ultimately, we will export a CSV of the dataframe to disk. This will make it easy to access the same data from both Python and R.

Jupyter Cell 88 []

In[14]:
```
%ls
```

```
01-iris-02-bayes-rule/  03-titanic.ipynb  doc/         lib/
01-seeds.ipynb          combined.pdf      Dockerfile   Makefile
02-iris.ipynb           data/             ipynb/       src/
```

Jupyter Cell 89 []

In[15]:
```
%mkdir -p data
```

Jupyter Cell 90 []

In[16]:
```
%ls
```

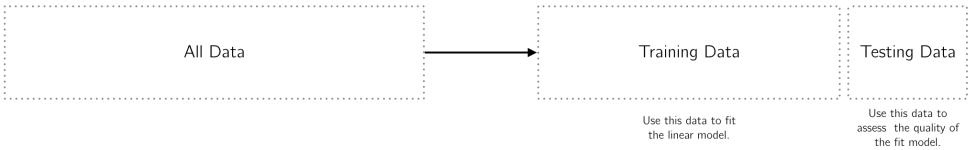
```
01-iris-02-bayes-rule/  03-titanic.ipynb  doc/         lib/
01-seeds.ipynb          combined.pdf      Dockerfile   Makefile
02-iris.ipynb           data/             ipynb/       src/
```

Jupyter Cell 91 []

In[17]:
```
iris_df.to_csv('data/iris.csv')
```

2.1.5 The Train-Test Split

Of course, we will not have access to the new data we will use at the time of fitting the model. We will have to simulate new data in some way. We do this, by creating **test** data using some fraction of the original data we started with.



Jupyter Cell 92 []

In[18]:
```
from sklearn.model_selection import train_test_split
```

Jupyter Cell 93 []

In[19]:
```
target, features = dmatrices("petal_width ~ petal_length", iris_df)
```

Of course, we will not have access to the new data we will use at the time of fitting the model. We will have to simulate new data in some way. We do this, by creating **test** data using some fraction of the original data we started with.

Jupyter Cell 94 []

In[20]:
```
(features_train,
 features_test,
 target_train,
 target_test) = train_test_split(features, target, random_state=42)
```

Jupyter Cell 95 []

In[21]:
```
(features_train.shape,
 target_train.shape,
 features_test.shape,
 target_test.shape)
```

Out[21]: ((112, 2), (112, 1), (38, 2), (38, 1))

Jupyter Cell 96 []

In[22]:
```
features_test[:5]
```

Out[22]:
```
array([[ 1. ,  4.7],
       [ 1. ,  1.7],
       [ 1. ,  6.9],
       [ 1. ,  4.5],
       [ 1. ,  4.8]])
```

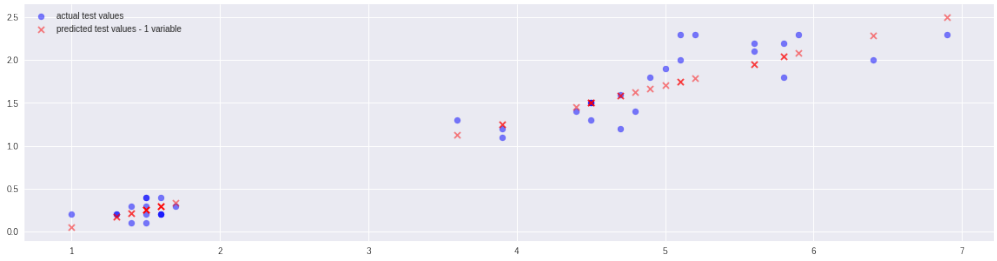
Jupyter Cell 97 []

In[23]:
```
linear_regression_model = LinearRegression(fit_intercept=False)

linear_regression_model.fit(features_train, target_train)

petal_width_prediction_1_var = (linear_regression_model
                                .predict(features_test))
```

Jupyter Cell 98 []

In[24]:
```
plt.figure(1, (20,5))

plt.scatter(features_test[:, 1], target_test,
            marker='o', color='blue', alpha=0.5,
            label='actual test values')
plt.scatter(features_test[:, 1], petal_width_prediction_1_var,
            marker='x', color='red', alpha=0.5,
            label='predicted test values - 1 variable')
plt.legend()
```

Out[24]: <matplotlib.legend.Legend at 0x7f7d08d88ef0>



2.1.6 Multicollinearity

How is the prediction affected by adding additional predictor variables?

Jupyter Cell 99 []

In[25]:
```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

%matplotlib inline

from patsy import dmatrices
from sklearn.linear_model import LinearRegression
```

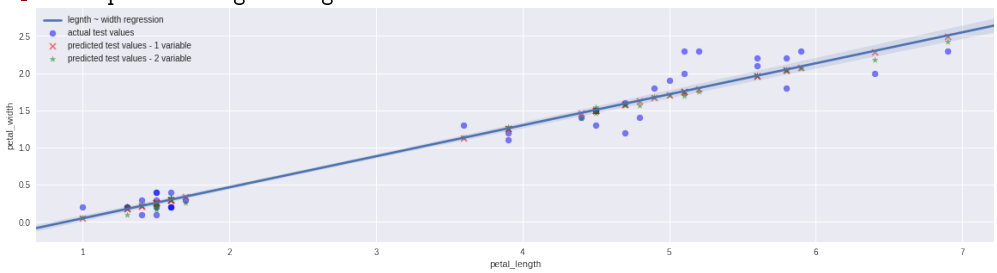
Jupyter Cell 100 []

In[26]:
```
from sklearn.model_selection import train_test_split
```

Jupyter Cell 101 []

In[27]:
```
target, features = dmatrices("petal_width ~ petal_length + sepal_length", iris_df)

(features_train,
 features_test,
 target_train,
 target_test) = train_test_split(features, target, random_state=42)

linear_regression_model = LinearRegression(fit_intercept=False)

linear_regression_model.fit(features_train, target_train)

petal_width_prediction_2_var = linear_regression_model.predict(features_test)

plt.figure(1, (20,5))


plt.scatter(features_test[:, 1], target_test,
            marker='o', color='blue', alpha=0.5, label='actual test values')
plt.scatter(features_test[:, 1], petal_width_prediction_1_var,
            marker='x', color='red', alpha=0.5, label='predicted test values - 1 variable')
plt.scatter(features_test[:, 1], petal_width_prediction_2_var,
            marker='*', color='green', alpha=0.5, label='predicted test values - 2 variable')
sns.regplot('petal_length', 'petal_width', data=iris_df, scatter=False, label='legnth ~ width regression
plt.legend()
```

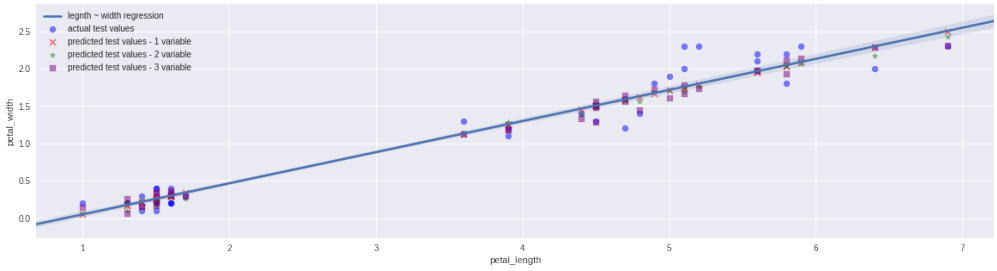
Out[27]: <matplotlib.legend.Legend at 0x7f7d0872ada0>



Jupyter Cell 102 []

In[28]:
```
target, features = dmatrices("petal_width ~ petal_length + sepal_length + sepal_width", iris_df)

(features_train,
 features_test,
 target_train,
 target_test) = train_test_split(features, target, random_state=42)

linear_regression_model = LinearRegression(fit_intercept=False)

linear_regression_model.fit(features_train, target_train)
petal_width_prediction_3_var = linear_regression_model.predict(features_test)

plt.figure(1, (20,5))

plt.scatter(features_test[:, 1], target_test,
            marker='o', color='blue', alpha=0.5, label='actual test values')
plt.scatter(features_test[:, 1], petal_width_prediction_1_var,
            marker='x', color='red', alpha=0.5, label='predicted test values - 1 variable')
plt.scatter(features_test[:, 1], petal_width_prediction_2_var,
            marker='*', color='green', alpha=0.5, label='predicted test values - 2 variable')
plt.scatter(features_test[:, 1], petal_width_prediction_3_var,
            marker='s', color='purple', alpha=0.5, label='predicted test values - 3 variable')
sns.regplot('petal_length', 'petal_width', data=iris_df, scatter=False, label='legnth ~ width regression

plt.legend()
```

Out[28]: <matplotlib.legend.Legend at 0x7f7d086b2a90>

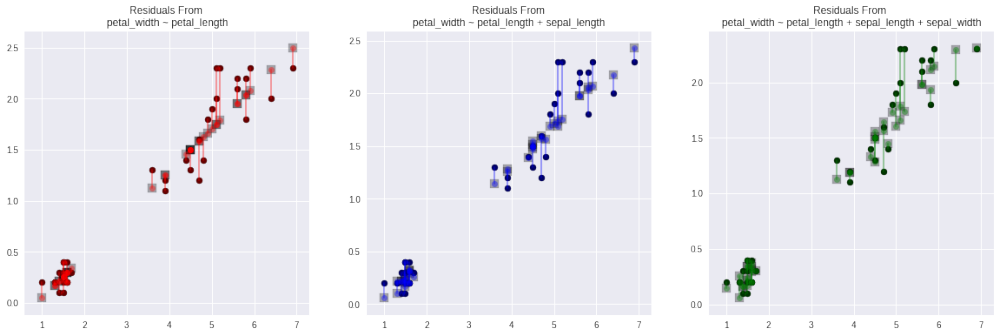


Jupyter Cell 103 []

In[29]:
```
x_values = features_test[:, 1]
y_values = target_test
y_hat_1_values = petal_width_prediction_1_var
y_hat_2_values = petal_width_prediction_2_var
y_hat_3_values = petal_width_prediction_3_var

y_hat = (y_hat_1_values, y_hat_2_values, y_hat_3_values)

points = list(zip(x_values, y_values, y_hat_1_values, y_hat_2_values, y_hat_3_values))
```

Jupyter Cell 104 []

In[30]:
```
_, ax = plt.subplots(1,3,figsize=(20,6))

for point in points:
    x, y, y_hat_1, y_hat_2, y_hat_3 = point
    ax[0].plot([x,x], [y,y_hat_1], 'ro-', alpha=0.4)
    ax[0].set_title('Residuals From \npetal_width ~ petal_length')
    ax[1].plot([x,x], [y,y_hat_2], 'bo-', alpha=0.4)
    ax[1].set_title('Residuals From \npetal_width ~ petal_length + sepal_length')
    ax[2].plot([x,x], [y,y_hat_3], 'go-', alpha=0.4)
    ax[2].set_title('Residuals From \npetal_width ~ petal_length + sepal_length + sepal_width')

for i, a in enumerate(ax):
    a.scatter(features_test[:, 1],
              target_test, marker='o', color='black')
    a.scatter(features_test[:, 1],
              y_hat[i], marker='s', s=100, alpha=0.3, color='black')
```



2.1.7 Using PCA to Plot Multidimensional Data in Two Dimensions

PCA is a complex and very powerful model typically used for dimensionality reduction. We will explore this model in greater detail later, but for now there is one application that is so useful that we will skip the details and just use it.

Jupyter Cell 105 []

In[31]:
```
from sklearn.datasets import load_iris
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))
```

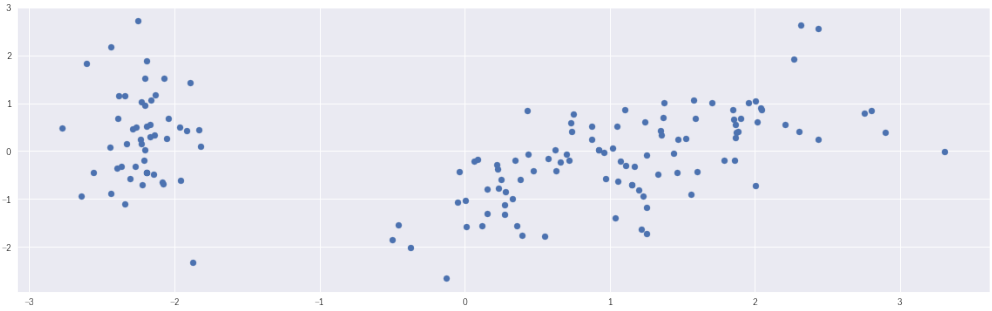
High-Dimensional Data

The Iris data we are looking at is an example of high-dimensional data. Actually, it is the smallest number of dimensions that we can really think of as "high-dimensional". You can easily imagine how to visualize data in one, two, or three dimensions, but as soon there is a fourth dimension, this becomes much more challenging.

Jupyter Cell 106 []

In[32]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

number_of_dimensions = 2
pca = PCA(number_of_dimensions)

features_scaled = StandardScaler().fit_transform(IRIS.data)
iris_2d = pca.fit_transform(features_scaled)
plt.scatter(iris_2d[:, 0], iris_2d[:, 1])
```

Out[32]: <matplotlib.collections.PathCollection at 0x7f7d03f6fb00>



Here, we have used PCA to reduce the dimensionality of our dataset from 4 to 2. Obviously, we have lost information, but this is okay. The purpose of running this algorithm is not to generate predictions, but to help us to visualize the data. At this, it was successful!

Coloring by Target

Jupyter Cell 107 []

In[33]:
```
labels = IRIS.target_names
labels
```

Out[33]:
```
array(['setosa', 'versicolor', 'virginica'],
      dtype='<U10')
```

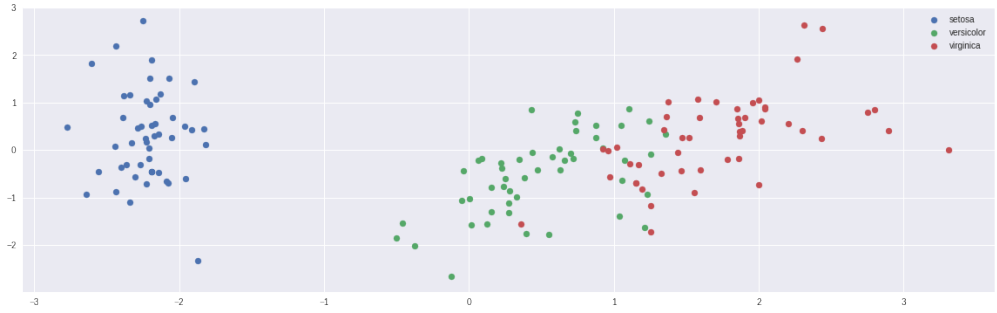
Jupyter Cell 108 []

In[34]:
```
named_target = np.array([labels[n] for n in IRIS.target])
```

Jupyter Cell 109 []

In[35]:
```
for label in labels:
    group_mask = named_target == label
    group = iris_2d[group_mask]
    plt.scatter(group[:, 0], group[:, 1], label=label)
plt.legend()
```

Out[35]: <matplotlib.legend.Legend at 0x7f7d03d13780>



2.2 Supervised and Unsupervised Learning

We think of a given dataset upon which we are working as a representation of some actual phenomenon. As data scientists, we seek a function, $\widehat{f}$ ("eff hat"), that we can use to approximate this actual phenomenon. We may take different approaches in developing this $\widehat{f}$.

In some cases, we have a set of input data, often called **features**, inputs, or independent variables, and we believe that these features can be uses to predict a **target**, output or dependent variable. If we seek to develop a model that fits a set of features to a target, this is known as **Supervised Learning**. The supervision comes from the fact that the targets or outputs are known. If the target consists of elements coming from a finite set of discrete categories e.g. $\{red, blue, green\}$, $\{heads, tails\}$, then we say that the task is a **classification** task and our $\widehat{f}$ is a classfication model. If the target consists of elements coming from a continuous range of values e.g. *Age* or *SalePrice*, then we say that the task is a **regression** task and our $\widehat{f}$ is a regression model.

NOTE: The reasoning behind the name "regression" is historical and is not consistent with the colloquial meaning of the word.

In other cases, we might seek to develop a model from a set of features without any corresponding target data. This type of model development is known as **Unsupervised Learning**. It is unsupervised because the targets are unknown. Common unsupervised learning tasks are **clustering**, in which we attempt to assign our data to a finite number of groups, and **dimensionality reduction**.

Jupyter Cell 110 []

In[None]:
```
from sklearn.datasets import load_iris
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))
```

Jupyter Cell 111 []

In[None]:
```
IRIS = load_iris()

feat_names = IRIS.feature_names

iris_df = pd.DataFrame(IRIS.data, columns=feat_names)

IRIS.target_names
```

2.2.1 Regression

Jupyter Cell 112 []

In[None]:
```
_, ax = plt.subplots(1,3, figsize=(20,6))

for i in range(3):
    sns.regplot(feat_names[i], feat_names[3],
                data=iris_df, ax=ax[i])
```

Jupyter Cell 113 []

In[None]:
```
from sklearn.linear_model import LinearRegression
linear_models = [LinearRegression(),
                 LinearRegression(),
                 LinearRegression()]
for feature, model in zip(feat_names[:3], linear_models):
    print("Fitting %s on petal width (cm) with linear regression." % feature)
    features = iris_df[[feature]]
    target = iris_df['petal width (cm)']
    model.fit(features, target)
```

2.2.2 Evaluate the Regression Models Using Mean Squared Error

Jupyter Cell 114 []

In[None]:
```
def MSE(actual, predicted):
    return sum((actual - predicted)**2)/len(actual)
```

Jupyter Cell 115 []

In[None]:
```
for feature, model in zip(feat_names[:3], linear_models):
    features = iris_df[[feature]]
    target = iris_df['petal width (cm)']
    print("Scoring linear regression model fit with %s." % feature)
    print("MSE: %f" % MSE(target, model.predict(features)))
```

Which feature appears to be least useful in helping to classify the flowers?

What was the accuracy of the model on the training data? What does this tell you about using the train-test split?

2.2.3 Classification

Jupyter Cell 116 []

In[None]:
```
from sklearn.datasets import load_iris
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))
```

Jupyter Cell 117 []

In[None]:
```
IRIS = load_iris()

feat_names = IRIS.feature_names

iris_df = pd.DataFrame(IRIS.data, columns=feat_names)

IRIS.target_names
```

Jupyter Cell 118 []

In[None]:
```
iris_df['target'] = IRIS.target_names[IRIS.target]
```

Jupyter Cell 119 []

In[None]:
```
_, ax = plt.subplots(1,4, figsize=(20,6))

for i in range(4):
    for iris_class in iris_df.target.unique():
        plotting_df = iris_df[iris_df.target == iris_class ]
        sns.distplot(plotting_df[feat_names[i]], ax=ax[i], label=iris_class)
        ax[i].legend()
```

Jupyter Cell 120 []

In[None]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

tree = DecisionTreeClassifier()
features = iris_df.drop('target', axis=1)
target_numerical = LabelEncoder().fit_transform(iris_df['target'])
```

Jupyter Cell 121 []

In[None]:
```
(features_train,
 features_validation,
 target_train,
 target_validation) = train_test_split(features, target_numerical)
```

2.2.4 Display the Classification Predictions and Actual

Jupyter Cell 122 []

In[None]:
```
tree.fit(features_train, target_train)
target_prediction = tree.predict(features_validation)
target_prediction
```

Jupyter Cell 123 []

In[None]:
```
target_validation
```

Jupyter Cell 124 []

In[None]:
```
difference = np.abs(target_validation - target_prediction)
difference
```

Jupyter Cell 125 []

In[None]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

number_of_dimensions = 2
pca = PCA(number_of_dimensions)

features_scaled = StandardScaler().fit_transform(features_validation)
iris_2d = pca.fit_transform(features_scaled)
```

Jupyter Cell 126 []

In[None]:
```
plt.figure(figsize=(20,5))
plt.scatter(x=iris_2d[:,0], y=iris_2d[:,1], c=difference)
```

2.2.5 Measure the Accuracy

Jupyter Cell 127 []

In[None]:
```
def accuracy(actual, predicted):
    return sum(np.abs(actual - predicted))/len(actual)
```

Jupyter Cell 128 []

In[None]:
```
accuracy(target_validation, target_prediction)
```

How well did the clustering algorithm do? What is the best number of clusters?

What would be some challenges in assessing cluster performance?

2.2.6 Clustering

Jupyter Cell 129 []

In[None]:
```
from sklearn.datasets import load_iris
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))
```

Jupyter Cell 130 []

In[None]:
```
IRIS = load_iris()

feat_names = IRIS.feature_names

iris_df = pd.DataFrame(IRIS.data, columns=feat_names)

IRIS.target_names
```

Jupyter Cell 131 []

In[None]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

number_of_dimensions = 2
pca = PCA(number_of_dimensions)

features_scaled = StandardScaler().fit_transform(iris_df)
iris_2d = pca.fit_transform(features_scaled)
```

Jupyter Cell 132 []

In[None]:
```
from sklearn.cluster import KMeans

number_of_clusters = [2,3,4,5]

_, ax = plt.subplots(1,4, figsize=(20,6))

for i, clusters in enumerate(number_of_clusters):
    kmeans = KMeans(n_clusters=clusters)
    kmeans.fit(iris_df)
    labels = ['cluster ' + str(label+1) for label in kmeans.labels_]
    sns.swarmplot(x=iris_2d[:,0], y=iris_2d[:,1], hue=labels, ax=ax[i])
    ax[i].set_xticklabels([])
    ax[i].set_yticklabels([])
    ax[i].legend(loc='lower right')
```

Jupyter Cell 133 []

In[None]:
```
for label in IRIS.target_names:
    group_mask = np.array([IRIS.target_names[n] for n in IRIS.target]) == label
    group = iris_2d[group_mask]
    plt.scatter(group[:, 0], group[:, 1], label=label)
plt.legend()
```

Jupyter Cell 134 []

In[None]:
```
iris_df['label'] = IRIS.target
iris_df.to_csv('data/iris.csv')
```

What percentage of the entire dataset is the first sample of three points? How well does this sample do in representing the entire dataset?

What does this code do?

Be as detailed as possible!

```
np.abs(iris_df.mean() - sample.mean())
```

Why is it important to use the absolute value of the error?

Why is it important to normalize the error by dividing by the standard deviation of the feature?

Why is it important to normalize the error by dividing by the standard deviation of the feature?

Prepare an Error Plot of all Four Features as a Function of `n`

What does the error we have been calculating represent?

Jupyter Cell 135 []

In[None]:
```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from IPython.display import display

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

Jupyter Cell 136 []

In[None]:
```
IRIS = load_iris()
```

Jupyter Cell 137 []

In[None]:
```
feat_names = IRIS.feature_names
iris_df = pd.DataFrame(IRIS.data, columns=feat_names)

iris_df['target'] = IRIS.target_names[IRIS.target]
```

Jupyter Cell 138 []

In[None]:
```
iris_df.columns
```

2.3 Sampling the Dataset

In this notebook, we begin to explore the iris dataset by sampling. First, let's sample three random points and examine them.

2.3.1 Visualizing the Distributions of the Features

2.3.2 `pd.melt()`

We can use `pandas.melt` to help with this visualization. Melt converts wide form data to long form data. So that

```
+---+---+---+
| A | B | C |
+===+===+===+
| 1 | 2 | 3 |
| 3 | 4 | 5 |
+---+---+---+
```

becomes

```
+-----+-----+
| var | val |
+=====+=====+
|  A  |  1  |
|  A  |  3  |
|  B  |  2  |
|  B  |  4  |
|  C  |  3  |
|  C  |  5  |
+-----+-----+
```

Here, is a sample of the data

Jupyter Cell 139 []

```
In[None]: samp = iris_df.sample(5)
          samp
```

And it becomes

Jupyter Cell 140 []

```
In[None]: samp_melt = pd.melt(samp.select_dtypes([float]))
          samp_melt
```

This is the exact format expected of the box plot in Seaborn.

Jupyter Cell 141 []

```
In[None]: iris_melt = pd.melt(iris_df.select_dtypes([float]))
```

Jupyter Cell 142 []

In[None]:
```
fig = plt.figure(figsize=(20,4))
sns.boxplot(x='variable', y='value', data=iris_melt)
plt.ylim(-1,9)

_, ax = plt.subplots(1,4, figsize=(20,4))
iris_numerical_df = iris_df.select_dtypes([float])

for i, feat in enumerate(iris_numerical_df.columns):
    sns.distplot(iris_numerical_df[feat], ax=ax[i])
    ax[i].set_xlim(-1,9)
    ax[i].axvline(iris_numerical_df[feat].mean())
```

Jupyter Cell 143 []

In[None]:
```
np.random.seed(42)
```

Jupyter Cell 144 []

In[None]:
```
sample_1 = iris_df.sample(3)
```

Jupyter Cell 145 []

In[None]:
```
sample_1
```

Jupyter Cell 146 []

In[None]:
```
iris_df.describe().T
```

Jupyter Cell 147 []

In[None]:
```
sample_1.describe().T
```

2.3.3 Visualize the Differences Using Seaborn

VIsualized with a Box Plot

Jupyter Cell 148 []

In[None]:
```
sample_1_melt = pd.melt(sample_1.select_dtypes([float]))
```

Jupyter Cell 149 []

In[None]:
```
_, ax = plt.subplots(1, 2, figsize=(20,5))

sns.boxplot(x='variable', y='value', data=sample_1_melt, ax=ax[0])
ax[0].set_title('Sample')

sns.boxplot(x='variable', y='value', data=iris_melt, ax=ax[1])
ax[1].set_title('Full Data Set');
```

Visualized with a Swarmplot

Jupyter Cell 150 []

In[None]:
```
_, ax = plt.subplots(1, 2, figsize=(20,5))

sns.swarmplot(x='variable', y='value', data=sample_1_melt, ax=ax[0])
ax[0].set_title('Sample')

sns.swarmplot(x='variable', y='value', data=iris_melt, ax=ax[1])
ax[1].set_title('Full Data Set');
```

Visualized with a Stripplot

Jupyter Cell 151 []

In[None]:
```
_, ax = plt.subplots(1, 2, figsize=(20,5))

sns.stripplot(x='variable', y='value', data=sample_1_melt, ax=ax[0])
ax[0].set_title('Sample')

sns.stripplot(x='variable', y='value', data=iris_melt, ax=ax[1])
ax[1].set_title('Full Data Set');
```

Visualized with a Violinplot

Jupyter Cell 152 []

In[None]:
```
_, ax = plt.subplots(1, 2, figsize=(20,5))

sns.violinplot(x='variable', y='value', data=sample_1_melt, ax=ax[0])
ax[0].set_title('Sample')

sns.violinplot(x='variable', y='value', data=iris_melt, ax=ax[1])
ax[1].set_title('Full Data Set');
```

Measure the Performance

Jupyter Cell 153 []

In[None]:
```
error_sample_1 = np.abs(iris_df.mean() - sample_1.mean())
error_sample_1
```

Normalized

Jupyter Cell 154 []

In[None]:
```
error_sample_1_normalized = np.abs((iris_df.mean() - sample_1.mean())/iris_df.std())
error_sample_1_normalized
```

2.3.4 A Second Sample

Jupyter Cell 155 []

In[None]:
```
sample_2 = iris_df.sample(3)

sample_2_melt = pd.melt(sample_2.select_dtypes([float]))
iris_melt = pd.melt(iris_df.select_dtypes([float]))

_, ax = plt.subplots(1, 2, figsize=(20,5))

sns.boxplot(x='variable', y='value', data=sample_2_melt, ax=ax[0])
ax[0].set_title('Sample')

sns.boxplot(x='variable', y='value', data=iris_melt, ax=ax[1])
ax[1].set_title('Full Data Set');
```

Jupyter Cell 156 []

In[None]:
```
error_sample_2_normalized = np.abs((iris_df.mean() - sample_2.mean())/iris_df.std())
```

Jupyter Cell 157 []

In[None]:
```
display(error_sample_1_normalized)
display(error_sample_2_normalized)
```

2.3.5 What about a larger sample?

Jupyter Cell 158 []

In[None]:
```
sample_3 = iris_df.sample(15)

sample_3_melt = pd.melt(sample_3.select_dtypes([float]))
iris_melt = pd.melt(iris_df.select_dtypes([float]))

_, ax = plt.subplots(1, 2, figsize=(20,5))

sns.boxplot(x='variable', y='value', data=sample_3_melt, ax=ax[0])
ax[0].set_title('Sample')

sns.boxplot(x='variable', y='value', data=iris_melt, ax=ax[1])
ax[1].set_title('Full Data Set');
```

Jupyter Cell 159 []

In[None]:
```
error_sample_3_normalized = np.abs((iris_df.mean() - sample_3.mean())/iris_df.std())
```

Jupyter Cell 160 []

In[None]:
```
display(error_sample_1_normalized)
display(error_sample_2_normalized)
display(error_sample_3_normalized)
```

2.3.6 Plot the Error as a Function of Sample Size

It might be useful to begin to think about the error in the mean as a function of sample size.

Jupyter Cell 161 []

In[None]:
```
def feature_error_by_n(data, feature, n):
    sample = data[feature].sample(n)
    error = np.abs((data[feature].mean() - sample.mean())/data[feature].std())
    return error
```

Jupyter Cell 162 []

In[None]:
```
iris_df.columns
```

Jupyter Cell 163 []

In[None]:
```
feature_error_by_n(iris_df, 'sepal length (cm)', 3)
```

Use a list comprehension to generate errors for every possible value of `n`.

Jupyter Cell 164 []

```
In[None]: sepal_length_error_by_n = [feature_error_by_n(iris_df, 'sepal length (cm)', n) for n in range(1,151)]
```

Jupyter Cell 165 []

```
In[None]: plt.plot(range(1,151), sepal_length_error_by_n, label='absolute error - sepal length mean by sample n')
          plt.legend()
```

2.4 Homework

Explain the bias-variance tradeoff.

Discuss the pros and cons of using the BIC to select a model.

2.5 Probability

2.5.1 What is 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.

2.5.2 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

2.5.3 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.

2.5.4 Probability of Two Events Occurring

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

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

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".

Conditional Probability

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

2.5.5 Bayes Rule

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

2.5.6 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*.

what is the 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 $p(\text{cancer}|\text{positive test})$?

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,

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

Why would the number be so small if our test is 99% accurate?

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.

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.

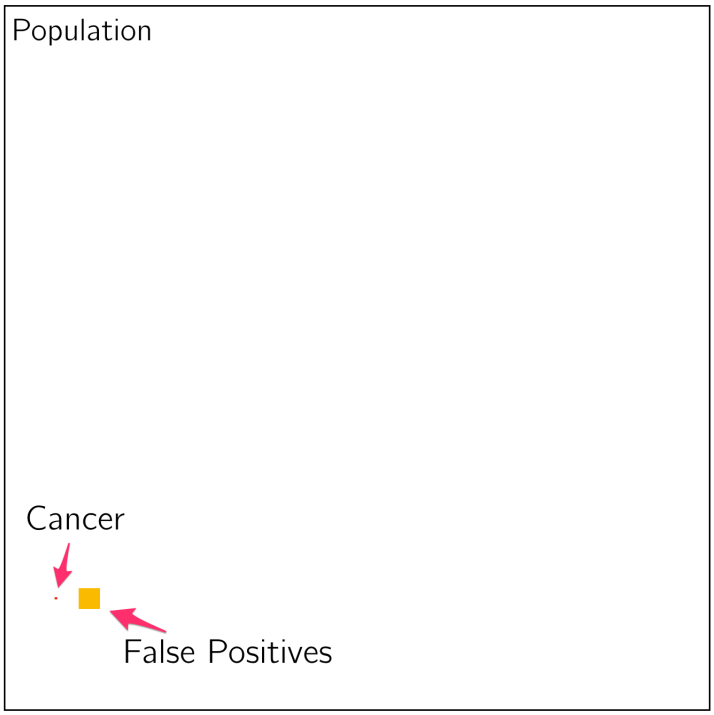


Figure 2.1: A population of 1000 patients

2.6 Probabilistic Model Selection

Jupyter Cell 166 []

```
In[None]: iris.data = read.csv("data/iris.csv", row.names='X')
          colnames(iris.data) = c('sepal_length','sepal_width','petal_length','petal_width','label')
```

Jupyter Cell 167 []

```
In[None]: head(iris.data)
```

Jupyter Cell 168 []

```
In[None]: iris.glm = glm("label ~ 1 + sepal_length + sepal_width + petal_length + petal_width", data = iris.data)
          summary(iris.glm)
```

2.6.1 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.

Jupyter Cell 169 []

```
In[None]: logLik(iris.glm)
```

"All models are wrong, but some are useful." - George Box
We might be concerned with one additional property - the **complexity** of the model.

William of Occam 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.

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

2.7 Bayesian Information Criterion

https://en.wikipedia.org/wiki/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

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

Jupyter Cell 170 []

```
In[None]: BIC(iris.glm)
```

Jupyter Cell 171 []

```
In[None]: 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
```

Jupyter Cell 172 []

In[None]:
```
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)
}
```

Jupyter Cell 173 []

In[None]:
```
BIC_of_model(iris.glm)
```

2.8 Model Selection

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

Jupyter Cell 174 []

In[None]:
```
model_1  = "label ~ 1 + sepal_length + sepal_width + petal_length + petal_width"
model_2a = "label ~ 1 + sepal_length + sepal_width + petal_length"
model_2b = "label ~ 1 + sepal_length + sepal_width                + petal_width"
model_2c = "label ~ 1 + sepal_length               + petal_length + petal_width"
model_2d = "label ~ 1                + sepal_width + petal_length + petal_width"
```

Jupyter Cell 175 []

In[None]:
```
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)
```

Jupyter Cell 176 []

In[None]:
```
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 )))
```

Jupyter Cell 177 []

In[None]:
```
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 )))
```

Jupyter Cell 178 []

In[None]:
```
model_1  = "label ~ 1 + sepal_length + sepal_width + petal_length + petal_width"
model_2c = "label ~ 1 + sepal_length               + petal_length + petal_width"
model_3a = "label ~ 1 + sepal_length               + petal_length "
model_3b = "label ~ 1 + sepal_length                                + petal_width"
model_3c = "label ~ 1                              + petal_length + petal_width"
```

Jupyter Cell 179 []

In[None]:
```
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)
```

Jupyter Cell 180 []

In[None]:
```
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 )))
```

We will be using a library `tqdm` to track the progress of our model fitting.

Jupyter Cell 181 []

In[None]:
```
!pip install tqdm
```

Jupyter Cell 182 []

In[None]:
```
from tqdm import tqdm
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from time import time
from lib.bic import BIC
```

Jupyter Cell 183 []

In[None]:
```
iris_df = pd.read_csv('data/iris.csv')
iris_df.drop('Unnamed: 0', axis=1, inplace=True)
```

These helper functions will help us to fit the model and track the time required for the fit.

Jupyter Cell 184 []

In[None]:
```
def fit_and_time(model, data):
    start = time()
    model = model.fit(data)
    end = time() - start
    return {'fit_time' : end, 'model' : model}

def process_results(results_list, data):
    df = pd.DataFrame(results_list)
    df['k'] = df.model.apply(lambda x: x.n_clusters)
    df['bic'] = df.model.apply(lambda x: BIC(x, data))
    df['sil_sc'] = df.model.apply(lambda x: silhouette_score(data, x.labels_))
    df.set_index('k', inplace=True)
    return df
```

Jupyter Cell 185 []

In[None]:
```
ks = range(2, 50)

kmeans_models = []

X = iris_df.drop('label', axis=1)

for k in tqdm(ks):
    kmeans_models.append(fit_and_time(KMeans(n_clusters=k, init="k-means++"), X))
kmeans_models_df = process_results(kmeans_models, X)
```

Jupyter Cell 186 []

In[None]:
```
import matplotlib.pyplot as plt
```

Jupyter Cell 187 []

In[None]:
```
_, ax = plt.subplots(1, 3, figsize=(20,5))
ax[0].plot(kmeans_models_df.index, kmeans_models_df.bic, label='BIC by cluster')
ax[0].legend()
ax[1].plot(kmeans_models_df.index, kmeans_models_df.sil_sc, label='Silhouette Score by cluster')
ax[1].legend()
ax[2].plot(kmeans_models_df.index, kmeans_models_df.fit_time, label='Fit by cluster')
ax[2].legend()
```

Jupyter Cell 188 []

In[None]:
```
X_sc = (X - X.mean())/X.std()
```

Jupyter Cell 189 []

In[None]:
```
ks = range(2, 50)

kmeans_sc_models = []

X = iris_df.drop('label', axis=1)

for k in tqdm(ks):
    kmeans_sc_models.append(fit_and_time(KMeans(n_clusters=k, init="k-means++"), X_sc))
kmeans_sc_models_df = process_results(kmeans_sc_models, X_sc)
```

Jupyter Cell 190 []

In[None]:
```
_, ax = plt.subplots(1, 3, figsize=(20,5))
ax[0].plot(kmeans_sc_models_df.index, kmeans_sc_models_df.bic, label='BIC by cluster')
ax[0].legend()
ax[1].plot(kmeans_sc_models_df.index, kmeans_sc_models_df.sil_sc, label='Silhouette Score by cluster')
ax[1].legend()
ax[2].plot(kmeans_sc_models_df.index, kmeans_sc_models_df.fit_time, label='Fit by cluster')
ax[2].legend()
```

Jupyter Cell 191 []

In[None]:
```
_, ax = plt.subplots(1, 2, figsize=(20,5))
ax[0].plot(kmeans_sc_models_df.index[:15], kmeans_sc_models_df.bic[:15], label='BIC by cluster')
ax[0].legend()
ax[1].plot(kmeans_sc_models_df.index[:15], kmeans_sc_models_df.sil_sc[:15], label='Silhouette Score by c
ax[1].legend()
```

Jupyter Cell 192 []

In[None]:
```
kmeans_sc_models_df.model.values[0]
```

Jupyter Cell 193 []

In[None]:
```
kmeans_2 = kmeans_sc_models_df.model.values[0]
kmeans_3 = kmeans_sc_models_df.model.values[1]
kmeans_4 = kmeans_sc_models_df.model.values[2]
kmeans_5 = kmeans_sc_models_df.model.values[3]
```

Jupyter Cell 194 []

In[None]:

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

number_of_dimensions = 2
pca = PCA(number_of_dimensions)

_, ax = plt.subplots(1,5, figsize=(20,6))
iris_2d = pca.fit_transform(X_sc)
ax[0].scatter(iris_2d[:, 0], iris_2d[:, 1], c=iris_df.label)
ax[0].set_title('Actual')
ax[1].scatter(iris_2d[:, 0], iris_2d[:, 1], c=kmeans_2.labels_)
ax[1].set_title('2 Clusters')
ax[2].scatter(iris_2d[:, 0], iris_2d[:, 1], c=kmeans_3.labels_)
ax[2].set_title('3 Clusters')
ax[3].scatter(iris_2d[:, 0], iris_2d[:, 1], c=kmeans_4.labels_)
ax[3].set_title('4 Clusters')
ax[4].scatter(iris_2d[:, 0], iris_2d[:, 1], c=kmeans_5.labels_)
ax[4].set_title('5 Clusters');
```

Chapter 3

The Titanic Data Set

Next, we will use one of the most famous data sets to begin to learn about exploratory data analysis and visualization. First, we define the task in terms of a well-defined problem statement.

Domain This is an introductory data set considered the "hello world" of data science. It is an ongoing competition on Kaggle allowing students of data science to prepare a model and make a submission to a competition while they are still learning the subject.

Problem This is a binary classification problem in which the challenge is to predict whether a passenger survived the sinking of the Titanic given the demographic data of the passengers. Here, the task T is a binary classification and the experience E is the list of passengers and their survival outcome. The machine learning challenge is to learn to perform this task using this experience.

Solution To solve this problem, we will programmatically generate a vector of integers using filtering and masking. This could be thought of as a kind of proto-decision tree.

Data A preliminary analysis of the data shows the following:

- there are 891 rows and 10 useful variable columns in the dataset. One of these columns is the target `Survived`. An 11th and 12th column are a unique id for each passenger and the name of each passenger, respectively, and have no predictive power.
- there are four integer value columns:
  - `Survived`
  - `Pclass`
  - `SibSp`
  - `Parch`
- there are two numerical value columns:
  - `Age`
  - `Fare`
- there are five factor columns:
  - `Sex`
  - `Ticket`
  - `Cabin`
  - `Embarked`
- The following are the summary statistics of the data:

```
   Survived    |   Pclass     |   Sex       |     Age       |     SibSp    |    Parch      |       Fare
 Min. :0.0000 | Min. :1.000 | female:314 | Min. : 0.42 | Min. :0.000 | Min. :0.0000 | Min. :  0.00
 Mean :0.3838 | Mean :2.309 | male  :577 | Mean :29.70 | Mean :0.523 | Mean :0.3816 | Mean : 32.20
 Max. :1.0000 | Max. :3.000 |            | Max. :80.00 | Max. :8.000 | Max. :6.0000 | Max. :512.33
              |             |            | NA's   :177 |
```

Benchmark We will use a naive guess based on the most common class as a benchmark. 61.6% of passengers did not survive. We will guess for our benchmark that there were no survivors.

Metrics As this is an beginning exercise, we will use the accuracy.

3.1 Measuring Accuracy

We have written two functions here to help us to measure the accuracy of a prediction vector. The first function is called `verify_length`. It takes two vectors and compares their length to make sure that they have the same length. This function is used in the second function as a preliminary check. If a prediction vector does not have the same length as a vector of actual values then there is a deeper problem that must be dealt with.

The second function is the `accuracy` function. This function takes two vectors: 1) a vector of actual values and 2) a vector of predicted values and compares them. It assigns a value of `TRUE` to each value that the prediction gets correct. Finally, all of the `TRUE` values are counted and this is divided by the length of the vector of actual values.

define accuracy metric

Jupyter Cell 195 []

In[39]:
```
verify_length <- function (v1, v2 ){
    if (length(v1) != length(v2)) {
        stop('length of vectors do not match')
    }
}

accuracy <- function (actual, predicted) {
    verify_length(actual, predicted)
    return(sum(actual == predicted)/length(actual))
}
```

For example we might have the following vector of the actual values:

a simple vector of actual values

Jupyter Cell 196 []

In[40]:
```
actual = c(1,1,0,0,1)
```

Our model might generate the following vector of predicted values:

a simple vector of predictions

Jupyter Cell 197 []

In[41]:
```
predicted = c(1,1,1,0,0)
```

For this simple result, we can look at it and tell that the predictions get 3 right and 2 wrong for an accuracy of 0.6.

assess accuracy of predictions

Jupyter Cell 198 []

```
In[42]: accuracy(actual, predicted)
```

0.6

3.2 Preliminary Analysis

We will start with some preliminary analysis on our data set.

3.2.1 Load the dataset using R

First, we love the data set using the R function `read.csv` and assign it to the variable `titanic`. Note that the `read.table` and `read.csv` in R are equivalent accept for the default args. `read.table` defaults to separating on white space. `read.csv` defaults to separating on commas. `read.csv` also defaults to the argument `header=T`.

load the dataset using `read.csv()`

Jupyter Cell 199 []

```
In[1]: ### BEGIN SOLUTION
       titanic <- read.csv('train.csv')
       ### END SOLUTION
```

Jupyter Cell 200 []

```
In[3]: stopifnot(dim(titanic) == c(891,12))
```

We displayed the dimension `dim()` and the structure `str()` of our datafrane. This is mostly done as a sanity check. We should have some idea of what the dimension and structure of our data is. By displaying these results immediately after loading the data, we can verify that the data has been loaded as we expect.

display the dimension of the data set

Jupyter Cell 201 []

```
In[4]: dim(titanic)
```

1. 891 2. 12

display the structure of the dataframe

Jupyter Cell 202 []

```
In[5]: str(titanic)
```

```
'data.frame':	891 obs. of  12 variables:
 $ PassengerId: int  1 2 3 4 5 6 7 8 9 10 ...
 $ Survived   : int  0 1 1 1 0 0 0 0 1 1 ...
 $ Pclass     : int  3 1 3 1 3 3 1 3 3 2 ...
 $ Name       : Factor w/ 891 levels "Abbing, Mr. Anthony",..: 109 191 358 277 16 559 520 629 417 581 ..
 $ Sex        : Factor w/ 2 levels "female","male": 2 1 1 1 2 2 2 2 1 1 ...
 $ Age        : num  22 38 26 35 35 NA 54 2 27 14 ...
 $ SibSp      : int  1 1 0 1 0 0 0 3 0 1 ...
 $ Parch      : int  0 0 0 0 0 0 0 1 2 0 ...
 $ Ticket     : Factor w/ 681 levels "110152","110413",..: 524 597 670 50 473 276 86 396 345 133 ...
 $ Fare       : num  7.25 71.28 7.92 53.1 8.05 ...
 $ Cabin      : Factor w/ 148 levels "","A10","A14",..: 1 83 1 57 1 1 131 1 1 1 ...
 $ Embarked   : Factor w/ 4 levels "","C","Q","S": 4 2 4 4 4 3 4 4 4 2 ...
```

3.2.2 The R Structure Object

I interpret the structure of our data frame in the following way. Each row in the structure object, `str(titanic)` represents a column in the data frame `titanic`. The value immediately following the `$` is the name of that column. The value immediately following the `:` is the data type of that column. The values following the datatype are the first few values of the data in the column itself.

Note that R has made some default decisions about the structure of our data. It has designated five columns as integer columns, five columns as factor columns, and two columns as numerical problems. These may or may not be accurate according to our own understanding of the data. This was done by R, doing its best to intuit the structure of the data during the read of the CSV file. For example, a reasonable case could be made that the the `Survived` column should not be an integer, nor should the `Pclass`.

3.2.3 Categorical Features In R

R stores categorical features using a special type of vector called a **factor**. The data is stored as a vector of integers. The factor has an additional attribute, however. It also has a vector of levels. The integer stored as data are actually references to the vector of names. We can think of the data stored in the Factor as a mapping to the vector of levels.

display that class of the `titanic$embarked` column

Jupyter Cell 203 []

```
In[15]: class(titanic$Embarked)
```

```
'factor'
```

display that levels of the `titanic$embarked` column

Jupyter Cell 204 []

```
In[23]: levels(titanic$Embarked)
```

1. '' 2. 'C' 3. 'Q' 4. 'S'

display that first few values of the `titanic$embarked` column

Jupyter Cell 205 []

```
In[26]: titanic$Embarked[1:5]
```

1. S 2. C 3. S 4. S 5. S

3.2.4 Completely Unique Columns

We can see from the structure of our data frame that it contains two columns that are completely unique. We are attempting to use the patterns in our data to make predictions about the survival of passengers during the Titanic disaster. This is done by identifying patterns in the data. If they column is completely unique there is no pattern to be identified there. Each passenger has its own unique value and there is really no immediate way to associate these unique values with each other. For this reason we will simply remove the completely unique columns. Prior to doing this, however, we should verify that they are in fact completely.

The two columns in question are `PassengerId` and `Name`. We will use the following method to establish that they are both completely unique:

1. We will take a measure of the number of passengers in the data set
2. We will take a measure of the number of unique values in each of the columns in question
3. If the values match we will consider the column safe for removal

store the number of passengers

Jupyter Cell 206 []

```
In[24]: number_of_passengers = length(titanic$PassengerId)
        number_of_passengers
```

891

display the length of the unique values in `titanic$passengerid` and `titanic$name`

Jupyter Cell 207 []

```
In[25]: length(unique(titanic$PassengerId)); length(unique(titanic$Name))
```

891

We note that the values do indeed match, therefore, it is safe to drop both of these columns from our dataframe. This can be done by assigning the `NULL` value to the named column. For example, we might do the following on a generic data frame and column

```
dataframe$mycolumn = NULL
```

drop the columns with completely unique values

Jupyter Cell 208 []

In[27]:
```
### BEGIN SOLUTION
titanic$PassengerId <- NULL
titanic$Name <- NULL
### END SOLUTION
```

Jupyter Cell 209 []

In[28]:
```
stopifnot(is.null(titanic$PassengerID))
stopifnot(is.null(titanic$Name))

### BEGIN HIDDEN TESTS
stopifnot(as.vector(titanic[4,]) == c('1','1','female', '35', '1','0','113803','53.1','C123','S'))
### END HIDDEN TESTS
```

3.2.5 Summarize The Data

Finally, having dropped the features deemed not immediately useful, we display the summary statistics of the dataframe using the `summary()` function. This function shows the quartile values of the data as well as mean and median for numerical features and the counts to the best of its ability for the factors.

Jupyter Cell 210 []

In[29]:
```
summary(titanic)
```

```
    Survived         Pclass          Sex           Age            SibSp
 Min.   :0.0000   Min.   :1.000   female:314   Min.   : 0.42   Min.   :0.000
 1st Qu.:0.0000   1st Qu.:2.000   male  :577   1st Qu.:20.12   1st Qu.:0.000
 Median :0.0000   Median :3.000                Median :28.00   Median :0.000
 Mean   :0.3838   Mean   :2.309                Mean   :29.70   Mean   :0.523
 3rd Qu.:1.0000   3rd Qu.:3.000                3rd Qu.:38.00   3rd Qu.:1.000
 Max.   :1.0000   Max.   :3.000                Max.   :80.00   Max.   :8.000
                                               NA's   :177
     Parch            Ticket         Fare                Cabin     Embarked
 Min.   :0.0000   1601    :  7   Min.   :  0.00             :687    :  2
 1st Qu.:0.0000   347082  :  7   1st Qu.:  7.91   B96 B98    :  4   C:168
 Median :0.0000   CA. 2343:  7   Median : 14.45   C23 C25 C27:  4   Q: 77
 Mean   :0.3816   3101295 :  6   Mean   : 32.20   G6         :  4   S:644
 3rd Qu.:0.0000   347088  :  6   3rd Qu.: 31.00   C22 C26    :  3
 Max.   :6.0000   CA 2144 :  6   Max.   :512.33   D          :  3
                  (Other) :852                    (Other)    :186
```

3.3 Preparing A Benchmark Model

Having performed a preliminary analysis of the data, we move onto preparing a benchmark model. First, we will do some analysis of the target column. Based upon this analysis we will think about what the best model for a benchmark might be.

We will make use of the R `table()` function to study the target column. This function builds a contingency table of the counts combinations of factor levels. Of course if only a Single column is passed to the function, it will just return a simple count.

display a contingency table of `titanic$survived`

Jupyter Cell 211 []

```
In[31]: table(titanic$Survived)
```

```
  0   1
549 342
```

From the result returns, we can see that the survival status is stored As either is 0, corresponding to did not survive, or a 1 corresponding to survived. We can use the helper function `prop.table()` to express the results a contingency table as fractions. Here, we can see that 0.61 Of the passengers did not survive. One thing we should immediately take note of is that our target column is not evenly distributed. An **evenly distributed** target column would have the exact same number of each possible outcome. As we grow in our data science practice we will learn more about dealing with an evenly distributed target. For now it is sufficient to simply take note of this fact.

display a proportion table of `titanic$survived`

Jupyter Cell 212 []

```
In[33]: prop.table(table(titanic$Survived))
```

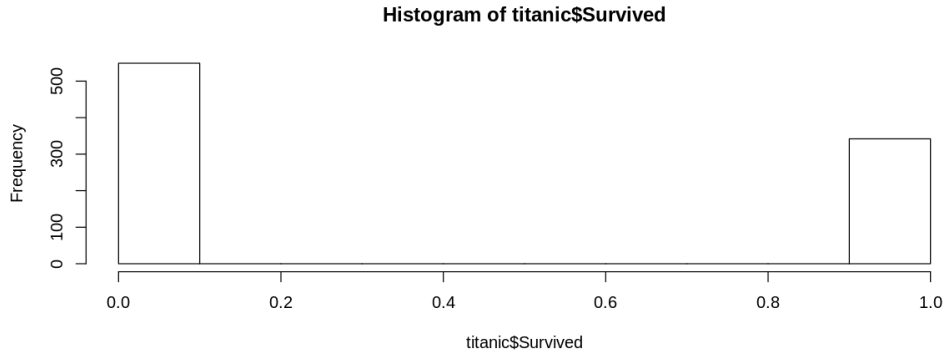
```
        0         1
0.6161616 0.3838384
```

Below, we use a histogram to show once more that the target is not evenly distributed. By default, the `hist()` function simply shows the counts for each measured value.

display a histogram of `titanic$Survived`

Jupyter Cell 213 []

In[34]:
```
library(repr)
options(repr.plot.width=10, repr.plot.height=4)

hist(titanic$Survived)
```



3.3.1 A Naïve Guess

We will use a naive guess based on the most common class as a benchmark. 61.6% of passengers did not survive. We will guess for our benchmark that there were no survivors. Note that we have done very little work and already have a better then 50-50 chance I've getting a correct answer simply by guessing that no one survived. This is one consideration for having an unevenly distributed target. Simply measuring accuracy may not give us a realistic sense of how well our model is doing. This is one reason why preparing a benchmark is so important. Had we not prepared at benchmark we might think that a 55% accuracy is deceny because it's better than the simple 50-50. This benchmark gives us a sense of what we need to do better than in order to prepare a model that adds value to the situation.

Create a vector called `no_survivors` **that is a list of predictions that no one survived.**

To create such a vector using R, we will use the replicate `rep()` function. This function takes a value and replicates it a given number of times.

Jupyter Cell 214 []

In[37]:
```
### BEGIN SOLUTION
no_survivors <- rep(0, number_of_passengers)
### END SOLUTION
```

Jupyter Cell 215 []

In[38]:
```
# HIDDEN TEST

### BEGIN HIDDEN TESTS
stopifnot(no_survivors == rep(0, length(titanic$Survived)))
### END HIDDEN TESTS
```

Once we have prepared this naïve guess, we can use the `accuracy` function we defined earlier to assess our benchmark as a vector of predictions.

accuracy of our naïve prediction

Jupyter Cell 216 []

In[43]:
```
accuracy(titanic$Survived, no_survivors)
```

0.616161616161616

As expected, we achieve an accuracy of $0.6\bar{1}$.

3.4 A Vectorized Solution To `fizzbuzz`

`fizzbuzz` is a canonical "coding interview" problem. You might want to read this humorous take by Joel Grus who attempts to use tensor for to solve the problem: http://joelgrus.com/2016/05/23/fizz-buzz-in-tensorflow/. 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

a first attempt at `fizzbuzz`

Jupyter Cell 217 []

In[86]:
```
fizzbuzz = function (n) {
    for (i in 1:n) {
        if (i %% 15 == 0) print("fizzbuzz")
        else if (i %% 3 == 0) print("fizz")
        else if (i %% 5 == 0) print("buzz")
        else print(i)
    }
}
fizzbuzz(15)
```

```
[1] 1
[1] 2
[1] "fizz"
[1] 4
[1] "buzz"
[1] "fizz"
[1] 7
[1] 8
[1] "fizz"
[1] "buzz"
[1] 11
[1] "fizz"
[1] 13
[1] 14
[1] "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

start the `solution` **vector**

Jupyter Cell 218 []

In[55]:
```
solution = 1:15
solution
```

```
1. 1 2. 2 3. 3 4. 4 5. 5 6. 6 7. 7 8. 8 9. 9 10. 10 11. 11 12. 12 13. 13 14. 14 15. 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).

create the mod 15 mask

Jupyter Cell 219 []

In[59]:
```
mod15_mask = (solution %% 15 == 0)
mod15_mask
```

1. FALSE 2. FALSE 3. FALSE 4. FALSE 5. FALSE 6. FALSE 7. FALSE 8. FALSE 9. FALSE 10. FALSE 11. FALSE 12. FALSE 13. FALSE 14. FALSE 15. TRUE

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

filter `solution` using the mind 15 mask

Jupyter Cell 220 []

In[60]:
```
solution[mod15_mask]
```

15

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

assign valued to the filtered `solution` vector

Jupyter Cell 221 []

In[61]:
```
solution[mod15_mask] = "fizzbuzz"
```

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

Jupyter Cell 222 []

In[62]:
```
solution
```

1. '1' 2. '2' 3. '3' 4. '4' 5. '5' 6. '6' 7. '7' 8. '8' 9. '9' 10. '10' 11. '11' 12. '12' 13. '13' 14. '14' 15. 'fizzbuzz'

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

a vectorized `fizzbuzz`

Jupyter Cell 223 []

In[87]:
```
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)
```

```
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. To read more about this, have a look at this blog post: http://www.noamross.net/blog/2014/4/16/vectorization-in-r–why.html

3.5 Incremental Model Improvement With Filters And Masks

And now begins the work of data scientist. We have established a benchmark model. We should now begin to refine upon this model seeking to continually improve the benchmark performance that we have. We can do this by using exploratory data analysis to study the features, especially as they relate to the target. If we find a feature that we believe exhibits some pattern of correspondence to our target we can use this to refine our model.

For this project, we are going to think of our model as simply the values stored in a vector of predictions. For example, we already have one model, a model called `no_survivors`, which is simply a vector of zeros. To improve upon this model we will use a mask to reduce the number of values we are looking at and then replace these values with a 1.

3.5.1 Randomized Model Improvement

What if we try to improve our model by simply randomly replacing zeros with one? We can do this using the `sample()` function

create a random mask

Jupyter Cell 224 []

In[76]:
```
random_mask = sample(c(TRUE,FALSE), number_of_passengers, replace = TRUE)
random_mask[1:10]
```

1. TRUE 2. FALSE 3. TRUE 4. TRUE 5. TRUE 6. TRUE 7. FALSE 8. FALSE 9. TRUE 10. FALSE

duplicate and filter to create random model

Jupyter Cell 225 []

In[80]:
```
random_model = rep(no_survivors)
random_model[random_mask] = 1
```

assess accuracy of random model

Jupyter Cell 226 []

In[82]:
```
accuracy(titanic$Survived, random_model)
```

0.482603815937149

As suspected, simply guessing is not better than guessing all zeros. It looks like we might actually justify our exorbitant salaries after all.

3.5.2 Use Proportion Tables To Look At Survival By Feature

Previously, we use a proportion table to look at a single feature, `Survived`. Next, We will use a proportion table to look at how two features interact with each other. Let's look at the structure of the dataframe again to remind ourselves which features we have available to us.

display the structure of the dataframe

Jupyter Cell 227 []

In[83]:
```
str(titanic)
```

```
'data.frame':	891 obs. of  10 variables:
 $ Survived: int  0 1 1 1 0 0 0 0 1 1 ...
 $ Pclass  : int  3 1 3 1 3 3 1 3 3 2 ...
 $ Sex     : Factor w/ 2 levels "female","male": 2 1 1 1 2 2 2 2 1 1 ...
 $ Age     : num  22 38 26 35 35 NA 54 2 27 14 ...
 $ SibSp   : int  1 1 0 1 0 0 0 3 0 1 ...
 $ Parch   : int  0 0 0 0 0 0 0 1 2 0 ...
 $ Ticket  : Factor w/ 681 levels "110152","110413",..: 524 597 670 50 473 276 86 396 345 133 ...
 $ Fare    : num  7.25 71.28 7.92 53.1 8.05 ...
 $ Cabin   : Factor w/ 148 levels "","A10","A14",..: 1 83 1 57 1 1 131 1 1 1 ...
 $ Embarked: Factor w/ 4 levels "","C","Q","S": 4 2 4 4 4 3 4 4 4 2 ...
```

First, we look at the proportions of `Pclass` and `Survived`. There are three different ways we can look at a proportion table.

1. The values of each combination as a proportion of the whole
2. The values in each row as a proportion of that row
3. The values in each column as a proportion of that column

whole proportions of `Pclass` versus `Survived`

Jupyter Cell 228 []

```
In[85]: prop.table(table(titanic$Pclass, titanic$Survived))
```

```
           0          1
1 0.08978676 0.15263749
2 0.10886644 0.09764310
3 0.41750842 0.13355780
```

proportions of `Pclass` versus `Survived` by row

Jupyter Cell 229 []

```
In[88]: prop.table(table(titanic$Pclass, titanic$Survived), 1)
```

```
          0         1
1 0.3703704 0.6296296
2 0.5271739 0.4728261
3 0.7576375 0.2423625
```

proportions of `Pclass` versus `Survived` by column

Jupyter Cell 230 []

```
In[89]: prop.table(table(titanic$Pclass, titanic$Survived), 2)
```

```
          0         1
1 0.1457195 0.3976608
2 0.1766849 0.2543860
3 0.6775956 0.3479532
```

whole proportions of `Sex` versus `Survived`

Jupyter Cell 231 []

```
In[90]: prop.table(table(titanic$Sex, titanic$Survived))
```

```
                0          1
female 0.09090909 0.26150393
male   0.52525253 0.12233446
```

proportions of `Sex` versus `Survived` by row

Jupyter Cell 232 []

```
In[91]: prop.table(table(titanic$Sex, titanic$Survived), 1)
```

```
                0         1
female 0.2579618 0.7420382
male   0.8110919 0.1889081
```

proportions of `Sex` versus `Survived` by column

Jupyter Cell 233 []

```
In[92]: prop.table(table(titanic$Sex, titanic$Survived), 2)
```

```
                0         1
female 0.1475410 0.6812865
male   0.8524590 0.3187135
```

Analyze Proportion Tables

Using the results obtained about prepare an analysis ofhHow these two features can be used to predict whether or not someone survived the sinking of the Titanic.

3.5.3 Targeted Model Improvement

We saw that randomly selecting values to be replaced by one did not improve our model. What if we use some more intelligent way to select values that should be replaced by a one in our vector of predictions? We just looked at two features and identified some patterns that showed it would be more likely to have survived the sinking of the ship. Based upon this work we might decide that it would be a better model to replace the prediction for all female passengers with a 1. We can do that using masks and filters.

create a mask of just women

Jupyter Cell 234 []

In[93]:
```
women_mask = titanic$Sex == 'female'
women_mask[1:10]
```

1. FALSE 2. TRUE 3. TRUE 4. TRUE 5. FALSE 6. FALSE 7. FALSE 8. FALSE 9. TRUE 10. TRUE

duplicate and filter to create a model, `women_survived`

Jupyter Cell 235 []

In[94]:
```
women_survived = rep(no_survivors)
women_survived[women_mask] = 1
```

assess accuracy of model, `women_survived`

Jupyter Cell 236 []

In[95]:
```
accuracy(titanic$Survived, women_survived)
```

0.78675645342312

Explaining Creation Of Prediction Vector

Explain in your own words the process by which the prediction vector, `women_survived`:

3.5.4 Can Another Feature Help?

Jupyter Cell 237 []

In[97]:
```
prop.table(table(titanic$Survived, titanic$Pclass, titanic$Sex))
```

```
, ,  = female


             1           2           3
  0 0.003367003 0.006734007 0.080808081
  1 0.102132435 0.078563412 0.080808081

, ,  = male


             1           2           3
  0 0.086419753 0.102132435 0.336700337
  1 0.050505051 0.019079686 0.052749719
```

create a mask of just first class

Jupyter Cell 238 []

In[98]:
```
first_class_mask = titanic$Pclass == 1
first_class_mask[1:10]
```

1. FALSE 2. TRUE 3. FALSE 4. TRUE 5. FALSE 6. FALSE 7. TRUE 8. FALSE 9. FALSE 10. FALSE

duplicate and filter to create a model, `women_and_first_class_survived`

Jupyter Cell 239 []

In[99]:
```
women_and_first_class_survived = rep(women_survived)
women_and_first_class_survived[first_class_mask] = 1
```

assess accuracy of model, `women_and_first_class_survived`

Jupyter Cell 240 []

In[100]:
```
accuracy(titanic$Survived, women_and_first_class_survived)
```

0.750841750841751

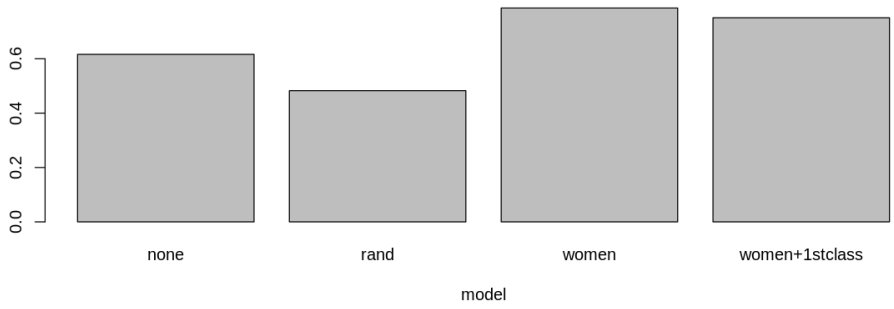
Jupyter Cell 241 []

In[101]:
```
scores = c(accuracy(titanic$Survived, no_survivors),
           accuracy(titanic$Survived, random_model),
           accuracy(titanic$Survived, women_survived),
           accuracy(titanic$Survived, women_and_first_class_survived))
```

3.5.5 Progress Report

Jupyter Cell 242 []

In[105]:
```
barplot(scores, xlab = 'model',
        names.arg = c('none','rand','women', 'women+1stclass'))
abline(h = max(scores))
```

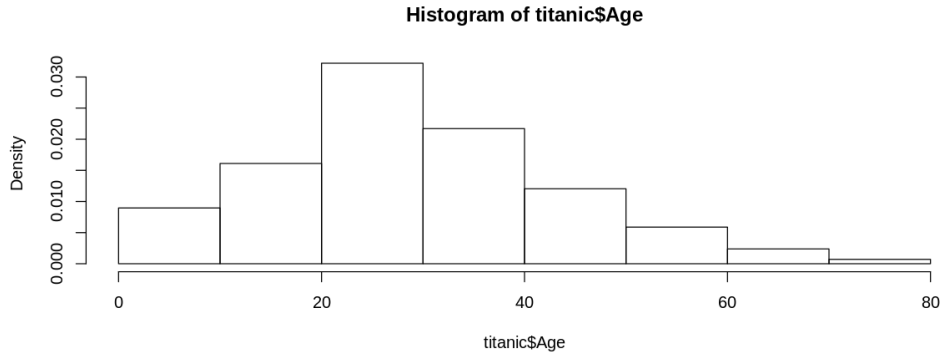


3.5.6 Age

Age is a numerical feature.
Age has missing values.

Jupyter Cell 243 []

In[110]:
```
hist(titanic$Age)
```



Jupyter Cell 244 []

In[119]:
```
missing_age_values_mask = is.na(titanic$Age)
```

Jupyter Cell 245 []

In[120]:
```
titanic$Survived[missing_age_values_mask]
```

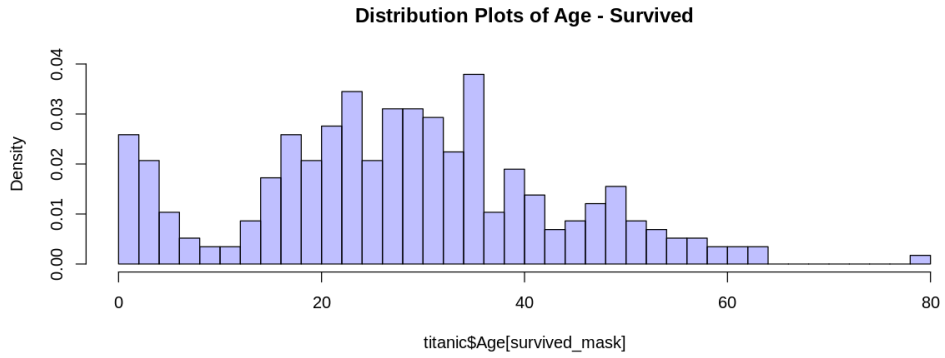
1. 0 2. 1 3. 1 4. 0 5. 1 6. 0 7. 1 8. 1 9. 1 10. 0 11. 0 12. 0 13. 1 14. 0 15. 1 16. 0 17. 1 18. 0 19. 0 20. 1 21. 0 22. 0 23. 0 24. 1 25. 1 26. 0 27. 0 28. 1 29. 0 30. 0 31. 0 32. 0 33. 1 34. 0 35. 0 36. 0 37. 0 38. 0 39. 1 40. 0 41. 1 42. 0 43. 0 44. 0 45. 0 46. 0 47. 0 48. 1 49. 0 50. 1 51. 0 52. 0 53. 0 54. 1 55. 0 56. 0 57. 0 58. 1 59. 1 60. 1 61. 1 62. 0 63. 1 64. 0 65. 1 66. 1 67. 0 68. 1 69. 0 70. 0 71. 1 72. 1 73. 0 74. 1 75. 1 76. 1 77. 0 78. 0 79. 0 80. 0 81. 0 82. 0 83. 0 84. 0 85. 0 86. 0 87. 1 88. 1 89. 0 90. 0 91. 1 92. 0 93. 0 94. 0 95. 0 96. 0 97. 0 98. 0 99. 0 100. 0 101. 0 102. 0 103. 0 104. 1 105. 0 106. 0 107. 0 108. 0 109. 0 110. 0 111. 1 112. 0 113. 1 114. 0 115. 0 116. 0 117. 0 118. 0 119. 0 120. 1 121. 0 122. 0 123. 0 124. 0 125. 1 126. 0 127. 0 128. 0 129. 0 130. 1 131. 0 132. 0 133. 0 134. 0 135. 1 136. 0 137. 0 138. 1 139. 0 140. 0 141. 1 142. 0 143. 0 144. 1 145. 1 146. 1 147. 0 148. 0 149. 1 150. 0 151. 0 152. 0 153. 1 154. 0 155. 0 156. 0 157. 0 158. 0 159. 0 160. 0 161. 0 162. 0 163. 0 164. 0 165. 0 166. 0 167. 1 168. 0 169. 0 170. 1 171. 0 172. 1 173. 0 174. 0 175. 0 176. 0 177. 0

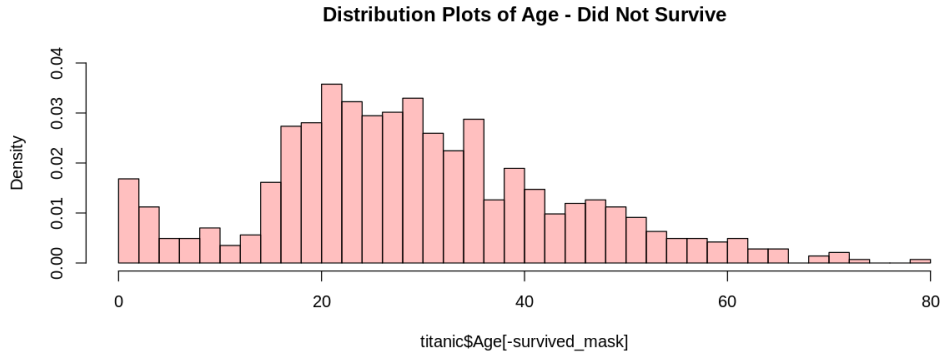
Jupyter Cell 246 []

In[149]:
```
survived_mask = as.logical(titanic$Survived)
```

Jupyter Cell 247 []

In[150]:
```
h1 = hist(titanic$Age[survived_mask], col=rgb(0,0,1,1/4),
          freq = F, breaks = 30, ylim = c(0,0.04),
          main='Distribution Plots of Age - Survived')
h2 = hist(titanic$Age[-survived_mask], col=rgb(1,0,0,1/4),
          freq = F, breaks = 30, ylim = c(0,0.04),
          main = 'Distribution Plots of Age - Did Not Survive')
```





create a mask of just children

Jupyter Cell 248 []

In[151]:
```
children_mask = titanic$Age < 10
```

duplicate and filter to create a model, `women_survived`

Jupyter Cell 249 []

In[152]:
```
women_and_children_survived = rep(women_survived)
women_and_children_survived[children_mask] = 1
```

assess accuracy of model, `women_survived`

Jupyter Cell 250 []

In[153]:
```
accuracy(titanic$Survived, women_and_children_survived)
```

0.793490460157127

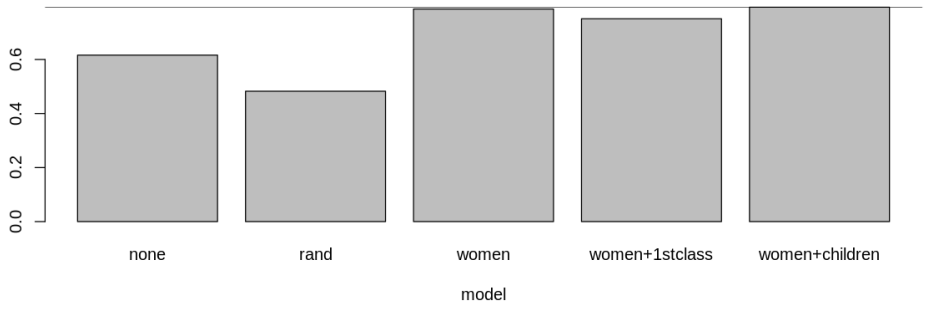
Jupyter Cell 251 []

In[154]:
```
scores = c(accuracy(titanic$Survived, no_survivors),
           accuracy(titanic$Survived, random_model),
           accuracy(titanic$Survived, women_survived),
           accuracy(titanic$Survived, women_and_first_class_survived),
           accuracy(titanic$Survived, women_and_children_survived))
```

3.5.7 Progress Report

Jupyter Cell 252 []

In[158]:
```
barplot(scores, xlab = 'model',
        names.arg = c('none','rand','women', 'women+1stclass', 'women+children'))
abline(h = max(scores))
```



Jupyter Cell 253 []

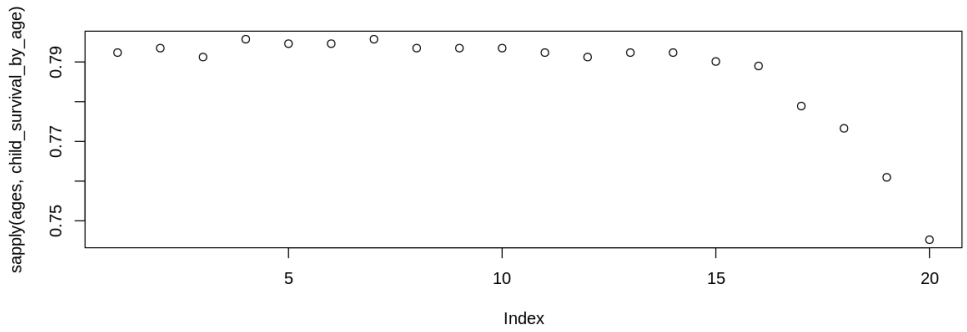
In[159]:
```
child_survival_by_age = function (age) {
    children_mask = titanic$Age < age

    women_and_children_survived = rep(women_survived)
    women_and_children_survived[children_mask] = 1

    return(accuracy(titanic$Survived, women_and_children_survived))
}
```

Jupyter Cell 254 []

In[160]:
```
ages = 1:20
```

Jupyter Cell 255 []

In[163]:
```
plot(sapply(ages, child_survival_by_age))
```



create a mask of just children

Jupyter Cell 256 []

In[164]:
```
children_mask = titanic$Age < 7
```

duplicate and filter to create a model, `women_survived`

Jupyter Cell 257 []

In[165]:
```
women_and_children_survived = rep(women_survived)
women_and_children_survived[children_mask] = 1
```

assess accuracy of model, `women_survived`

Jupyter Cell 258 []

In[166]:
```
accuracy(titanic$Survived, women_and_children_survived)
```

0.795735129068462

Jupyter Cell 259 []

In[167]:
```
scores = c(accuracy(titanic$Survived, no_survivors),
           accuracy(titanic$Survived, random_model),
           accuracy(titanic$Survived, women_survived),
           accuracy(titanic$Survived, women_and_first_class_survived),
           accuracy(titanic$Survived, women_and_children_survived))
```

3.5.8 Progress Report

Jupyter Cell 260 []

In[168]:
```
barplot(scores, xlab = 'model',
        names.arg = c('none','rand','women', 'women+1stclass', 'women+children'))
abline(h = max(scores))
```