Introduction To Data Science

Joshua Cook

Jun 06, 2018

CONTENTS

CHAPTER

ONE

THE SEEDS DATASET

1.1 The Seeds Dataset

Welcome to data science! This text is designed to teach you at the practice of data science by working on Data sets. 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. Furthermore, an emphasis will be placed upon practicing data science using the Jupyter notebook server.

The fastest way to install Jupyter is by following the instructions here: http://jupyter.readthedocs.io/en/latest/install.html. That said, I recommend taking the time to install Jupyter using Docker. Installing Jupyter using

1.2 Interactive Programming

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

1.2.1 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.

1.2.2 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__" <http://ibiblio.org/g2swap/byteofpython/read/module-name.html>‘__:.

1.2.3 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:

```
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.

```
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.

```
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.

```
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!'
```

1.2.4 How to Program Interactively

Define a variable `my_integer` that is equal to 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.

```
In [6]: my_integer
```

```
Out[6]: 5
```

Redefine `my_integer` to be equal to 23.

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

Once more display the result.

```
In [8]: my_integer
```

Out[8]: 23

1.2.5 Plotting a Strange Attractor

This is not a very interesting use of interactive programming.

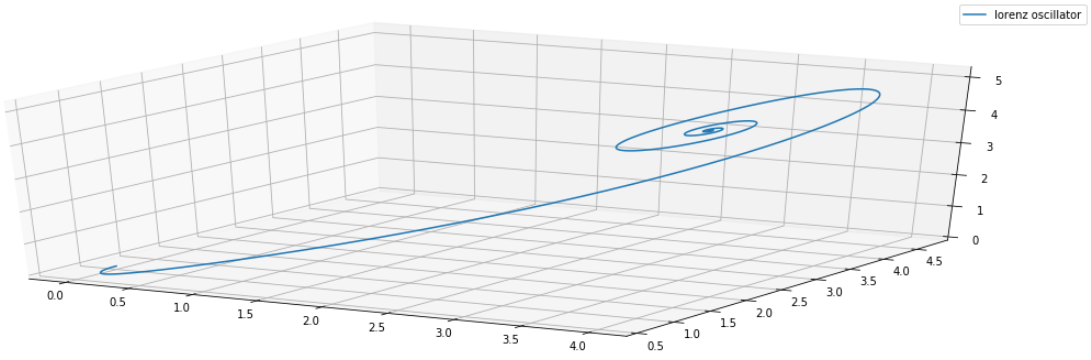
https://en.wikipedia.org/wiki/Attractor#Strange_attractor

```
In [9]: from scipy.integrate import odeint as odeint
        import matplotlib.pyplot as plt
        import numpy as np
        from mpl_toolkits.mplot3d import Axes3D
```

```
In [10]: 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)
```

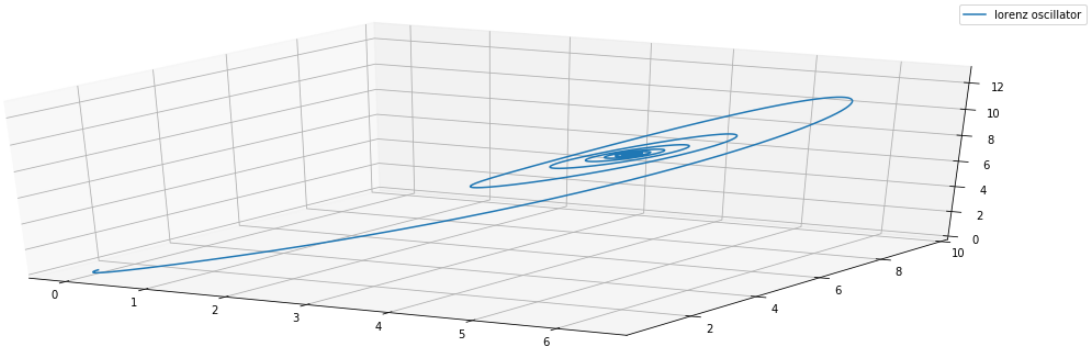
```
In [11]: fig = plt.figure(figsize=(20,6))
         fig.gca(projection='3d')

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

Out[11]: <matplotlib.legend.Legend at 0x7f15568014e0>



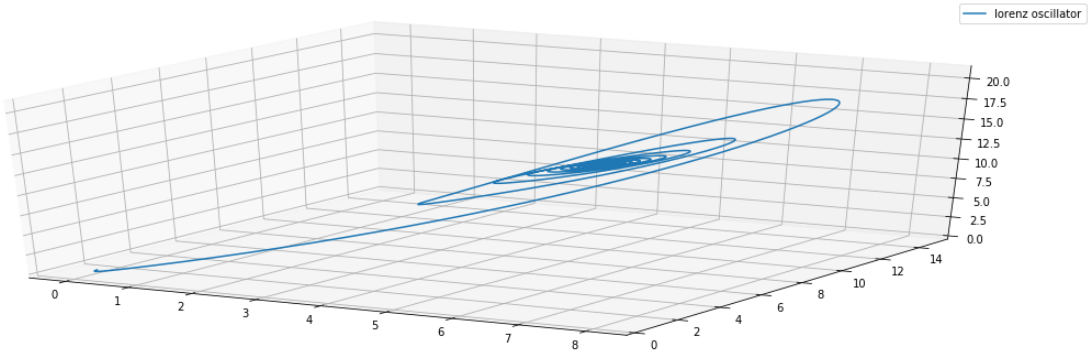

```
In [12]: fig = plt.figure(figsize=(20,6))
         fig.gca(projection='3d')

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

Out[12]: <matplotlib.legend.Legend at 0x7f15546c42e8>



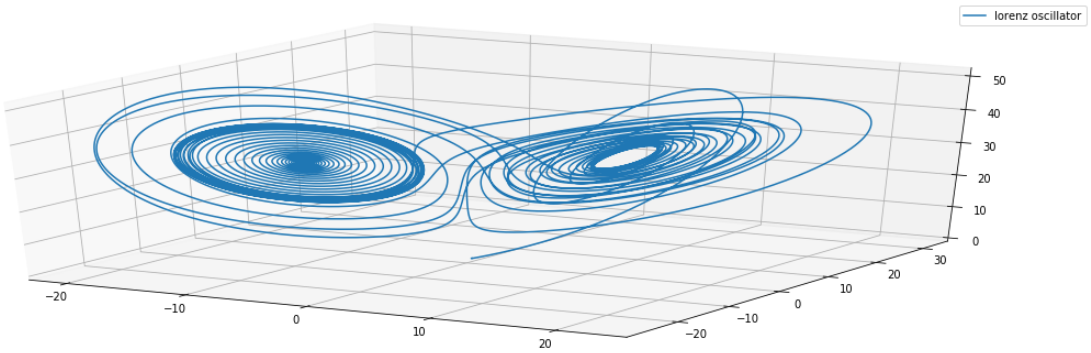
In [13]:
```
fig = plt.figure(figsize=(20,6))
fig.gca(projection='3d')

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

Out[13]: <matplotlib.legend.Legend at 0x7f158029d8d0>



In [14]:
```
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()
```

Out[14]: <matplotlib.legend.Legend at 0x7f1552d10a58>



What do you think the `rho` parameter does?

1.3 Python

IPython magic are special commands that can be used to interact with your System.

Enter the following into an IPython session:

```
In [1]: (an_integer,
         a_list,
         a_dictionary,
         a_set,
         a_tuple) = 1, [1,2,3], {'k': 1}, {1,2,2,3}, (1,2)
```

1.3.1 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
```

```
In [2]: from sys import getsizeof

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

In [3]: sizeof_and_value(an_integer)

Out[3]: (28, 1)

In [4]: sizeof_and_value(a_tuple)

Out[4]: (64, (1, 2))
```

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.

```
In [5]: 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:

```
In [6]: 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
```

1.3.2 IPython Magic

1.3.3 %whos

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

```
In [7]: %whos
```

```
Variable           Type                              Data/Info
-----------------------------------------------------------
a_dictionary       dict                              n=1
a_list             list                              n=3
a_set              set                               {1, 2, 3}
a_tuple            tuple                             n=2
an_integer         int                               1
getsizeof          builtin_function_or_method        <built-in function getsizeof>
sizeof_and_value   function                          <function sizeof_and_value at 0x7f18d0307e18>
type_and_value     function                          <function type_and_value at 0x7f18d0307ea0>
```

1.3.4 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

```
In [8]: %pwd
```

```
Out[8]: '/home/jovyan/introductiontodatascience/source/01-seeds'
```

cd - change directory

```
In [9]: %cd src/
```

```
[Errno 2] No such file or directory: 'src/'
/home/jovyan/introductiontodatascience/source/01-seeds
```

```
In [10]: %pwd
```

```
Out[10]: '/home/jovyan/introductiontodatascience/source/01-seeds'
```

ls - list files

```
In [11]: %ls
```

```
01-00-seeds.ipynb  01-01-interactive-programming.ipynb  01-03.ipynb
01-00-seeds.rst    01-02-Python.ipynb                   01-04.ipynb
```

```
In [12]: # HIDDEN TEST

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

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-12-1f0ebf75cbe5> in <module>()
      3 ### BEGIN HIDDEN TESTS
      4 import os
----> 5 assert os.getcwd().split('/')[-1] == 'src'
      6 ### END HIDDEN TESTS

AssertionError:
```

1.3.5 IPython Magic commands

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

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

```
In [ ]: %run a_simple_script.py
```

```
In [ ]: %matplotlib inline
```

```
In [ ]: %run a_simple_script.py
```

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

1.4 The Python Numerical Stack

Consists of:

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

```
In [1]: 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.

```
In [2]: from sklearn.linear_model import LinearRegression
```

1.4.1 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

```
In [3]: seeds_data = pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/00236/see
                                header=None, sep="\s+")
        seeds_data.columns = [
        "area",
        "perimeter",
        "compactness",
        "length of kernel",
        "width of kernel ",
        "asymmetry coefficient ",
        "length of kernel groove",
        "Class"
        ]
```

```
In [4]: seeds_data.shape
```

```
Out[4]: (210, 8)
```

1.4.2 What does `.shape` do?

Dataframes

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

```
In [5]: seeds_data.head()
```

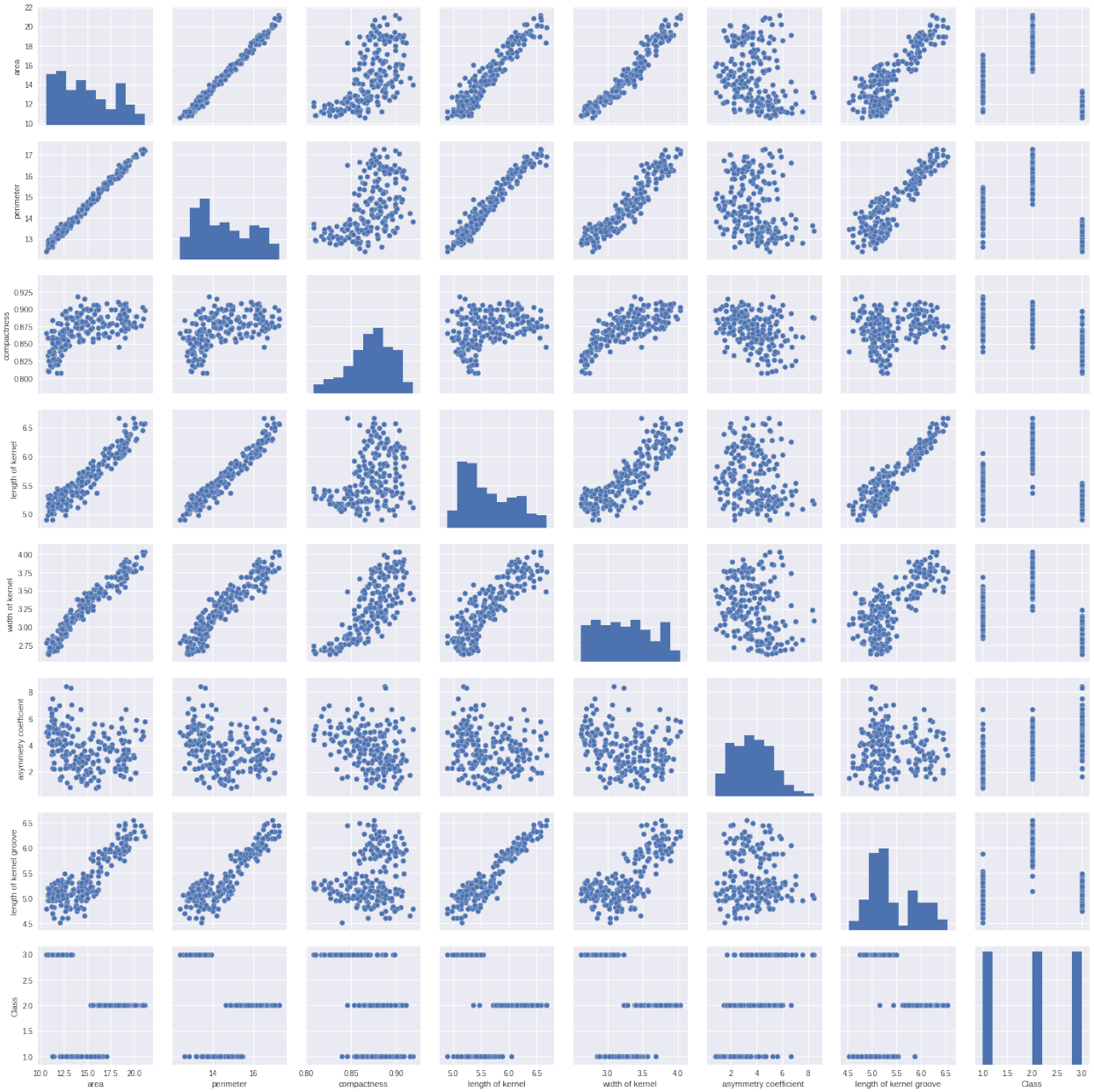
```
Out[5]: area  perimeter  compactness  length of kernel  width of kernel   \
        0  15.26      14.84       0.8710             5.763             3.312
        1  14.88      14.57       0.8811             5.554             3.333
        2  14.29      14.09       0.9050             5.291             3.337
        3  13.84      13.94       0.8955             5.324             3.379
        4  16.14      14.99       0.9034             5.658             3.562

           asymmetry coefficient   length of kernel groove  Class
        0                  2.221                    5.220      1
        1                  1.018                    4.956      1
        2                  2.699                    4.825      1
        3                  2.259                    4.805      1
        4                  1.355                    5.175      1
```

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.

```
In [6]: sns.pairplot(seeds_data)
```

```
Out[6]: <seaborn.axisgrid.PairGrid at 0x7f1cec6cac18>
```



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]
```

```
In [7]: def square_number(x):
            return x**2
```

```
In [8]: [square_number(i) for i in (1,2,3,4,5)]
```

```
Out[8]: [1, 4, 9, 16, 25]
```

1.4.3 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]
```

```
In [9]: 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
```

```
In [10]: 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
```

Remove Unit and White Space from Feature Name

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

```
In [11]: seeds_data.columns
```

```
Out[11]: Index(['area', 'perimeter', 'compactness', 'length of kernel',
                'width of kernel ', 'asymmetry coefficient ', 'length of kernel groove',
                'Class'],
               dtype='object')
```

```
In [12]: def remove_unit_and_white_space(feature_name):
             feature_name = feature_name.replace(' (cm)','')
             feature_name = feature_name.replace(' ', '_')
             return feature_name
```

```
In [13]: seeds_data_features_names = [remove_unit_and_white_space(name) for name in seeds_data.columns
```

```
In [14]: seeds_data_features_names
```

```
Out[14]: ['area',
          'perimeter',
          'compactness',
          'length_of_kernel',
          'width_of_kernel_',
          'asymmetry_coefficient_',
```

```
          'length_of_kernel_groove',
          'Class']
```

```
In [15]: seeds_data.columns = seeds_data_features_names
         seeds_data.head()
```

```
Out[15]:   area  perimeter  compactness  length_of_kernel  width_of_kernel_  \
         0  15.26      14.84       0.8710             5.763             3.312
         1  14.88      14.57       0.8811             5.554             3.333
         2  14.29      14.09       0.9050             5.291             3.337
         3  13.84      13.94       0.8955             5.324             3.379
         4  16.14      14.99       0.9034             5.658             3.562

            asymmetry_coefficient_  length_of_kernel_groove  Class
         0                   2.221                    5.220      1
         1                   1.018                    4.956      1
         2                   2.699                    4.825      1
         3                   2.259                    4.805      1
         4                   1.355                    5.175      1
```

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.

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

```
01-00-seeds.ipynb                       01-04-prediction.ipynb
01-00-seeds.rst                         data/
01-01-interactive-programming.ipynb     Untitled1.ipynb
01-02-Python.ipynb                      Untitled2.ipynb
01-03-the-python-numerical-stack.ipynb
```

```
In [17]: %mkdir -p data
```

```
In [18]: %ls
```

```
01-00-seeds.ipynb                       01-04-prediction.ipynb
01-00-seeds.rst                         data/
01-01-interactive-programming.ipynb     Untitled1.ipynb
01-02-Python.ipynb                      Untitled2.ipynb
01-03-the-python-numerical-stack.ipynb
```

```
In [19]: seeds_data.to_csv('data/seeds_data.csv')
```

1.5 Prediction

1.5.1 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 $\widehat{f}$ ("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

```
In [1]: %pwd
```

```
Out[1]: '/home/jovyan/introductiontodatascience/source/01-seeds'
```

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?

```
In [2]: %ls
```

```
01-00-seeds.ipynb                       01-05-the-train-test-split.ipynb
01-01-interactive-programming.ipynb     01-06-inference.ipynb
01-02-Python.ipynb                      data/
01-03-the-python-numerical-stack.ipynb  doc/
01-04-prediction.ipynb
```

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

        %matplotlib inline

        from patsy import dmatrices
```

```
In [4]: seeds_data = pd.read_csv('data/seeds_data.csv', index_col=0)
```

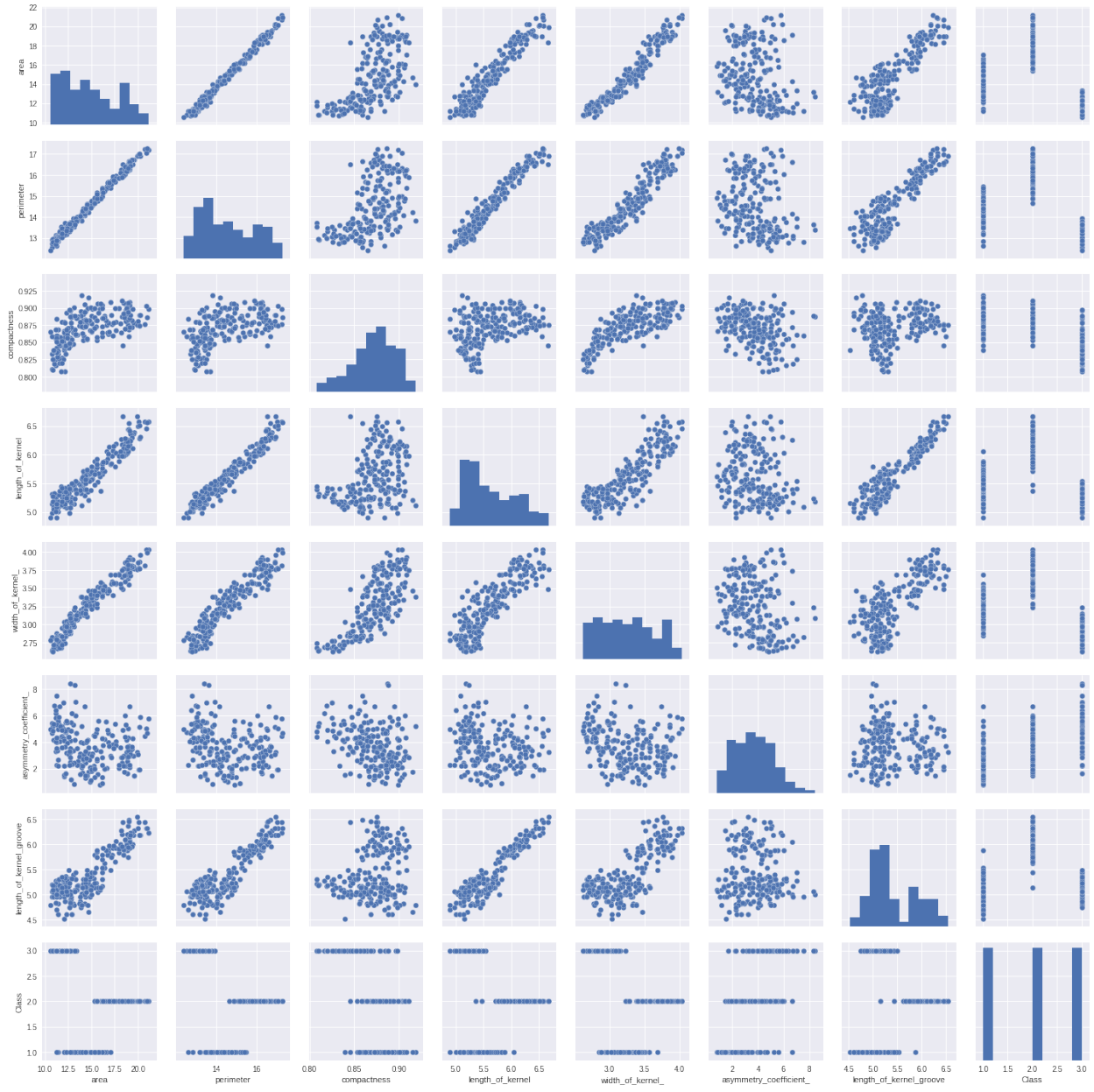
```
In [5]: seeds_data.describe()
```

```
Out[5]: area   perimeter  compactness  length_of_kernel  \
        count  210.000000  210.000000   210.000000        210.000000
        mean    14.847524   14.559286     0.870999          5.628533
        std      2.909699    1.305959     0.023629          0.443063
        min     10.590000   12.410000     0.808100          4.899000
        25%     12.270000   13.450000     0.856900          5.262250
        50%     14.355000   14.320000     0.873450          5.523500
        75%     17.305000   15.715000     0.887775          5.979750
        max     21.180000   17.250000     0.918300          6.675000

               width_of_kernel_  asymmetry_coefficient_  length_of_kernel_groove  \
        count         210.000000              210.000000               210.000000
        mean            3.258605                3.700201                 5.408071
        std             0.377714                1.503557                 0.491480
        min             2.630000                0.765100                 4.519000
        25%             2.944000                2.561500                 5.045000
        50%             3.237000                3.599000                 5.223000
        75%             3.561750                4.768750                 5.877000
        max             4.033000                8.456000                 6.550000

                    Class
        count  210.000000
        mean     2.000000
        std      0.818448
        min      1.000000
```

```
25%       1.000000
50%       2.000000
75%       3.000000
max       3.000000
```

In [6]:
```
plt.figure(1, (20,10))

sns.pairplot(seeds_data)
```

Out[6]:
```
<seaborn.axisgrid.PairGrid at 0x7f3bae6df0f0>

<Figure size 1440x720 with 0 Axes>
```

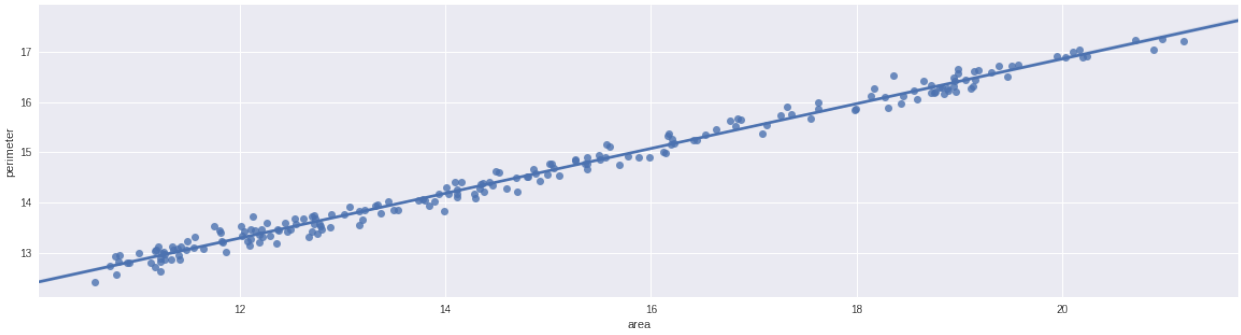


Having a look at the pair plot, we might say that we are able to the uniformity of cell shape using the uniformity of cell size.

In [7]:
```
plt.figure(1, (20,5))
```

```
In [7]: sns.regplot('area','perimeter', data=seeds_data)
```

```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3ba9e56b70>
```



1.5.2 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**.

1.5.3 Build a Simple Regression Model

```
In [8]: target, features = dmatrices("perimeter ~ area", seeds_data)
```

```
In [9]: from sklearn.linear_model import LinearRegression
```

```
In [10]: linear_regression_model = LinearRegression(fit_intercept=False)
         linear_regression_model.fit(features, target)
```

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

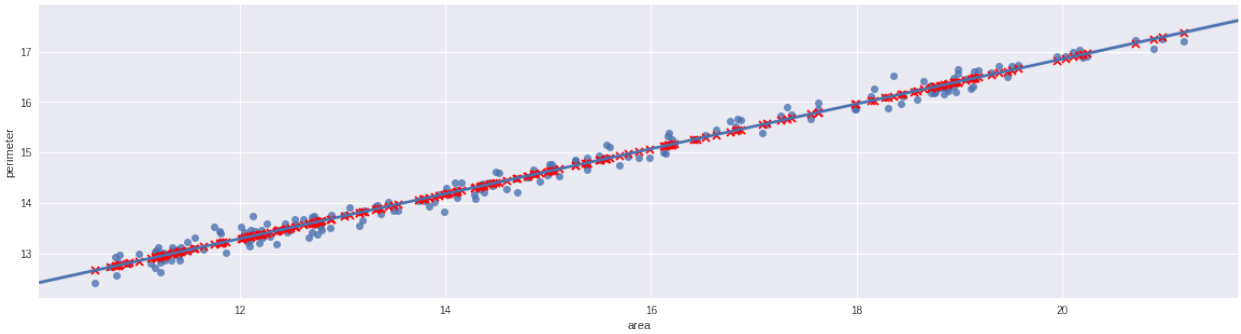
1.5.4 Plot the Results

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

We then plot the predictions versus the actual values.

```
In [11]: 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')
```

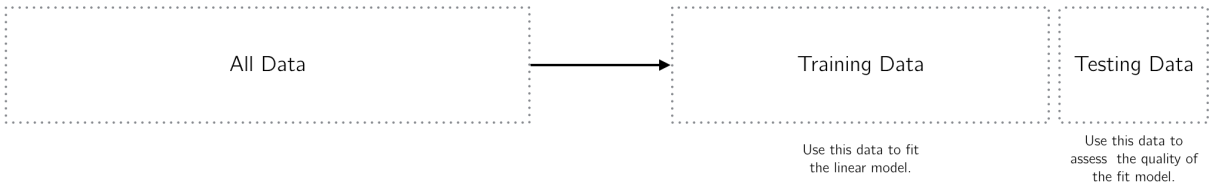
```
Out[11]: <matplotlib.collections.PathCollection at 0x7f3ba0a1e710>
```



What does this plot show us?

1.6 The Train-Test Split

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



1.6.1 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**.

1.6.2 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.

1.6.3 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.

```
In [1]: 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.model_selection import train_test_split
        from sklearn.linear_model import LinearRegression
```

```
In [2]: seeds_data = pd.read_csv('data/seeds_data.csv')
```

```
In [3]: 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.

```
In [4]: (features_train,
         features_test,
         target_train,
         target_test) = train_test_split(features, target, random_state=42)
```

```
In [5]: (features_train.shape,
         target_train.shape,
         features_test.shape,
         target_test.shape)
```

```
Out[5]: ((157, 2), (157, 1), (53, 2), (53, 1))
```

```
In [6]: features_test[:5]
```

```
Out[6]: array([[  1.  ,  13.16],
               [  1.  ,  11.27],
               [  1.  ,  19.51],
               [  1.  ,  12.76],
               [  1.  ,  11.42]])
```

```
In [7]: 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))
```

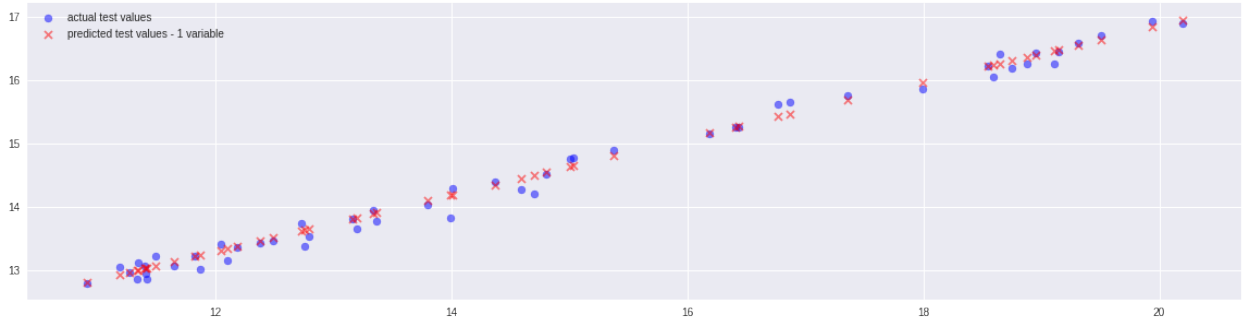
```
In [8]: seeds_data = pd.read_csv('data/seeds_data.csv')
```

```
In [9]: target, features = dmatrices("perimeter ~ area", seeds_data)
```

```
In [10]: (features_train,
          features_test,
          target_train,
          target_test) = train_test_split(features, target, random_state=42)
```

```
In [11]: 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[11]: <matplotlib.legend.Legend at 0x7f3436144b70>
```



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

1.7 Inference

1.7.1 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

```
In [1]: seeds.data = read.csv('data/seeds_data.csv', row.names=1)
```

Sanity Check

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

	area	perimeter	compactness	length_of_kernel	width_of_kernel_	asymmetry_coefficient_	length_of_kernel_groove
0	15.26	14.84	0.8710	5.763	3.312	2.221	5.220
1	14.88	14.57	0.8811	5.554	3.333	1.018	4.956
2	14.29	14.09	0.9050	5.291	3.337	2.699	4.825
3	13.84	13.94	0.8955	5.324	3.379	2.259	4.805
4	16.14	14.99	0.9034	5.658	3.562	1.355	5.175
5	14.38	14.21	0.8951	5.386	3.312	2.462	4.956

```
In [3]: summary(seeds.data)
```

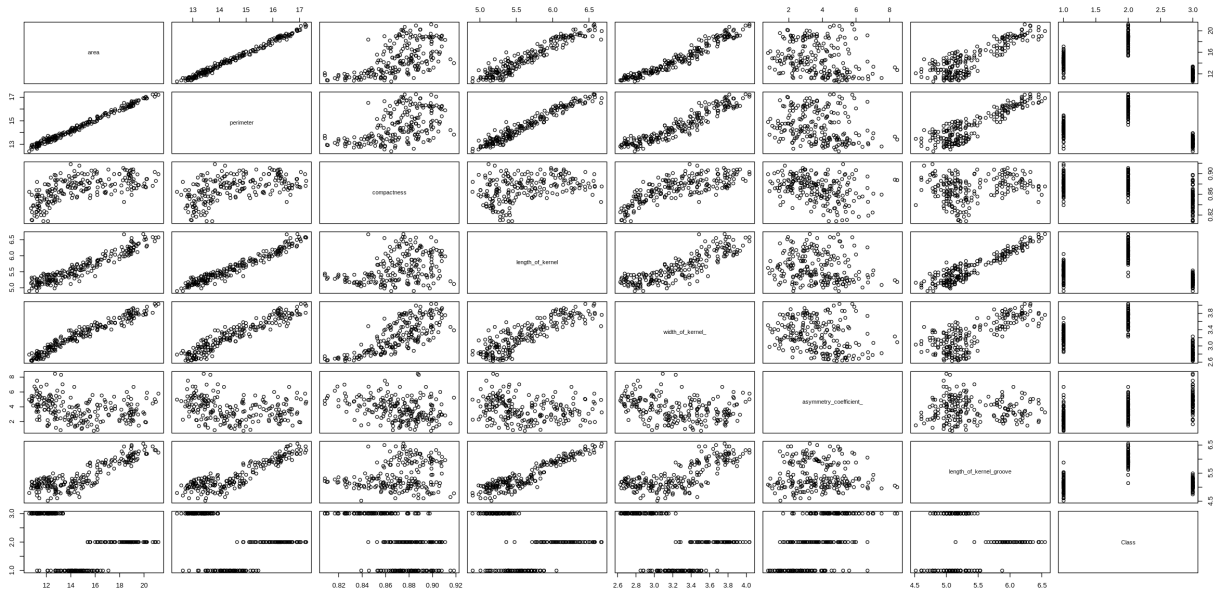
```
area          perimeter       compactness     length_of_kernel
 Min.   :10.59   Min.   :12.41   Min.   :0.8081   Min.   :4.899
 1st Qu.:12.27   1st Qu.:13.45   1st Qu.:0.8569   1st Qu.:5.262
 Median :14.36   Median :14.32   Median :0.8734   Median :5.524
 Mean   :14.85   Mean   :14.56   Mean   :0.8710   Mean   :5.629
 3rd Qu.:17.30   3rd Qu.:15.71   3rd Qu.:0.8878   3rd Qu.:5.980
 Max.   :21.18   Max.   :17.25   Max.   :0.9183   Max.   :6.675
 width_of_kernel_ asymmetry_coefficient_ length_of_kernel_groove     Class
 Min.   :2.630    Min.   :0.7651         Min.   :4.519           Min.   :1
 1st Qu.:2.944    1st Qu.:2.5615         1st Qu.:5.045           1st Qu.:1
```

```
 Median :3.237    Median :3.5990         Median :5.223          Median :2
 Mean   :3.259    Mean   :3.7002         Mean   :5.408          Mean   :2
 3rd Qu.:3.562    3rd Qu.:4.7687         3rd Qu.:5.877          3rd Qu.:3
 Max.   :4.033    Max.   :8.4560         Max.   :6.550          Max.   :3
```

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

In [5]:
```
options(repr.plot.width=20, repr.plot.height=10)
```

In [6]:
```
pairs(seeds.data)
```



In [7]:
```
library(ggplot2)
```

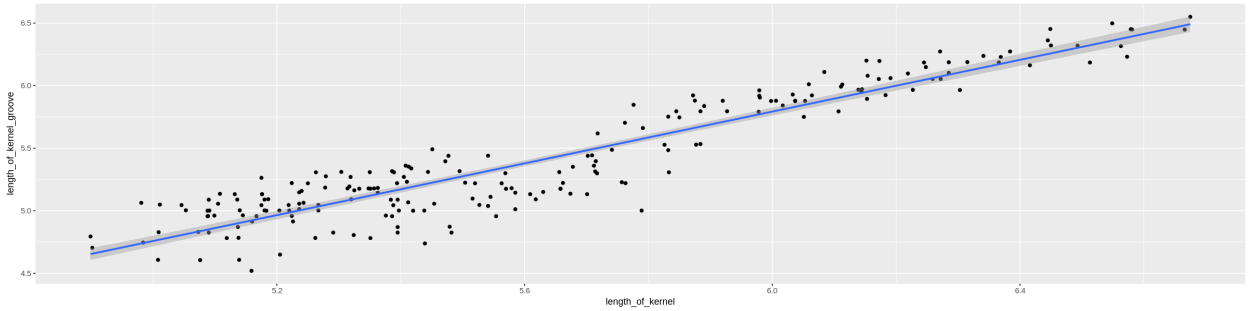
In [8]:
```
str(seeds.data)
```

```
'data.frame':   210 obs. of  8 variables:
 $ area                   : num  15.3 14.9 14.3 13.8 16.1 ...
 $ perimeter              : num  14.8 14.6 14.1 13.9 15 ...
 $ compactness            : num  0.871 0.881 0.905 0.895 0.903 ...
 $ length_of_kernel       : num  5.76 5.55 5.29 5.32 5.66 ...
 $ width_of_kernel_       : num  3.31 3.33 3.34 3.38 3.56 ...
 $ asymmetry_coefficient_ : num  2.22 1.02 2.7 2.26 1.35 ...
 $ length_of_kernel_groove: num  5.22 4.96 4.83 4.8 5.17 ...
 $ Class                  : int  1 1 1 1 1 1 1 1 1 1 ...
```

In [9]:
```
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')
```

Data type cannot be displayed:



1.7.2 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**.

```
In [10]: lm_1_var = lm('length_of_kernel_groove ~ length_of_kernel', seeds.data)
         lm_1_var


Call:
lm(formula = "length_of_kernel_groove ~ length_of_kernel", data = seeds.data)

Coefficients:
     (Intercept)  length_of_kernel
          -0.416             1.035


In [11]: lm('length_of_kernel_groove ~ .', seeds.data)


Call:
lm(formula = "length_of_kernel_groove ~ .", data = seeds.data)

Coefficients:
            (Intercept)                    area               perimeter
              -1.147083                0.067233                0.096897
           compactness        length_of_kernel        width_of_kernel_
              1.258737                0.806318               -0.562471
asymmetry_coefficient_                   Class
              0.003122                0.166367
```

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

TWO

THE IRIS DATASET

2.1 The Iris Data Set

2.1.1 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

```
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 <https://en.wikipedia.org/wiki/Iris_flower_data_set>`__
```

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

In [3]: type(IRIS.data)

Out[3]: numpy.ndarray

In [4]: IRIS.data.shape

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

What does `.shape` do?

Dataframes

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

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

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

Out[6]:

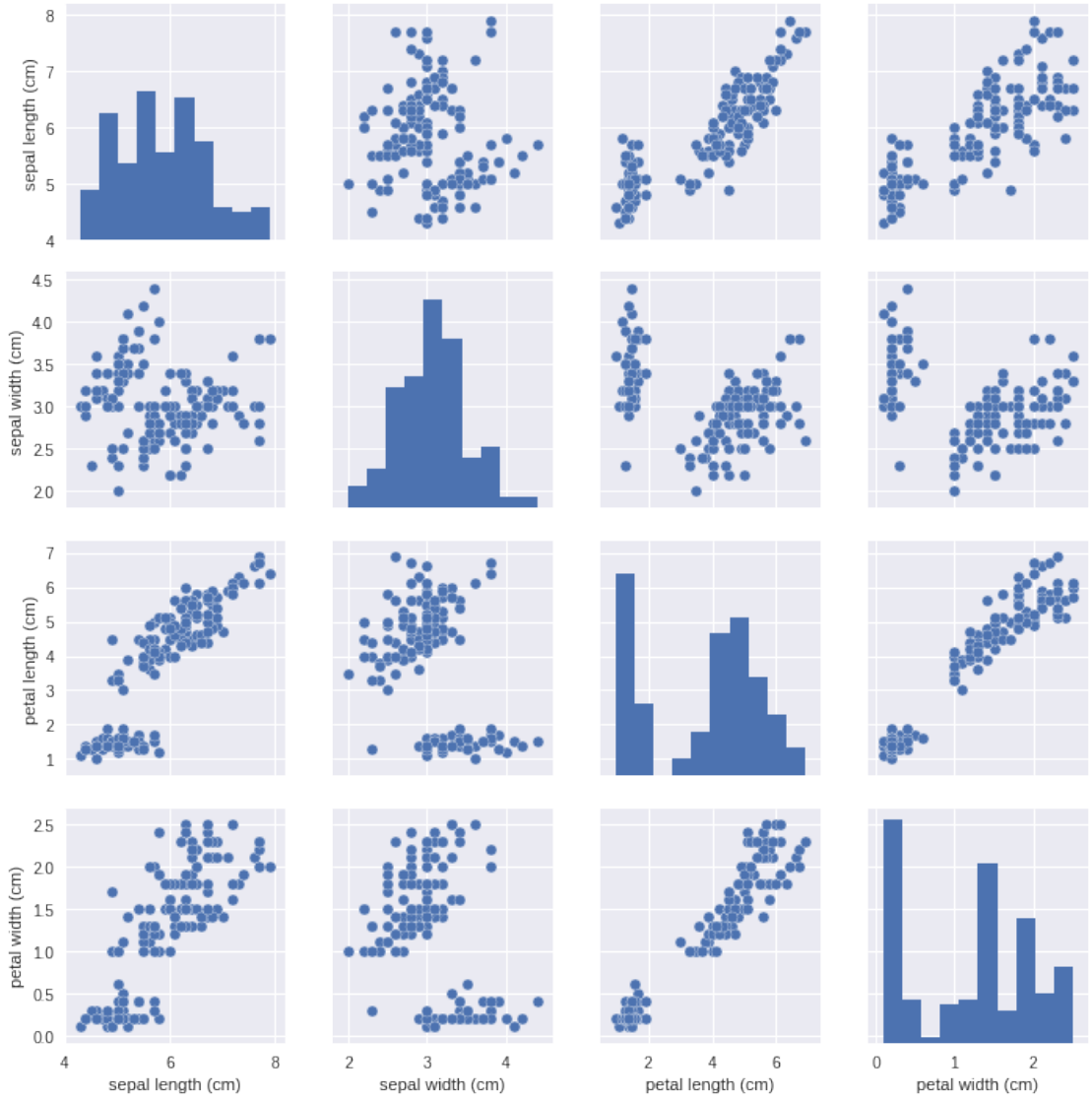
	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

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.

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

```
Out[7]: <seaborn.axisgrid.PairGrid at 0x7f50ca7b7e80>
```



Remove Unit and White Space from Feature Name

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

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

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

```
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)']
```

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

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

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

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

```
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
```

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.

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

```
02-01-iris.ipynb          02-05-probabilistic-model-selection.ipynb
02-02-learning.ipynb      02-06-cluster-modeling.ipynb
02-03-sampling.ipynb      Untitled3.ipynb
02-04-probability.ipynb
```

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

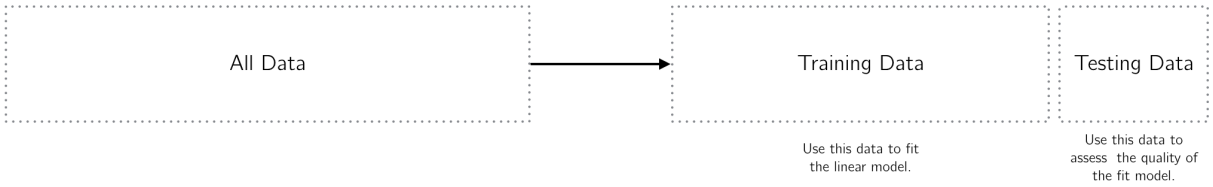
```
In [16]: %ls
```

```
02-01-iris.ipynb          02-05-probabilistic-model-selection.ipynb
02-02-learning.ipynb      02-06-cluster-modeling.ipynb
02-03-sampling.ipynb      data/
02-04-probability.ipynb   Untitled3.ipynb
```

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

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.




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

```
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.

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

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

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

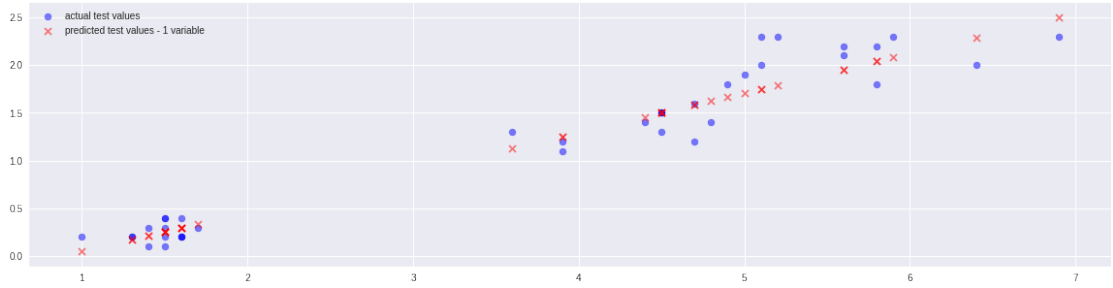
```
In [22]: features_test[:5]
```

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

```
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))
```

```
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 0x7f50c56b3cf8>
```



Multicollinearity

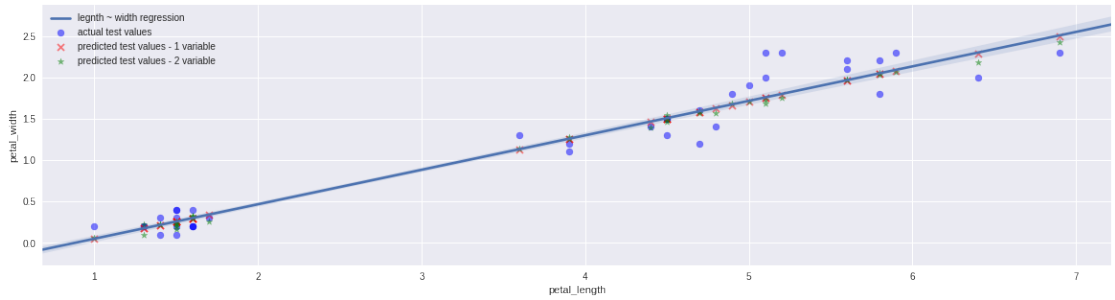
How is the prediction affected by adding additional predictor variables?

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
```

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

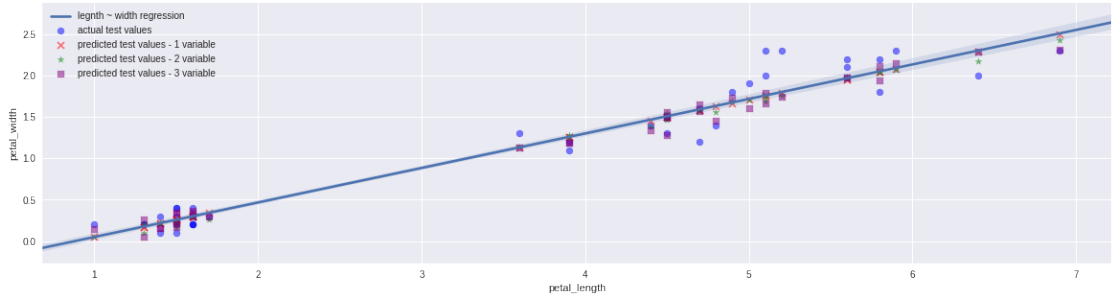
In [27]:
```
target, features = dmatrices("petal_width ~ petal_length + sepal_length", iris_df)

(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 variab
plt.scatter(features_test[:, 1], petal_width_prediction_2_var,
            marker='*', color='green', alpha=0.5, label='predicted test values - 2 vari
sns.regplot('petal_length', 'petal_width', data=iris_df, scatter=False, label='legnth ~
plt.legend()
```

Out[27]:
```
<matplotlib.legend.Legend at 0x7f50c567bcc0>
```



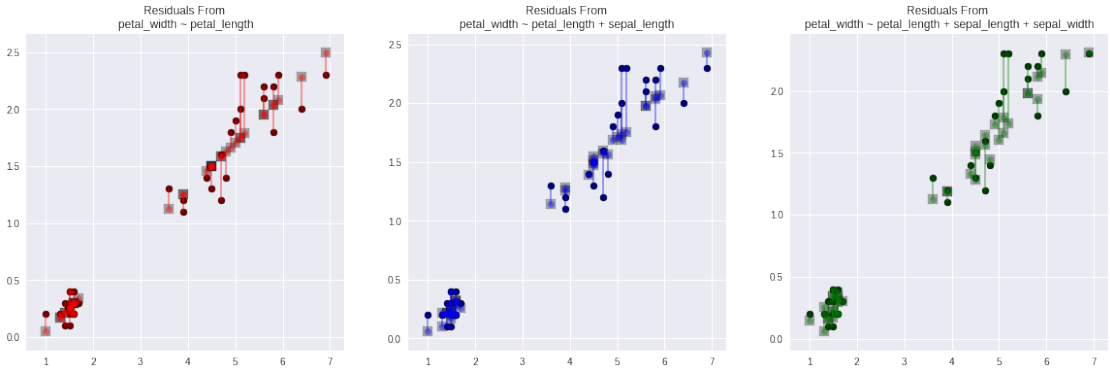
In [28]:
```
target, features = dmatrices("petal_width ~ petal_length + sepal_length + sepal_width",

(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 variab
plt.scatter(features_test[:, 1], petal_width_prediction_2_var,
            marker='*', color='green', alpha=0.5, label='predicted test values - 2 vari
plt.scatter(features_test[:, 1], petal_width_prediction_3_var,
            marker='s', color='purple', alpha=0.5, label='predicted test values - 3 var
sns.regplot('petal_length', 'petal_width', data=iris_df, scatter=False, label='legnth ~

plt.legend()
```

Out[28]:
```
<matplotlib.legend.Legend at 0x7f50c56038d0>
```



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))
```

```
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

         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')
```



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.

```
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))
```

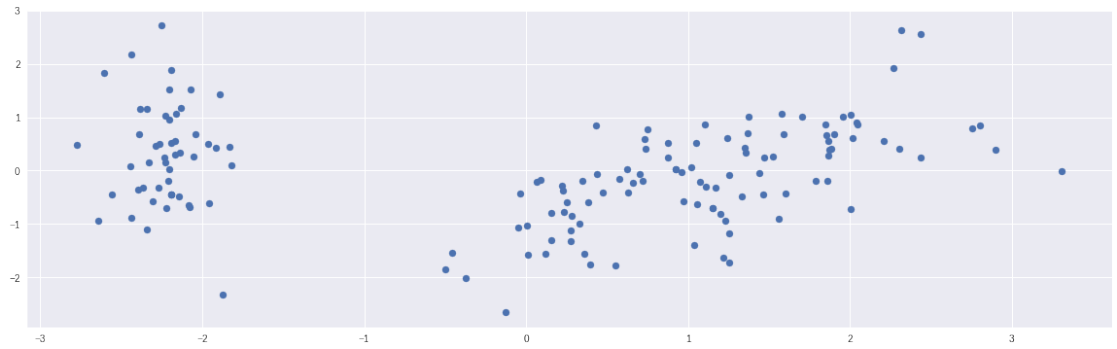
High-Dimensional Data

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.

```
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 0x7f50c4ed6748>
```



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

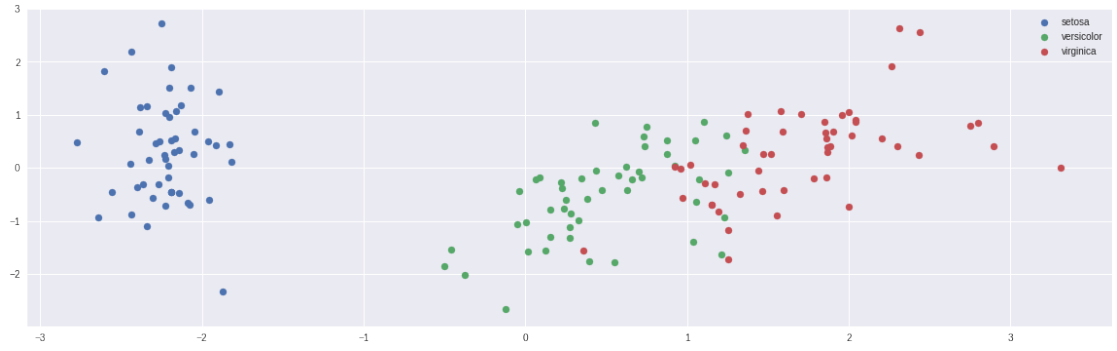
```
In [33]: labels = IRIS.target_names
         labels
```

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

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

```
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 0x7f50c4c7a710>
```



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.

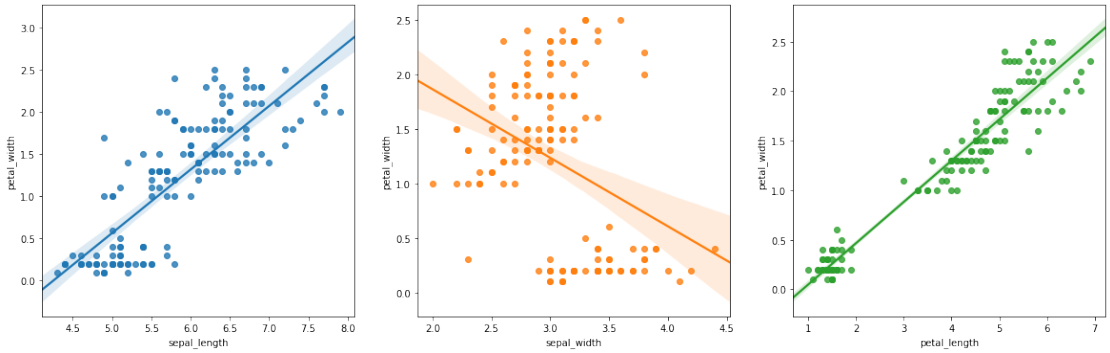
In other cases, we might seek to develop a model from a set of features without any corresponding target data. This type of model development is known as **Unsupervised Learning**. It is unsupervised because the targets are unknown. Common unsupervised learning tasks are **clustering**, in which we attempt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

```
In [1]: 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))

        IRIS = load_iris()
        iris_df = pd.DataFrame(IRIS.data, columns=['sepal_length', 'sepal_width', 'petal_length'
        labels = IRIS.target_names
```

```
In [2]: feat_names = iris_df.columns
```

2.2.1 Regression

```
In [3]: _, 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])
```




```
In [4]: 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 with linear regression." % feature)
            features = iris_df[[feature]]
            target = iris_df['petal_width']
            model.fit(features, target)
```

```
Fitting sepal_length on petal_width with linear regression.
Fitting sepal_width on petal_width with linear regression.
Fitting petal_length on petal_width with linear regression.
```

2.2.2 The Mean Squared Error

One common metric for assessing the performance of a regression model is the expectation of the squared error also known as the **Mean Squared Error**

$$MSE = \mathbb{E}\left[(y - \widehat{y})^2\right] = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2$$

In layman's terms this is the average of the squared difference between the actual and the predicted value. Squaring the difference means that

1. The sign of the difference is is irrelevant i.e. overestimating is equivalent to underestimating
2. Larger errors will be more impactful than smaller errors when squared

e.g. an error of 0.1 becomes 0.01, an error of 10 becomes 100

```
In [5]: def MSE(actual, predicted):
            return sum((actual - predicted)**2)/len(actual)
```

```
In [6]: for feature, model in zip(feat_names[:3], linear_models):
            features = iris_df[[feature]]
            target = iris_df['petal_width']
            print("Scoring linear regression model fit with %s." % feature)
            print("MSE: %f" % MSE(target, model.predict(features)))
```

```
Scoring linear regression model fit with sepal_length.
MSE: 0.191466
Scoring linear regression model fit with sepal_width.
MSE: 0.504986
Scoring linear regression model fit with petal_length.
MSE: 0.042290
```

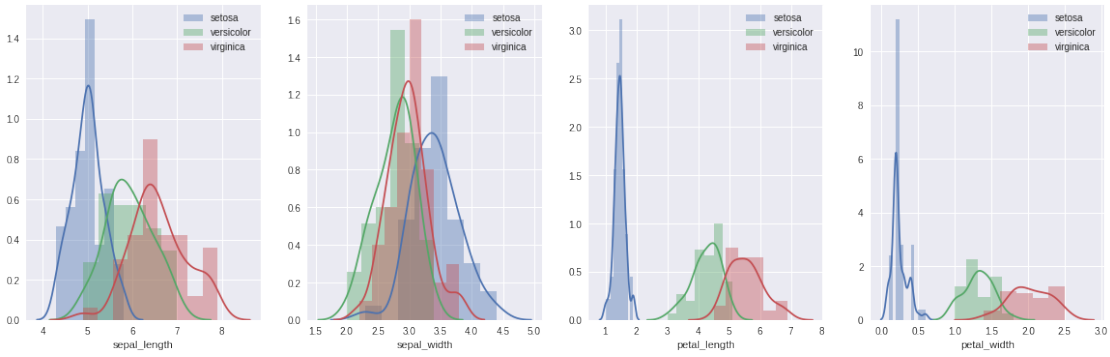
2.2.3 Classification

```
In [7]: 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))
```

```
In [8]: iris_df['target'] = IRIS.target_names[IRIS.target]
```

```
In [9]: _, 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()
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```




```
In [10]: 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'])
```

```
In [11]: (features_train,
          features_validation,
          target_train,
          target_validation) = train_test_split(features, target_numerical)
```

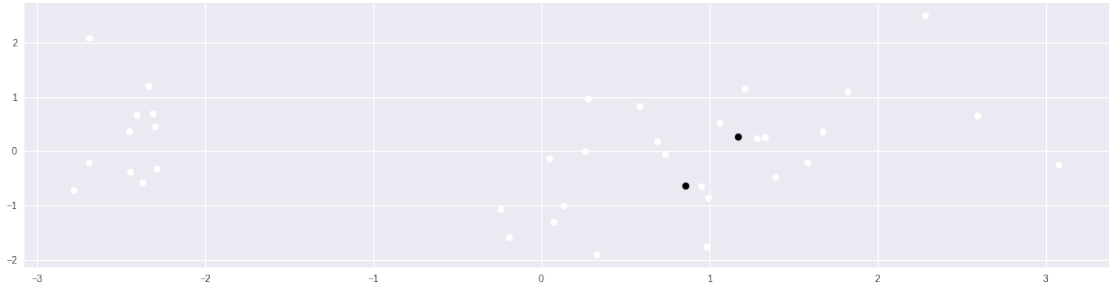
2.2.4 Display the Classification Predictions and Actual

```
In [12]: tree.fit(features_train, target_train)
         target_prediction = tree.predict(features_validation)
         target_prediction
```

```
Out[12]: array([1, 1, 0, 2, 2, 0, 2, 2, 2, 2, 1, 1, 2, 0, 2, 1, 1, 0, 1, 1, 2, 2, 2,
                0, 0, 2, 0, 1, 0, 0, 1, 2, 2, 1, 1, 1, 0, 0])
```

```
In [13]: target_validation
```

```
Out[13]: array([1, 1, 0, 1, 2, 0, 2, 2, 2, 2, 1, 1, 2, 0, 2, 1, 1, 0, 1, 1, 2, 2, 2,
                0, 0, 2, 0, 1, 0, 0, 1, 2, 1, 1, 1, 1, 0, 0])

In [14]: difference = np.abs(target_validation - target_prediction)
         difference

Out[14]: array([0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0])

In [15]: from sklearn.decomposition import PCA
         from sklearn.preprocessing import StandardScaler

         number_of_dimensions = 2
         pca = PCA(number_of_dimensions)

         features_scaled_validation = StandardScaler().fit_transform(features_validation)
         iris_2d_validation = pca.fit_transform(features_scaled_validation)

In [16]: plt.figure(figsize=(20,5))
         plt.scatter(x=iris_2d_validation[:,0], y=iris_2d_validation[:,1], c=difference)

Out[16]: <matplotlib.collections.PathCollection at 0x7f6182388550>
```



```
In [17]: for i, label in enumerate(labels):
             group_mask = target_validation == i
             group = iris_2d_validation[group_mask]
             plt.scatter(group[:, 0], group[:, 1], label=label)
         plt.legend()

Out[17]: <matplotlib.legend.Legend at 0x7f618239ca90>
```

2.2.5 Measure the Accuracy

```
In [18]: def accuracy(actual, predicted):
             return 1 - sum(np.abs(actual - predicted))/len(actual)

In [19]: accuracy(target_validation, target_prediction)
```

Out[19]: 0.94736842105263164

2.2.6 Clustering

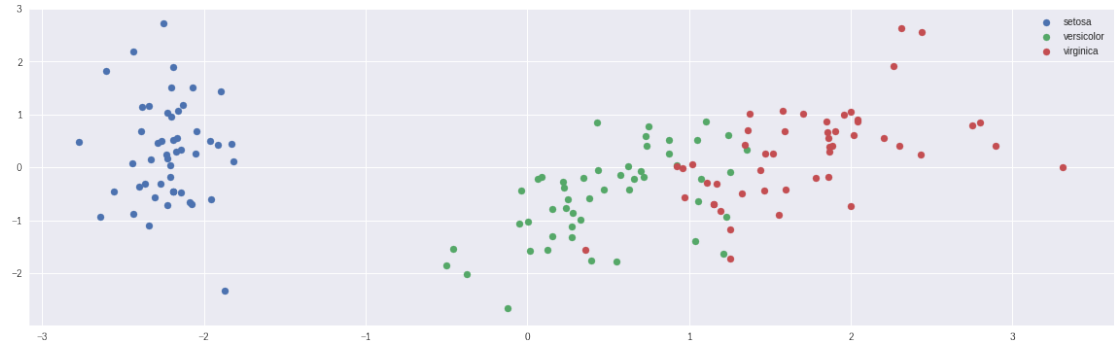
In [20]:
```
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

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

In [21]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

number_of_dimensions = 2
pca = PCA(number_of_dimensions)
```

In [22]:
```
from sklearn.cluster import KMeans

number_of_clusters = [2,3,4,5]

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

features_scaled = StandardScaler().fit_transform(IRIS.data)
iris_2d = pca.fit_transform(features_scaled)
named_target = np.array([labels[n] for n in IRIS.target])

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

In [23]:
```
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[23]: <matplotlib.legend.Legend at 0x7f61801d80f0>



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

```
In [1]: 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))
```

```
        IRIS = load_iris()
        iris_df = pd.DataFrame(IRIS.data, columns=['sepal_length', 'sepal_width', 'petal_length'
        labels = IRIS.target_names
```

```
In [2]: samp = iris_df.sample(5)
        samp
```

```
Out[2]: sepal_length  sepal_width  petal_length  petal_width
        0             5.1          3.5           1.4          0.2
        77            6.7          3.0           5.0          1.7
        94            5.6          2.7           4.2          1.3
        81            5.5          2.4           3.7          1.0
        126           6.2          2.8           4.8          1.8
```

And it becomes

```
In [3]: samp_melt = pd.melt(samp.select_dtypes([float]))
        samp_melt
```

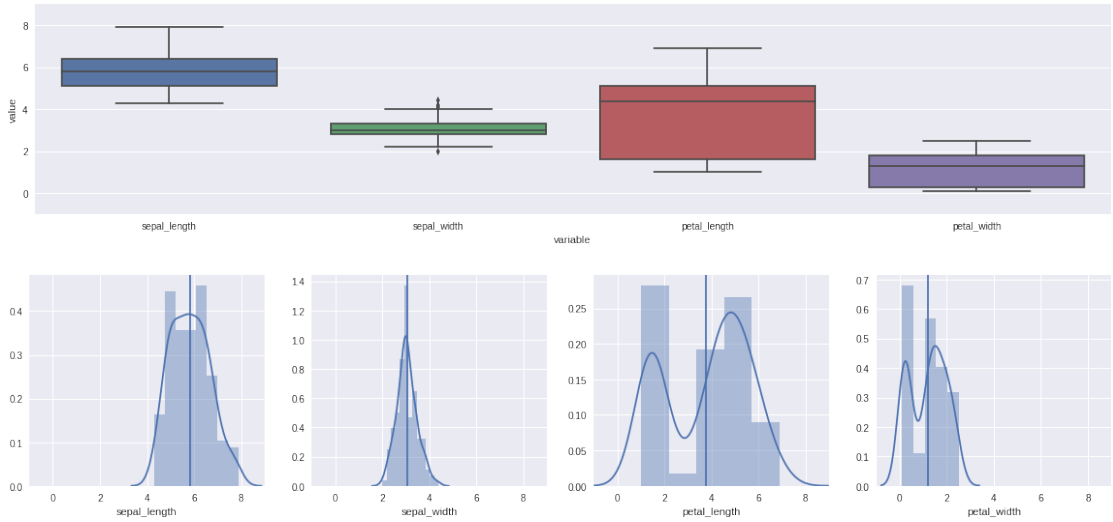
```
Out[3]: variable  value
        0   sepal_length    5.1
        1   sepal_length    6.7
        2   sepal_length    5.6
        3   sepal_length    5.5
        4   sepal_length    6.2
        5    sepal_width    3.5
        6    sepal_width    3.0
        7    sepal_width    2.7
        8    sepal_width    2.4
        9    sepal_width    2.8
        10  petal_length    1.4
        11  petal_length    5.0
        12  petal_length    4.2
        13  petal_length    3.7
        14  petal_length    4.8
        15   petal_width    0.2
        16   petal_width    1.7
        17   petal_width    1.3
        18   petal_width    1.0
        19   petal_width    1.8
```

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

```
In [4]: iris_melt = pd.melt(iris_df.select_dtypes([float]))
```

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

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

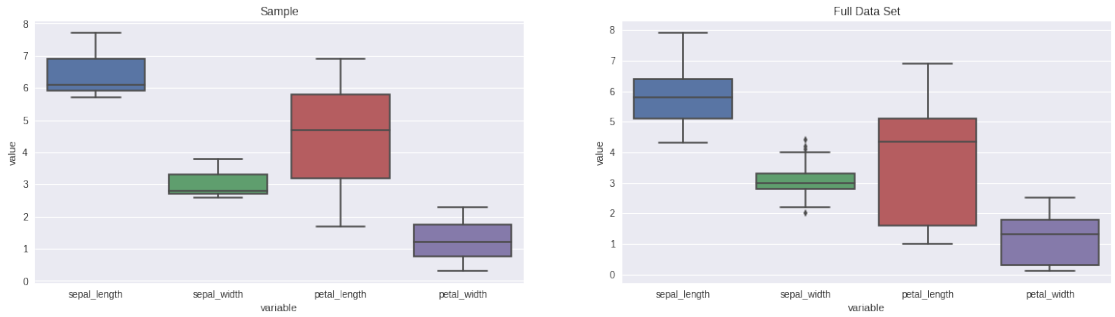



```
In [6]: np.random.seed(42)

In [7]: sample_1 = iris_df.sample(3)

In [8]: sample_1

Out[8]: sepal_length  sepal_width  petal_length  petal_width
        73            6.1          2.8           4.7          1.2
        18            5.7          3.8           1.7          0.3
        118           7.7          2.6           6.9          2.3

In [9]: iris_df.describe().T

Out[9]: count       mean       std  min  25%   50%  75%  max
        sepal_length  150.0  5.843333  0.828066  4.3  5.1  5.80  6.4  7.9
        sepal_width   150.0  3.054000  0.433594  2.0  2.8  3.00  3.3  4.4
        petal_length  150.0  3.758667  1.764420  1.0  1.6  4.35  5.1  6.9
        petal_width   150.0  1.198667  0.763161  0.1  0.3  1.30  1.8  2.5

In [10]: sample_1.describe().T

Out[10]: count       mean       std  min   25%  50%   75%  max
         sepal_length    3.0  6.500000  1.058301  5.7  5.90  6.1  6.90  7.7
         sepal_width     3.0  3.066667  0.642910  2.6  2.70  2.8  3.30  3.8
         petal_length    3.0  4.433333  2.610236  1.7  3.20  4.7  5.80  6.9
         petal_width     3.0  1.266667  1.001665  0.3  0.75  1.2  1.75  2.3
```

2.3.3 Visualize the Differences Using Seaborn

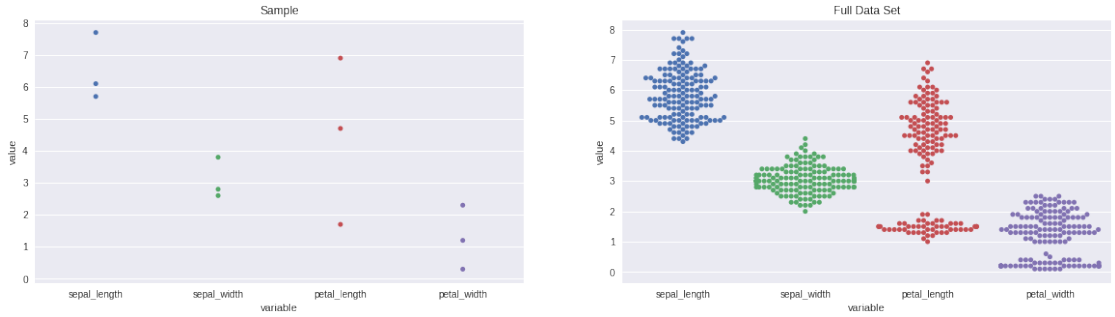
VIsualized with a Box Plot

```
In [11]: sample_1_melt = pd.melt(sample_1.select_dtypes([float]))

In [12]: _, 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');
```



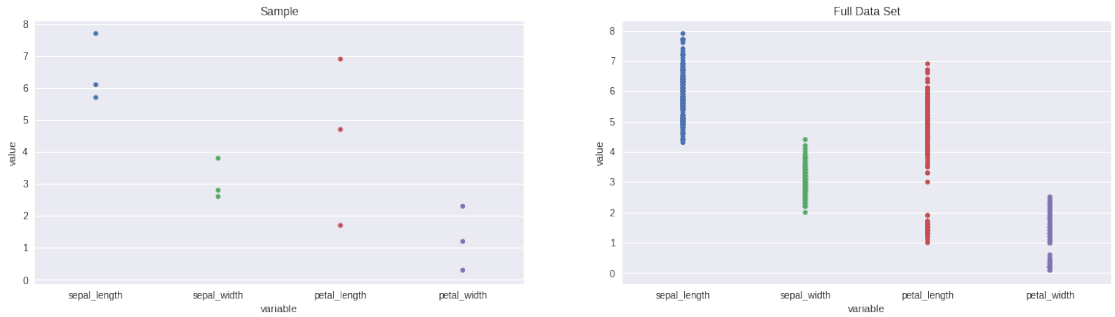
Visualized with a Swarmplot

```
In [13]: _, 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');
```



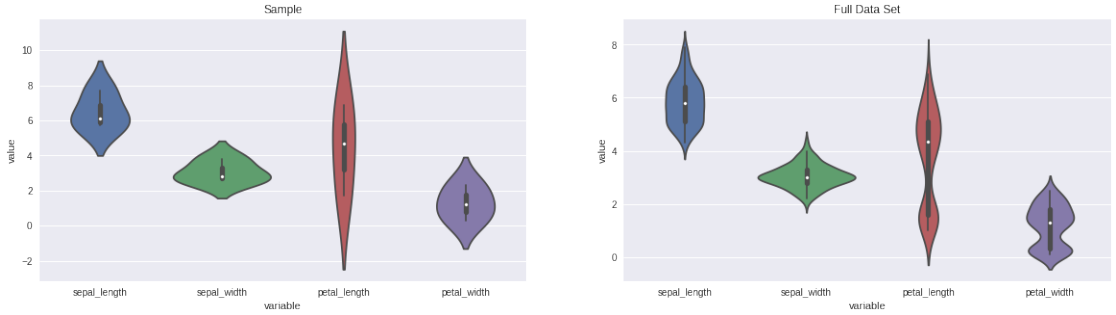
Visualized with a Stripplot

```
In [14]: _, 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

```
In [15]: _, 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

```
In [16]: error_sample_1 = np.abs(iris_df.mean() - sample_1.mean())
         error_sample_1

Out[16]: sepal_length    0.656667
         sepal_width     0.012667
         petal_length    0.674667
         petal_width     0.068000
         dtype: float64
```

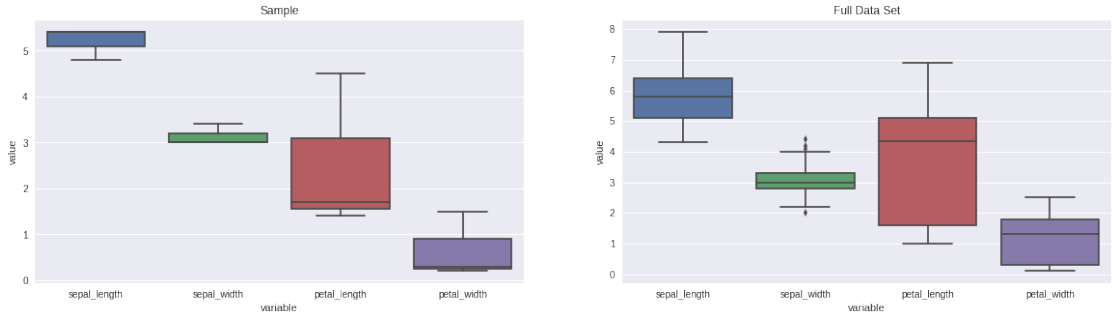
Normalized

```
In [17]: error_sample_1_normalized = np.abs((iris_df.mean() - sample_1.mean())/iris_df.std())
         error_sample_1_normalized

Out[17]: sepal_length    0.793012
         sepal_width     0.029213
         petal_length    0.382373
         petal_width     0.089103
         dtype: float64
```

2.3.4 A Second Sample

```
In [18]: 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');
```




```
In [19]: error_sample_2_normalized = np.abs((iris_df.mean() - sample_2.mean())/iris_df.std())
```

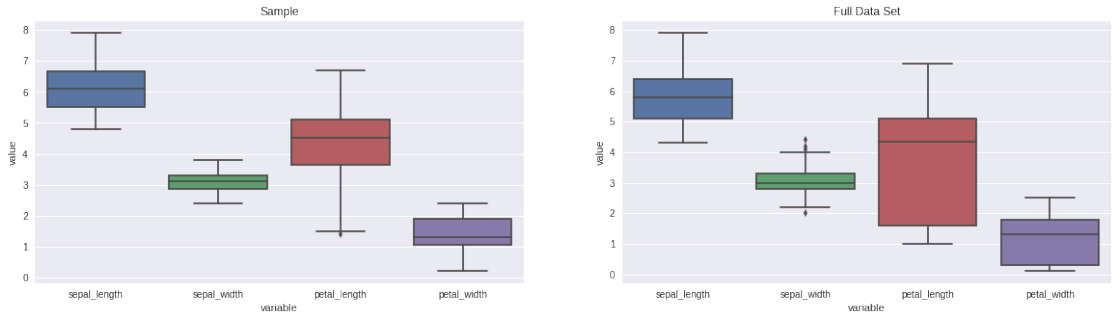
```
In [20]: display(error_sample_1_normalized)
         display(error_sample_2_normalized)
```

```
sepal_length    0.793012
sepal_width     0.029213
petal_length    0.382373
petal_width     0.089103
dtype: float64
```

```
sepal_length    0.776911
sepal_width     0.182967
petal_length    0.694468
petal_width     0.697101
dtype: float64
```

2.3.5 What about a larger sample?

```
In [21]: 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');
```




```
In [22]: error_sample_3_normalized = np.abs((iris_df.mean() - sample_3.mean())/iris_df.std())
```

```
In [23]: display(error_sample_1_normalized)
         display(error_sample_2_normalized)
         display(error_sample_3_normalized)
```

```
sepal_length    0.793012
sepal_width     0.029213
petal_length    0.382373
petal_width     0.089103
dtype: float64
```

```
sepal_length    0.776911
sepal_width     0.182967
petal_length    0.694468
petal_width     0.697101
dtype: float64
```

```
sepal_length    0.406570
sepal_width     0.121465
petal_length    0.219902
petal_width     0.185195
dtype: float64
```

```
In [2]: 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
```

```
In [3]: iris_df.columns
```

```
Out[3]: Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width'], dtype='object')
```

```
In [4]: feature_error_by_n(iris_df, 'sepal_length', 3)
```

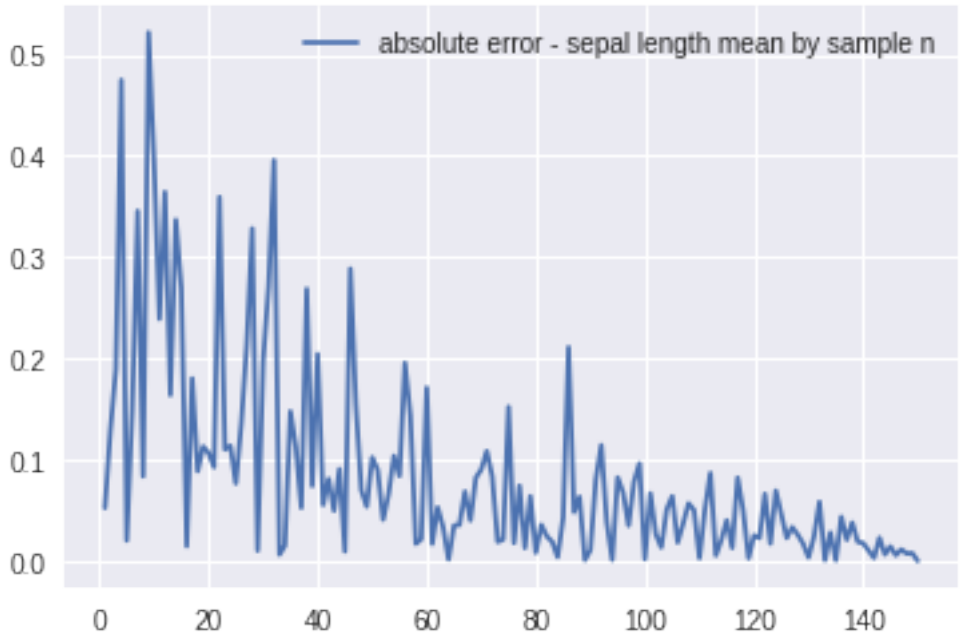
```
Out[4]: 0.1730940663922016
```

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

```
In [5]: sepal_length_error_by_n = [feature_error_by_n(iris_df, 'sepal_length', n) for n in range
```

```
In [6]: plt.plot(range(1,151), sepal_length_error_by_n, label='absolute error - sepal length mea
        plt.legend()
```

```
Out[6]: <matplotlib.legend.Legend at 0x7fbc034f5b70>
```




```
In [7]: iris_df['label'] = IRIS.target

In [8]: iris_df.to_csv('data/iris.csv')

In [1]: 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))

        IRIS = load_iris()
        iris_df = pd.DataFrame(IRIS.data, columns=['sepal_length', 'sepal_width', 'petal_length'
        labels = IRIS.target_names
```

2.4 The Bias-Variance Tradeoff

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

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

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

to reconcile with their meaning when thinking about the Bias-Variance Tradeoff.

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

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

2.4.1 The Gory Details

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

Let's consider sources of possible test error:

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

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

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

Model Assessment

Consider these eight models using the four features in our dataset, petal length x_{pl}, petal width x_{pw}, sepal length x_{sl}, and sepal width x_{sw}

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

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

```
In [2]: samples = []
        for _ in range(10):
            samples.append(iris_df.sample(50))
```

```
In [3]: from patsy import dmatrices
```

```
In [4]: from sklearn.linear_model import LinearRegression
        import numpy as np
```

```
In [5]: def MSE(actual, predicted):
            return sum((actual - predicted)**2)/len(actual)

        def test_func(model_description):

            test = dict()

            test['samples'] = [dmatrices(model_description, sample) for sample in samples]
            test['models']  = [LinearRegression(fit_intercept=False) for _ in range(10)]
            test['scores']  = []

            for model, sample in zip(test['models'], test['samples']):
                target = sample[0]
                features = sample[1]
                model.fit(features, target)
                test['scores'].append(MSE(model.predict(features), target))

            test['scores'] = np.array(test['scores'])
            results = { 'description' : model_description }
            results['bias'] = test['scores'].mean()
            results['variance'] = test['scores'].std()
            return test, results
```

```
In [6]: test_1, results_1 = test_func("petal_width ~ 1")
        test_2, results_2 = test_func("petal_width ~ 1 + petal_length")
        test_3, results_3 = test_func("petal_width ~ 1 + sepal_length")
        test_4, results_4 = test_func("petal_width ~ 1 + sepal_width")
        test_5, results_5 = test_func("petal_width ~ 1 + petal_length + sepal_width")
        test_6, results_6 = test_func("petal_width ~ 1 + petal_length + sepal_length")
        test_7, results_7 = test_func("petal_width ~ 1 + sepal_length + sepal_width")
        test_8, results_8 = test_func("petal_width ~ 1 + petal_length + sepal_length + sepal_wid
        results = [
            results_1,
            results_2,
            results_3,
            results_4,
            results_5,
            results_6,
            results_7,
            results_8
        ]
```

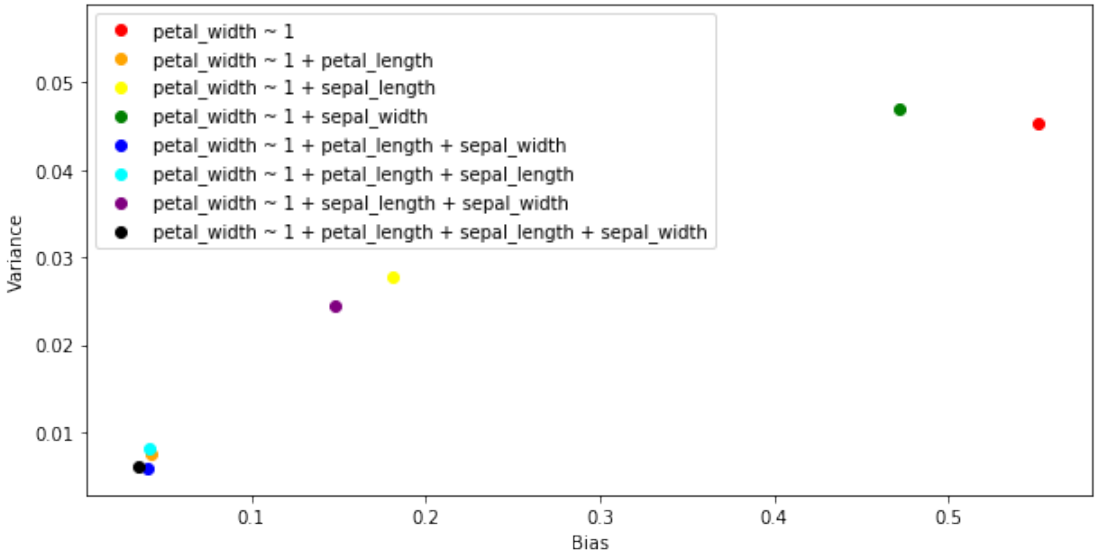
```
In [7]: results = pd.DataFrame(results)
        results['color'] = ['red','orange', 'yellow','green','blue','cyan','purple' ,'black']
        results
```

```
Out[7]: bias                                        description  variance  \
        0  0.551216                                   petal_width ~ 1  0.045366
        1  0.042561                petal_width ~ 1 + petal_length  0.007566
```

```
2  0.181284                    petal_width ~ 1 + sepal_length  0.027866
3  0.471672                     petal_width ~ 1 + sepal_width  0.046972
4  0.040000       petal_width ~ 1 + petal_length + sepal_width  0.006017
5  0.041027      petal_width ~ 1 + petal_length + sepal_length  0.008187
6  0.147898       petal_width ~ 1 + sepal_length + sepal_width  0.024406
7  0.034831  petal_width ~ 1 + petal_length + sepal_length ...  0.006047

    color
0     red
1  orange
2  yellow
3   green
4    blue
5    cyan
6  purple
7   black
```

```
In [9]: plt.figure(figsize=(10,5))
        for i in results.index:
            plt.scatter(results.loc[i].bias, results.loc[i].variance, c=results.loc[i].color, la
        plt.xlabel('Bias')
        plt.ylabel('Variance')
        plt.legend()
```

```
Out[9]: <matplotlib.legend.Legend at 0x7f5b8f678cc0>
```



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.

2.6 Probabilistic Model Selection

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

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

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

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

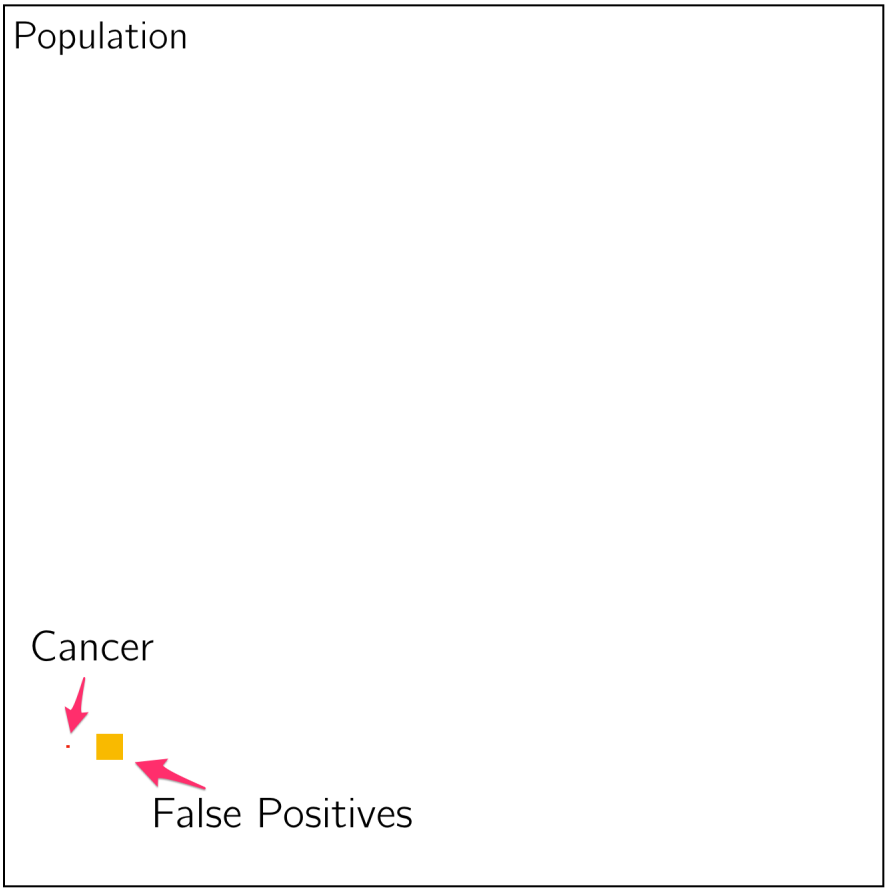


Fig. 1: A population of 1000 patients

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

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

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

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

2.6.1 Generating Model Hypotheses

For this particular task we will consider the hypothesis space to be all of the linear models possible given our data sets. If we recognize that the set of all of the models possible with `dataset_1` is a subset of all of the models possible with `dataset_2` and likewise with `dataset_3` and `dataset_4`, then we can take the size of this hypothesis space to be all of the possible models made using `dataset_2` and all of the possible models made using `dataset_4`.

This is a vast space.

To get a sense of the size of this hypothesis space, let us consider a simpler data set, one that has just two features, x_1 and x_2. We note that there are four possible models using these two features:

$$\begin{aligned}
\widehat{f_1} &= \beta_0 \\
\widehat{f_2} &= \beta_0 + \beta_1 x_1 \\
\widehat{f_3} &= \beta_0 + \beta_1 x_2 \\
\widehat{f_4} &= \beta_0 + \beta_1 x_1 + \beta_2 x_2
\end{aligned}$$

Similarly for a data set with three features, there will be eight possible models:

$$\begin{aligned}
\widehat{f_1} &= \beta_0 \\
\widehat{f_2} &= \beta_0 + \beta_1 x_1 \\
\widehat{f_3} &= \beta_0 + \beta_1 x_2 \\
\widehat{f_4} &= \beta_0 + \beta_1 x_3 \\
\widehat{f_5} &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 \\
\widehat{f_6} &= \beta_0 + \beta_1 x_1 + \beta_2 x_3 \\
\widehat{f_7} &= \beta_0 + \beta_1 x_2 + \beta_2 x_3 \\
\widehat{f_8} &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3
\end{aligned}$$

Essentially we are forming what is known as a Power Set of possible feature combinations. The number of elements in a Power Set is given by 2^p where p is the number of elements in the set. For our current dataset we have 390 features. This is a hypothesis space With a dimension of 2.5×10^{117}. If we trained one model per second, it would take us 8×10^{109} years to search the entire hypothesis space. For perspective Physicists estimate that the universe is approximately 13.8×10^9 years old.

In other words, we are not going to be able to exhaustively search this hypothesis space.

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

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

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

```
In [3]: iris.glm = glm("label ~ 1 + sepal_length + sepal_width + petal_length + petal_width", da
        summary(iris.glm)


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

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

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

(Dispersion parameter for gaussian family taken to be 0.04798457)

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

Number of Fisher Scoring iterations: 2
```

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.

```
In [4]: logLik(iris.glm)
```

```
'log Lik.' 17.46751 (df=6)
```

“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

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

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

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

```
In [5]: BIC(iris.glm)
```

-4.87121487462612

```
In [6]: 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
```

```
'log Lik.' -4.871215 (df=6)
```

```
In [7]: 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)
        }
```

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

```
'log Lik.' -4.871215 (df=6)
```

2.8 Model Selection

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

```
In [9]: 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"
```

```
In [10]: 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)
```

```
In [11]: 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 )))
```

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

```
In [12]: 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 )))
```

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

```
In [13]: 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"
```

```
In [14]: 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)
```

```
In [15]: 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 )))
```

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

2.9 Cluster Modeling

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

```
In [1]: 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
        from sklearn.datasets import load_iris
```

```
In [2]: 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.

```
In [3]: iris_df.head()
```

```
Out[3]: sepal_length  sepal_width  petal_length  petal_width  label
        0           5.1          3.5           1.4          0.2      0
        1           4.9          3.0           1.4          0.2      0
        2           4.7          3.2           1.3          0.2      0
        3           4.6          3.1           1.5          0.2      0
        4           5.0          3.6           1.4          0.2      0
```

```
In [4]: IRIS = load_iris()
        iris_df['target'] = [IRIS.target_names[t] for t in IRIS.target]
```

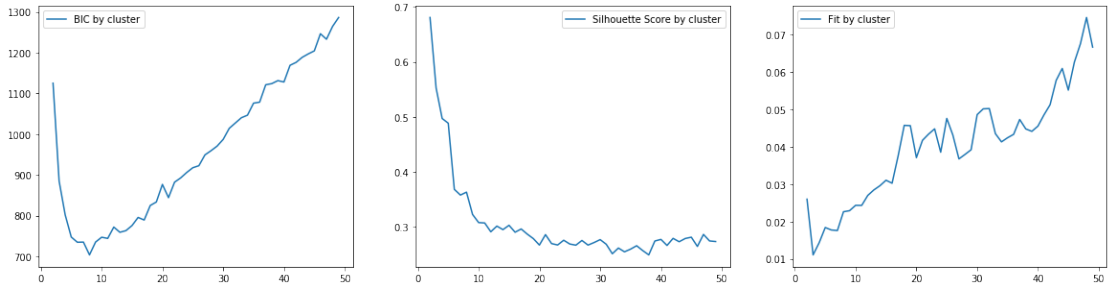
```
In [5]: def fit_and_time(model, data):
            start = time()
            model = model.fit(data)
            end = time() - start
            return {'fit_time' : end, 'model' : model}

        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
```

```
In [6]: ks = range(2, 50)

        kmeans_models = []

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

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

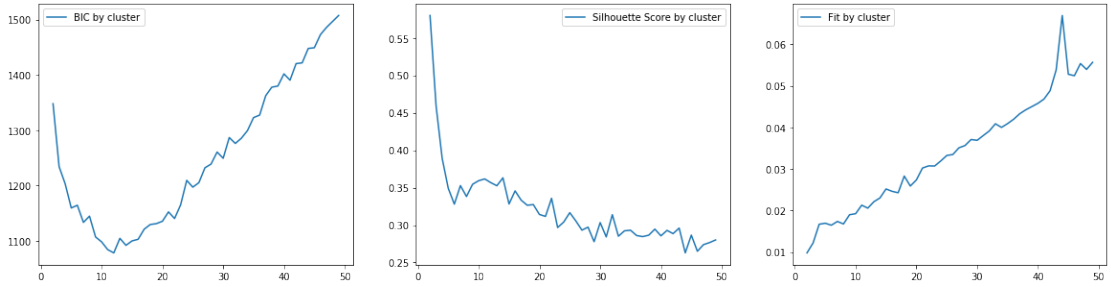
```
In [7]: import matplotlib.pyplot as plt
```

```
In [8]: _, 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 c
        ax[1].legend()
        ax[2].plot(kmeans_models_df.index, kmeans_models_df.fit_time, label='Fit by cluster')
        ax[2].legend()
```

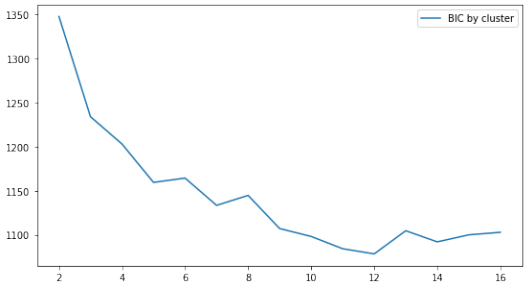
```
Out[8]: <matplotlib.legend.Legend at 0x7fc86c4845f8>
```

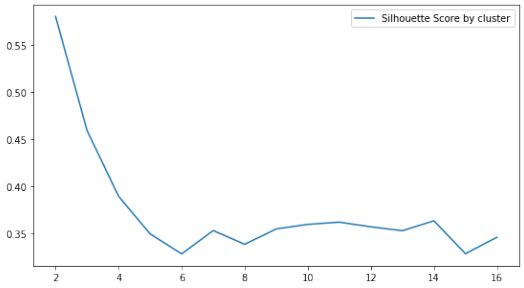



```
In [9]: X_sc = (X - X.mean())/X.std()

In [10]: ks = range(2, 50)

         kmeans_sc_models = []

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

         for k in 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)

In [11]: _, 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 Scc
         ax[1].legend()
         ax[2].plot(kmeans_sc_models_df.index, kmeans_sc_models_df.fit_time, label='Fit by clust
         ax[2].legend()

Out[11]: <matplotlib.legend.Legend at 0x7fc86c3617b8>
```




```
In [12]: _, 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
         ax[0].legend()
         ax[1].plot(kmeans_sc_models_df.index[:15], kmeans_sc_models_df.sil_sc[:15], label='Silh
         ax[1].legend()

Out[12]: <matplotlib.legend.Legend at 0x7fc86c2c09b0>
```



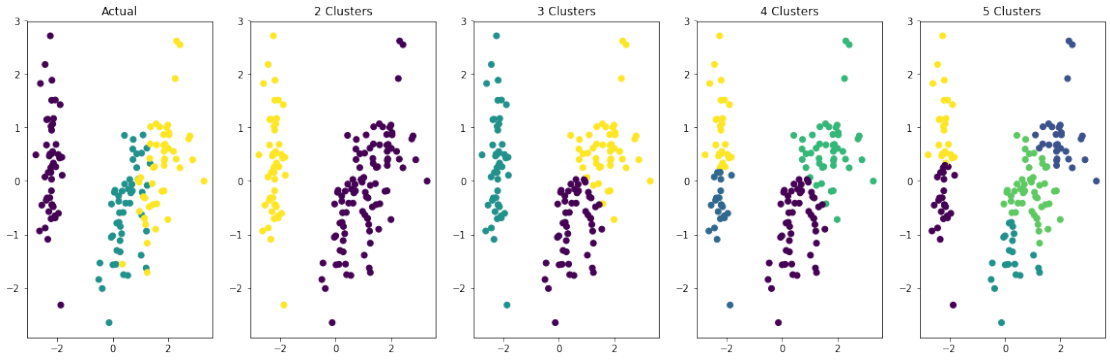



```
In [13]: kmeans_sc_models_df.model.values[0]
```

```
Out[13]: KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
             n_clusters=2, n_init=10, n_jobs=1, precompute_distances='auto',
             random_state=None, tol=0.0001, verbose=0)
```

```
In [14]: 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]
```

```
In [15]: 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

THREE

THE TITANIC DATASET

3.1 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.

3.2 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.

3.2.1 define accuracy metric

```
In [1]: 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:

3.2.2 a simple vector of actual values

```
In [2]: actual = c(1,1,0,0,1)
```

Our model might generate the following vector of predicted values:

3.2.3 a simple vector of predictions

```
In [3]: 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.

3.2.4 assess accuracy of predictions

```
In [4]: accuracy(actual, predicted)
```

0.6

3.3 Preliminary Analysis

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

3.3.1 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()`

```
In [1]: titanic <- read.csv('titanic.csv')
```

```
In [2]: 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

```
In [3]: dim(titanic)
```

1. 891 2. 12

display the structure of the dataframe

```
In [4]: 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 41
 $ 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 ...
```

3.3.2 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`.

3.3.3 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

```
In [5]: class(titanic$Embarked)
```

'factor'

display that levels of the `titanic$embarked` column

```
In [6]: levels(titanic$Embarked)
```

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

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

```
In [7]: titanic$Embarked[1:5]
```

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

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

3.3.4 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

```
In [8]: number_of_passengers = length(titanic$PassengerId)
        number_of_passengers
```

891

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

```
In [9]: length(unique(titanic$PassengerId)); length(unique(titanic$Name))
```

891891 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

```
In [10]: titanic$PassengerId <- NULL
         titanic$Name <- NULL
```

```
In [11]: stopifnot(is.null(titanic$PassengerID))
         stopifnot(is.null(titanic$Name))
         stopifnot(as.vector(titanic[4,]) == c('1','1','female', '35', '1','0','113803','53.1','
```

3.3.5 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.

```
In [12]: 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
```

3.4 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.

3.4.1 display a contingency table of `titanic$survived`

```
In [1]: titanic <- read.csv('titanic.csv')
```

```
In [2]: 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 $0.6\bar{1}$ 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.

3.4.2 display a proportion table of `titanic$survived`

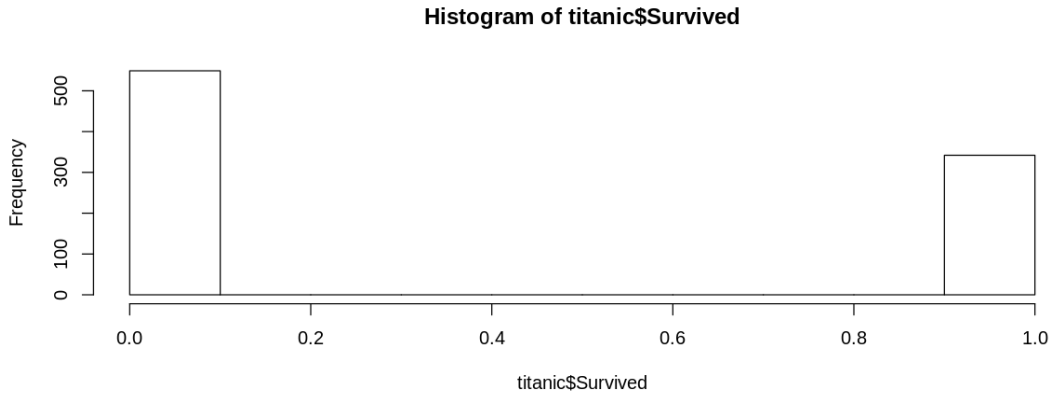
```
In [3]: prop.table(table(titanic$Survived))
```

```
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.

3.4.3 display a histogram of `titanic$Survived`

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

        hist(titanic$Survived)
```



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.

3.4.4 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.

```
In [5]: number_of_passengers = length(titanic$Survived)
```

```
In [6]: no_survivors <- rep(0, number_of_passengers)
```

```
In [7]: stopifnot(no_survivors == rep(0, length(titanic$Survived)))
```

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.

3.4.5 accuracy of our naïve prediction

```
In [8]: source('metrics.r')
```

```
In [9]: accuracy(titanic$Survived, no_survivors)
```

0.616161616161616 As expected, we achieve an accuracy of $0.6\bar{1}$.

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

3.4.6 a first attempt at `fizzbuzz`

```
In [10]: 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

3.4.7 start the `solution` vector

```
In [11]: 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).

3.4.8 create the mod 15 mask

```
In [12]: 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`.

3.4.9 filter `solution` using the mind 15 mask

```
In [13]: solution[mod15_mask]
```

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

3.4.10 assign valued to the filtered `solution` vector

```
In [14]: solution[mod15_mask] = "fizzbuzz"
```

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

```
In [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. 'fizzbuzz' We can repeat this technique to build an entire solution to the problem.

3.4.11 a vectorized `fizzbuzz`

```
In [16]: 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.

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

```
In [1]: source('init.r')
```

3.5.1 Randomized Model Improvement

create a random mask

```
In [2]: random_mask = sample(c(TRUE,FALSE), number_of_passengers, replace = TRUE)
        random_mask[1:10]
```

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

duplicate and filter to create random model

```
In [3]: random_model = rep(no_survivors)
        random_model[random_mask] = 1
```

assess accuracy of random model

```
In [4]: accuracy(titanic$Survived, random_model)
```

0.503928170594837 As suspected, simply guessing is not better than guessing all zeros. It looks like we might actually justify our exorbitant salaries after all.

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

```
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 41
 $ 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`

```
In [6]: 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

```
In [7]: 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

```
In [8]: 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`

```
In [9]: 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

```
In [10]: 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

```
In [11]: 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.

3.5.2 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

```
In [12]: 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`

```
In [13]: women_survived = rep(no_survivors)
         women_survived[women_mask] = 1
```

assess accuracy of model, `women_survived`

```
In [14]: 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`:

3.5.3 Can Another Feature Help?

```
In [15]: 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

```
In [16]: 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`

```
In [17]: 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`

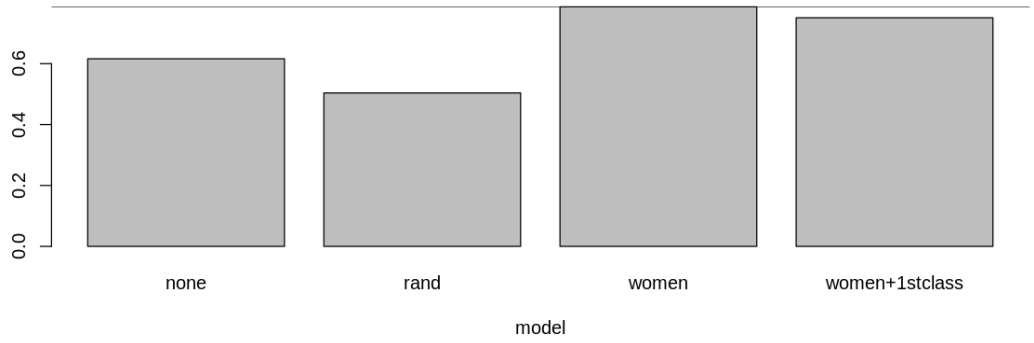
```
In [18]: accuracy(titanic$Survived, women_and_first_class_survived)
```

0.750841750841751

```
In [19]: 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))
```

Progress Report

```
In [20]: barplot(scores, xlab = 'model',
                 names.arg = c('none','rand','women', 'women+1stclass'))
         abline(h = max(scores))
```



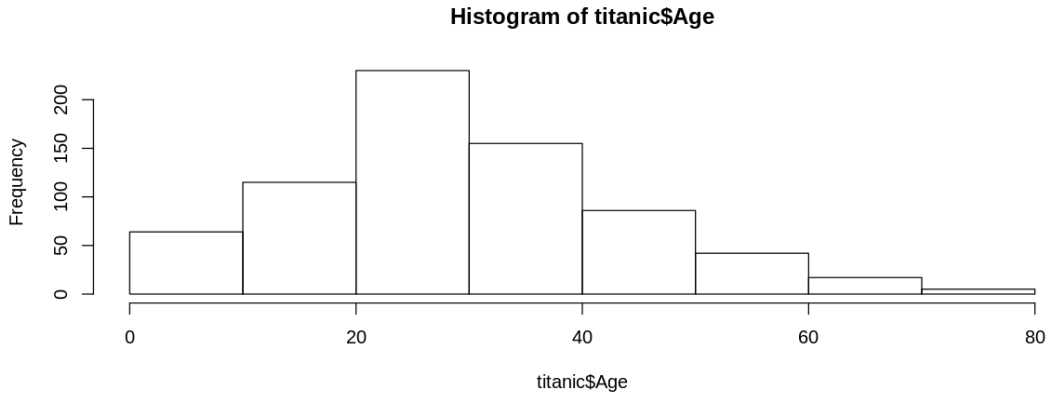

```
In [1]: source('init.r')
```

3.6 Numerical Features as Categorical Features

Age is a numerical feature.

Age has missing values.

```
In [2]: hist(titanic$Age)
```



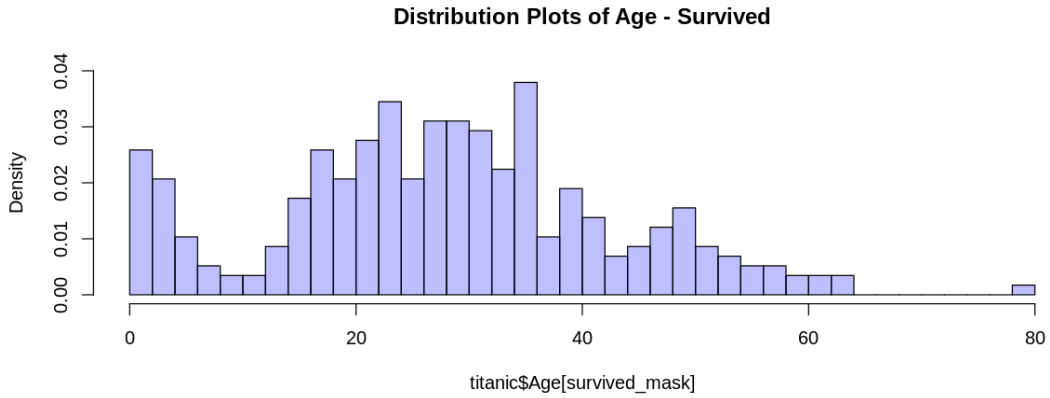

```
In [3]: missing_age_values_mask = is.na(titanic$Age)
```

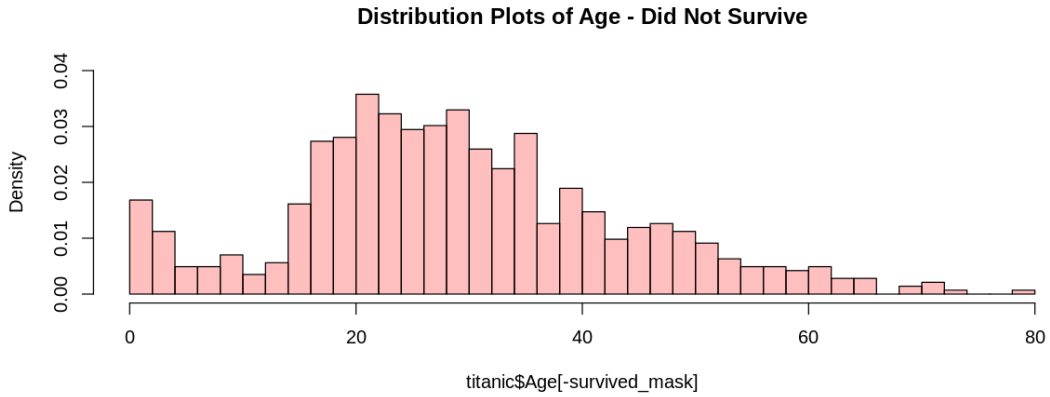
```
In [4]: head(titanic$Survived[missing_age_values_mask])
```

1. 0 2. 1 3. 1 4. 0 5. 1 6. 0

```
In [5]: survived_mask = as.logical(titanic$Survived)
```

```
In [6]: 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')
```





3.6.1 create a mask of just children

```
In [7]: children_mask = titanic$Age < 10
```

3.6.2 duplicate and filter to create a model, `women_survived`

```
In [8]: women_and_children_survived = rep(women_survived)
        women_and_children_survived[children_mask] = 1
```

3.6.3 assess accuracy of model, `women_survived`

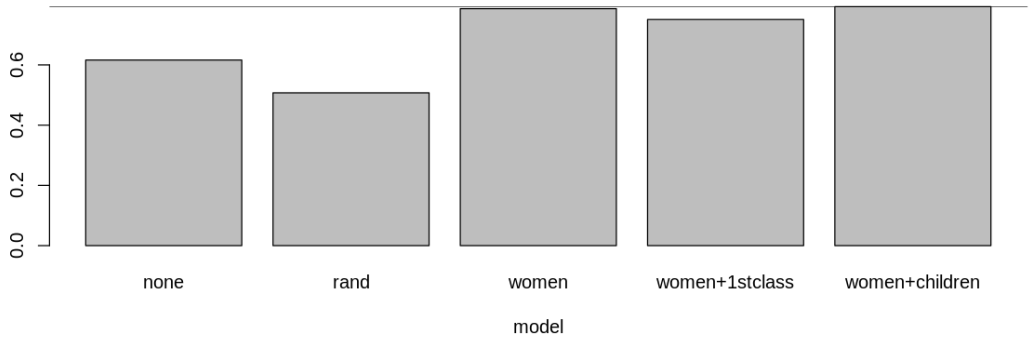
```
In [9]: accuracy(titanic$Survived, women_and_children_survived)
```

0.793490460157127

```
In [10]: 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))
```

3.6.4 Progress Report

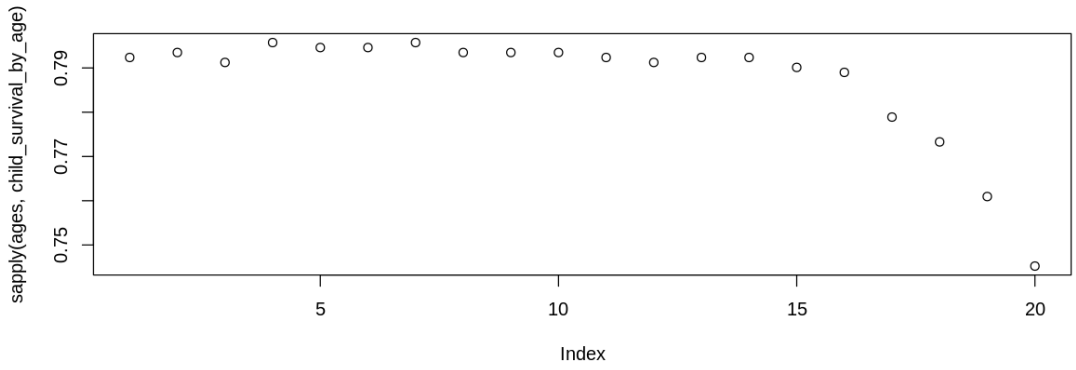
```
In [11]: barplot(scores, xlab = 'model',
                 names.arg = c('none','rand','women', 'women+1stclass', 'women+children'))
         abline(h = max(scores))
```




```
In [12]: 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))
         }
```

```
In [13]: ages = 1:20
```

```
In [14]: plot(sapply(ages, child_survival_by_age))
```



3.6.5 create a mask of just children

```
In [15]: children_mask = titanic$Age < 7
```

3.6.6 duplicate and filter to create a model, `women_survived`

```
In [16]: women_and_children_survived = rep(women_survived)
         women_and_children_survived[children_mask] = 1
```

3.6.7 assess accuracy of model, `women_survived`

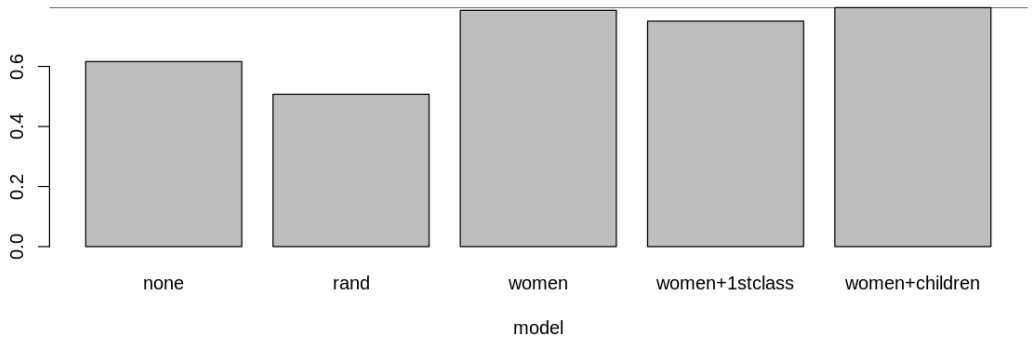
```
In [17]: accuracy(titanic$Survived, women_and_children_survived)
```

0.795735129068462

```
In [18]: 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))
```

3.6.8 Progress Report

```
In [19]: barplot(scores, xlab = 'model',
                 names.arg = c('none','rand','women', 'women+1stclass', 'women+children'))
         abline(h = max(scores))
```



CHAPTER

FOUR

THE WHOLESALE CUSTOMER DATASET

4.1 The Wholesale Customer Dataset

Domain This is a canonical dataset taken from the UCI Machine Learning Repository.

It is data collected from the sale of products by a grocery wholesaler in Portugal. The values are in "monetary units".

Problem While this task includes labels, it is typically used for clustering or unsupervised learning work. Here we will keep the labels for comparison but will be exploring this dataset in the context of EDA and unsupervised learning in an EDA setting.

Two columns will work as label for this set `region` and `channel`.

Solution We will be working toward a customer segmentation generated by a cluster analysis.

Data The following analysis shows:

- there are 440 rows and 8 useful variable columns in the dataset. Two of these columns are target features `Region` and `Channel`.
- there are six integer value columns:
- `Fresh`
- `Milk`
- `Grocery`
- `Frozen`
- `Detergents_Paper`
- `Delicatessen`

Benchmark As this is an EDA and Unsupervised Learning task, we will not define an explicit benchmark.

Metrics We will not define a metric for this project at this time.

4.2 Moments

> In mathematics, a moment is a specific quantitative measure, used in both mechanics and statistics, of the shape of a set of points.

https://en.wikipedia.org/wiki/Moment_(mathematics)

If the points represent mass: - the zeroth moment is the total mass - the first moment divided by the total mass is the center of mass - the second moment is the rotational inertia

If the points represent probability density:

- the zeroth moment is the total probability (i.e. one)
- the first moment is the mean
- the second central moment is the variance
- the third central moment is the skewness
- the fourth central moment (with normalization and shift) is the kurtosis.

4.2.1 Continuous Probability

The k-th moment of a real-valued continuous function, a **probability density function**, $f(x)$ of a real variable about a value c is

$$\mu_k = \int_{-\infty}^{\infty} (x - c)^k \, f(x) \, \mathrm{d}x$$

4.2.2 Discrete Probabilty

The k-th moment of a real-valued discrete function, a **probability mass function**, $p(x)$ of a real variable about a value c is

$$\mu_k = \sum (x - c)^k p(x)$$

Simple Case: the Mean

Consider the first moment for a discrete valued probabilty where each value has an equal chance of being observed.

Then, for a list of n values

$$\mathcal{D} = \{x_1, \ldots, x_n\}$$

each with probabilty $p(x_k) = \frac{1}{n}$

Then

$$\mu_1 = \sum (x)^1 p(x) = \sum x \frac{1}{n} = \frac{1}{n} \sum x$$

This is the well-known mean you are used to. We typically refer to it simply as μ

$$\mu = \frac{1}{n} \sum x$$

We also call this an **Expected Value**, $\mathbb{E}[x]$, and for the common case

$$\mathrm{E}[x] = \mu$$

In python:

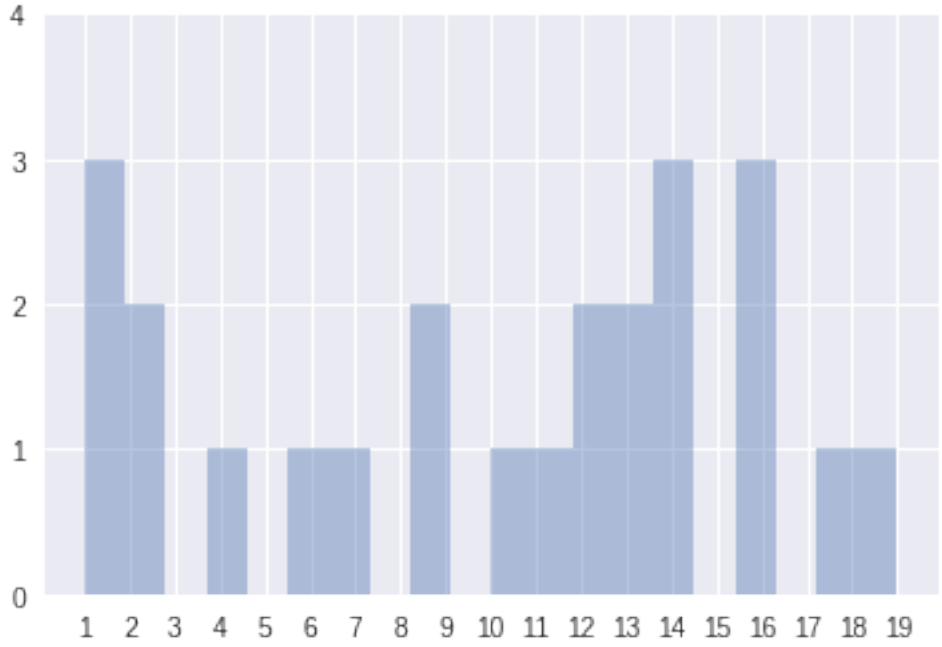
```
In [1]: import numpy as np
        import seaborn as sns
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [2]: D = np.array([2, 11,  7, 19, 14,  9,  9,  4, 16, 12, 16, 18, 13, 16, 10, 13,  1, 6, 12,
        n = len(D)
        n
```

```
Out[2]: 24
```

```
In [3]: sns.distplot(D, bins=20, kde=False)
        plt.xticks(list(range(1,20)))
        plt.yticks(list(range(5)));
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```




```
In [4]: mu = 1/(n)*np.sum(D)
        mu
```

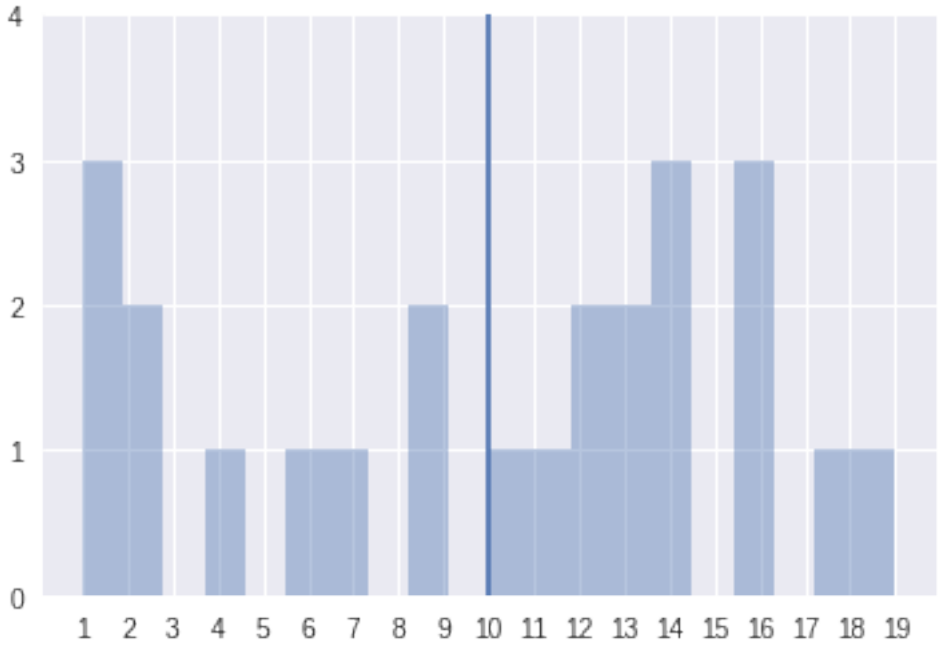
```
Out[4]: 10.0
```

```
In [5]: D.mean()
```

```
Out[5]: 10.0
```

```
In [6]: sns.distplot(D, bins=20, kde=False)
        plt.axvline(D.mean())
        plt.xticks(list(range(1,20)))
        plt.yticks(list(range(5)));
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



Simple Case: the Variance

Consider the second **central** moment for a discrete valued probabilty where each value has an equal chance of being observed. We call it **central** because we will center this value around the mean.

Then

$$\mu_2 = \sum (x-\mu)^2 p(x) = \frac{1}{n}\sum (x-\mu)^2$$

This is just the expected value of $(x-\mu)^2$. We call this the **variance**, denoted σ^2.

$$\sigma^2 = \mathbb{E}\left[(x-\mu)^2\right]$$

In python:

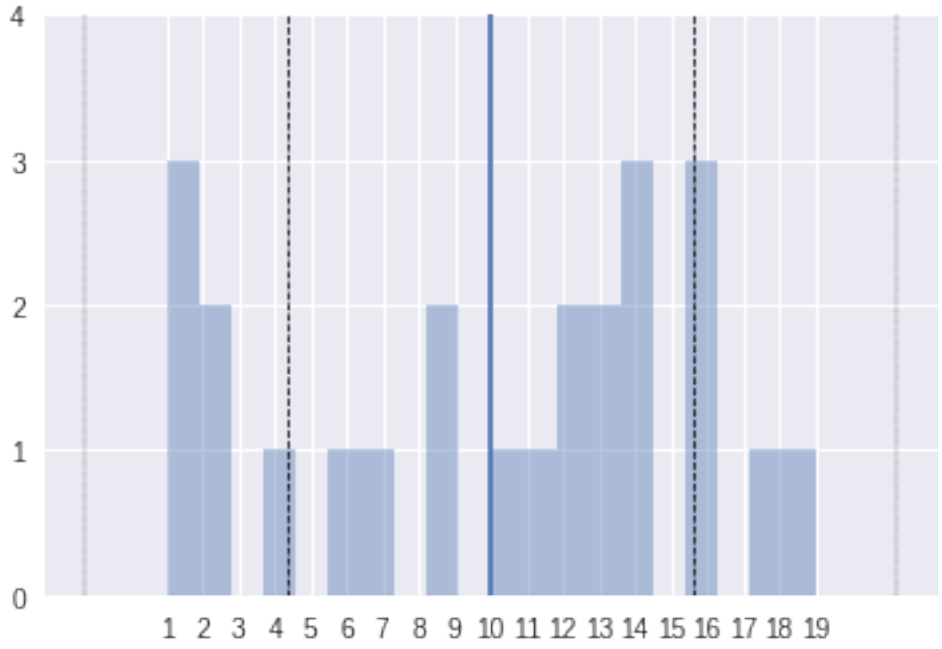
```
In [7]: var = 1/n*np.sum((D-mu)**2)
        var

Out[7]: 31.75

In [8]: D.var()

Out[8]: 31.75
```

Note that the square root of the variance is the standard deviation.

```
In [9]: np.sqrt(var)

Out[9]: 5.634713834792322

In [10]: D.std()

Out[10]: 5.634713834792322

In [11]: np.sqrt((3*(-2.2)**2+(0.8)**2+(2.8)**2)/5)

Out[11]: 2.1447610589527217
```

```
In [12]: sns.distplot(D, bins=20, kde=False)
         plt.axvline(D.mean()+D.std(), color='black', lw=.75, ls="dashed")
         plt.axvline(D.mean()-D.std(), color='black', lw=.75, ls="dashed")
         plt.axvline(D.mean()+2*D.std(), color='black', lw=.25, ls="dashed")
         plt.axvline(D.mean()-2*D.std(), color='black', lw=.25, ls="dashed")
         plt.axvline(D.mean())
         plt.xticks(list(range(1,20)))
         plt.yticks(list(range(5)));
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



Advanced Cases: Skew and Kurtosis

4.2.3 Skewness

In probability theory and statistics, skewness is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean.

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

Skewness is the third **standardized** moment.

$$\gamma = \mathrm{E}\left[\left(\frac{X - \mu}{\sigma}\right)^3\right]$$

In python:

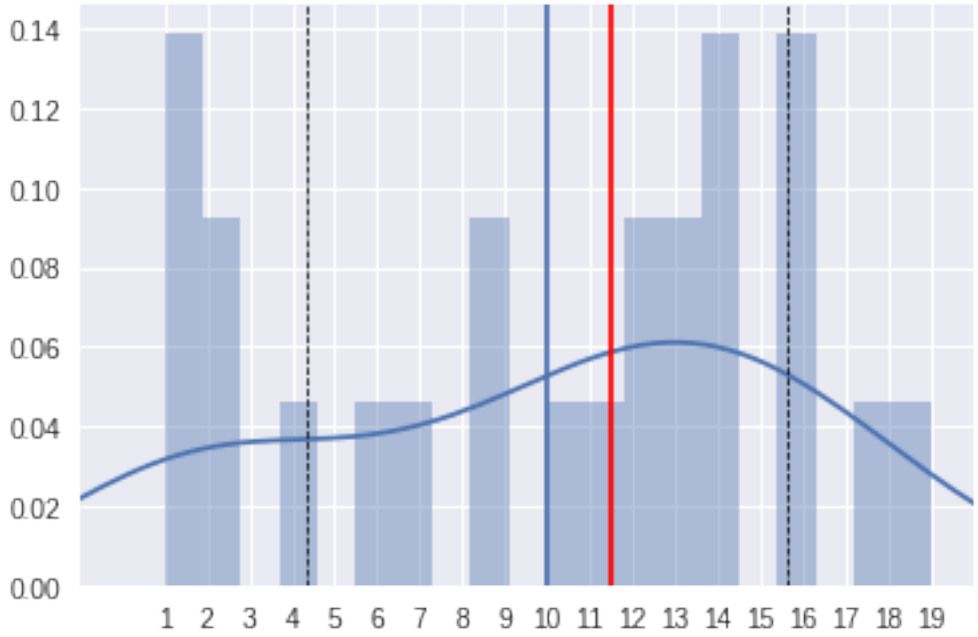
```
In [13]: skew = 1/n*np.sum(((D-mu)/np.sqrt(var))**3)
         skew
```

```
Out[13]: -0.31860989629428982
```

```
In [14]: import scipy.stats as st
```

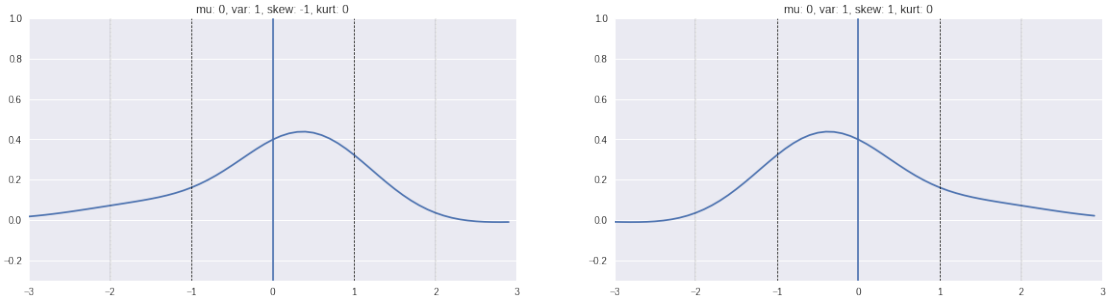
```
In [15]: st.skew(D)
```

```
Out[15]: -0.31860989629428965

In [16]: np.median(D)

Out[16]: 11.5

In [17]: sns.distplot(D, bins=20, kde=True)
         plt.axvline(D.mean()+D.std(), color='black', lw=.75, ls="dashed")
         plt.axvline(D.mean()-D.std(), color='black', lw=.75, ls="dashed")
         plt.axvline(D.mean()+2*D.std(), color='black', lw=.25, ls="dashed")
         plt.axvline(D.mean()-2*D.std(), color='black', lw=.25, ls="dashed")
         plt.axvline(D.mean())
         plt.axvline(np.median(D), color="red")
         plt.xlim(-1,20)
         plt.xticks(list(range(1,20)));

/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```




```
In [18]: from statsmodels.sandbox.distributions.extras import pdf_mvsk

In [19]: _, ax = plt.subplots(1,2,figsize=(20,5))

         mvsk = [0,1,-1,0]
         pdffunc = pdf_mvsk(mvsk)
         rng = np.arange(-3, 3, 0.1)
         ax[0].plot(rng, pdffunc(rng))
         ax[0].axvline(1, color='black', lw=.75, ls="dashed")
         ax[0].axvline(-1, color='black', lw=.75, ls="dashed")
         ax[0].axvline(2, color='black', lw=.25, ls="dashed")
         ax[0].axvline(-2, color='black', lw=.25, ls="dashed")
         ax[0].axvline(0)
         ax[0].set_xlim(-3,3)
         ax[0].set_ylim(-0.3, 1)
         ax[0].set_title("mu: {}, var: {}, skew: {}, kurt: {}".format(*mvsk))

         mvsk = [0,1,1,0]
         pdffunc = pdf_mvsk(mvsk)
```

```
ax[1].plot(rng, pdffunc(rng))
ax[1].axvline(1, color='black', lw=.75, ls="dashed")
ax[1].axvline(-1, color='black', lw=.75, ls="dashed")
ax[1].axvline(2, color='black', lw=.25, ls="dashed")
ax[1].axvline(-2, color='black', lw=.25, ls="dashed")
ax[1].axvline(0)
ax[1].set_xlim(-3,3)
ax[1].set_ylim(-0.3, 1)
ax[1].set_title("mu: {}, var: {}, skew: {}, kurt: {}".format(*mvsk));
```



4.2.4 Kurtosis

In probability theory and statistics, kurtosis (from Greek: kurtos, meaning "curved, arching") is a measure of the "tailedness" of the probability distribution of a real-valued random variable. In a similar way to the concept of skewness, kurtosis is a descriptor of the shape of a probability distribution and, just as for skewness, there are different ways of quantifying it for a theoretical distribution and corresponding ways of estimating it from a sample from a population.

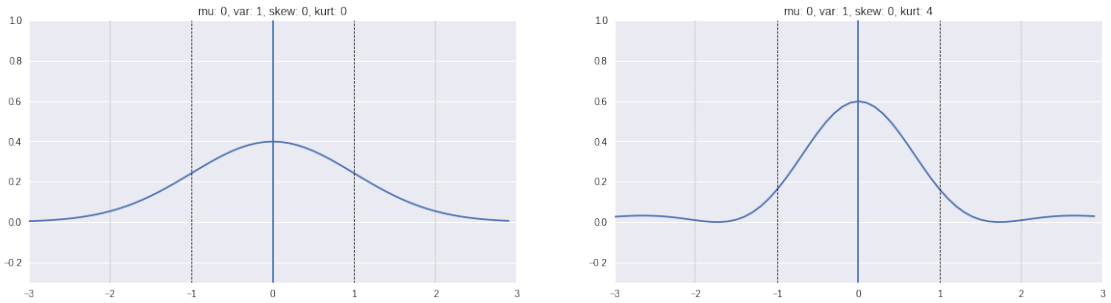
https://en.wikipedia.org/wiki/Kurtosis

Kurtosis is the fourth **standardized** moment.

$$\gamma = \mathrm{E}\left[\left(\frac{X - \mu}{\sigma}\right)^4\right]$$

```
In [20]: _, ax = plt.subplots(1,2,figsize=(20,5))

         mvsk = [0,1,0,0]
         pdffunc = pdf_mvsk(mvsk)
         rng = np.arange(-3, 3, 0.1)
         ax[0].plot(rng, pdffunc(rng))
         ax[0].axvline(1, color='black', lw=.75, ls="dashed")
         ax[0].axvline(-1, color='black', lw=.75, ls="dashed")
         ax[0].axvline(2, color='black', lw=.25, ls="dashed")
         ax[0].axvline(-2, color='black', lw=.25, ls="dashed")
         ax[0].axvline(0)
         ax[0].set_xlim(-3,3)
         ax[0].set_ylim(-0.3, 1)
         ax[0].set_title("mu: {}, var: {}, skew: {}, kurt: {}".format(*mvsk))

         mvsk = [0,1,0,4]
         pdffunc = pdf_mvsk(mvsk)
         ax[1].plot(rng, pdffunc(rng))
         ax[1].axvline(1, color='black', lw=.75, ls="dashed")
         ax[1].axvline(-1, color='black', lw=.75, ls="dashed")
```

```
ax[1].axvline(2, color='black', lw=.25, ls="dashed")
ax[1].axvline(-2, color='black', lw=.25, ls="dashed")
ax[1].axvline(0)
ax[1].set_xlim(-3,3)
ax[1].set_ylim(-0.3, 1)
ax[1].set_title("mu: {}, var: {}, skew: {}, kurt: {}".format(*mvsk));
```



In python:

```
In [21]: kurt = 1/n*np.sum(((D-mu)/np.sqrt(var))**4)
         kurt
```

```
Out[21]: 1.8607477214954438
```

Kurtosis is the average (or expected value) of the standardized data raised to the fourth power. Any standardized values that are less than 1 (i.e., data within one standard deviation of the mean, where the "peak" would be), contribute virtually nothing to kurtosis, since raising a number that is less than 1 to the fourth power makes it closer to zero. The only data values (observed or observable) that contribute to kurtosis in any meaningful way are those outside the region of the peak; i.e., the outliers. Therefore, kurtosis measures outliers only; it measures nothing about the "peak."

```
In [22]: st.kurtosis(D, fisher=False)
```

```
Out[22]: 1.860747721495443
```

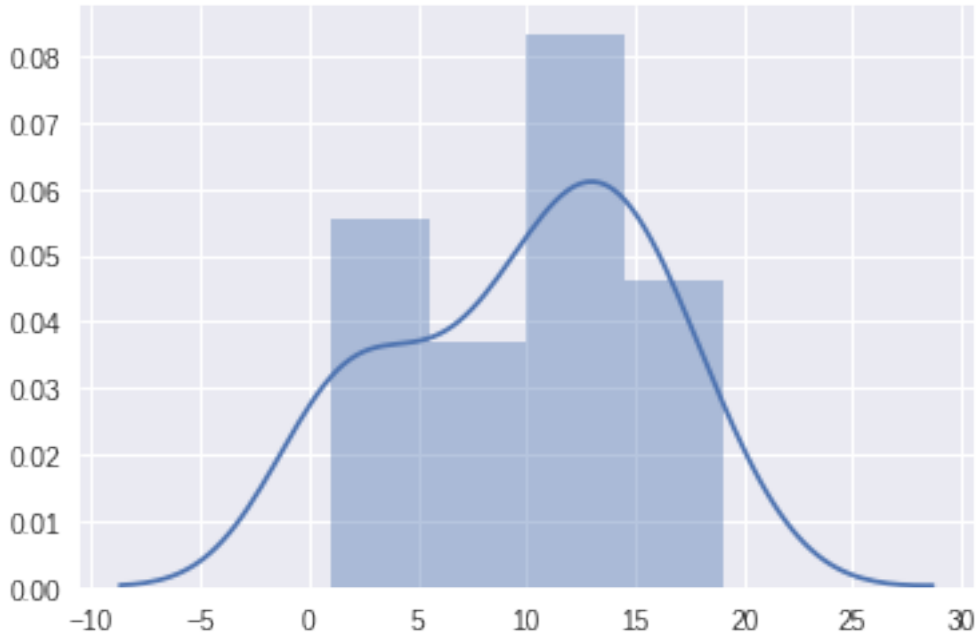
Describing a Distribution

Using these four values, we can describe the distribution of data.

```
In [23]: print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
         sns.distplot(D);
```

```
mu: 10.0, var: 31.75, skew: -0.31860989629428965, kurt: 1.860747721495443
```

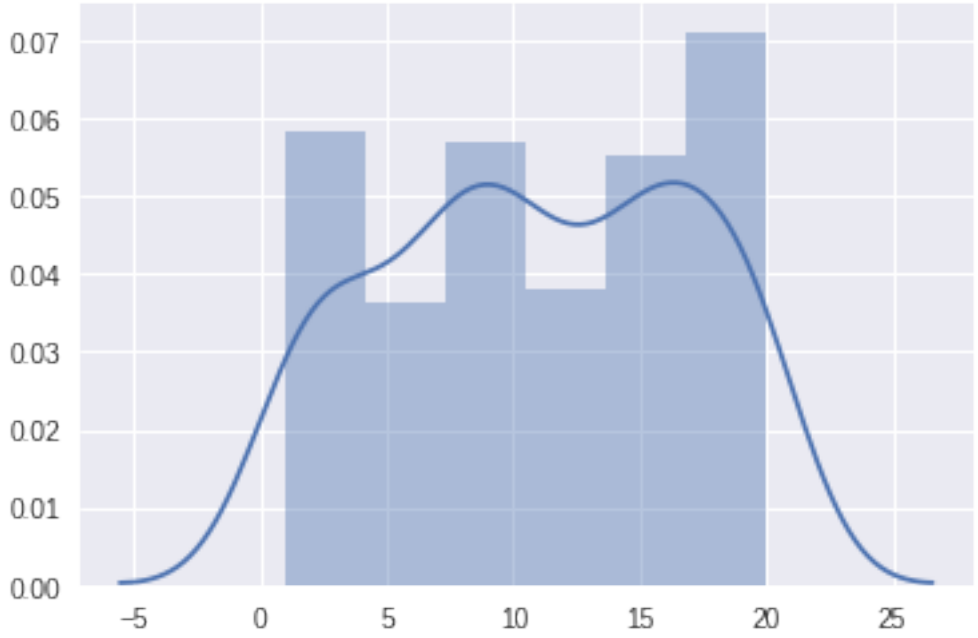
```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```




```
In [24]: n = 200
         D = np.random.randint(1,21,n)
         print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
         sns.distplot(D);
```

```
mu: 10.95, var: 35.4575, skew: -0.095678878462892, kurt: 1.805668100181942
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

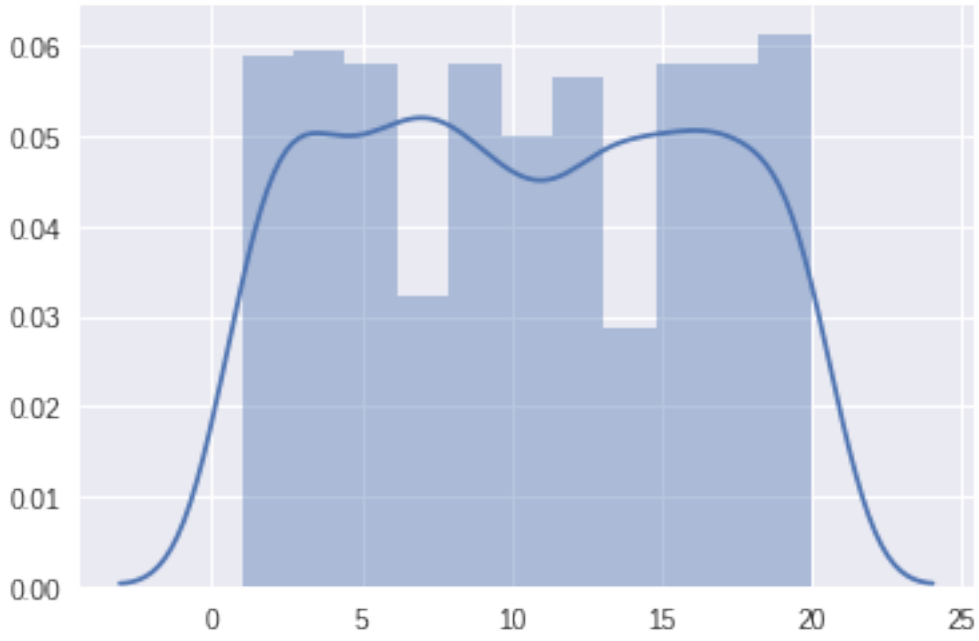



```
In [25]: n = 2000
         D = np.random.randint(1,21,n)
         print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
```

```
        sns.distplot(D);
```

```
mu: 10.479, var: 34.222559, skew: 0.01624997831549287, kurt: 1.7595753732515274
```

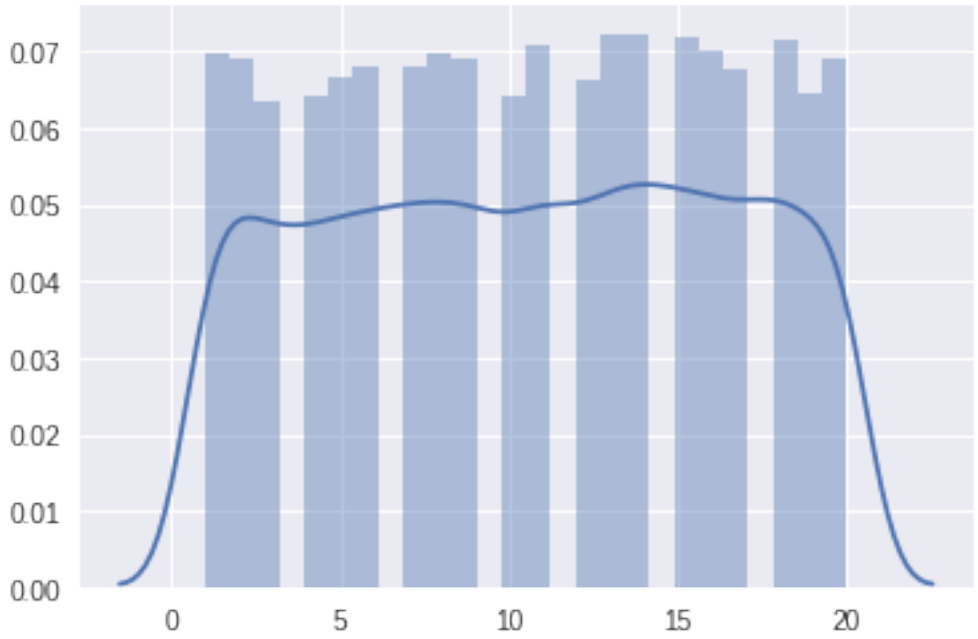
```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```




```
In [26]: n = 20000
         D = np.random.randint(1,21,n)
         print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
         sns.distplot(D);
```

```
mu: 10.5711, var: 33.079444790000004, skew: -0.028223514161604082, kurt: 1.8047816496038298
```

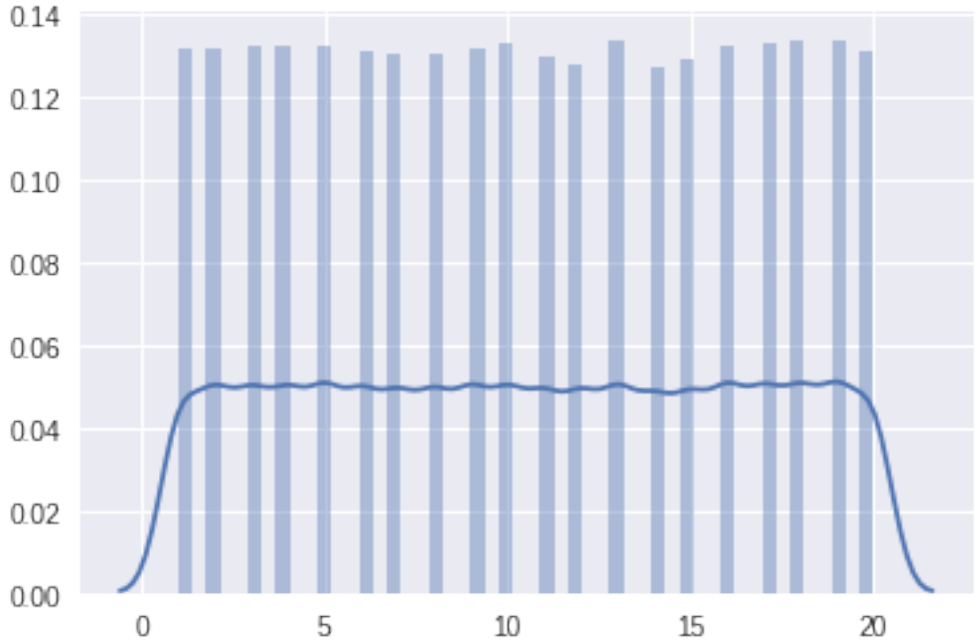
```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```




```
In [27]: n = 200000
         D = np.random.randint(1,21,n)
         print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
         sns.distplot(D);
```

```
mu: 10.505555, var: 33.40199914197501, skew: 0.000378249840623098, kurt: 1.787109241713362
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

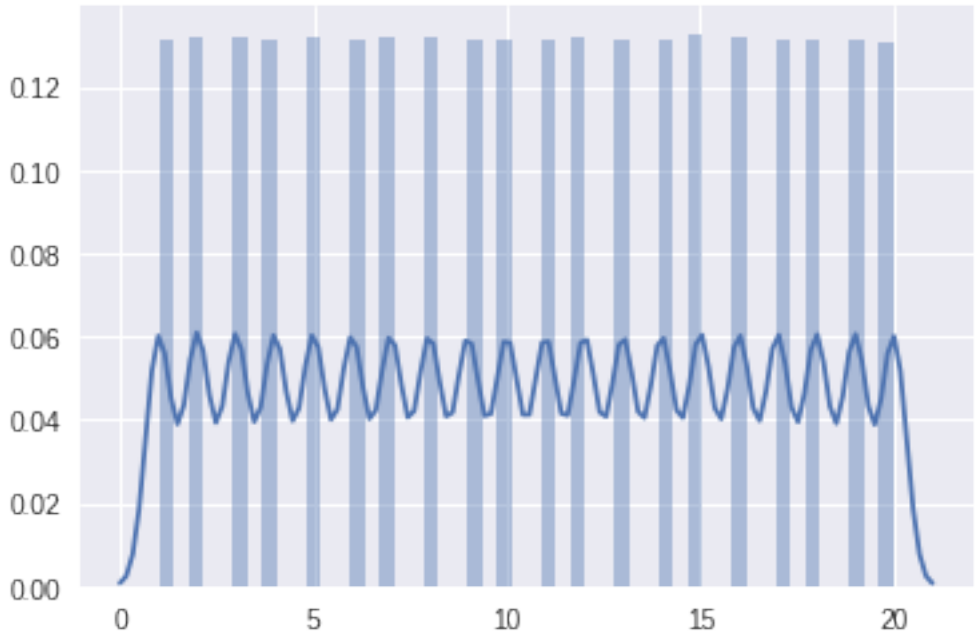



```
In [28]: n = 2000000
         D = np.random.randint(1,21,n)
         print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
```

```
        sns.distplot(D);
```

```
mu: 10.4979305, var: 33.21148071716976, skew: -0.00015445161897586403, kurt: 1.7944532070580905
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



`randint` draws from a uniform distribution

```
np.random.randint()

Return random integers from the "discrete uniform" distribution of
the specified dtype in the "half-open" interval [`low`, `high`). If
`high` is None (the default), then results are from [0, `low`).
```

4.2.5 What about `randn`?

```
In [29]: D
```

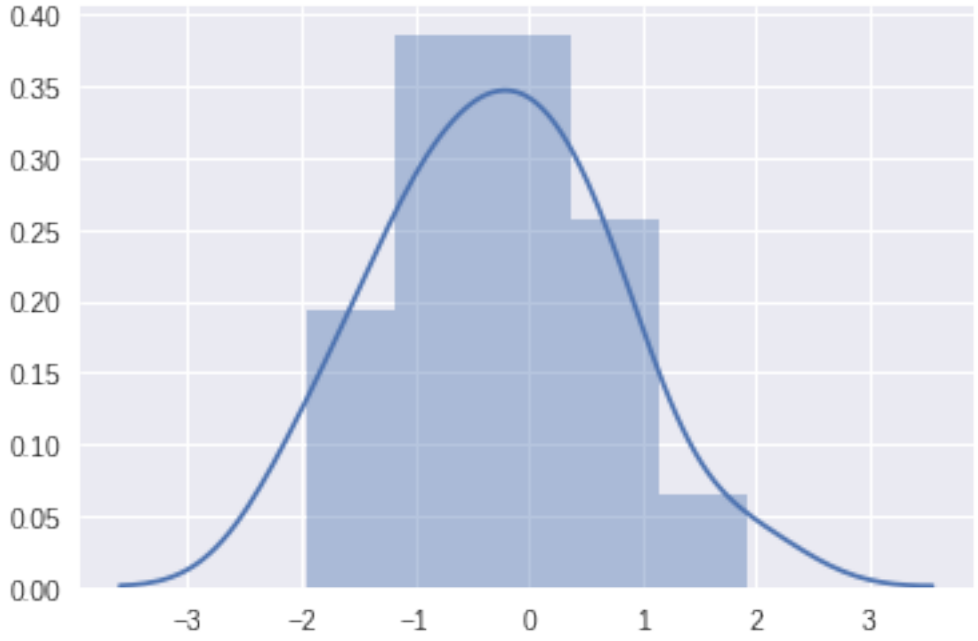
```
Out[29]: array([9, 8, 3, ..., 6, 2, 3])
```

```
In [30]: n = 20
         D = np.random.randn(n)
         print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
         sns.distplot(D)
```

```
mu: -0.28245330636521315, var: 0.9347880474961198, skew: 0.16365363902416516, kurt: 2.6550436086
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[30]: <matplotlib.axes._subplots.AxesSubplot at 0x7f31796311d0>
```

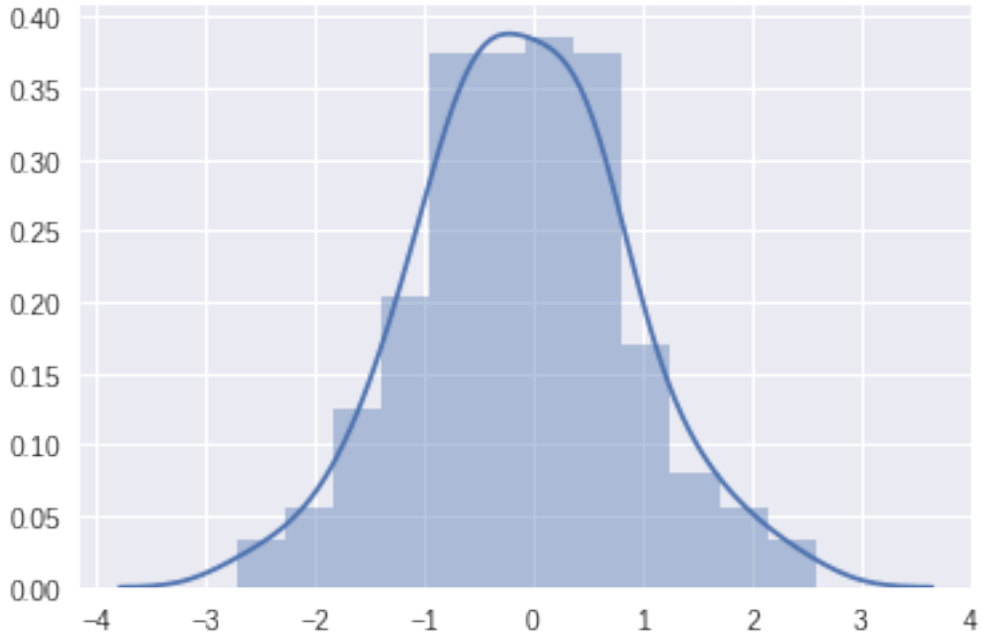



```
In [31]: n = 200
         D = np.random.randn(n)
         print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
         sns.distplot(D)
```

```
mu: -0.10670609324476409, var: 0.9358965680910097, skew: 0.053742654096183604, kurt: 3.111110304
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3179730e80>
```



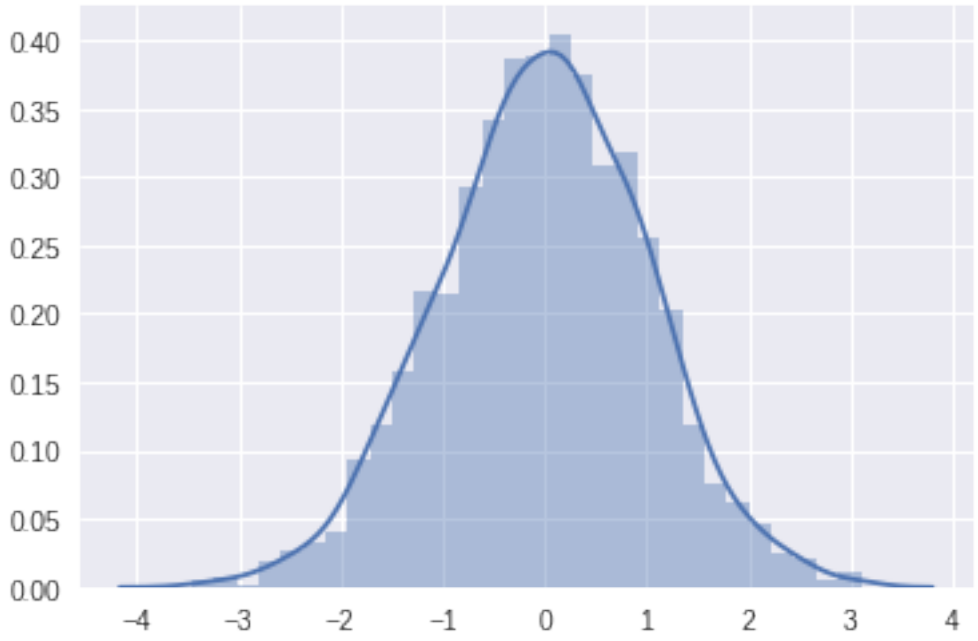

```
In [32]: n = 2000
         D = np.random.randn(n)
```

```
        print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
        sns.distplot(D)
```

```
mu: -0.015046940863861649, var: 1.024674407845896, skew: -0.07536331180038155, kurt: 3.036372040
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x7f31798a2400>
```

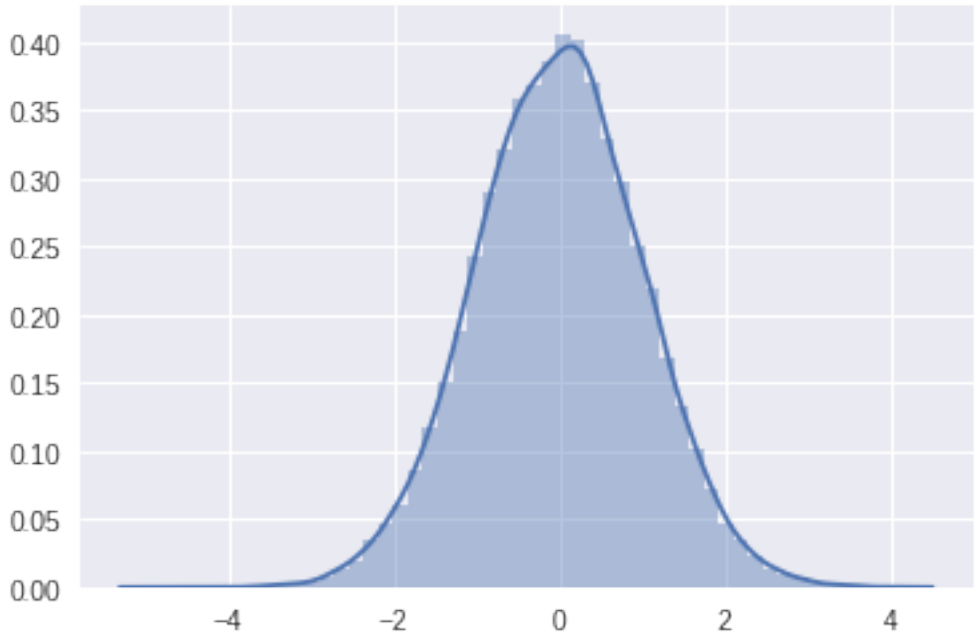



```
In [33]: n = 20000
         D = np.random.randn(n)
         print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
         sns.distplot(D)
```

```
mu: -0.007566473638051492 5, var: 0.9989193780229353, skew: -0.007125244530337679, kurt: 3.049333
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x7f317d1cdb00>
```

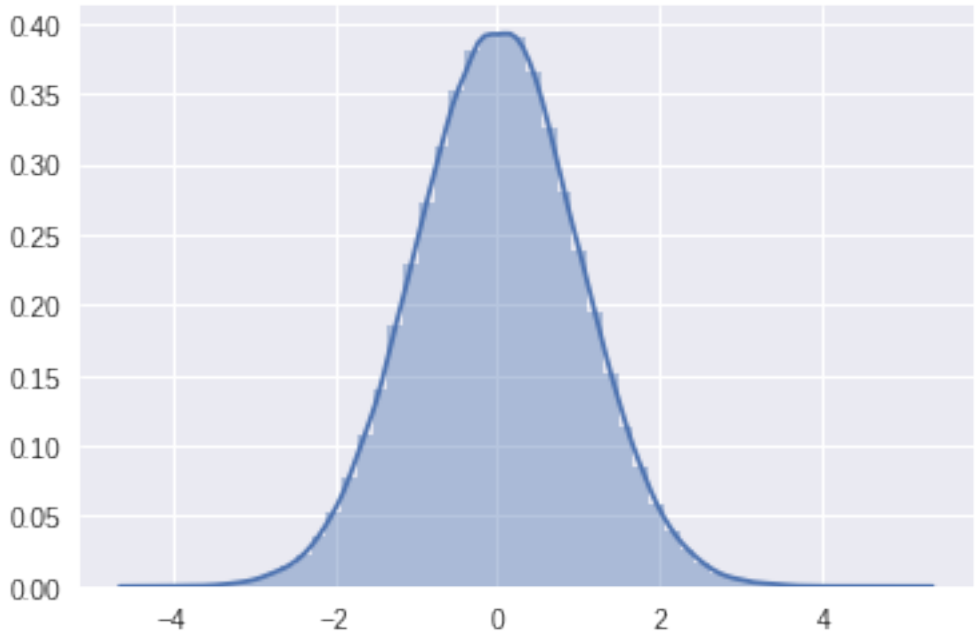



```
In [34]: n = 200000
         D = np.random.randn(n)
         print("mu: {}, var: {}, skew: {}, kurt: {}".format(D.mean(), D.var(), st.skew(D), st.ku
         sns.distplot(D)
```

```
mu: 2.0811176896789058e-05, var: 1.0014134403607624, skew: 0.0008737443030560927, kurt: 3.007041
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```

```
Out[34]: <matplotlib.axes._subplots.AxesSubplot at 0x7f317989aba8>
```



4.3 Preliminary EDA

```
In [1]: customers <- read.table('Wholesale_customers_data.csv', sep=",", header = T)
```

```
In [2]: head(customers)
```

Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicatessen
2	3	12669	9656	7561	214	2674	1338
2	3	7057	9810	9568	1762	3293	1776
2	3	6353	8808	7684	2405	3516	7844
1	3	13265	1196	4221	6404	507	1788
2	3	22615	5410	7198	3915	1777	5185
2	3	9413	8259	5126	666	1795	1451

```
In [3]: dim(customers); str(customers)
```

1. 440 2. 8

```
'data.frame':   440 obs. of  8 variables:
 $ Channel         : int  2 2 2 1 2 2 2 2 1 2 ...
 $ Region          : int  3 3 3 3 3 3 3 3 3 3 ...
 $ Fresh           : int  12669 7057 6353 13265 22615 9413 12126 7579 5963 6006 ...
 $ Milk            : int  9656 9810 8808 1196 5410 8259 3199 4956 3648 11093 ...
 $ Grocery         : int  7561 9568 7684 4221 7198 5126 6975 9426 6192 18881 ...
 $ Frozen          : int  214 1762 2405 6404 3915 666 480 1669 425 1159 ...
 $ Detergents_Paper: int  2674 3293 3516 507 1777 1795 3140 3321 1716 7425 ...
 $ Delicatessen    : int  1338 1776 7844 1788 5185 1451 545 2566 750 2098 ...
```

```
In [4]: customers$Channel = factor(customers$Channel)
        customers$Region = factor(customers$Region)
```

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

```
'data.frame':   440 obs. of  8 variables:
 $ Channel         : Factor w/ 2 levels "1","2": 2 2 2 1 2 2 2 2 1 2 ...
 $ Region          : Factor w/ 3 levels "1","2","3": 3 3 3 3 3 3 3 3 3 3 ...
 $ Fresh           : int  12669 7057 6353 13265 22615 9413 12126 7579 5963 6006 ...
 $ Milk            : int  9656 9810 8808 1196 5410 8259 3199 4956 3648 11093 ...
 $ Grocery         : int  7561 9568 7684 4221 7198 5126 6975 9426 6192 18881 ...
 $ Frozen          : int  214 1762 2405 6404 3915 666 480 1669 425 1159 ...
 $ Detergents_Paper: int  2674 3293 3516 507 1777 1795 3140 3321 1716 7425 ...
 $ Delicatessen    : int  1338 1776 7844 1788 5185 1451 545 2566 750 2098 ...
```

```
In [6]: cust_sum = summary(customers)
        cust_sum
```

```
Channel Region      Fresh             Milk          Grocery
 1:298   1: 77   Min.   :     3   Min.   :   55   Min.   :    3
 2:142   2: 47   1st Qu.:  3128   1st Qu.: 1533   1st Qu.: 2153
         3:316   Median :  8504   Median : 3627   Median : 4756
                 Mean   : 12000   Mean   : 5796   Mean   : 7951
                 3rd Qu.: 16934   3rd Qu.: 7190   3rd Qu.:10656
                 Max.   :112151   Max.   :73498   Max.   :92780
     Frozen        Detergents_Paper   Delicatessen
 Min.   :   25.0   Min.   :    3.0   Min.   :    3.0
 1st Qu.:  742.2   1st Qu.:  256.8   1st Qu.:  408.2
 Median : 1526.0   Median :  816.5   Median :  965.5
 Mean   : 3071.9   Mean   : 2881.5   Mean   : 1524.9
 3rd Qu.: 3554.2   3rd Qu.: 3922.0   3rd Qu.: 1820.2
 Max.   :60869.0   Max.   :40827.0   Max.   :47943.0
```

```
In [7]: table(customers$Channel, customers$Region)
```

```
1   2   3
  1  59  28 211
  2  18  19 105
```

```
In [8]: customer_features = Filter(is.numeric, customers)
```

```
In [9]: library(repr)
        options(repr.plot.width=20, repr.plot.height=6)
```

```
In [10]: sum_vals = data.frame(feature=colnames(customer_features))
         sum_vals['mean_'] = sapply(customer_features, mean)
         sum_vals['median_'] = sapply(customer_features, median)
         sum_vals['sd_'] = sapply(customer_features, sd)
         sum_vals
```

feature	mean_	median_	sd_
Fresh	12000.298	8504.0	12647.329
Milk	5796.266	3627.0	7380.377
Grocery	7951.277	4755.5	9503.163
Frozen	3071.932	1526.0	4854.673
Detergents_Paper	2881.493	816.5	4767.854
Delicatessen	1524.870	965.5	2820.106

```
In [11]: library(reshape2)
```

```
In [12]: melt(sum_vals)
```

```
Using feature as id variables
```

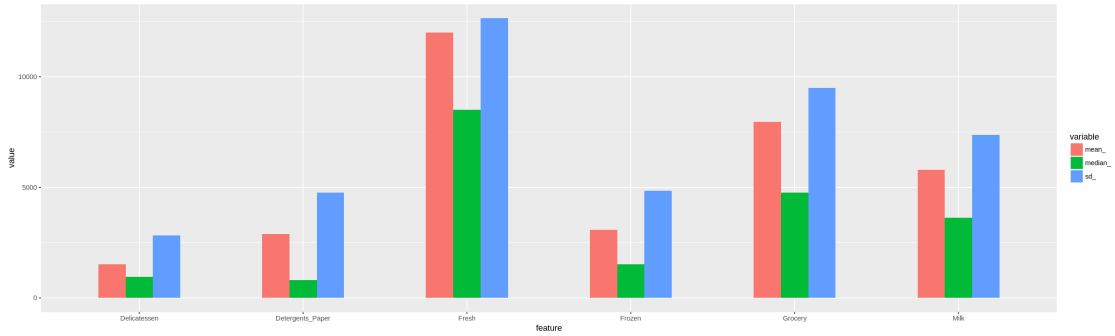
feature	variable	value
Fresh	mean_	12000.298
Milk	mean_	5796.266
Grocery	mean_	7951.277
Frozen	mean_	3071.932
Detergents_Paper	mean_	2881.493
Delicatessen	mean_	1524.870
Fresh	median_	8504.000
Milk	median_	3627.000
Grocery	median_	4755.500
Frozen	median_	1526.000
Detergents_Paper	median_	816.500
Delicatessen	median_	965.500
Fresh	sd_	12647.329
Milk	sd_	7380.377
Grocery	sd_	9503.163
Frozen	sd_	4854.673
Detergents_Paper	sd_	4767.854
Delicatessen	sd_	2820.106

```
In [13]: library(ggplot2)
```

```
         ggplot(melt(sum_vals), aes(x = feature, y = value, fill = variable)) +
             geom_bar(stat = "identity", width=0.5, position = "dodge")
```

```
Using feature as id variables
```

Data type cannot be displayed:




```
In [1]: customers <- read.table('Wholesale_customers_data.csv', sep=",", header = T)
        customer_features = Filter(is.numeric, customers)
```

4.4 Sampling

We can use the Survey Monkey Sample Size Calculator to consider what sample sizes we will need for our data to representative of the whole.

4.4.1 Test 1 - `n = 5`

```
In [2]: sum_vals = data.frame(feature=colnames(customer_features))
        sum_vals['mean_'] = sapply(customer_features, mean)
        sum_vals
```

feature	mean_
Channel	1.322727
Region	2.543182
Fresh	12000.297727
Milk	5796.265909
Grocery	7951.277273
Frozen	3071.931818
Detergents_Paper	2881.493182
Delicatessen	1524.870455

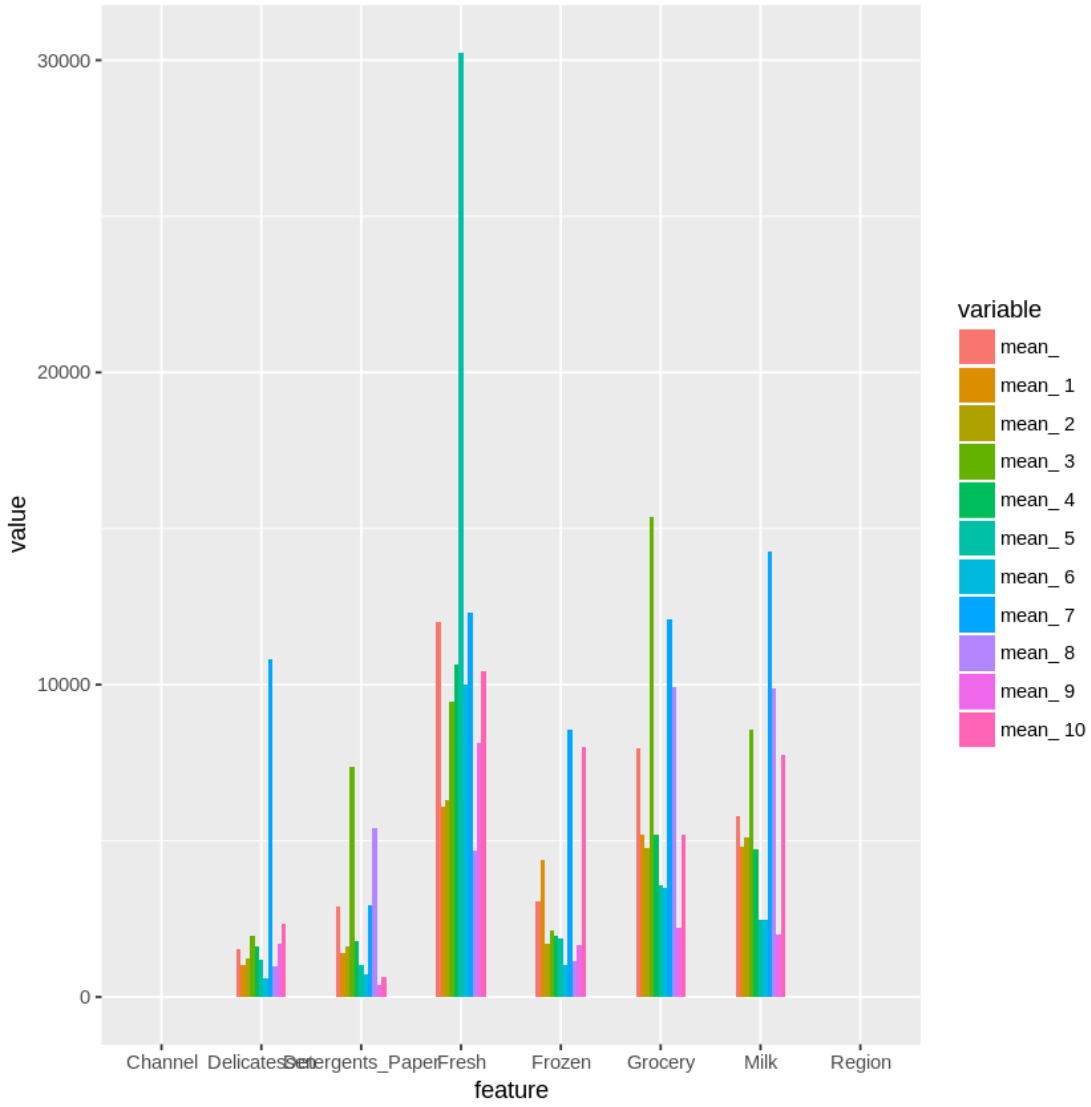
```
In [3]: library(dplyr, warn.conflicts = FALSE)
        samples = list()
        for (i in 1:10) {
            samples[[i]] = sample_n(customer_features, 5)
        }

In [4]: for (i in 1:10) {
            sum_vals[paste('mean_', i)] = sapply(samples[[i]], mean)
        }

In [5]: sum_vals
```

feature	mean_	mean_ 1	mean_ 2	mean_ 3	mean_ 4	mean_ 5	mean_ 6	mean_ 7	mean_ 8
Channel	1.322727	1.0	1.2	1.6	1.4	1.0	1.2	1.4	1.4
Region	2.543182	3.0	2.6	3.0	2.2	2.2	3.0	2.2	3.0
Fresh	12000.297727	6078.8	6294.8	9434.4	10634.0	30249.6	10006.6	12314.0	4696.2
Milk	5796.265909	4828.0	5126.0	8565.8	4725.4	2451.2	2479.4	14286.6	9900.4
Grocery	7951.277273	5215.0	4749.0	15366.8	5205.6	3562.8	3490.6	12102.8	9903.8
Frozen	3071.931818	4375.0	1700.8	2133.0	1939.8	1866.4	1025.4	8541.4	1131.4
Detergents_Paper	2881.493182	1412.8	1637.8	7384.4	1787.4	1027.0	740.4	2944.2	5408.4
Delicatessen	1524.870455	1040.6	1253.8	1972.0	1637.2	1203.6	600.0	10804.8	966.4

In [6]:
```
library(reshape2)
library(ggplot2)

ggplot(melt(sum_vals), aes(x = feature, y = value, fill = variable)) +
    geom_bar(stat = "identity", width=0.5, position = "dodge")
```

```
Using feature as id variables
```

Data type cannot be displayed:



4.4.2 Test 2 - `n = 50`

```
In [7]: sum_vals = data.frame(feature=colnames(customer_features))
        sum_vals['mean_'] = sapply(customer_features, mean)
        sum_vals
```

feature	mean_
Channel	1.322727
Region	2.543182
Fresh	12000.297727
Milk	5796.265909
Grocery	7951.277273
Frozen	3071.931818
Detergents_Paper	2881.493182
Delicatessen	1524.870455

```
In [8]: samples = list()
        for (i in 1:10) {
```

```
            samples[[i]] = sample_n(customer_features, 50)
        }
```

```
In [9]: for (i in 1:10) {
            sum_vals[paste('mean_', i)] = sapply(samples[[i]], mean)
        }
```

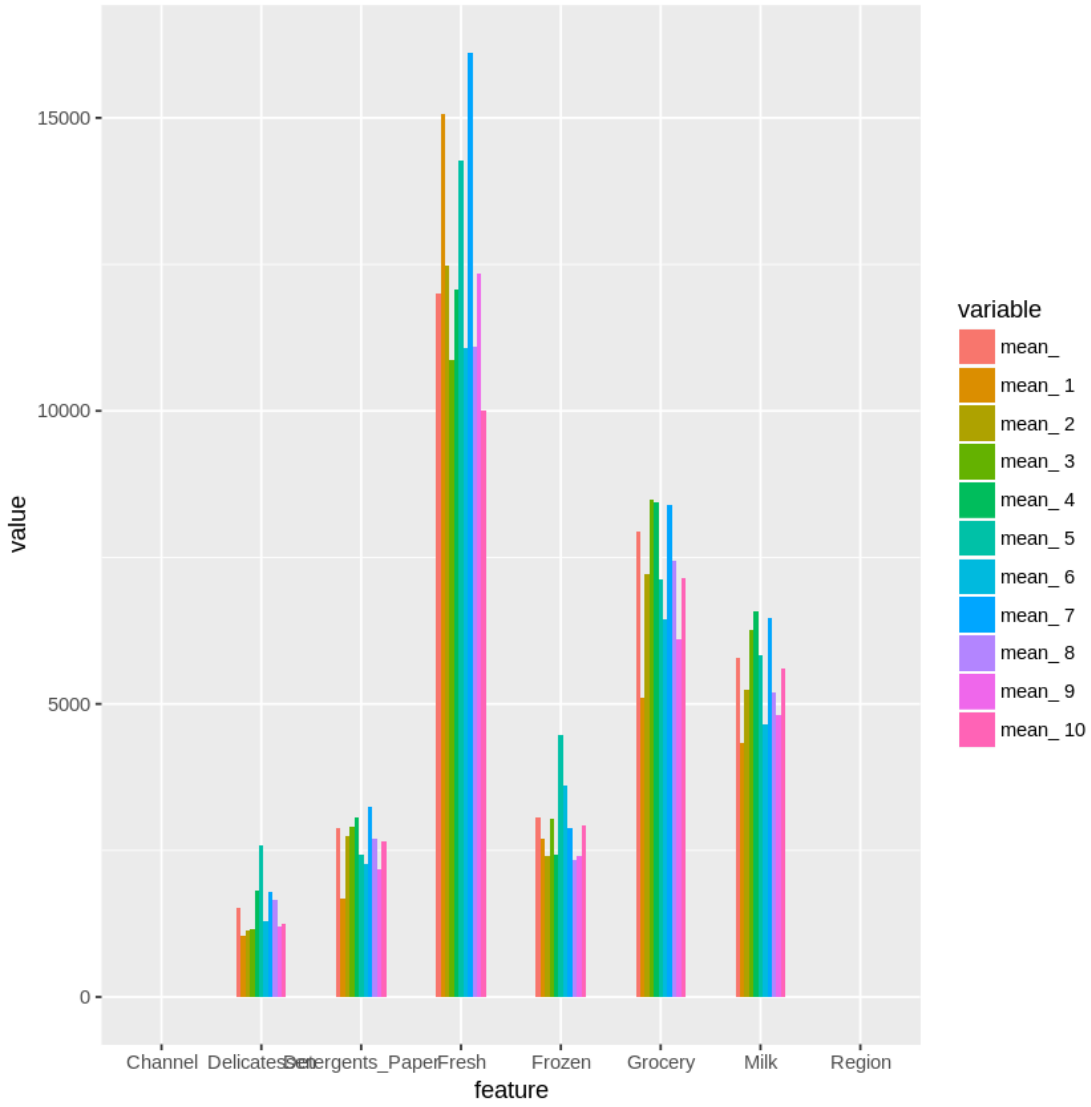
```
In [10]: sum_vals
```

feature	mean_	mean_ 1	mean_ 2	mean_ 3	mean_ 4	mean_ 5	mean_ 6	mean_ 7	me
Channel	1.322727	1.24	1.30	1.26	1.38	1.26	1.32	1.32	1.3
Region	2.543182	2.58	2.60	2.42	2.58	2.68	2.40	2.58	2.7
Fresh	12000.297727	15075.74	12484.90	10865.40	12073.56	14279.60	11072.20	16117.72	110
Milk	5796.265909	4325.60	5236.72	6255.24	6571.76	5826.96	4641.40	6477.76	518
Grocery	7951.277273	5100.40	7213.46	8488.42	8434.02	7125.42	6454.28	8391.26	744
Frozen	3071.931818	2691.42	2398.20	3031.08	2435.60	4478.56	3598.74	2892.28	234
Detergents_Paper	2881.493182	1673.16	2741.30	2900.46	3058.48	2429.24	2261.46	3249.62	270
Delicatessen	1524.870455	1040.22	1127.10	1162.96	1808.50	2588.38	1284.04	1800.90	165

```
In [11]: library(reshape2)
         library(ggplot2)

         ggplot(melt(sum_vals), aes(x = feature, y = value, fill = variable)) +
             geom_bar(stat = "identity", width=0.5, position = "dodge")
```

```
Using feature as id variables
```

Data type cannot be displayed:



4.4.3 Test 2 - `n = 150`

```
In [12]: sum_vals = data.frame(feature=colnames(customer_features))
         sum_vals['mean_'] = sapply(customer_features, mean)
         sum_vals
```

feature	mean_
Channel	1.322727
Region	2.543182
Fresh	12000.297727
Milk	5796.265909
Grocery	7951.277273
Frozen	3071.931818
Detergents_Paper	2881.493182
Delicatessen	1524.870455

```
In [13]: samples = list()
         for (i in 1:10) {
```

```
    samples[[i]] = sample_n(customer_features, 150)
}
```

In [14]:
```
for (i in 1:10) {
    sum_vals[paste('mean_', i)] = sapply(samples[[i]], mean)
}
```

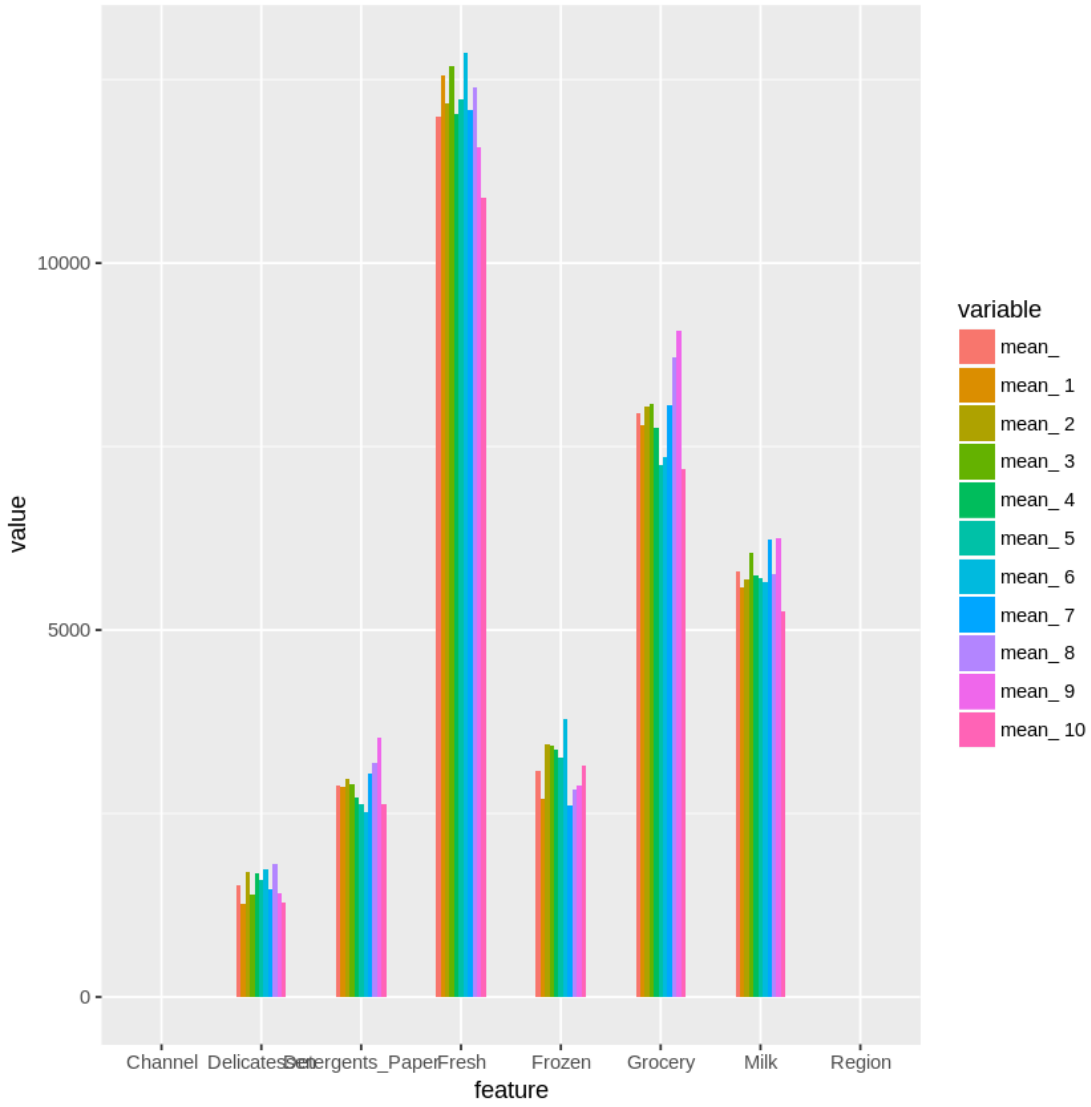
In [15]:
```
sum_vals
```

feature	mean_	mean_ 1	mean_ 2	mean_ 3	mean_ 4	mean_ 5	me
Channel	1.322727	1.346667	1.333333	1.326667	1.346667	1.320000	1.2
Region	2.543182	2.553333	2.613333	2.560000	2.446667	2.573333	2.3
Fresh	12000.297727	12551.373333	12169.140000	12686.913333	12039.500000	12237.960000	12
Milk	5796.265909	5588.453333	5688.786667	6043.613333	5751.866667	5701.246667	56
Grocery	7951.277273	7794.293333	8043.053333	8073.806667	7750.426667	7241.660000	73
Frozen	3071.931818	2702.393333	3447.700000	3432.253333	3362.500000	3260.666667	37
Detergents_Paper	2881.493182	2867.653333	2965.793333	2907.440000	2710.446667	2624.446667	25
Delicatessen	1524.870455	1268.853333	1700.986667	1400.220000	1682.153333	1597.653333	17

In [16]:
```
library(reshape2)
library(ggplot2)

ggplot(melt(sum_vals), aes(x = feature, y = value, fill = variable)) +
    geom_bar(stat = "identity", width=0.5, position = "dodge")
```

```
Using feature as id variables
```

Data type cannot be displayed:



4.5 The Z-Score

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

        %matplotlib inline
        from IPython.display import display
```

```
In [16]: customers = pd.read_csv('data/Wholesale_customers_data.csv')
         customers.Region = customers.Region.astype('category')
         customers.Channel = customers.Channel.astype('category')
         customer_features = customers.select_dtypes([int])

         display(customers.info())
         display(customers.describe())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 440 entries, 0 to 439
Data columns (total 8 columns):
Channel             440 non-null category
Region              440 non-null category
Fresh               440 non-null int64
Milk                440 non-null int64
Grocery             440 non-null int64
Frozen              440 non-null int64
Detergents_Paper    440 non-null int64
Delicatessen        440 non-null int64
dtypes: category(2), int64(6)
memory usage: 21.6 KB
```

```
None
```

```
Fresh          Milk       Grocery        Frozen  \
count     440.000000    440.000000    440.000000    440.000000
mean    12000.297727   5796.265909   7951.277273   3071.931818
std     12647.328865   7380.377175   9503.162829   4854.673333
min         3.000000     55.000000      3.000000     25.000000
25%      3127.750000   1533.000000   2153.000000    742.250000
50%      8504.000000   3627.000000   4755.500000   1526.000000
75%     16933.750000   7190.250000  10655.750000   3554.250000
max    112151.000000  73498.000000  92780.000000  60869.000000

       Detergents_Paper  Delicatessen
count        440.000000    440.000000
mean        2881.493182   1524.870455
std         4767.854448   2820.105937
min            3.000000      3.000000
25%          256.750000    408.250000
50%          816.500000    965.500000
75%         3922.000000   1820.250000
max        40827.000000  47943.000000
```

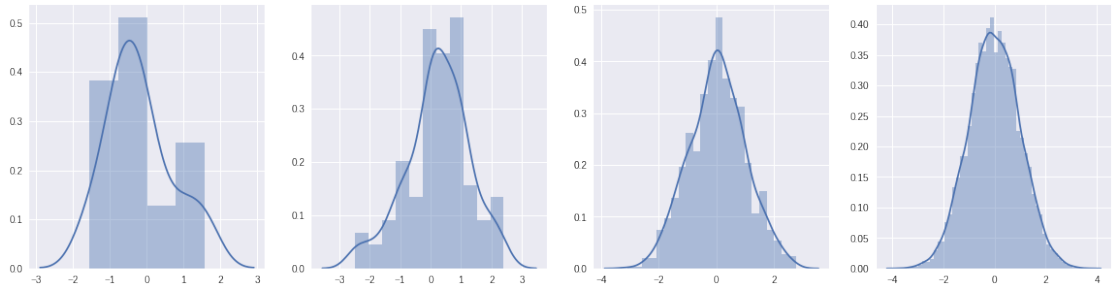
4.5.1 The Normal Distribution

The normal distribution is arguably the most commonly used distribution in all of statistics. **Normality** is an assumption that underlies many statistical tests and serves as a convenient model for the distribution of many (but not all!) variables.

The normal distribution relies on two parameters: - The population mean - The population standard deviation

If a variable follows a Normal distribution exactly, its mean, median, and mode will all be equal.

```
In [8]: fig = plt.figure(figsize=(20,5))

        for i in range(1,5):
            yy = np.random.normal(size=10**i)
            fig.add_subplot(1,4,i)
            sns.distplot(yy)
```



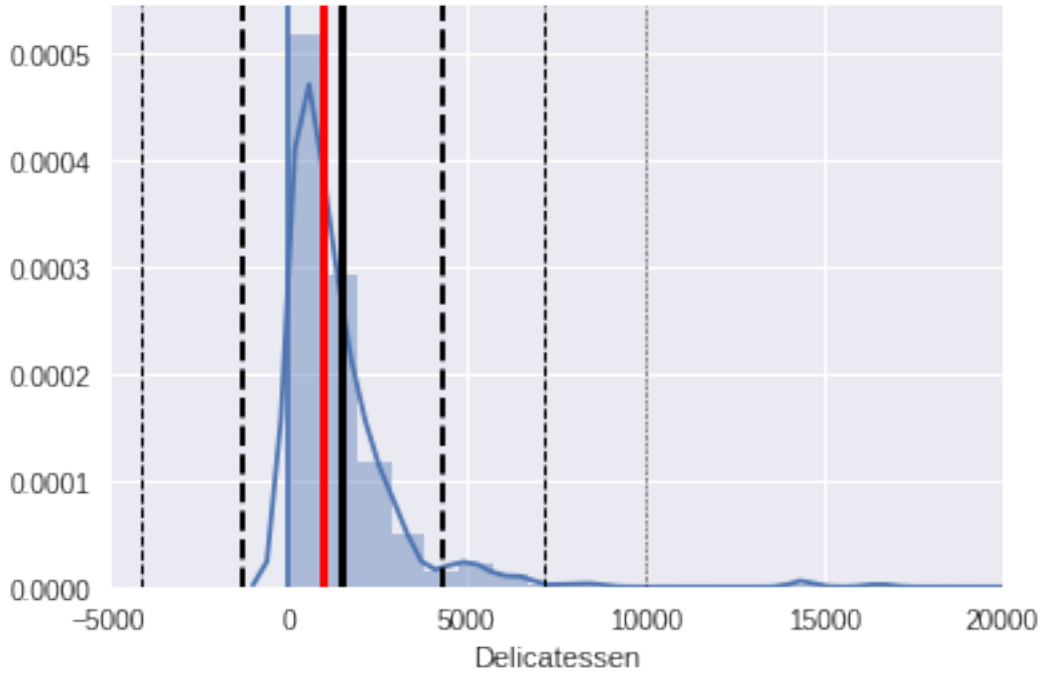
The 68-95-99.7 Rule

It is often beneficial to identify how extreme (or far away from the expected value) a particular observation is within the context of a distribution.

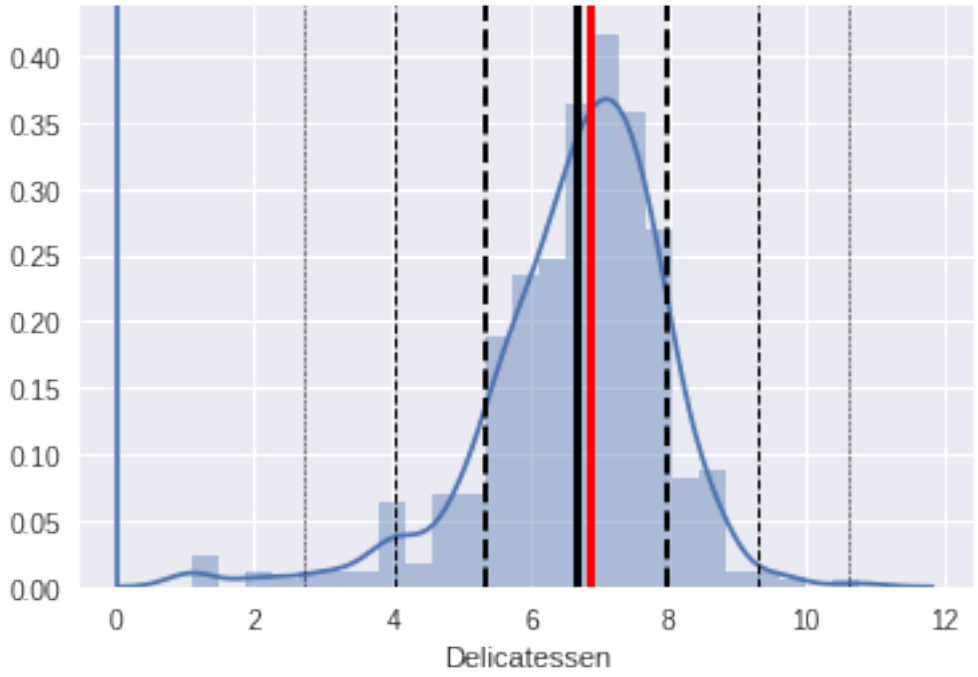
It is possible to show that, for a Normal distribution: - 68% of observations from a population will fall within ± 1 standard deviation of the population mean. - 95% of observations from a population will fall within ± 2 standard deviations of the population mean. - 99.7% of observations from a population will fall within ± 3 standard deviations of the population mean.

Below is a visual representation of the 68-95-99.7 rule on the Delicatessen distribution:

```
In [12]: sns.distplot(customers.Delicatessen)
         plt.axvline(customers.Delicatessen.mean(), color='black', lw=3)
         plt.axvline(customers.Delicatessen.median(), color='red', lw=3)
         plt.axvline((customers.Delicatessen.mean() - customers.Delicatessen.std()),
                     color='black', lw=2, ls="dashed")
         plt.axvline((customers.Delicatessen.mean() + customers.Delicatessen.std()),
                     color='black', lw=2, ls="dashed")
         plt.axvline((customers.Delicatessen.mean() + 2*customers.Delicatessen.std()),
                     color='black', lw=1, ls="dashed")
         plt.axvline((customers.Delicatessen.mean() - 2*customers.Delicatessen.std()),
                     color='black', lw=1, ls="dashed")
         plt.axvline((customers.Delicatessen.mean() + 3*customers.Delicatessen.std()),
                     color='black', lw=.5, ls="dashed")
         plt.axvline((customers.Delicatessen.mean() - 3*customers.Delicatessen.std()),
                     color='black', lw=.5, ls="dashed")
         plt.axvline(0)
         plt.xlim(-5000,20000)

Out[12]: (-5000, 20000)
```




```
In [11]: sns.distplot(np.log(customers.Delicatessen))
         plt.axvline(np.log(customers.Delicatessen).mean(), color='black', lw=3)
         plt.axvline(np.log(customers.Delicatessen).median(), color='red', lw=3)
         plt.axvline((np.log(customers.Delicatessen).mean() - np.log(customers.Delicatessen).std
                     color='black', lw=2, ls="dashed")
         plt.axvline((np.log(customers.Delicatessen).mean() + np.log(customers.Delicatessen).std
                     color='black', lw=2, ls="dashed")
         plt.axvline((np.log(customers.Delicatessen).mean() + 2*np.log(customers.Delicatessen).s
                     color='black', lw=1, ls="dashed")
         plt.axvline((np.log(customers.Delicatessen).mean() - 2*np.log(customers.Delicatessen).s
                     color='black', lw=1, ls="dashed")
         plt.axvline((np.log(customers.Delicatessen).mean() + 3*np.log(customers.Delicatessen).s
                     color='black', lw=.5, ls="dashed")
         plt.axvline((np.log(customers.Delicatessen).mean() - 3*np.log(customers.Delicatessen).s
                     color='black', lw=.5, ls="dashed")
         plt.axvline(0)

Out[11]: <matplotlib.lines.Line2D at 0x7f130c91bac8>
```



4.5.2 Definition: z-score

The z-score of an observation quantifies how many standard deviations the observation is away from the population mean:

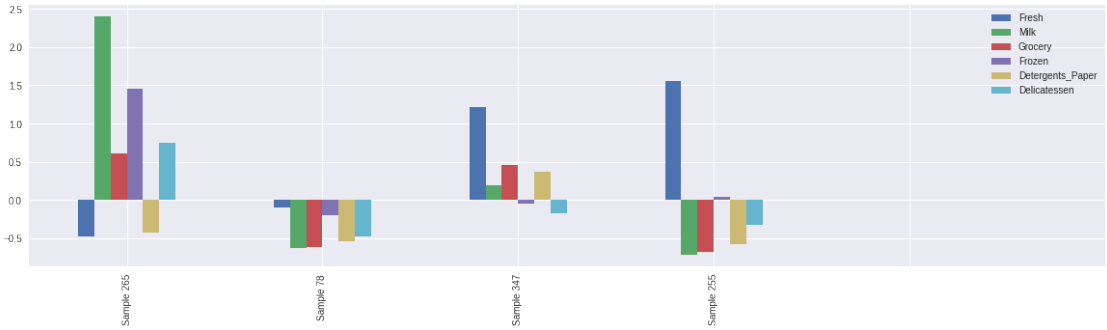
$$z_i = \frac{x_i - \text{population mean of x}}{\text{standard deviation of x}}$$

```
In [17]: customer_feature_z_scores = (customer_features -  customer_features.mean())/customer_fe
```

```
In [21]: np.random.seed(42)
         sample = customer_feature_z_scores.sample(4)
         sample
```

```
Out[21]: Fresh      Milk   Grocery    Frozen  Detergents_Paper  Delicatessen
         265 -0.481627  2.402416  0.604822  1.459021         -0.430276      0.748599
         78  -0.097594 -0.626156 -0.619191 -0.201029         -0.541227     -0.481496
         347  1.216044  0.188030  0.458765 -0.054161          0.364841     -0.178316
         255  1.550818 -0.719105 -0.679382  0.035032         -0.581078     -0.322637
```

```
In [22]: sample.plot(kind='bar', figsize=(20,5))
         labels = ["Sample {}".format(i) for i in sample.index]
         plt.xticks(range(sample.shape[0]+2),labels);
```



4.6 Central Limit Theorem

Normality underlies many of the inferential techniques that we use in data science.

Consider the random variable X. We can take a sample from this population of size n and find the mean of that sample. Let's call this sample mean x_1. We can take another sample from this population, also of size n, and find the mean of that sample. Let's call this sample mean x_1. We can do this over and over until we've calculated the mean of every possible sample of size n. If we plotted every sample mean on a histogram, we get another distribution called "the sampling distribution of $\bar{X}$."

This distribution, the sampling distribution of :math:'bar{X}', is Normally distributed even if the distribution of :math:'X' is not. (That is, unless some rare conditions are violated).

We can formally define the central limit theorm like so:

> In probability theory, the central limit theorem states that, when independent random variables are added, their sum tends toward a normal distribution (commonly known as a bell curve), even if the original variables themselves are not normally distributed. In more precise terms, given certain conditions, the arithmetic mean of a sufficiently large number of iterates of independent random variables — each with a well-defined (finite) expected value and finite variance — will be approximately normally distributed, regardless of the underlying distribution.

Some properties that arise from the central limit theorem include:

> If X $N(\mu, \sigma)$, then $\bar{X}$ is exactly $N(\mu, \frac{\sigma}{\sqrt{n}})$
>
> If X is not normally distributed, then $\bar{X}$ is approximately $N(\mu, \frac{\sigma}{\sqrt{n}})$ if the sample size n is at least 30. As n increases, $\bar{X}$ becomes asymptotically normally distributed.
>
> If $\bar{X}$ is normally distributed, then we can use inferential methods that rely on our sample mean, $\bar{x}$

4.6.1 Additional resources

http://blog.vctr.me/posts/central-limit-theorem.html

http://www.usablestats.com/lessons/central_limit

http://blog.minitab.com/blog/michelle-paret/explaining-the-central-limit-theorem-with-bunnies-and-dragons-v2

In the next few notebooks, we are going to do some Unsupervised Exploration of the `customer` table in our Database.

> What does a data scientist do? PCA on the `customer` table. - Joshua Cook

```
In [41]: from scipy.stats import skew

In [42]: skew(customer_features)

Out[42]: array([  2.55258269,   4.03992212,   3.57518722,   5.88782573,
                  3.61945758,  11.11353365])

In [43]: import random

         random.sample(range(10), 2)

Out[43]: [3, 1]

In [44]: stats = customer_features.describe().T
         stats['skew'] = skew(customer_features)
         stats

Out[44]: count          mean           std   min      25%     50%  \
         Fresh             440.0  12000.297727  12647.328865   3.0  3127.75  8504.0
         Milk              440.0   5796.265909   7380.377175  55.0  1533.00  3627.0
         Grocery           440.0   7951.277273   9503.162829   3.0  2153.00  4755.5
         Frozen            440.0   3071.931818   4854.673333  25.0   742.25  1526.0
         Detergents_Paper  440.0   2881.493182   4767.854448   3.0   256.75   816.5
         Delicatessen      440.0   1524.870455   2820.105937   3.0   408.25   965.5

                              75%       max       skew
         Fresh            16933.75  112151.0   2.552583
         Milk              7190.25   73498.0   4.039922
         Grocery          10655.75   92780.0   3.575187
         Frozen            3554.25   60869.0   5.887826
         Detergents_Paper  3922.00   40827.0   3.619458
         Delicatessen      1820.25   47943.0  11.113534
```

Sampling the Dataset

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

```
In [45]: np.random.seed(42)

In [49]: sample = customer_features.sample(3)

In [50]: sample

Out[50]: Fresh   Milk  Grocery  Frozen  Detergents_Paper  Delicatessen
         265    5909  23527    13699   10155              830          3636
         78    10766   1175     2067    2096              301           167
         347   27380   7184    12311    2809             4621          1022

In [51]: stats

Out[51]: count          mean           std   min      25%     50%  \
         Fresh             440.0  12000.297727  12647.328865   3.0  3127.75  8504.0
         Milk              440.0   5796.265909   7380.377175  55.0  1533.00  3627.0
         Grocery           440.0   7951.277273   9503.162829   3.0  2153.00  4755.5
         Frozen            440.0   3071.931818   4854.673333  25.0   742.25  1526.0
         Detergents_Paper  440.0   2881.493182   4767.854448   3.0   256.75   816.5
         Delicatessen      440.0   1524.870455   2820.105937   3.0   408.25   965.5
```

```
                          75%       max       skew
Fresh                16933.75  112151.0   2.552583
Milk                  7190.25   73498.0   4.039922
Grocery              10655.75   92780.0   3.575187
Frozen                3554.25   60869.0   5.887826
Detergents_Paper      3922.00   40827.0   3.619458
Delicatessen          1820.25   47943.0  11.113534
```

Sampling for a Statistical Description

We are able to take the mean and standard deviation of the data, but what if we want to visualize it?

Of course, this dataset is small, but we might want techniques that work even when the dataset is very large.

Let's start by looking at 1% of the data.

```
In [52]: sample_1pct_1 = customer_features.sample(5)

In [53]: sample_1pct_1.mean()

Out[53]: Fresh               14123.6
         Milk                 5801.0
         Grocery              5475.8
         Frozen               2812.6
         Detergents_Paper     1097.8
         Delicatessen         1987.6
         dtype: float64
```

4.6.2 How does this compare to the actual mean?

```
In [54]: sample_1pct_1.mean() - stats['mean']

Out[54]: Fresh               2123.302273
         Milk                   4.734091
         Grocery            -2475.477273
         Frozen              -259.331818
         Detergents_Paper   -1783.693182
         Delicatessen         462.729545
         dtype: float64
```

Let's think about this in terms of percent error.

```
In [55]: (sample_1pct_1.mean() - stats['mean'])/stats['mean']

Out[55]: Fresh               0.176937
         Milk                0.000817
         Grocery            -0.311331
         Frozen             -0.084420
         Detergents_Paper   -0.619017
         Delicatessen        0.303455
         dtype: float64
```

4.6.3 How does it do?

4.6.4 Let's try it again

```
In [56]: sample_1pct_2 = customer_features.sample(5)
```

```
In [57]: sample_1pct_2.mean() - stats['mean']
```

```
Out[57]: Fresh              -2157.497727
         Milk                 484.534091
         Grocery             1619.922727
         Frozen              -310.531818
         Detergents_Paper     524.306818
         Delicatessen          39.729545
         dtype: float64
```

```
In [58]: (sample_1pct_2.mean() - stats['mean'])/stats['mean']
```

```
Out[58]: Fresh              -0.179787
         Milk                0.083594
         Grocery             0.203731
         Frozen              -0.101087
         Detergents_Paper    0.181957
         Delicatessen        0.026054
         dtype: float64
```

4.6.5 How does it do?

4.6.6 Repeatedly Sample

Let's do it 10 times.

```
In [59]: sample_means = []
         for _ in range(10):
             sample_means.append(customer_features.sample(5).mean())

         sample_means = np.array(sample_means)
         (sample_means.mean(axis=0)-stats['mean'])/stats['mean']
```

```
Out[59]: Fresh              -0.105892
         Milk                0.108534
         Grocery             -0.013132
         Frozen              0.000081
         Detergents_Paper    0.103324
         Delicatessen        -0.288294
         Name: mean, dtype: float64
```

And 50 times.

```
In [60]: sample_means = []
         for _ in range(50):
             sample_means.append(customer_features.sample(5).mean())

         sample_means = np.array(sample_means)
         (sample_means.mean(axis=0)-stats['mean'])/stats['std']
```

```
Out[60]: Fresh              -0.001005
         Milk                0.014526
         Grocery             -0.009778
         Frozen              0.080827
         Detergents_Paper    -0.029035
         Delicatessen        0.037616
         dtype: float64
```

And 100 times.

```
In [61]: sample_means = []
         for _ in range(100):
             sample_means.append(customer_features.sample(5).mean())

         sample_means = np.array(sample_means)
         (sample_means.mean(axis=0)-stats['mean'])/stats['std']
```

```
Out[61]: Fresh              -0.056725
         Milk               -0.032065
         Grocery            -0.039822
         Frozen              0.003006
         Detergents_Paper   -0.040477
         Delicatessen        0.030977
         dtype: float64
```

4.6.7 What do we notice?

4.6.8 Take a larger sample

Totally different. Which makes sense . . . we're only taking 1% of the data!

What if we take a sample of 10% of the data?

```
In [62]: sample_10pct_1 = customer_features.sample(44)
         (sample_10pct_1.mean() - stats['mean'])/stats['mean']
```

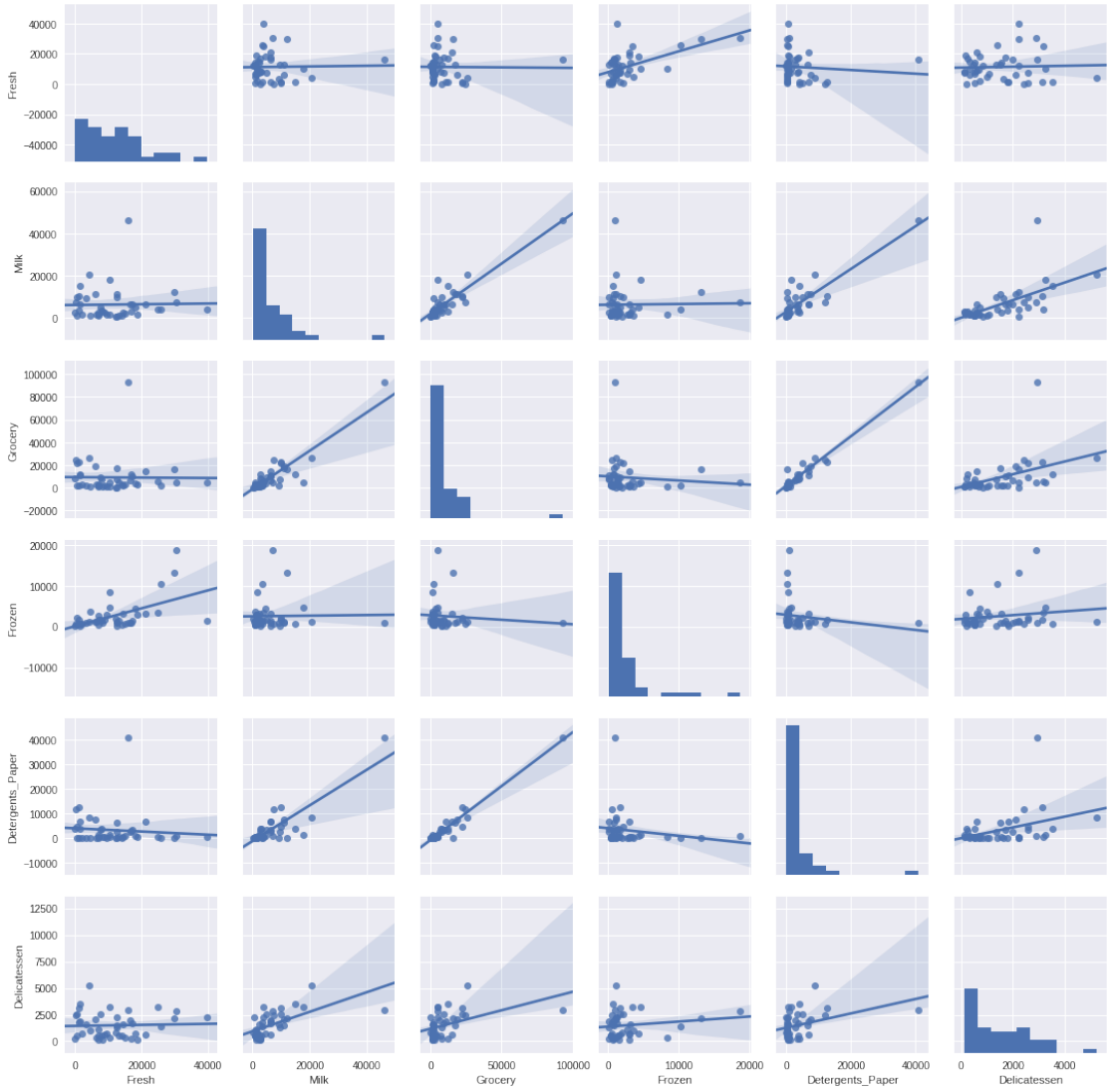
```
Out[62]: Fresh              -0.053236
         Milk                0.063137
         Grocery             0.153777
         Frozen             -0.166944
         Detergents_Paper    0.145184
         Delicatessen       -0.017353
         dtype: float64
```

4.6.9 Is this sample good enough for plotting?

https://stats.stackexchange.com/questions/2541/is-there-a-reference-that-suggest-using-30-as-a-large-enough-sample-size

```
In [63]: sns.pairplot(sample_10pct_1, kind='reg')
```

```
Out[63]: <seaborn.axisgrid.PairGrid at 0x7fec4ff2c2b0>
```



4.7 Correlation and Redundancy

I claim that there is correlation and redundancy in the `customer` table. What I mean by this is that some features are linear combinations of other features.

Let's examine redundancy by dropping a feature and seeing if the other features can predict it.

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

        %matplotlib inline
        from IPython.display import display

        customers = pd.read_csv('Wholesale_customers_data.csv')
        customers.Region = customers.Region.astype('category')
```

```
        customers.Channel = customers.Channel.astype('category')
        customer_features = customers.select_dtypes([int])
```

```
In [2]: from sklearn.model_selection import train_test_split
        from sklearn.tree import DecisionTreeRegressor
```

```
In [3]: def calculate_r_2_for_feature(data,feature):
            new_data = data.drop(feature, axis=1)
            target = data[feature]

            X_train, \
            X_test,  \
            y_train, \
            y_test = train_test_split(
                new_data,target,test_size=0.25
            )

            regressor = DecisionTreeRegressor()
            regressor.fit(X_train,y_train)

            score = regressor.score(X_test,y_test)
            return score
```

```
In [4]: calculate_r_2_for_feature(customer_features,'Detergents_Paper')
```

```
Out[4]: 0.68955180233880187
```

```
In [5]: print("{:24} {}".format("Delicatessen: ", calculate_r_2_for_feature(customer_features,'D
        print("{:24} {}".format("Degergents_paper: ", calculate_r_2_for_feature(customer_feature
        print("{:24} {}".format("Fresh: ", calculate_r_2_for_feature(customer_features,'Fresh'))
        print("{:24} {}".format("Frozen: ", calculate_r_2_for_feature(customer_features,'Frozen'
        print("{:24} {}".format("Grocery: ", calculate_r_2_for_feature(customer_features,'Grocer
        print("{:24} {}".format("Milk: ", calculate_r_2_for_feature(customer_features,'Milk')))
```

```
Delicatessen:            -0.10110199949377607
Degergents_paper:        0.7237296133962476
Fresh:                   -0.15561863327423753
Frozen:                  0.15408964681328152
Grocery:                 0.5711225907682309
Milk:                    0.36936099492906893
```

But this is subject to randomness. There is randomness in my `train_test_split`. Let's do the whole thing many times and take the average.

```
In [6]: def mean_r2_for_feature(data, feature):
            scores = []
            for _ in range(100):
                scores.append(calculate_r_2_for_feature(data, feature))

            scores = np.array(scores)
            return scores.mean()
```

```
In [7]: print("{:24} {}".format("Delicatessen: ", mean_r2_for_feature(customer_features,'Delicat
        print("{:24} {}".format("Detergents_Paper: ", mean_r2_for_feature(customer_features,'Det
        print("{:24} {}".format("Fresh: ", mean_r2_for_feature(customer_features,'Fresh')))
        print("{:24} {}".format("Frozen: ", mean_r2_for_feature(customer_features,'Frozen')))
        print("{:24} {}".format("Grocery: ", mean_r2_for_feature(customer_features,'Grocery')))
        print("{:24} {}".format("Milk: ", mean_r2_for_feature(customer_features,'Milk')))
```

```
Delicatessen:            -2.9743955952146393
Detergents_Paper:        0.664581339638863
Fresh:                   -0.72646632081378
Frozen:                  -0.8326070538178221
```

```
Grocery:                  0.6781021997531815
Milk:                     0.028011623136967624
```

```
In [8]: print("{:24} {}".format("Delicatessen: ", mean_r2_for_feature(customer_features,'Delicat
        print("{:24} {}".format("Detergents_Paper: ", mean_r2_for_feature(customer_features,'Det
        print("{:24} {}".format("Fresh: ", mean_r2_for_feature(customer_features,'Fresh')))
        print("{:24} {}".format("Frozen: ", mean_r2_for_feature(customer_features,'Frozen')))
        print("{:24} {}".format("Grocery: ", mean_r2_for_feature(customer_features,'Grocery')))
        print("{:24} {}".format("Milk: ", mean_r2_for_feature(customer_features,'Milk')))
```

```
Delicatessen:             -2.8077039808500466
Detergents_Paper:         0.6552670432973136
Fresh:                    -0.7709724608776678
Frozen:                   -1.1212585099954124
Grocery:                  0.669636174477285
Milk:                     0.1539517179009045
```

4.7.1 Discussion

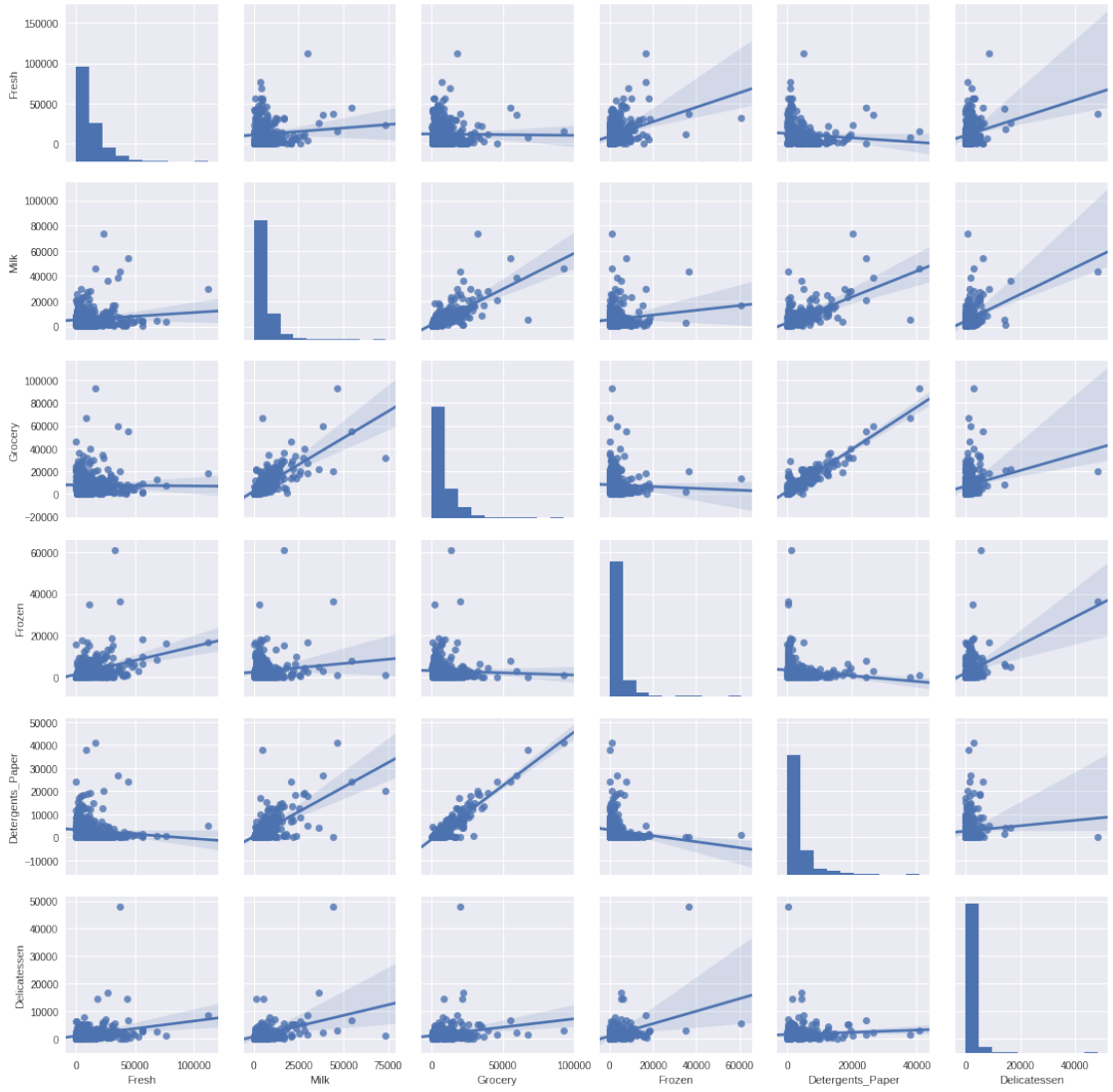
What does this tell us?

Visualize Redundancy

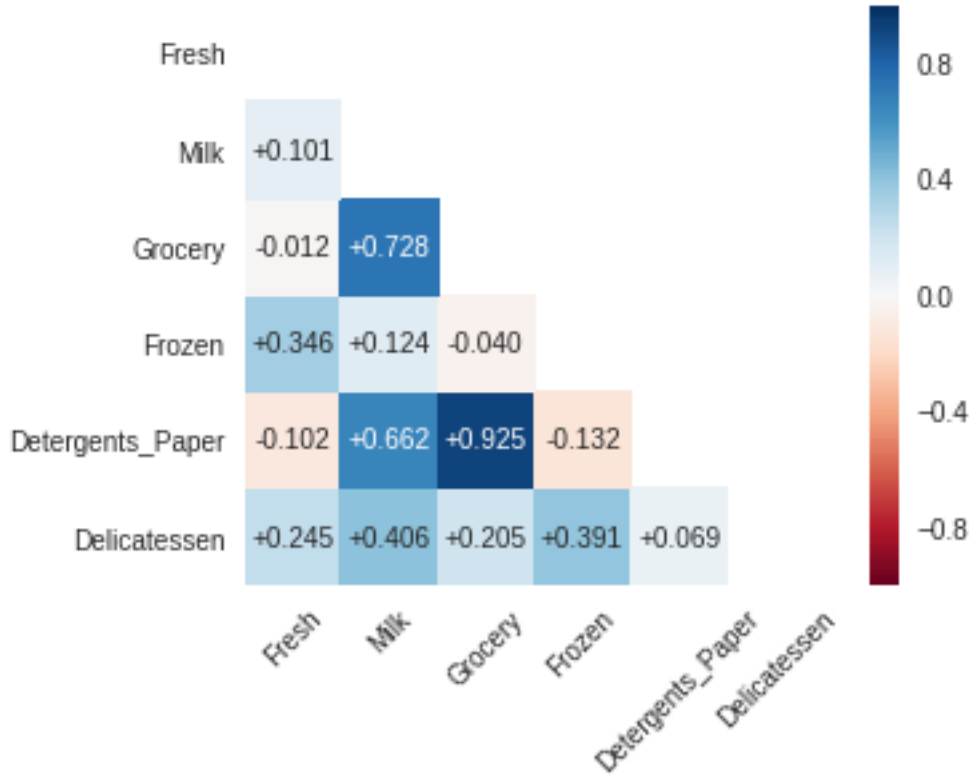
Study the correlation of the data.

```
In [9]: import time
        start = time.time()
        sns.pairplot(customer_features, kind='reg')
        print(time.time() - start)
```

```
14.507956266403198
```




```
In [10]: corr = customer_features.corr()
         mask = np.zeros_like(corr)
         mask[np.triu_indices_from(mask, 0)] = True
         with sns.axes_style("white"):
             ax = sns.heatmap(corr, mask=mask, square=True, annot=True,
                              cmap='RdBu', fmt='+.3f')
             plt.xticks(rotation=45, ha='center')
```



In [11]:
```
from sklearn.preprocessing import StandardScaler
import scipy.stats as st
```

In [12]:
```
sample_1pct_1 = customer_features.sample(5)
```

In [13]:
```
samp_stats = sample_1pct_1.describe().T
samp_stats['skew'] = st.skew(sample_1pct_1)
samp_stats['kurt'] = st.kurtosis(sample_1pct_1)
samp_stats
```

Out[13]:

	count	mean	std	min	25%	50%	75%	max	skew	kurt
Fresh	5.0	10107.6	12759.709452	622.0	1725.0	4983.0	11594.0	31614.0	1.126619	-0.316084
Milk	5.0	3366.6	3202.526003	55.0	489.0	3651.0	4859.0	7779.0	0.247699	-1.298135
Grocery	5.0	6646.2	5858.136026	137.0	1495.0	6633.0	12144.0	12822.0	-0.014954	-1.708406
Frozen	5.0	5051.8	7303.444544	75.0	824.0	3242.0	3252.0	17866.0	1.358587	0.066258
Detergents_Paper	5.0	2697.8	3485.200525	7.0	111.0	912.0	4424.0	8035.0	0.757249	-1.035105
Delicatessen	5.0	1648.8	1278.895696	8.0	615.0	2157.0	2435.0	3029.0	-0.304111	-1.543527

In [14]:
```
stats = customer_features.describe().T
stats['skew'] = st.skew(customer_features)
stats['kurt'] = st.kurtosis(customer_features)
stats
```

Out[14]:

	count	mean	std	min	25%	50%	\
Fresh	440.0	12000.297727	12647.328865	3.0	3127.75	8504.0	

```
Milk              440.0   5796.265909   7380.377175  55.0  1533.00  3627.0
Grocery           440.0   7951.277273   9503.162829   3.0  2153.00  4755.5
Frozen            440.0   3071.931818   4854.673333  25.0   742.25  1526.0
Detergents_Paper  440.0   2881.493182   4767.854448   3.0   256.75   816.5
Delicatessen      440.0   1524.870455   2820.105937   3.0   408.25   965.5

                       75%       max       skew        kurt
Fresh             16933.75  112151.0   2.552583   11.392124
Milk               7190.25   73498.0   4.039922   24.376349
Grocery           10655.75   92780.0   3.575187   20.664153
Frozen             3554.25   60869.0   5.887826   54.056180
Detergents_Paper   3922.00   40827.0   3.619458   18.780528
Delicatessen       1820.25   47943.0  11.113534  168.747781
```

MANY OF THE TOOLS WE WILL USE WILL ASSUME NORMAL DATA

You are already familiar with standardization.

$$Z = \frac{X - \mu}{\sigma}$$

```
In [17]: scaler = StandardScaler()
         scaler.fit(customer_features)
         customer_sc = scaler.transform(customer_features)

         customer_sc_df = pd.DataFrame(customer_sc, columns=customer_features.columns)

         sc_stats = customer_sc_df.describe().T
         sc_stats['skew'] = st.skew(customer_features)
         sc_stats['kurt'] = st.kurtosis(customer_features)
         display(stats)
         display(sc_stats)
```

```
count          mean           std   min      25%     50%  \
Fresh             440.0  12000.297727  12647.328865   3.0  3127.75  8504.0
Milk              440.0   5796.265909   7380.377175  55.0  1533.00  3627.0
Grocery           440.0   7951.277273   9503.162829   3.0  2153.00  4755.5
Frozen            440.0   3071.931818   4854.673333  25.0   742.25  1526.0
Detergents_Paper  440.0   2881.493182   4767.854448   3.0   256.75   816.5
Delicatessen      440.0   1524.870455   2820.105937   3.0   408.25   965.5

                       75%       max       skew        kurt
Fresh             16933.75  112151.0   2.552583   11.392124
Milk               7190.25   73498.0   4.039922   24.376349
Grocery           10655.75   92780.0   3.575187   20.664153
Frozen             3554.25   60869.0   5.887826   54.056180
Detergents_Paper   3922.00   40827.0   3.619458   18.780528
Delicatessen       1820.25   47943.0  11.113534  168.747781
```

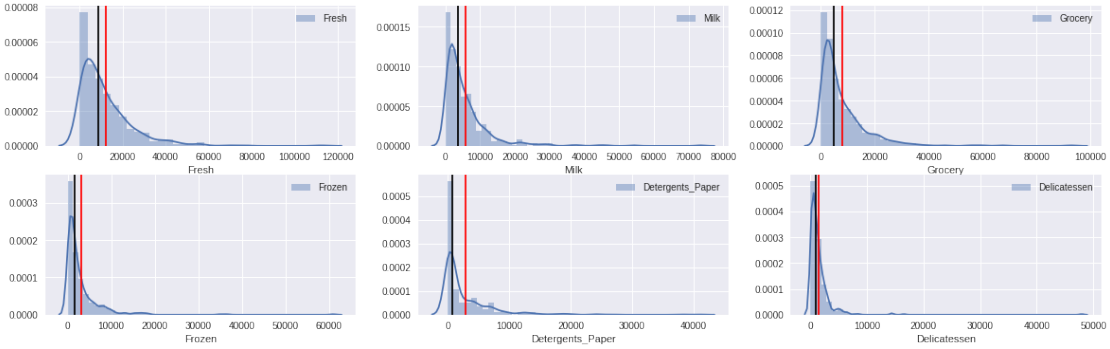
```
count          mean       std       min       25%       50%  \
Fresh             440.0 -3.431598e-17  1.001138 -0.949683 -0.702334 -0.276760
Milk              440.0  0.000000e+00  1.001138 -0.778795 -0.578306 -0.294258
Grocery           440.0 -4.037175e-17  1.001138 -0.837334 -0.610836 -0.336668
Frozen            440.0  3.633457e-17  1.001138 -0.628343 -0.480431 -0.318804
Detergents_Paper  440.0  2.422305e-17  1.001138 -0.604416 -0.551135 -0.433600
```

```
Delicatessen      440.0 -8.074349e-18  1.001138 -0.540264 -0.396401 -0.198577

                       75%        max       skew        kurt
Fresh             0.390523   7.927738   2.552583   11.392124
Milk              0.189092   9.183650   4.039922   24.376349
Grocery           0.284911   8.936528   3.575187   20.664153
Frozen            0.099464  11.919002   5.887826   54.056180
Detergents_Paper  0.218482   7.967672   3.619458   18.780528
Delicatessen      0.104860  16.478447  11.113534  168.747781
```

Visualizing Data Transformation

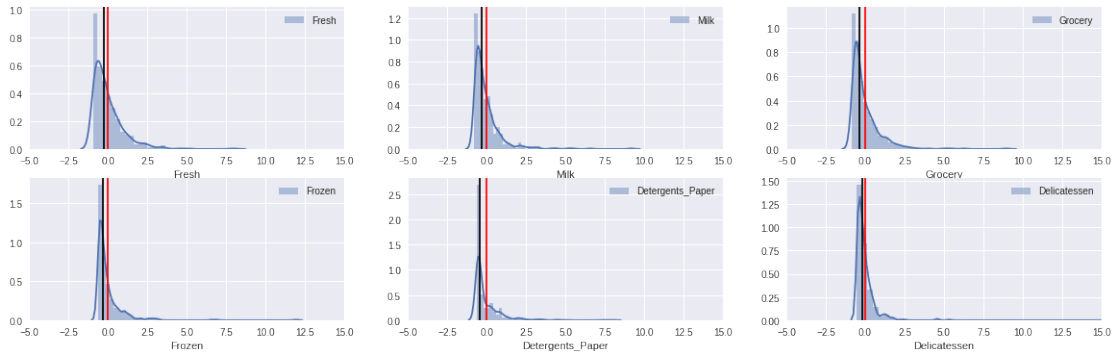
```
In [16]: fig = plt.figure(figsize=(20,6))
         for i, col in enumerate(customer_features.columns):
             fig.add_subplot(231+i)
             sns.distplot(customer_features[col], label=col)
             plt.axvline(customer_features[col].mean(), c='red')
             plt.axvline(customer_features[col].median(), c='black')
             plt.legend()
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```




```
In [17]: fig = plt.figure(figsize=(20,6))
         for i, col in enumerate(customer_sc_df.columns):
             fig.add_subplot(231+i)
             sns.distplot(customer_sc_df[col], label=col)
             plt.axvline(customer_sc_df[col].mean(), c='red')
             plt.axvline(customer_sc_df[col].median(), c='black')
             plt.legend()
             plt.xlim(-5,15)
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



MANY OF THE TOOLS WE WILL USE WILL ASSUME NORMAL DATA

4.8 Deskew the Data

We will look at two common approaches to deskewing data:

- the log transform
- scaling by the Box-Cox test

For purposes of comparison, we will keep both transforms.

We have previously looked at scaling data as a preprocessing step. Note that scaling of data will have no effect on its skewness.

Another way we can verify this is via a test of skewness.

To perform this test we can use the `scipy.stats.skewtest`.

This function tests the null hypothesis that the skewness of the population that the sample was drawn from is the same as that of a corresponding normal distribution. Remember, a low p-value means reject the null hypothesis i.e the data is skewed!

```
In [1]: import scipy.stats as st
```

```
In [2]: import numpy as np
        import pandas as pd
        import scipy.stats as stats
        import seaborn as sns
        import matplotlib.pyplot as plt

        %matplotlib inline
        from IPython.display import display

        customers = pd.read_csv('Wholesale_customers_data.csv')
        customers.Region = customers.Region.astype('category')
        customers.Channel = customers.Channel.astype('category')
        customer_features = customers.select_dtypes([int])
```

```
In [3]: from sklearn.preprocessing import StandardScaler
```

```
In [4]: scaler = StandardScaler()
        customer_sc = scaler.fit_transform(customer_features)
        customer_sc_df = pd.DataFrame(customer_sc, columns=customer_features.columns)
```

```
In [5]: for col in customer_sc_df.columns:
            original_col_skewtest = st.skewtest(customer_features[col])
            scaled_col_skewtest = st.skewtest(customer_sc_df[col])
            print("{}\norig skew test: {} \nscaled skew test: {}\n\n".format(col,
                                                                             original_col_skewte
                                                                             scaled_col_skewtest
```

```
Fresh
orig skew test: SkewtestResult(statistic=13.363200236723891, pvalue=9.920555601203282e-41)
scaled skew test: SkewtestResult(statistic=13.363200236723884, pvalue=9.920555601204277e-41)


Milk
orig skew test: SkewtestResult(statistic=16.597463367777181, pvalue=7.2698939324027666e-62)
scaled skew test: SkewtestResult(statistic=16.597463367777177, pvalue=7.2698939324029738e-62)


Grocery
orig skew test: SkewtestResult(statistic=15.727788993926845, pvalue=9.7558667104486924e-56)
scaled skew test: SkewtestResult(statistic=15.727788993926845, pvalue=9.7558667104486924e-56)


Frozen
orig skew test: SkewtestResult(statistic=19.301887122446967, pvalue=5.1783108829290055e-83)
scaled skew test: SkewtestResult(statistic=19.301887122446967, pvalue=5.1783108829290055e-83)


Detergents_Paper
orig skew test: SkewtestResult(statistic=15.815137437204189, pvalue=2.4467517316177991e-56)
scaled skew test: SkewtestResult(statistic=15.815137437204193, pvalue=2.44675173161759e-56)


Delicatessen
orig skew test: SkewtestResult(statistic=23.905153021167603, pvalue=2.707223345502464e-126)
scaled skew test: SkewtestResult(statistic=23.90515302116761, pvalue=2.7072233455021563e-126)
```

4.8.1 Deskew by taking the log of the data

Many times the skew of data can be easily removed by taking the log of the data. Let's do so here.

We will then scale the data after deskewing.

```
In [6]: customer_log_df = np.log(1+customer_features)
        scaler.fit(customer_log_df)
        customer_log_sc = scaler.transform(customer_log_df)
        customer_log_sc_df = pd.DataFrame(customer_log_sc, columns=customer_features.columns)
```

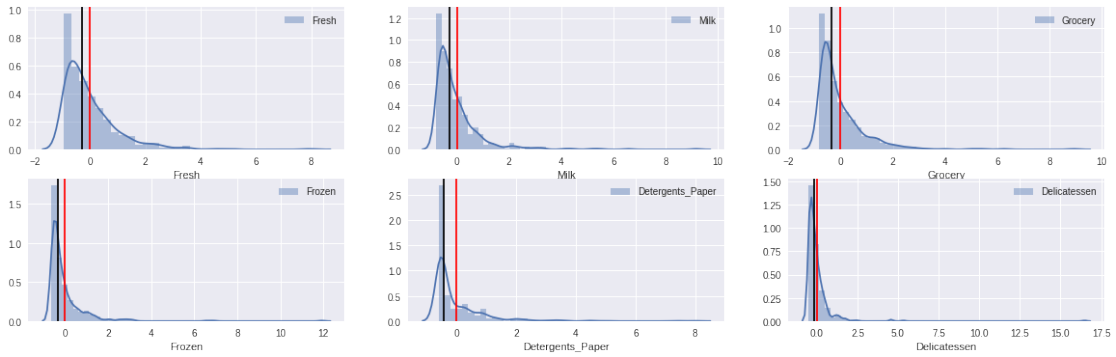
```
In [7]: for col in customer_log_df.columns:
            original_col_skewtest = st.skewtest(customer_features[col])
            scaled_col_skewtest = st.skewtest(customer_sc_df[col])
            original_log_col_skewtest = st.skewtest(customer_log_df[col])
            scaled_log_col_skewtest = st.skewtest(customer_log_sc_df[col])
            print("""{}
            orig:       {}
            scaled:     {}
            orig log:   {}
            scaled log: {}
```

```
            """.format(col,
                       original_col_skewtest,
                       scaled_col_skewtest,
                       original_log_col_skewtest,
                       scaled_log_col_skewtest))
```

```
Fresh
    orig:       SkewtestResult(statistic=13.363200236723891, pvalue=9.920555601203282e-41)
    scaled:     SkewtestResult(statistic=13.363200236723884, pvalue=9.9205556012042771e-41)
    orig log:   SkewtestResult(statistic=-10.102809507315 93, pvalue=5.3681802134267123e-24)
    scaled log: SkewtestResult(statistic=-10.102809507315 93, pvalue=5.3681802134267123e-24)


Milk
    orig:       SkewtestResult(statistic=16.597463367777181, pvalue=7.2698939324027666e-62)
    scaled:     SkewtestResult(statistic=16.597463367777177, pvalue=7.2698939324029738e-62)
    orig log:   SkewtestResult(statistic=-1.9212199962370617, pvalue=0.054703978253562636)
    scaled log: SkewtestResult(statistic=-1.9212199962370418, pvalue=0.054703978253565155)


Grocery
    orig:       SkewtestResult(statistic=15.727788993926845, pvalue=9.7558667104486924e-56)
    scaled:     SkewtestResult(statistic=15.727788993926845, pvalue=9.7558667104486924e-56)
    orig log:   SkewtestResult(statistic=-5.358422081971046, pvalue=8.395192825534008e-08)
    scaled log: SkewtestResult(statistic=-5.3584220819710255, pvalue=8.3951928255349503e-08)


Frozen
    orig:       SkewtestResult(statistic=19.301887122446967, pvalue=5.1783108829290055e-83)
    scaled:     SkewtestResult(statistic=19.301887122446967, pvalue=5.1783108829290055e-83)
    orig log:   SkewtestResult(statistic=-2.9755954096465325, pvalue=0.0029242037232686466)
    scaled log: SkewtestResult(statistic=-2.9755954096465387, pvalue=0.0029242037232685837)


Detergents_Paper
    orig:       SkewtestResult(statistic=15.815137437204189, pvalue=2.4467517316177991e-56)
    scaled:     SkewtestResult(statistic=15.815137437204193, pvalue=2.44675173161759e-56)
    orig log:   SkewtestResult(statistic=-2.0207555302417473, pvalue=0.043305078313325168)
    scaled log: SkewtestResult(statistic=-2.0207555302417313, pvalue=0.043305078313325168)


Delicatessen
    orig:       SkewtestResult(statistic=23.905153021167603, pvalue=2.707223345502464e-126)
    scaled:     SkewtestResult(statistic=23.90515302116761, pvalue=2.7072233455021563e-126)
    orig log:   SkewtestResult(statistic=-7.8572860996062319, pvalue=3.9254555181798112e-15)
    scaled log: SkewtestResult(statistic=-7.8572860996062319, pvalue=3.9254555181798112e-15)
```

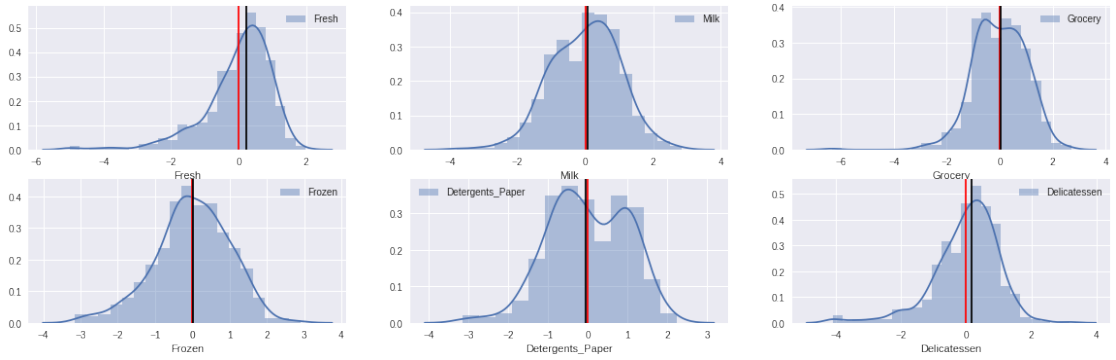
```
In [8]: fig = plt.figure(figsize=(20,6))
        for i, col in enumerate(customer_sc_df.columns):
            fig.add_subplot(231+i)
            sns.distplot(customer_sc_df[col], label=col)
            plt.axvline(customer_sc_df[col].mean(), c='red')
            plt.axvline(customer_sc_df[col].median(), c='black')
            plt.legend()
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```




```
In [9]: fig = plt.figure(figsize=(20,6))
        for i, col in enumerate(customer_log_sc_df.columns):
            fig.add_subplot(231+i)
            sns.distplot(customer_log_sc_df[col], label=col)
            plt.axvline(customer_log_sc_df[col].mean(), c='red')
            plt.axvline(customer_log_sc_df[col].median(), c='black')
            plt.legend()
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



4.8.2 Deskew by Box-Cox Test

The box cox test works by identifying the optimum power, λ to raise the data where

$$\mathbf{x_i}' = \frac{\mathbf{x_i}^{\lambda} - 1}{\lambda}$$

The implementation in Python is

```
y = (x**lmbda - 1) / lmbda,  for lmbda > 0
    log(x),                  for lmbda = 0
```

`boxcox` requires the input data to be positive.

```
In [10]: customer_box_cox_df = pd.DataFrame()
         for col in customer_features.columns:
             box_cox_trans = st.boxcox(customer_features[col])[0]
             customer_box_cox_df[col] = pd.Series(box_cox_trans)
```

```
In [11]: scaler.fit(customer_box_cox_df)
         customer_box_cox_sc = scaler.transform(customer_box_cox_df)
         customer_box_cox_sc_df = pd.DataFrame(customer_box_cox_sc, columns=customer_features.co
```

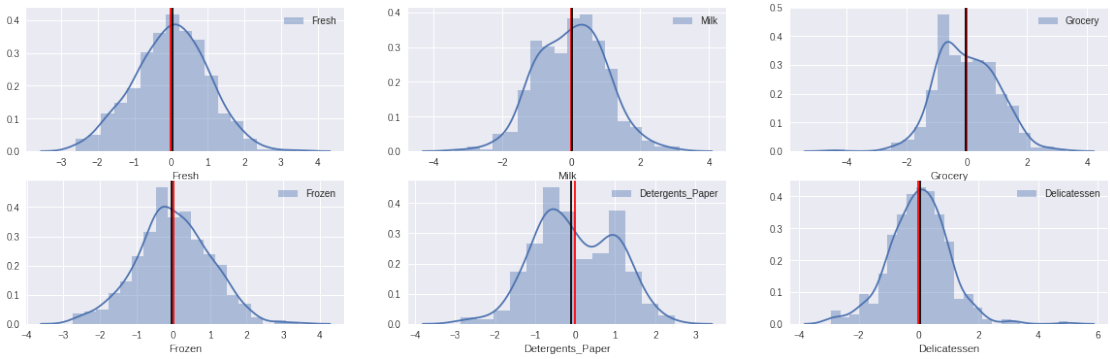
```
In [12]: for col in customer_log_df.columns:
             original_col_skewtest = st.skewtest(customer_features[col])
             scaled_col_skewtest = st.skewtest(customer_sc_df[col])
             original_log_col_skewtest = st.skewtest(customer_log_df[col])
             scaled_log_col_skewtest = st.skewtest(customer_log_sc_df[col])
             original_box_cox_col_skewtest = st.skewtest(customer_box_cox_df[col])
             scaled_box_cox_col_skewtest = st.skewtest(customer_box_cox_sc_df[col])
             print("""{}
             orig:           {}
             scaled:         {}
             orig log:       {}
             scaled log:     {}
             orig box-cox:   {}
             scaled box-cox: {}

             """.format(col,
                        original_col_skewtest,
                        scaled_col_skewtest,
                        original_log_col_skewtest,
                        scaled_log_col_skewtest,
                        original_box_cox_col_skewtest,
                        scaled_box_cox_col_skewtest))
```

```
Fresh
    orig:           SkewtestResult(statistic=13.363200236723891, pvalue=9.920555601203282e-41)
    scaled:         SkewtestResult(statistic=13.363200236723884, pvalue=9.9205556012042771e-41)
    orig log:       SkewtestResult(statistic=-10.10280950731593, pvalue=5.3681802134267123e-24)
    scaled log:     SkewtestResult(statistic=-10.10280950731593, pvalue=5.3681802134267123e-24)
    orig box-cox:   SkewtestResult(statistic=-0.35994737974045543, pvalue=0.71888648448291859)
    scaled box-cox: SkewtestResult(statistic=-0.35994737974044694, pvalue=0.71888648448292491)


Milk
    orig:           SkewtestResult(statistic=16.597463367777181, pvalue=7.2698939324027666e-62)
    scaled:         SkewtestResult(statistic=16.597463367777177, pvalue=7.2698939324029738e-62)
    orig log:       SkewtestResult(statistic=-1.9212199962370617, pvalue=0.054703978253562636)
    scaled log:     SkewtestResult(statistic=-1.9212199962370418, pvalue=0.054703978253565155)
    orig box-cox:   SkewtestResult(statistic=0.0130277609362587, pvalue=0.98960564471404966)
    scaled box-cox: SkewtestResult(statistic=0.013027760936236086, pvalue=0.98960564471406764)


Grocery
    orig:           SkewtestResult(statistic=15.727788993926845, pvalue=9.7558667104486924e-56)
    scaled:         SkewtestResult(statistic=15.727788993926845, pvalue=9.7558667104486924e-56)
    orig log:       SkewtestResult(statistic=-5.358422081971046, pvalue=8.395192825534008e-08)
    scaled log:     SkewtestResult(statistic=-5.3584220819710255, pvalue=8.3951928255349503e-08)
    orig box-cox:   SkewtestResult(statistic=0.30768956032739458, pvalue=0.75831856425784494)
    scaled box-cox: SkewtestResult(statistic=0.30768956032738215, pvalue=0.75831856425785427)


Frozen
    orig:           SkewtestResult(statistic=19.301887122446967, pvalue=5.1783108829290055e-83)
    scaled:         SkewtestResult(statistic=19.301887122446967, pvalue=5.1783108829290055e-83)
    orig log:       SkewtestResult(statistic=-2.9755954096465325, pvalue=0.0029242037232686466)
    scaled log:     SkewtestResult(statistic=-2.9755954096465387, pvalue=0.0029242037232685837)
    orig box-cox:   SkewtestResult(statistic=0.025747761582376901, pvalue=0.9794585282352174)
    scaled box-cox: SkewtestResult(statistic=0.025747761582365615, pvalue=0.97945852823522639)
```

```
Detergents_Paper
    orig:           SkewtestResult(statistic=15.815137437204189, pvalue=2.4467517316177991e-56)
    scaled:         SkewtestResult(statistic=15.815137437204193, pvalue=2.44675173161759e-56)
    orig log:       SkewtestResult(statistic=-2.0207555302417473, pvalue=0.043305078313332351)
    scaled log:     SkewtestResult(statistic=-2.0207555302417313, pvalue=0.043305078313325168)
    orig box-cox:   SkewtestResult(statistic=-0.1501016224622185, pvalue=0.88068443994892309)
    scaled box-cox: SkewtestResult(statistic=-0.15010162246221934, pvalue=0.88068443994892243)


Delicatessen
    orig:           SkewtestResult(statistic=23.905153021167603, pvalue=2.707223345502464e-126)
    scaled:         SkewtestResult(statistic=23.90515302116761, pvalue=2.7072233455021563e-126)
    orig log:       SkewtestResult(statistic=-7.8572860996062319, pvalue=3.9254555181798112e-15)
    scaled log:     SkewtestResult(statistic=-7.8572860996062319, pvalue=3.9254555181798112e-15)
    orig box-cox:   SkewtestResult(statistic=0.90677255021156078, pvalue=0.36452708858901794)
    scaled box-cox: SkewtestResult(statistic=0.90677255021155789, pvalue=0.3645270885890195)
```

```
In [13]: fig = plt.figure(figsize=(20,6))
         for i, col in enumerate(customer_box_cox_sc_df.columns):
             fig.add_subplot(231+i)
             sns.distplot(customer_box_cox_sc_df[col], label=col)
             plt.axvline(customer_box_cox_sc_df[col].mean(), c='red')
             plt.axvline(customer_box_cox_sc_df[col].median(), c='black')
             plt.legend()
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py:6462: UserWarning: The 'normed'
  warnings.warn("The 'normed' kwarg is deprecated, and has been "
```



4.9 Identifying and Removing Outliers

```
In [3]: %run load_data.py
```

```
In [4]: %matplotlib inline
```

To identify outliers in the data, we will use what is the Tukey Method.

- leverages the Interquartile Range
- isn't dependent on distributional assumptions
- ignores the mean and standard deviation
- making it resistant to being influenced by the extreme values in the range

Tukey's Method: look for points that are more than 1.5 times the Inter-quartile range above the third quartile or below the first quartile.

```
In [8]: def feature_outliers(dataframe, col, param=1.5):
            Q1 = np.percentile(dataframe[col], 25)
            Q3 = np.percentile(dataframe[col], 75)
            tukey_window = param*(Q3-Q1)
            less_than_Q1 = dataframe[col] < Q1 - tukey_window
        #     print(less_than_Q1)
            greater_than_Q3 = dataframe[col] > Q3 + tukey_window
            tukey_mask = (less_than_Q1 | greater_than_Q3)
            return dataframe.loc[tukey_mask]
```

```
In [11]: feature_outliers(customer_features, 'Grocery')
```

Out[11]:

	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicatessen
28	4113	20484	25957	1158	8604	5206
43	630	11095	23998	787	9529	72
47	44466	54259	55571	7782	24171	6465
49	4967	21412	28921	1798	13583	1163
56	4098	29892	26866	2616	17740	1340
61	35942	38369	59598	3254	26701	2017
65	85	20959	45828	36	24231	1423
77	12205	12697	28540	869	12034	1009
85	16117	46197	92780	1026	40827	2944
86	22925	73498	32114	987	20070	903
92	9198	27472	32034	3232	18906	5130
109	1406	16729	28986	673	836	3
145	22039	8384	34792	42	12591	4430
163	5531	15726	26870	2367	13726	446
201	4484	14399	24708	3549	14235	1681
205	1107	11711	23596	955	9265	710
211	12119	28326	39694	4736	19410	2870
216	2532	16599	36486	179	13308	674
251	6134	23133	33586	6746	18594	5121
304	161	7460	24773	617	11783	2410
331	11223	14881	26839	1234	9606	1102
333	8565	4980	67298	131	38102	1215
343	1689	6964	26316	1456	15469	37
437	14531	15488	30243	437	14841	1867

```
In [13]: for col in customer_log_sc_df:
             print(col, feature_outliers(customer_features, col).shape)
```

```
Fresh (20, 6)
Milk (28, 6)
Grocery (24, 6)
Frozen (43, 6)
Detergents_Paper (30, 6)
Delicatessen (27, 6)
```

What if we count the rows that show up as an outlier more than once?

```
In [14]: from collections import Counter
```

```
In [17]: def multiple_outliers(dataframe, count=2):
             raw_outliers = []
             for col in dataframe:
                 outlier_df = feature_outliers(dataframe, col)
                 raw_outliers += list(outlier_df.index)

             outlier_count = Counter(raw_outliers)
```

```
            outliers = [k for k,v in outlier_count.items() if v >= count]
            return outliers
```

```
In [18]: len(multiple_outliers(customer_features))
```

```
Out[18]: 41
```

```
In [19]: len(multiple_outliers(customer_sc_df))
```

```
Out[19]: 41
```

```
In [20]: len(multiple_outliers(customer_log_sc_df))
```

```
Out[20]: 5
```

```
In [21]: len(multiple_outliers(customer_box_cox_sc_df))
```

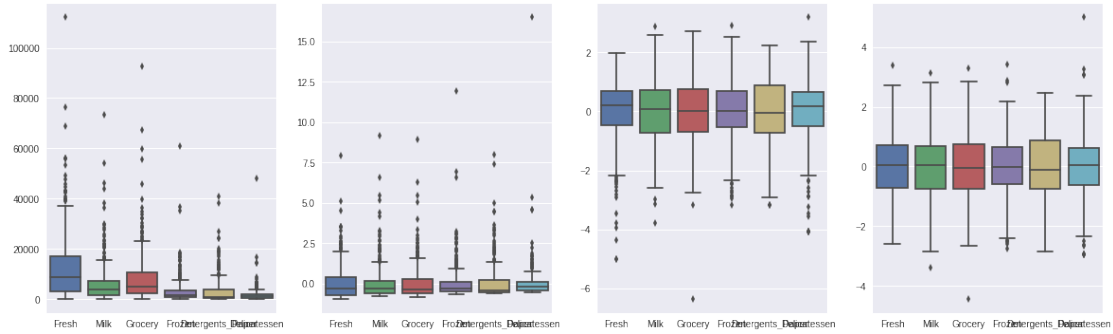
```
Out[21]: 2
```

```
In [11]: customer_log_sc_df.shape
```

```
Out[11]: (440, 6)
```

```
In [26]: _, ax = plt.subplots(1,4,figsize=(20,6))

         for i, df in enumerate([customer_features, customer_sc_df, customer_log_sc_df, customer
             sns.boxplot(df, ax=ax[i])
```

```
/opt/conda/lib/python3.6/site-packages/seaborn/categorical.py:2171: UserWarning: The boxplot API
  warnings.warn(msg, UserWarning)
```




```
In [22]: customer_features_outliers_removed = customer_features.drop(multiple_outliers(customer_
         customer_sc_df_outliers_removed = customer_sc_df.drop(multiple_outliers(customer_sc_df)
         customer_log_sc_df_outliers_removed = customer_log_sc_df.drop(multiple_outliers(custome
         customer_box_cox_sc_df_outliers_removed = customer_box_cox_sc_df.drop(multiple_outliers
```

```
In [23]: (customer_features_outliers_removed.shape,
          customer_sc_df_outliers_removed.shape,
          customer_log_sc_df_outliers_removed.shape,
          customer_box_cox_sc_df_outliers_removed.shape)
```

```
Out[23]: ((399, 6), (399, 6), (435, 6), (438, 6))
```

4.10 Principal Component Analysis

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

```
%matplotlib inline
from IPython.display import display

from sklearn.preprocessing import StandardScaler
import scipy.stats as st

customers = pd.read_csv('Wholesale_customers_data.csv')
customers.Region = customers.Region.astype('category')
customers.Channel = customers.Channel.astype('category')
customer_features = customers.select_dtypes([int])

scaler = StandardScaler()
customer_sc = scaler.fit_transform(customer_features)
customer_sc_df = pd.DataFrame(customer_sc, columns=customer_features.columns)

customer_log_df = np.log(1+customer_features)
scaler.fit(customer_log_df)
customer_log_sc = scaler.transform(customer_log_df)
customer_log_sc_df = pd.DataFrame(customer_log_sc, columns=customer_features.columns)

customer_box_cox_df = pd.DataFrame()
for col in customer_features.columns:
    box_cox_trans = st.boxcox(customer_features[col])[0]
    customer_box_cox_df[col] = pd.Series(box_cox_trans)

scaler.fit(customer_box_cox_df)
customer_box_cox_sc = scaler.transform(customer_box_cox_df)
customer_box_cox_sc_df = pd.DataFrame(customer_box_cox_sc, columns=customer_features.col
```

In [2]:
```
def feature_outliers(dataframe, col, param=1.5):
    Q1 = np.percentile(dataframe[col], 25)
    Q3 = np.percentile(dataframe[col], 75)
    tukey_window = param*(Q3-Q1)
    less_than_Q1 = dataframe[col] < Q1 - tukey_window
    greater_than_Q3 = dataframe[col] > Q3 + tukey_window
    tukey_mask = (less_than_Q1 | greater_than_Q3)
    return dataframe[tukey_mask]
```

In [3]:
```
from collections import Counter
```

In [4]:
```
def multiple_outliers(dataframe, count=2):
    raw_outliers = []
    for col in dataframe:
        outlier_df = feature_outliers(dataframe, col)
        raw_outliers += list(outlier_df.index)

    outlier_count = Counter(raw_outliers)
    outliers = [k for k,v in outlier_count.items() if v >= count]
    return outliers
```

In [5]:
```
customer_features_outliers_removed = customer_features.drop(multiple_outliers(customer_f
customer_sc_df_outliers_removed = customer_sc_df.drop(multiple_outliers(customer_sc_df))
customer_log_sc_df_outliers_removed = customer_log_sc_df.drop(multiple_outliers(customer
customer_box_cox_sc_df_outliers_removed = customer_box_cox_sc_df.drop(multiple_outliers(
```

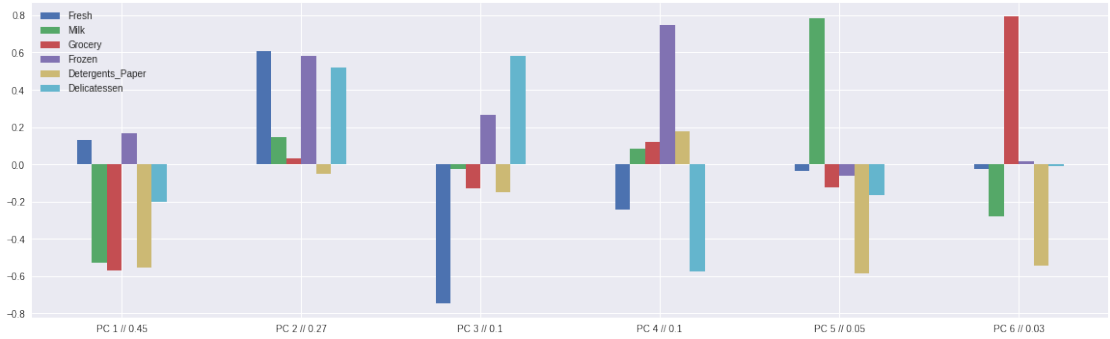
In [6]:
```
(customer_features.shape,
 customer_sc_df.shape,
 customer_log_sc_df.shape,
 customer_box_cox_sc_df.shape,
 customer_features_outliers_removed.shape,
 customer_sc_df_outliers_removed.shape,
```

```
          customer_log_sc_df_outliers_removed.shape,
          customer_box_cox_sc_df_outliers_removed.shape)
```

Out[6]:
```
((440, 6),
 (440, 6),
 (440, 6),
 (440, 6),
 (399, 6),
 (399, 6),
 (435, 6),
 (438, 6))
```

In [7]:
```
from sklearn.decomposition import PCA
```

In [8]:
```
pca_box_cox = PCA()
pca_box_cox.fit(customer_box_cox_sc_df_outliers_removed)

principal_component_loadings_box_cox = \
 pd.DataFrame(pca_box_cox.components_,
 columns=customer_box_cox_sc_df_outliers_removed.columns)

explained_variance_ratio_box_cox = pca_box_cox.explained_variance_ratio_
pca_labels_box_cox = ['PC ' + str(i+1) +' // ' + str(round(ratio,2)) for i, ratio in enu

ax = principal_component_loadings_box_cox.plot(kind='bar', figsize=(20,6), rot=0)
ax.set_xticklabels(pca_labels_box_cox);
```



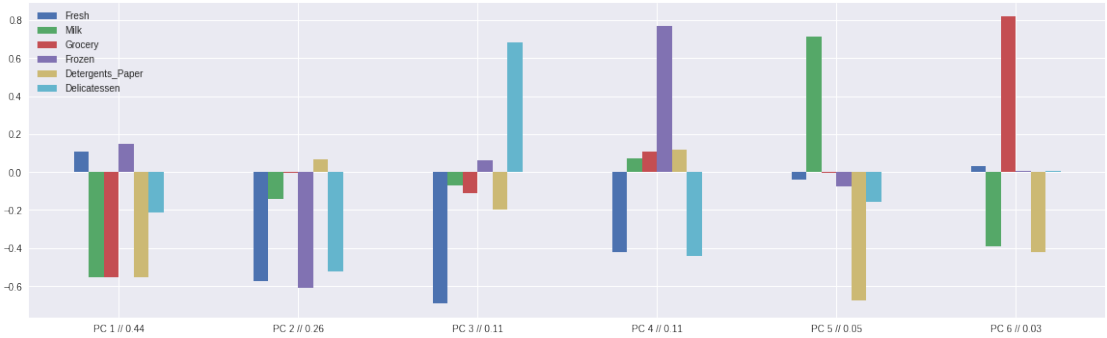
In [21]:
```
pca_log = PCA()
pca_log.fit(customer_log_sc_df_outliers_removed)
```

Out[21]:
```
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

In [22]:
```
principal_component_loadings_log = pd.DataFrame(pca_log.components_, columns=customer_l
```

In [23]:
```
explained_variance_ratio_log = pca_log.explained_variance_ratio_
pca_labels_log = ['PC ' + str(i+1) +' // ' + str(round(ratio,2)) for i, ratio in enumer
```

In [24]:
```
ax = principal_component_loadings_log.plot(kind='bar', figsize=(20,6), rot=0)
ax.set_xticklabels(pca_labels_log);
```



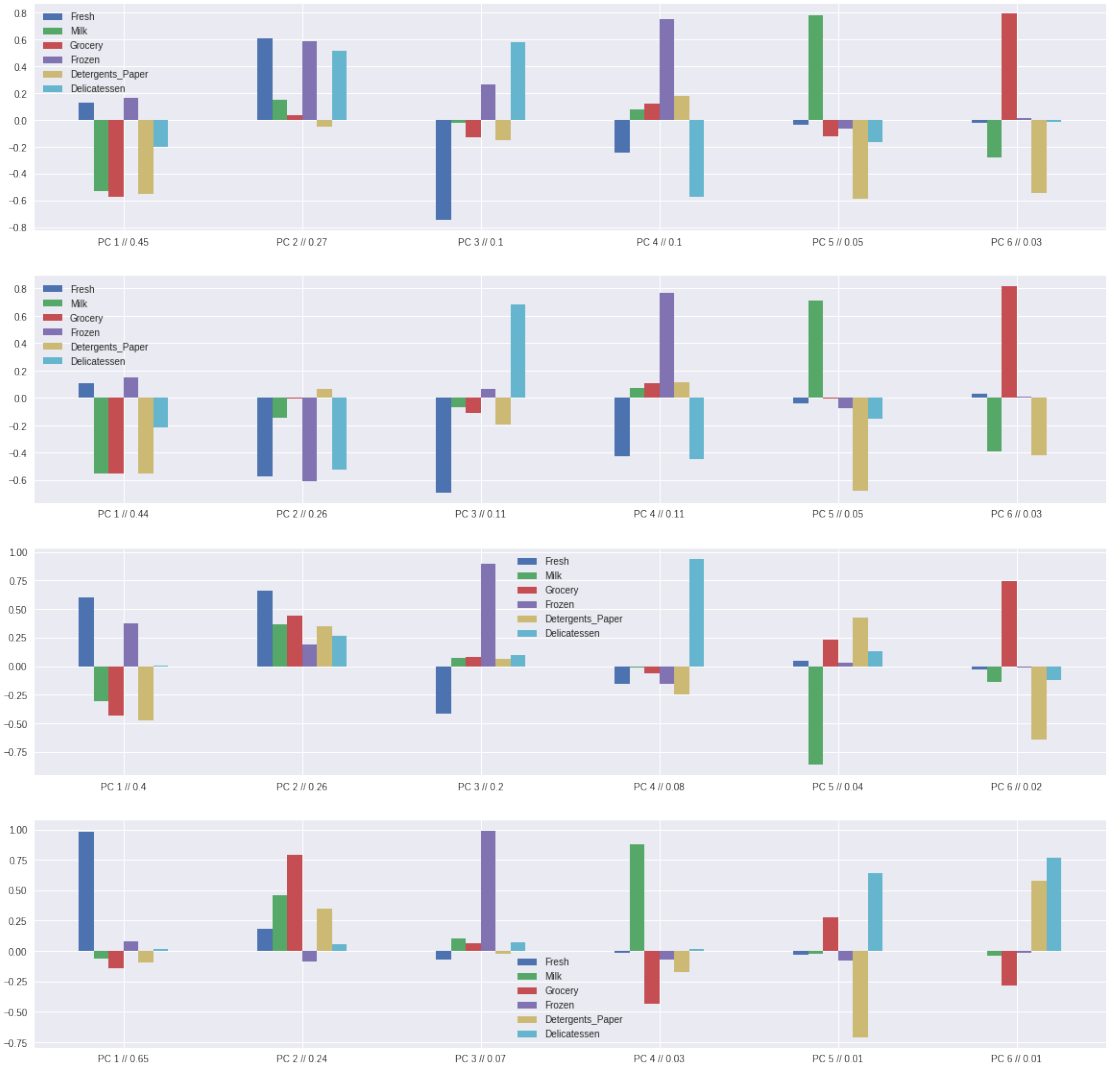

```
In [25]: pca_orig = PCA()
         pca_orig.fit(customer_features_outliers_removed)
         pca_scaled = PCA()
         pca_scaled.fit(customer_sc_df_outliers_removed)
```

```
Out[25]: PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
           svd_solver='auto', tol=0.0, whiten=False)
```

```
In [26]: principal_component_loadings_orig = pd.DataFrame(pca_orig.components_, columns=customer
         principal_component_loadings_scaled = pd.DataFrame(pca_scaled.components_, columns=cust
```

```
In [27]: explained_variance_ratio_orig = pca_orig.explained_variance_ratio_
         explained_variance_ratio_scaled = pca_scaled.explained_variance_ratio_
         pca_labels_orig = ['PC ' + str(i+1) +' // ' + str(round(ratio,2)) for i, ratio in enume
         pca_labels_scaled = ['PC ' + str(i+1) +' // ' + str(round(ratio,2)) for i, ratio in enu
```

```
In [28]: _, ax = plt.subplots(4,1,figsize=(20,20))
         principal_component_loadings_box_cox.plot(kind='bar', rot=0, ax=ax[0])
         ax[0].set_xticklabels(pca_labels_box_cox)
         principal_component_loadings_log.plot(kind='bar', rot=0, ax=ax[1])
         ax[1].set_xticklabels(pca_labels_log)
         principal_component_loadings_scaled.plot(kind='bar', rot=0, ax=ax[2])
         ax[2].set_xticklabels(pca_labels_scaled)
         principal_component_loadings_orig.plot(kind='bar', rot=0, ax=ax[3])
         ax[3].set_xticklabels(pca_labels_orig);
```

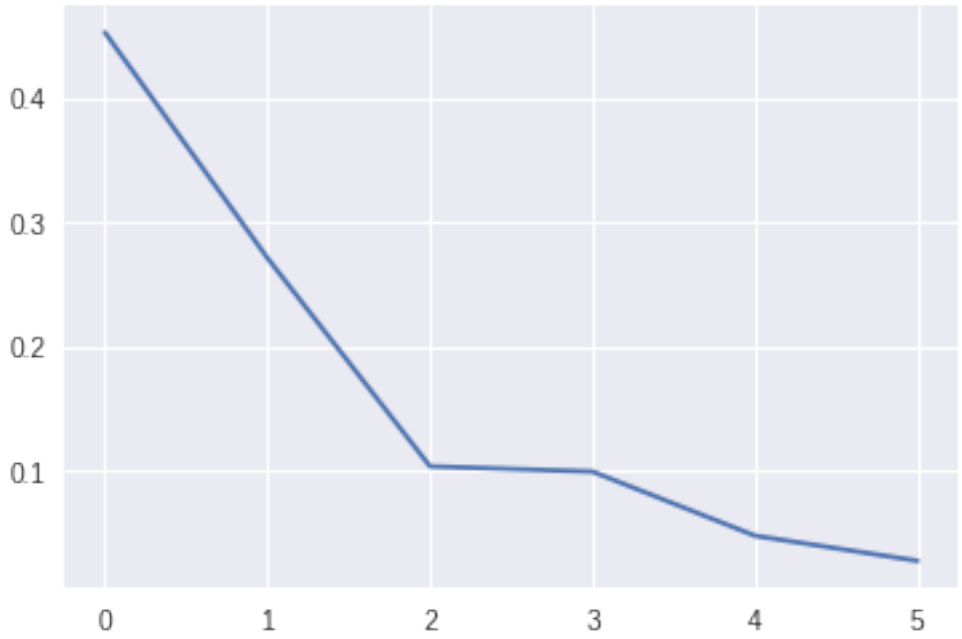


4.10.1 Scree Plot

Look for "the bend". This will tell you how many components to keep.

```
In [34]: plt.plot(pca_box_cox.explained_variance_ratio_)
```

```
Out[34]: [<matplotlib.lines.Line2D at 0x7fdaecd7e208>]
```




```
In [35]: pca = PCA(2)
         pca.fit(customer_box_cox_sc_df_outliers_removed)
```

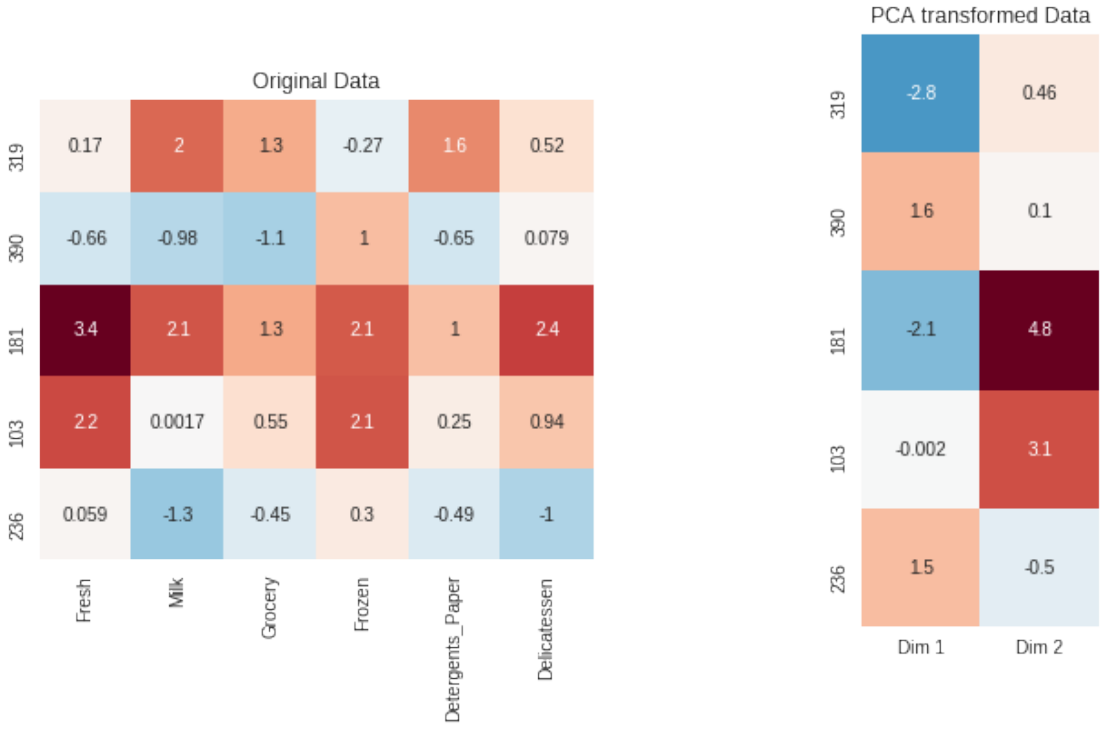
```
Out[35]: PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
           svd_solver='auto', tol=0.0, whiten=False)
```

```
In [36]: sample = customer_box_cox_sc_df_outliers_removed.sample(5)
```

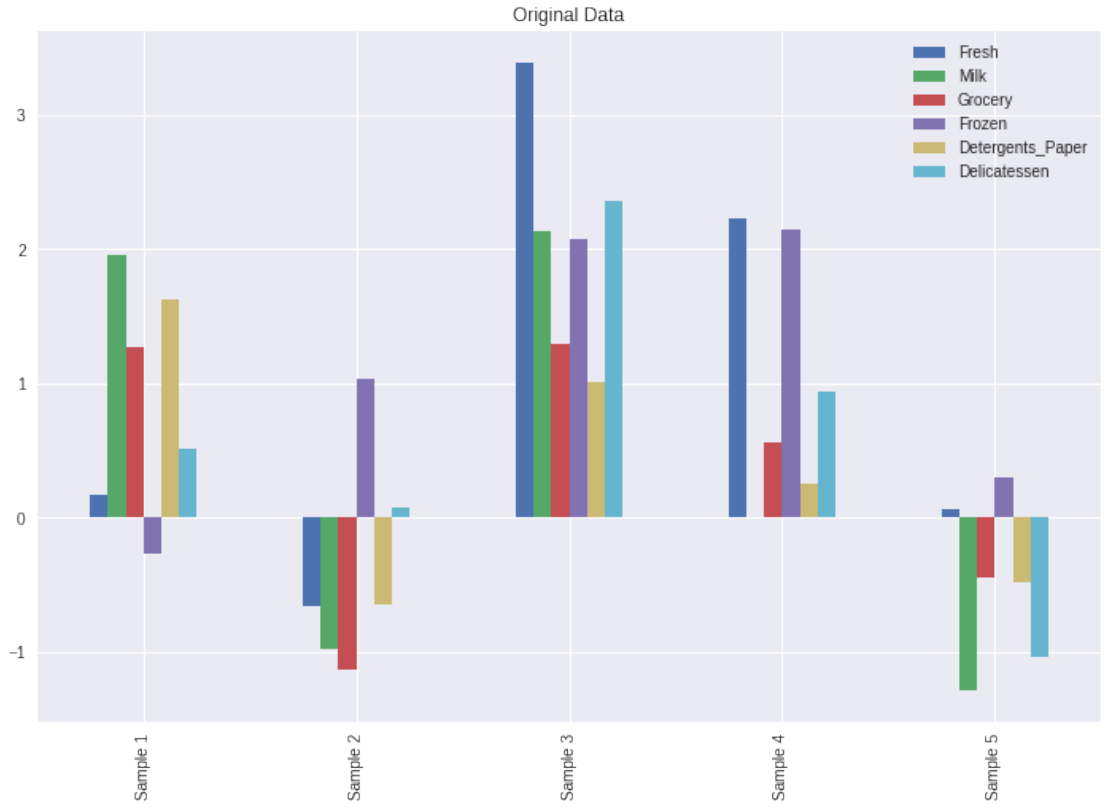
```
In [37]: customer_pca_df = pd.DataFrame(pca.transform(customer_box_cox_sc_df),
                                        columns=['Dim 1', 'Dim 2'],
                                        index=customer_box_cox_sc_df.index)
         sample_pca_df = pd.DataFrame(pca.transform(sample),
                                      columns=['Dim 1', 'Dim 2'],
                                      index=sample.index)
```

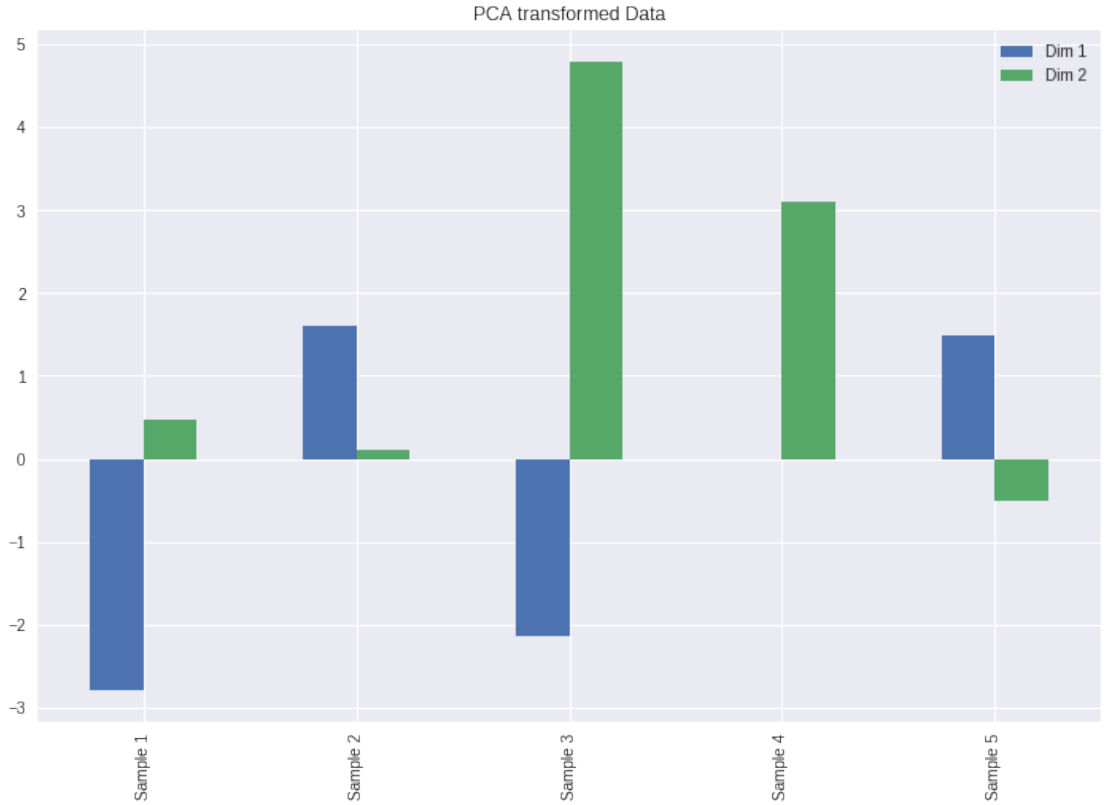
```
In [38]: fig = plt.figure(figsize=(12,6))
         fig.add_subplot(121)
         plt.title("Original Data")
         sns.heatmap(sample, annot=True, cbar=False, square=True)
         fig.add_subplot(122)
         plt.title("PCA transformed Data")
         sns.heatmap(sample_pca_df, annot=True, cbar=False, square=True)
```

```
Out[38]: <matplotlib.axes._subplots.AxesSubplot at 0x7fdaecd11f98>
```



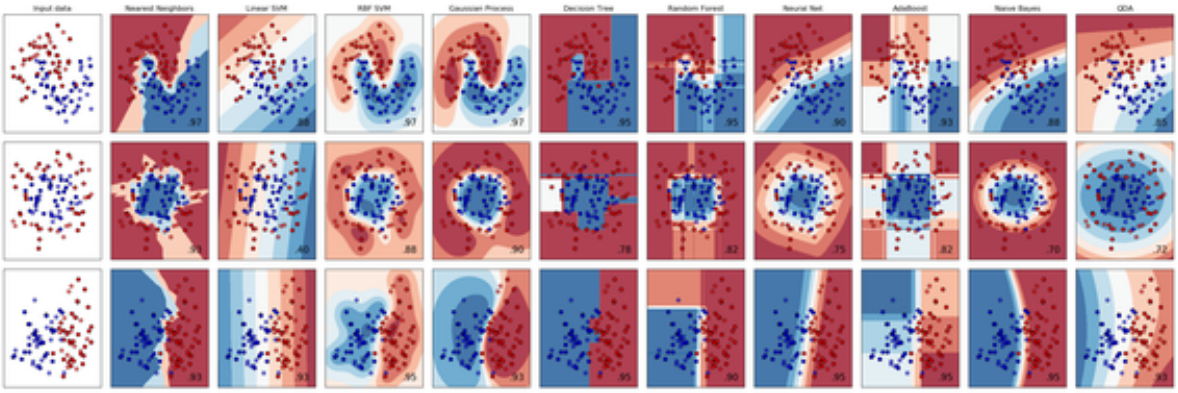

```
In [39]: sample.plot(kind='bar', figsize=(12,8))
         plt.title("Original Data")
         _ = plt.xticks(range(5),['Sample 1','Sample 2','Sample 3', 'Sample 4','Sample 5'])

         sample_pca_df.plot(kind='bar', figsize=(12,8))
         plt.title("PCA transformed Data")
         _ = plt.xticks(range(5),['Sample 1','Sample 2','Sample 3', 'Sample 4','Sample 5'])
```


Original Data
Fresh
Milk
Grocery
Frozen
Detergents_Paper
Delicatessen
3
2
1
0
−1
Sample 1
Sample 2
Sample 3
Sample 4
Sample 5


PCA transformed Data
Dim 1
Dim 2
5
4
3
2
1
0
−1
−2
−3
Sample 1
Sample 2
Sample 3
Sample 4
Sample 5

4.11 scikit-learn

Machine Learning in Python



4.11.1 Estimator Objects

Fitting data: the main API implemented by scikit-learn is that of the estimator. An estimator is any object that learns from data; it may be a

- a **predictor**
  - classification algorithm
  - regression algorithm
  - clustering algorithm
- a **transformer** that extracts/filters useful features from raw data

All estimator objects expose a fit method that takes a dataset (usually a 2-d array):

```
estimator.fit(data)
```

Estimator parameters: All the parameters of an estimator can be set when it is instantiated or by modifying the corresponding attribute:

```
estimator = Estimator(param1=1, param2=2)
estimator.param1
```

Estimated parameters: When data is fitted with an estimator, parameters are estimated from the data at hand. All the estimated parameters are attributes of the estimator object ending by an underscore:

```
estimator.estimated_param_
```

4.11.2 the sklearn API

```
In [1]: from sklearn import datasets

In [2]: IRIS = datasets.load_iris()

In [3]: from sklearn.linear_model import LogisticRegression
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.neighbors import KNeighborsClassifier
```

```
        from sklearn.svm import SVC

        from sklearn.preprocessing import StandardScaler
```

```
In [4]: scaler = StandardScaler()
        scaler.fit(IRIS.data)
        X = scaler.transform(IRIS.data)
        y = IRIS.target
```

```
In [5]: model = LogisticRegression()
        model.fit(X, y)
        model.score(X, y)
```

```
Out[5]: 0.92666666666666664
```

```
In [6]: model = DecisionTreeClassifier()
        model.fit(X, y)
        model.score(X, y)
```

```
Out[6]: 1.0
```

```
In [7]: model = KNeighborsClassifier()
        model.fit(X, y)
        model.score(X, y)
```

```
Out[7]: 0.95333333333333337
```

```
In [8]: model = SVC()
        model.fit(X, y)
        model.score(X, y)
```

```
Out[8]: 0.97333333333333338
```

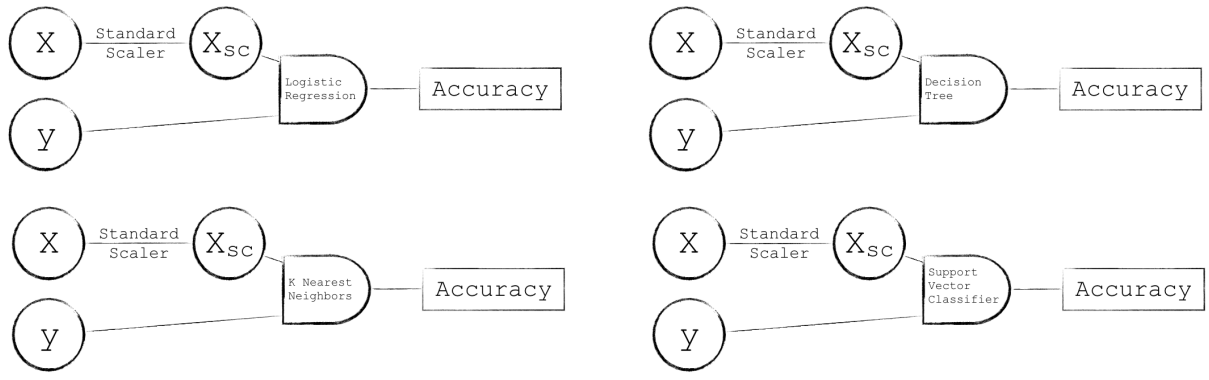
```
In [10]: model = SVC()
         model.fit(X, y)
         predictions = model.predict(X)
```

```
In [20]: from sklearn.metrics import f1_score
```

```
In [21]: f1_score(y, predictions, average='micro')
```

```
Out[21]: 0.97333333333333338
```

4.11.3 Machine Learning Pipelines



4.12 What to Know About PCA

```
In [1]: run load_data.py

In [2]: from sklearn.decomposition import PCA

In [3]: X = customer_box_cox_sc_df_outliers_removed

        pca = PCA()
        pca.fit(X)
        customer_box_cox_sc_df_outliers_removed_pca = pca.transform(X)
```

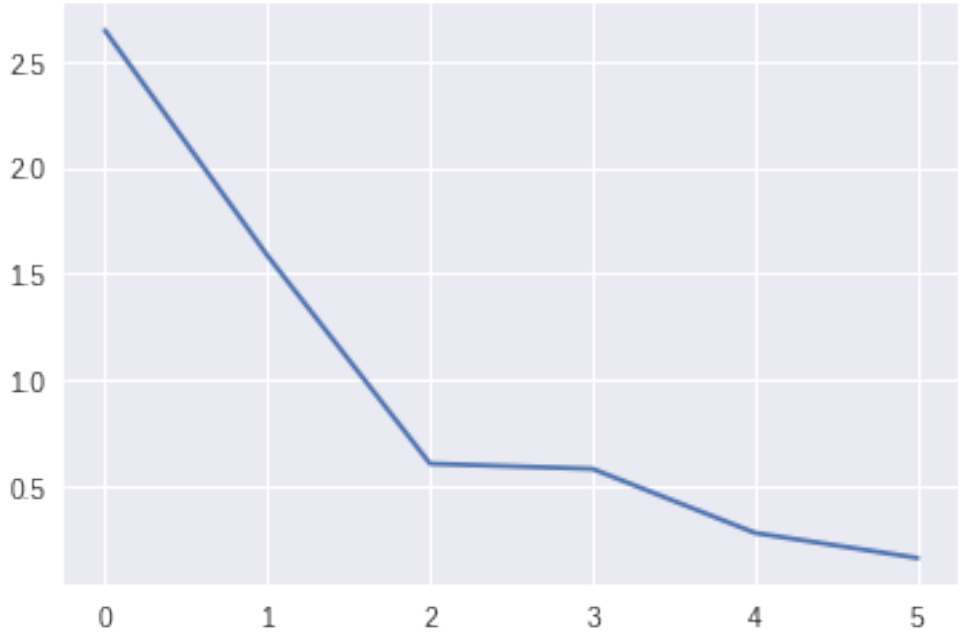
The process of fitting a PCA model gives us three important outputs:

- the **explained variance**, L
- the **loadings**, P
- the **transformed data**, X_p

```
In [4]: L = pca.explained_variance_

In [5]: P = pca.components_
        P_df = pd.DataFrame(P, columns=X.columns)

In [6]: X_p = customer_box_cox_sc_df_outliers_removed_pca
```

4.12.1 Use the Explained Variance to Identify Component Significance

```
In [7]: plt.plot(L)

Out[7]: [<matplotlib.lines.Line2D at 0x7fcd5dc46550>]
```



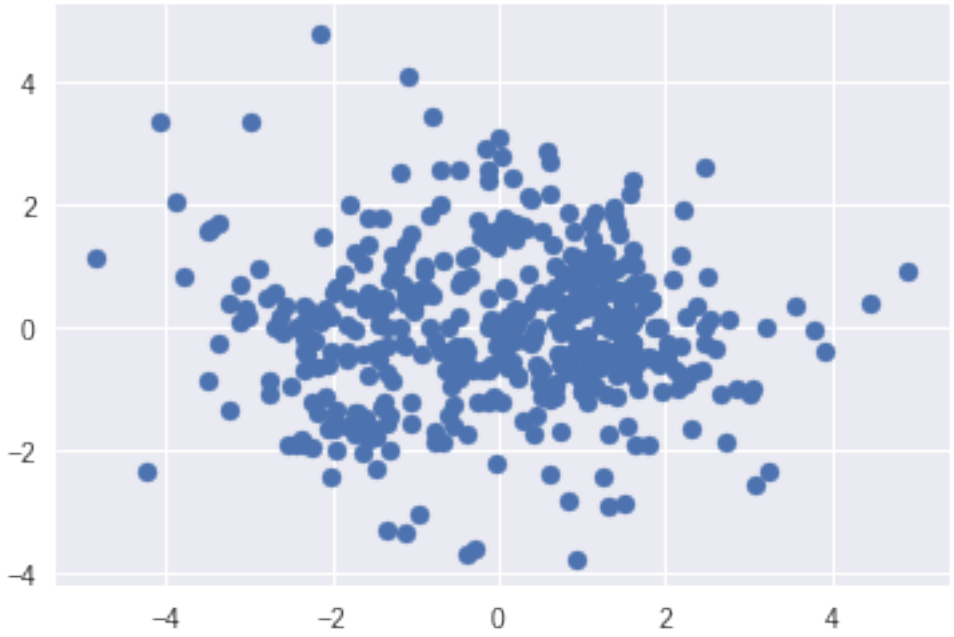
4.12.2 Use the Loadings to Explain the Components

```
In [8]: P_df.plot(kind="bar", figsize=(20,5))
```

```
Out[8]: <matplotlib.axes._subplots.AxesSubplot at 0x7fcd5dc02438>
```

4.12.3 Use the Transformed Data in your ML Pipeline

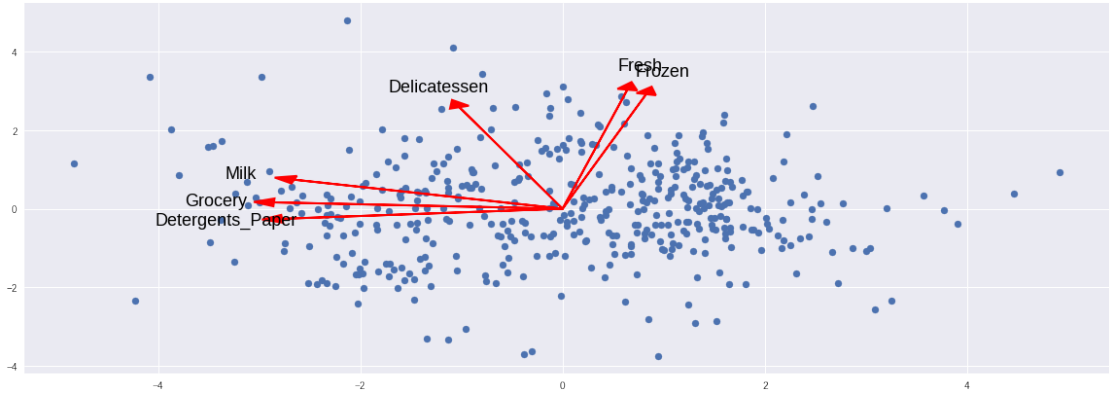
```
In [15]: X = customer_box_cox_sc_df_outliers_removed

         pca = PCA(n_components=2)
         pca.fit(X)
         X_p = pd.DataFrame(pca.transform(X), index=X.index)

In [17]: plt.scatter(X_p[0], X_p[1])

Out[17]: <matplotlib.collections.PathCollection at 0x7f2b68f4f160>
```



4.12.4 One Last Thing to Think About - The Biplot

```
In [18]: feature_vectors = pca.components_.T

In [24]: plt.figure(figsize=(20,7))
         plt.scatter(X_p[0], X_p[1])
         for i, v in enumerate(feature_vectors):
             plt.arrow(0, 0, 5*v[0], 5*v[1],
                       head_width=0.2, head_length=0.2, linewidth=2, color='red')
```

```
plt.text(v[0]*6, v[1]*6, X.columns[i], color='black',
         ha='center', va='center', fontsize=18)
```




```
In [1]: run load_data.py
```

4.13 Evaluating Model Pipelines

We will evaluate a total of 24 model pipelines:

1. the original data
2. the original data with outliers removed
3. the original data transformed by a PCA with 2 components
4. the original data with outliers removed transformed by a PCA with 2 components
5. the original data transformed by a PCA with 3 components
6. the original data with outliers removed transformed by a PCA with 3 components
7. scaled data
8. scaled data with outliers removed
9. scaled data transformed by a PCA with 2 components
10. scaled data with outliers removed transformed by a PCA with 2 components
11. scaled data transformed by a PCA with 3 components
12. scaled data with outliers removed transformed by a PCA with 3 components
13. log transformed, scaled data
14. log transformed, scaled data with outliers removed
15. log transformed, scaled data transformed by a PCA with 2 components
16. log transformed, scaled data with outliers removed transformed by a PCA with 2 components
17. log transformed, scaled data transformed by a PCA with 3 components
18. log transformed, scaled data with outliers removed transformed by a PCA with 3 components
19. box-cox transformed, scaled data
20. box-cox transformed, scaled data with outliers removed
21. box-cox transformed, scaled data transformed by a PCA with 2 components

22. box-cox transformed, scaled data with outliers removed transformed by a PCA with 2 components
23. box-cox transformed, scaled data transformed by a PCA with 3 components
24. box-cox transformed, scaled data with outliers removed transformed by a PCA with 3 components

4.14 Experiment Design

We will pass each of these transformed data sets to a Gaussian Mixture Model and then assess the model using the BIC.

```
In [2]: from sklearn.mixture import GaussianMixture

In [3]: original_data = [
            ('original', customer_features),
            ('original - no outliers', customer_features_outliers_removed),
            ('original - pca, 2 components', customer_features_pca_2),
            ('original - pca, 3 components', customer_features_pca_3),
            ('original - no outliers, pca, 2 components', customer_features_outliers_removed_pca
            ('original - no outliers, pca, 3 components', customer_features_outliers_removed_pca
        ]

        scaled_data = [
            ('scaled', customer_sc),
            ('scaled - no outliers', customer_sc_outliers_removed),
            ('scaled - pca, 2 components', customer_sc_pca_2),
            ('scaled - pca, 3 components', customer_sc_pca_3),
            ('scaled - no outliers, pca, 2 components', customer_sc_outliers_removed_pca_2),
            ('scaled - no outliers, pca, 3 components', customer_sc_outliers_removed_pca_3),
        ]

        log_transformed_data = [
            ('log transformed, scaled', customer_log_sc),
            ('log transformed, scaled - no outliers', customer_log_sc_outliers_removed),
            ('log transformed, scaled - pca, 2 components', customer_log_sc_pca_2),
            ('log transformed, scaled - pca, 3 components', customer_log_sc_pca_3),
            ('log transformed, scaled - no outliers, pca, 2 components', customer_log_sc_outlier
            ('log transformed, scaled - no outliers, pca, 3 components', customer_log_sc_outlier
        ]

        box_cox_transformed_data = [
            ('box-cox transformed, scaled', customer_box_cox_sc),
            ('box-cox transformed, scaled - no outliers', customer_box_cox_sc_outliers_removed),
            ('box-cox transformed, scaled - pca, 2 components', customer_box_cox_sc_pca_2),
            ('box-cox transformed, scaled - pca, 3 components', customer_box_cox_sc_pca_3),
            ('box-cox transformed, scaled - no outliers, pca, 2 components', customer_box_cox_sc
            ('box-cox transformed, scaled - no outliers, pca, 3 components', customer_box_cox_sc
        ]

In [4]: def fit_and_score(data, n_components=2):
            model = GaussianMixture(n_components=n_components)
            model.fit(data)
            return model.bic(data)

In [5]: n = 2

        results_2_clusters = []

        for name, data in original_data:
```

```
        results_2_clusters.append({
            'name' : name,
            'n' : n,
            'BIC' : fit_and_score(data, n)
        })

    for name, data in scaled_data:
        results_2_clusters.append({
            'name' : name,
            'n' : n,
            'BIC' : fit_and_score(data, n)
        })

    for name, data in log_transformed_data:
        results_2_clusters.append({
            'name' : name,
            'n' : n,
            'BIC' : fit_and_score(data, n)
        })

    for name, data in box_cox_transformed_data:
        results_2_clusters.append({
            'name' : name,
            'n' : n,
            'BIC' : fit_and_score(data, n)
        })
```

In [8]:
```
pd.DataFrame(results_2_clusters).sort_values('BIC')
```

Out[8]:

	BIC	n	name
10	1885.559367	2	scaled - no outliers, pca, 2 components
11	2412.065861	2	scaled - no outliers, pca, 3 components
7	2448.476967	2	scaled - no outliers
8	2586.743957	2	scaled - pca, 2 components
16	3042.170333	2	log transformed, scaled - no outliers, pca, 2 ...
22	3135.999109	2	box-cox transformed, scaled - no outliers, pca...
14	3154.867206	2	log transformed, scaled - pca, 2 components
20	3185.304232	2	box-cox transformed, scaled - pca, 2 components
9	3415.376180	2	scaled - pca, 3 components
17	4060.888354	2	log transformed, scaled - no outliers, pca, 3 ...
15	4170.830084	2	log transformed, scaled - pca, 3 components
23	4201.912396	2	box-cox transformed, scaled - no outliers, pca...
21	4258.879750	2	box-cox transformed, scaled - pca, 3 components
6	5818.549199	2	scaled
13	6284.272757	2	log transformed, scaled - no outliers
19	6323.642586	2	box-cox transformed, scaled - no outliers
18	6395.343130	2	box-cox transformed, scaled
12	6434.366911	2	log transformed, scaled
4	16482.713328	2	original - no outliers, pca, 2 components
2	18695.306294	2	original - pca, 2 components
5	23814.617363	2	original - no outliers, pca, 3 components
3	27114.512185	2	original - pca, 3 components
1	44268.638982	2	original - no outliers
0	50650.969076	2	original

In [9]:
```
n = 3

results_3_clusters = []
```

```
    for name, data in original_data:
        results_3_clusters.append({
            'name' : name,
            'n' : n,
            'BIC' : fit_and_score(data, n)
        })

    for name, data in scaled_data:
        results_3_clusters.append({
            'name' : name,
            'n' : n,
            'BIC' : fit_and_score(data, n)
        })

    for name, data in log_transformed_data:
        results_3_clusters.append({
            'name' : name,
            'n' : n,
            'BIC' : fit_and_score(data, n)
        })

    for name, data in box_cox_transformed_data:
        results_3_clusters.append({
            'name' : name,
            'n' : n,
            'BIC' : fit_and_score(data, n)
        })
```

```
In [10]: pd.DataFrame(results_3_clusters).sort_values('BIC')
```

Out[10]:

	BIC	n	name
10	1727.215879	3	scaled - no outliers, pca, 2 components
7	2255.218493	3	scaled - no outliers
11	2257.444963	3	scaled - no outliers, pca, 3 components
8	2474.559794	3	scaled - pca, 2 components
16	3062.346233	3	log transformed, scaled - no outliers, pca, 2 ...
14	3157.549693	3	log transformed, scaled - pca, 2 components
22	3160.667316	3	box-cox transformed, scaled - no outliers, pca...
20	3188.782693	3	box-cox transformed, scaled - pca, 2 components
9	3359.809511	3	scaled - pca, 3 components
6	3660.361663	3	scaled
17	4114.867543	3	log transformed, scaled - no outliers, pca, 3 ...
15	4241.780549	3	log transformed, scaled - pca, 3 components
23	4252.371646	3	box-cox transformed, scaled - no outliers, pca...
21	4288.178164	3	box-cox transformed, scaled - pca, 3 components
13	6235.027181	3	log transformed, scaled - no outliers
12	6268.999409	3	log transformed, scaled
19	6308.349875	3	box-cox transformed, scaled - no outliers
18	6375.112641	3	box-cox transformed, scaled
4	16312.036164	3	original - no outliers, pca, 2 components
2	18633.951812	3	original - pca, 2 components
5	23709.912674	3	original - no outliers, pca, 3 components
3	27038.853476	3	original - pca, 3 components
1	44192.156132	3	original - no outliers
0	50086.677037	3	original

4.15 One More Thing . . . What About Those Labels?

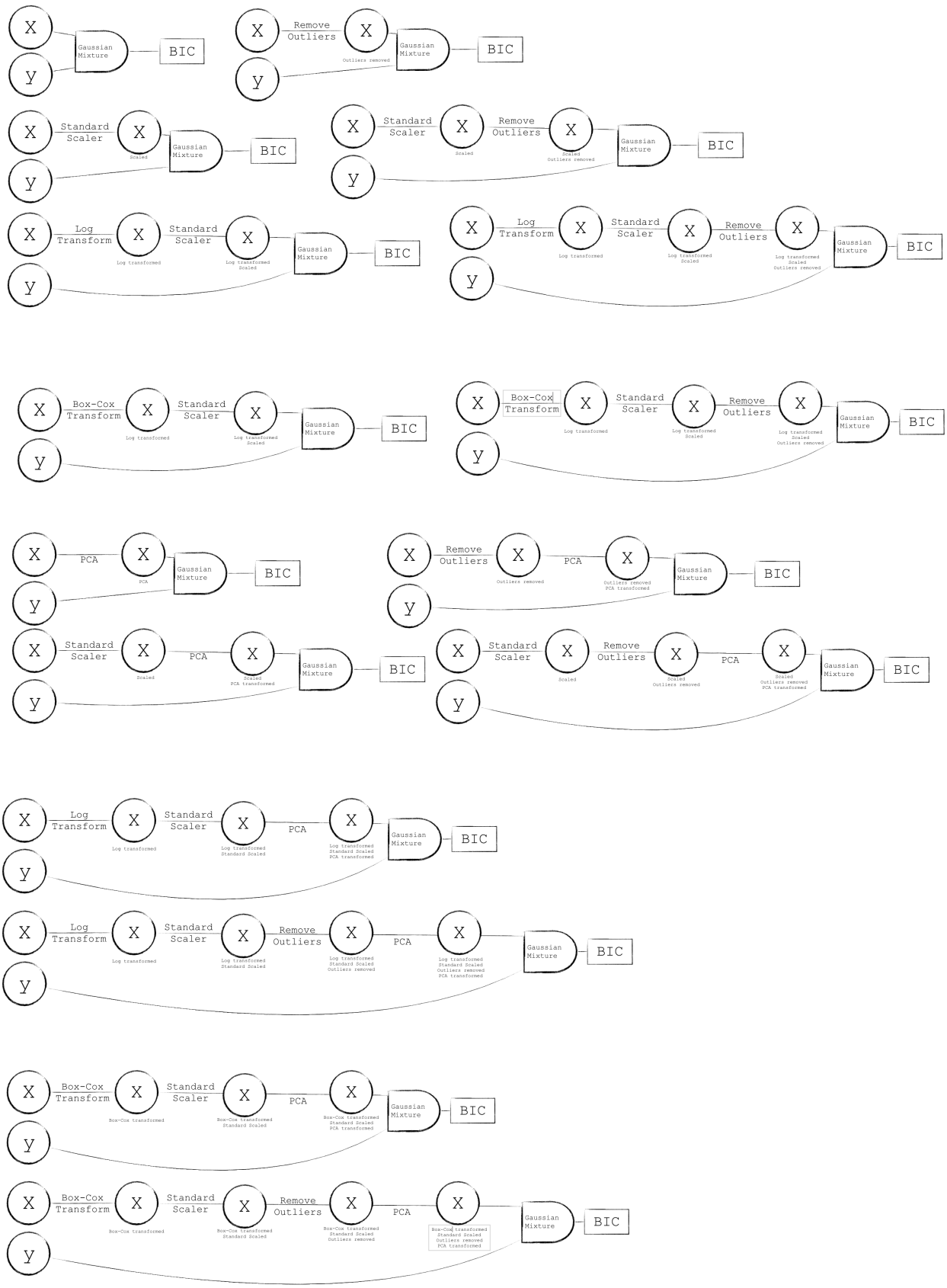
```
In [11]: channel = customers.Channel.astype(int) - 1
         # region = customers.Region

In [12]: from sklearn.metrics import accuracy_score

         def fit_and_score_predictions(data, labels, n_components=2):
             model = GaussianMixture(n_components=n_components)
             model.fit(data)
             predictions = model.predict(data)
             labels_pos = labels
             labels_neg = (labels == 0).astype(int)
             return max(accuracy_score(labels_pos, predictions), accuracy_score(labels_neg, pred

In [13]: customer_sc_outliers_removed.shape

Out[13]: (399, 6)

In [14]: original_data_with_labels = [
             ('original', customer_features, channel),
             ('original - no outliers', customer_features_outliers_removed, channel_original_out
             ('original - pca, 2 components', customer_features_pca_2, channel),
             ('original - pca, 3 components', customer_features_pca_3, channel),
             ('original - no outliers, pca, 2 components', customer_features_outliers_removed_pc
             ('original - no outliers, pca, 3 components', customer_features_outliers_removed_pc
         ]

         scaled_data_with_labels = [
             ('scaled', customer_sc, channel),
             ('scaled - no outliers', customer_sc_outliers_removed, channel_scaled_outliers_remo
             ('scaled - pca, 2 components', customer_sc_pca_2, channel),
             ('scaled - pca, 3 components', customer_sc_pca_3, channel),
             ('scaled - no outliers, pca, 2 components', customer_sc_outliers_removed_pca_2, cha
             ('scaled - no outliers, pca, 3 components', customer_sc_outliers_removed_pca_3, cha
         ]

         log_transformed_data_with_labels = [
             ('log transformed, scaled', customer_log_sc, channel),
             ('log transformed, scaled - no outliers', customer_log_sc_outliers_removed, channel
             ('log transformed, scaled - pca, 2 components', customer_log_sc_pca_2, channel),
             ('log transformed, scaled - pca, 3 components', customer_log_sc_pca_3, channel),
             ('log transformed, scaled - no outliers, pca, 2 components', customer_log_sc_outlie
             ('log transformed, scaled - no outliers, pca, 3 components', customer_log_sc_outlie
         ]

         box_cox_transformed_data_with_labels = [
             ('box-cox transformed, scaled', customer_box_cox_sc, channel),
             ('box-cox transformed, scaled - no outliers', customer_box_cox_sc_outliers_removed,
             ('box-cox transformed, scaled - pca, 2 components', customer_box_cox_sc_pca_2, chan
             ('box-cox transformed, scaled - pca, 3 components', customer_box_cox_sc_pca_3, chan
             ('box-cox transformed, scaled - no outliers, pca, 2 components', customer_box_cox_s
             ('box-cox transformed, scaled - no outliers, pca, 3 components', customer_box_cox_s
         ]

In [15]: n = 2

         results_2_accuracy = []

         for name, data, label in original_data_with_labels:
```

```
        results_2_accuracy.append({
            'name' : name,
            'n' : n,
            'accuracy' : fit_and_score_predictions(data, label, n)
        })

    for name, data, label in scaled_data_with_labels:
        results_2_accuracy.append({
            'name' : name,
            'n' : n,
            'accuracy' : fit_and_score_predictions(data, label, n)
        })

    for name, data, label in log_transformed_data_with_labels:
        results_2_accuracy.append({
            'name' : name,
            'n' : n,
            'accuracy' : fit_and_score_predictions(data, label, n)
        })

    for name, data, label in box_cox_transformed_data_with_labels:
        results_2_accuracy.append({
            'name' : name,
            'n' : n,
            'accuracy' : fit_and_score_predictions(data, label, n)
        })
```

In [16]:
```
pd.DataFrame(results_2_accuracy).sort_values('accuracy', ascending=False)
```

Out[16]:

	accuracy	n	name
12	0.895455	2	log transformed, scaled
13	0.891954	2	log transformed, scaled - no outliers
18	0.890909	2	box-cox transformed, scaled
19	0.890411	2	box-cox transformed, scaled - no outliers
23	0.878995	2	box-cox transformed, scaled - no outliers, pca...
16	0.878161	2	log transformed, scaled - no outliers, pca, 2 ...
22	0.876712	2	box-cox transformed, scaled - no outliers, pca...
21	0.872727	2	box-cox transformed, scaled - pca, 3 components
14	0.870455	2	log transformed, scaled - pca, 2 components
20	0.870455	2	box-cox transformed, scaled - pca, 2 components
10	0.842105	2	scaled - no outliers, pca, 2 components
7	0.817043	2	scaled - no outliers
1	0.817043	2	original - no outliers
2	0.718182	2	original - pca, 2 components
8	0.709091	2	scaled - pca, 2 components
0	0.686364	2	original
15	0.677273	2	log transformed, scaled - pca, 3 components
17	0.673563	2	log transformed, scaled - no outliers, pca, 3 ...
6	0.670455	2	scaled
3	0.663636	2	original - pca, 3 components
9	0.656818	2	scaled - pca, 3 components
4	0.596491	2	original - no outliers, pca, 2 components
11	0.558897	2	scaled - no outliers, pca, 3 components
5	0.551378	2	original - no outliers, pca, 3 components


X
y
Gaussian Mixture
BIC
Remove Outliers
Outliers removed
Standard Scaler
Scaled
Log Transform
Log transformed
Box-Cox Transform
Box-Cox transformed
PCA
Log transformed Standard Scaled
Log transformed Standard Scaled PCA transformed
Log transformed Standard Scaled Outliers removed
Log transformed Standard Scaled Outliers removed PCA transformed
Box-Cox transformed Standard Scaled
Box-Cox transformed Standard Scaled PCA transformed
Box-Cox transformed Standard Scaled Outliers removed
Box-Cox transformed Standard Scaled Outliers removed PCA transformed

CHAPTER

FIVE

THE AMES, IOWA HOUSING DATASET

5.1 Ingest the Data

This notebook contains the basic commands required to ingest the data for our work. Note that all of these commands were added to the file, `src/load_data-01.r` so that in subsequent notebooks the data is loaded via script.

5.1.1 Join the Data Sets

Often you will receive data describing the same instances from multiple data sources. The original Ames, Iowa housing data has been arbitrarily split in order to allow us the opportunity to practice joining data from different sources.

```
In [1]: zoning_df = read.csv('data/zoning.csv')
        listing_df = read.csv('data/listing.csv')
        sale_df = read.csv('data/sale.csv')
```

```
In [2]: head(zoning_df)
```

Id	MSSubClass	MSZoning	LotFrontage	LotArea	LotShape	LandContour	Utilities	LotConfig	LandSlope
1	60	RL	65	8450	Reg	Lvl	AllPub	Inside	Gtl
2	20	RL	80	9600	Reg	Lvl	AllPub	FR2	Gtl
3	60	RL	68	11250	IR1	Lvl	AllPub	Inside	Gtl
4	70	RL	60	9550	IR1	Lvl	AllPub	Corner	Gtl
5	60	RL	84	14260	IR1	Lvl	AllPub	FR2	Gtl
6	50	RL	85	14115	IR1	Lvl	AllPub	Inside	Gtl

```
In [3]: head(listing_df)
```

Id	Street	Alley	OverallQual	OverallCond	YearBuilt	YearRemodAdd	RoofStyle	RoofMatl	Exterior1st	..
1	Pave		7	5	2003	2003	Gable	CompShg	VinylSd	..
2	Pave		6	8	1976	1976	Gable	CompShg	MetalSd	..
3	Pave		7	5	2001	2002	Gable	CompShg	VinylSd	..
4	Pave		7	5	1915	1970	Gable	CompShg	Wd Sdng	..
5	Pave		8	5	2000	2000	Gable	CompShg	VinylSd	..
6	Pave		5	5	1993	1995	Gable	CompShg	VinylSd	..

```
In [4]: head(sale_df)
```

Id	MoSold	YrSold	SaleType	SaleCondition	SalePrice
1	2	2008	WD	Normal	208500
2	5	2007	WD	Normal	181500
3	9	2008	WD	Normal	223500
4	2	2006	WD	Abnorml	140000
5	12	2008	WD	Normal	250000
6	10	2009	WD	Normal	143000

Here, we join the three datasets using the `merge` command using the column `Id` as reference.

```
In [5]: housing_df = merge(zoning_df, listing_df, by="Id")
        housing_df = merge(housing_df, sale_df, by="Id")
```

```
In [6]: head(housing_df)
```

Id	MSSubClass	MSZoning	LotFrontage	LotArea	LotShape	LandContour	Utilities	LotConfig	LandSlope
1	60	RL	65	8450	Reg	Lvl	AllPub	Inside	Gtl
2	20	RL	80	9600	Reg	Lvl	AllPub	FR2	Gtl
3	60	RL	68	11250	IR1	Lvl	AllPub	Inside	Gtl
4	70	RL	60	9550	IR1	Lvl	AllPub	Corner	Gtl
5	60	RL	84	14260	IR1	Lvl	AllPub	FR2	Gtl
6	50	RL	85	14115	IR1	Lvl	AllPub	Inside	Gtl

```
In [7]: dim(housing_df)
```

1. 1460 2. 81

```
In [8]: str(Filter(is.numeric, housing_df))
```

```
'data.frame':   1460 obs. of  38 variables:
 $ Id           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ MSSubClass   : int  60 20 60 70 60 50 20 60 50 190 ...
 $ LotFrontage  : num  65 80 68 60 84 85 75 NA 51 50 ...
 $ LotArea      : int  8450 9600 11250 9550 14260 14115 10084 10382 6120 7420 ...
 $ OverallQual  : int  7 6 7 7 8 5 8 7 7 5 ...
 $ OverallCond  : int  5 8 5 5 5 5 5 6 5 6 ...
 $ YearBuilt    : int  2003 1976 2001 1915 2000 1993 2004 1973 1931 1939 ...
 $ YearRemodAdd : int  2003 1976 2002 1970 2000 1995 2005 1973 1950 1950 ...
 $ MasVnrArea   : int  196 0 162 0 350 0 186 240 0 0 ...
 $ BsmtFinSF1   : int  706 978 486 216 655 732 1369 859 0 851 ...
 $ BsmtFinSF2   : int  0 0 0 0 0 0 0 32 0 0 ...
 $ BsmtUnfSF    : int  150 284 434 540 490 64 317 216 952 140 ...
 $ TotalBsmtSF  : int  856 1262 920 756 1145 796 1686 1107 952 991 ...
 $ FirstFlrSF   : int  856 1262 920 961 1145 796 1694 1107 1022 1077 ...
 $ SecondFlrSF  : int  854 0 866 756 1053 566 0 983 752 0 ...
 $ LowQualFinSF : int  0 0 0 0 0 0 0 0 0 0 ...
 $ GrLivArea    : int  1710 1262 1786 1717 2198 1362 1694 2090 1774 1077 ...
 $ BsmtFullBath : int  1 0 1 1 1 1 1 1 0 1 ...
 $ BsmtHalfBath : int  0 1 0 0 0 0 0 0 0 0 ...
 $ FullBath     : int  2 2 2 1 2 1 2 2 2 1 ...
 $ HalfBath     : int  1 0 1 0 1 1 0 1 0 0 ...
 $ BedroomAbvGr : int  3 3 3 3 4 1 3 3 2 2 ...
 $ KitchenAbvGr : int  1 1 1 1 1 1 1 1 2 2 ...
 $ TotRmsAbvGrd : int  8 6 6 7 9 5 7 7 8 5 ...
 $ Fireplaces   : int  0 1 1 1 1 0 1 2 2 2 ...
 $ GarageYrBlt  : int  2003 1976 2001 1998 2000 1993 2004 1973 1931 1939 ...
 $ GarageCars   : int  2 2 2 3 3 2 2 2 2 1 ...
 $ GarageArea   : int  548 460 608 642 836 480 636 484 468 205 ...
 $ WoodDeckSF   : int  0 298 0 0 192 40 255 235 90 0 ...
 $ OpenPorchSF  : int  61 0 42 35 84 30 57 204 0 4 ...
 $ EnclosedPorch: int  0 0 0 272 0 0 0 228 205 0 ...
 $ ThreeSsnPorch: int  0 0 0 0 0 320 0 0 0 0 ...
```

```
 $ ScreenPorch  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ PoolArea     : int  0 0 0 0 0 0 0 0 0 0 ...
 $ MiscVal      : int  0 0 0 0 0 700 0 350 0 0 ...
 $ MoSold       : int  2 5 9 2 12 10 8 11 4 1 ...
 $ YrSold       : int  2008 2007 2008 2006 2008 2009 2007 2009 2008 2008 ...
 $ SalePrice    : int  208500 181500 223500 140000 250000 143000 307000 200000 129900 118000 ...
```

```
In [9]: rownames(housing_df) <- housing_df$Id
        housing_df$Id <- NULL
```

5.1.2 Typecast Categorical Features

Several features are categorical in nature in spite of the fact that the data is stored as integer values. We must explicitly cast these features as `factor` type features.

```
In [10]: housing_df$MSSubClass <- as.factor(housing_df$MSSubClass)
         housing_df$OverallQual <- as.factor(housing_df$OverallQual)
         housing_df$OverallCond <- as.factor(housing_df$OverallCond)
         housing_df$BsmtFullBath <- as.factor(housing_df$BsmtFullBath)
         housing_df$BsmtHalfBath <- as.factor(housing_df$BsmtHalfBath)
         housing_df$FullBath <- as.factor(housing_df$FullBath)
         housing_df$HalfBath <- as.factor(housing_df$HalfBath)
         housing_df$BedroomAbvGr <- as.factor(housing_df$BedroomAbvGr)
         housing_df$KitchenAbvGr <- as.factor(housing_df$KitchenAbvGr)
         housing_df$TotRmsAbvGrd <- as.factor(housing_df$TotRmsAbvGrd)
         housing_df$Fireplaces <- as.factor(housing_df$Fireplaces)
         housing_df$GarageCars <- as.factor(housing_df$GarageCars)
         housing_df$MoSold <- as.factor(housing_df$MoSold)
```

5.2 Impute Missing Values

This data set contains many columns with missing values. This notebook deals with accounting for missing values. Note that all of these commands were added to the file, `src/load_data-02.r` so that in subsequent notebooks the data is loaded via script.

There is a challenge to handling these missing values specific to the dataset. The file `doc/data_description.txt` contains a detailed description of each feature in this data set. Here, we see the description for `MasVnrType`.

```
MasVnrType: Masonry veneer type

       BrkCmn   Brick Common
       BrkFace  Brick Face
       CBlock   Cinder Block
       None None
       Stone    Stone
```

Note that one attribute for this feature is the value `None` meaning that the house in question does not have a Veneer. This is common in the data set. Unfortunately, upon loading the data, these `None` values will be taken to mean that the data is missing when they should be taken to mean `None`. Adding further complication to this, is the fact that there are features that contain actual missing values.

```
In [1]: source('src/load_data-01.r')
```

```
In [2]: dim(housing_df)
```

1. 1460 2. 81

```
In [3]: head(housing_df)
```

X	MSSubClass	MSZoning	LotFrontage	LotArea	LotShape	LandContour	Utilities	LotConfig	LandSlope
1	60	RL	65	8450	Reg	Lvl	AllPub	Inside	Gtl
2	20	RL	80	9600	Reg	Lvl	AllPub	FR2	Gtl
3	60	RL	68	11250	IR1	Lvl	AllPub	Inside	Gtl
4	70	RL	60	9550	IR1	Lvl	AllPub	Corner	Gtl
5	60	RL	84	14260	IR1	Lvl	AllPub	FR2	Gtl
6	50	RL	85	14115	IR1	Lvl	AllPub	Inside	Gtl

5.2.1 Impute `nan` Values

We begin by investigating the dataset for `nan` or "not a number" values. This value was added to numerical features with missing data. These should be taken as actual missing values.

```
In [4]: nan_sums = colSums(is.na(housing_df))
        nan_sums[nan_sums > 0]
```

LotFrontage 259 **MasVnrArea** 8 **GarageYrBlt** 81 We will impute the missing values by simply signing to them the mean of the extant values. There is a bit of a trick to this. As shown below taking the mean of a feature with missing values will return a `nan`.

```
In [5]:  mean(housing_df$LotFrontage)
```

<NA> This can be by passing the argument `na.rm=T` to the `mean()` function.

```
In [6]: mean(housing_df$LotFrontage, na.rm=T)
```

70.0499583680267 Here we assign these mean values to variables.

```
In [7]: mean_LotFrontage <- mean(housing_df$LotFrontage, na.rm=T)
        mean_MasVnrArea <- mean(housing_df$MasVnrArea, na.rm=T)
        mean_GarageYrBlt <- mean(housing_df$GarageYrBlt, na.rm=T)
```

Next, we use the `is.na` function to create a mask for the missing values. We then assign to the missing values the calculated mean value.

```
In [8]: housing_df$LotFrontage[is.na(housing_df$LotFrontage)] <- mean_LotFrontage
        housing_df$MasVnrArea[is.na(housing_df$MasVnrArea)] <- mean_MasVnrArea
        housing_df$GarageYrBlt[is.na(housing_df$GarageYrBlt)] <- mean_GarageYrBlt
```

```
In [9]: nan_sums = colSums(is.na(housing_df))
        nan_sums[nan_sums > 0]
```

5.2.2 Handling `None` Values

Here, we write a set of helper functions to help us to identify the `None` values for our categorical features. Note, that they are currently being stored as simply an empty string. We use masking by creating a mask for empty string values.

```
In [10]: count_empty_values <- function (feature) {
             empty_string_mask = housing_df[feature] == ""
             return(length(housing_df[feature][empty_string_mask]))
         }

         count_empty_total <- function (){
             for (feature in colnames(housing_df)) {
                 empty_count <- count_empty_values(feature)
                 if (empty_count > 0) {
                     print(paste(feature, empty_count))
```

```
                }
            }
        }
```

```
In [11]: count_empty_total()
```

```
[1] "Alley 1369"
[1] "MasVnrType 8"
[1] "BsmtQual 37"
[1] "BsmtCond 37"
[1] "BsmtExposure 38"
[1] "BsmtFinType1 37"
[1] "BsmtFinType2 38"
[1] "Electrical 1"
[1] "FireplaceQu 690"
[1] "GarageType 81"
[1] "GarageFinish 81"
[1] "GarageQual 81"
[1] "GarageCond 81"
[1] "PoolQC 1453"
[1] "Fence 1179"
[1] "MiscFeature 1406"
```

Here, we note that missing values for one of the features should be taken to mean `nan`

```
Electrical: Electrical system

       SBrkr    Standard Circuit Breakers & Romex
       FuseA    Fuse Box over 60 AMP and all Romex wiring (Average)
       FuseF    60 AMP Fuse Box and mostly Romex wiring (Fair)
       FuseP    60 AMP Fuse Box and mostly knob & tube wiring (poor)
       Mix  Mixed
```

To handle for this, we create two lists one where an empty string signifies `None` and the other where the empty string signifies `nan`.

```
In [12]: empty_means_without <-c("Alley","BsmtQual","BsmtCond","BsmtExposure","BsmtFinType1",
                                 "BsmtFinType2", "FireplaceQu","GarageType","GarageFinish",
                                 "GarageQual","GarageCond","PoolQC","Fence","MiscFeature","MasVn

         empty_means_NA <- c("Electrical")
```

We then right a series of helper functions to use the masking on on empty string to properly handle all of these empty strings. Note that where an empty string signifies `None` we assigned the value `"without"`.

```
In [13]: replace_empty_with_without <- function(feature) {
             levels(feature) <- c(levels(feature), "without")
             empty_string_mask <- feature == ''
             feature[empty_string_mask] <- "without"
             return(feature)
         }

         replace_empty_with_NA <- function(feature) {
             levels(feature) <- c(levels(feature), NA)
             empty_string_mask <- feature == ''
             feature[empty_string_mask] <- NA
             return(feature)
         }
```

```
In [14]: for (feature in empty_means_without) {
             housing_df[,feature] <- replace_empty_with_without(housing_df[,feature])
         }
```

```
        for (feature in empty_means_NA) {
            housing_df[,feature] <- replace_empty_with_NA(housing_df[,feature])
        }
```

```
In [15]: count_empty_total()
```

```
[1] "Electrical 1"
```

```
In [16]: nan_sums = colSums(is.na(housing_df))
         nan_sums[nan_sums > 0]
```

Electrical: 1 Following this, we still have a `nan` value that has not been dealt with. As the affected data amounts to less than .1% our our total data, we simply drop the affected row.

```
In [17]: housing_df <- na.omit(housing_df)
```

```
In [18]: dim(housing_df)
```

1. 1459 2. 81

5.3 Basic EDA

In this notebook, we perform basic data analysis for our dataset. This mostly consists of preparing distribution plots for the numerical features. We also begin to explore the technique of preparing distribution plots for numerical features separated by a categorical feature.

```
In [1]: source('src/load_data-02.r')
        source('src/multiplot.r')
```

```
In [2]: dim(housing_df)
```

1. 1451 2. 80

```
In [3]: head(housing_df)
```

MSSubClass	MSZoning	LotFrontage	LotArea	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neig
60	RL	65	8450	Reg	Lvl	AllPub	Inside	Gtl	Coll
20	RL	80	9600	Reg	Lvl	AllPub	FR2	Gtl	Veer
60	RL	68	11250	IR1	Lvl	AllPub	Inside	Gtl	Coll
70	RL	60	9550	IR1	Lvl	AllPub	Corner	Gtl	Crav
60	RL	84	14260	IR1	Lvl	AllPub	FR2	Gtl	NoR
50	RL	85	14115	IR1	Lvl	AllPub	Inside	Gtl	Mitc

```
In [4]: count_empty_total()
```

```
In [5]: str(Filter(is.numeric, housing_df))
```

```
'data.frame':   1451 obs. of  24 variables:
 $ LotFrontage  : num  65 80 68 60 84 ...
 $ LotArea      : int  8450 9600 11250 9550 14260 14115 10084 10382 6120 7420 ...
 $ YearBuilt    : int  2003 1976 2001 1915 2000 1993 2004 1973 1931 1939 ...
 $ YearRemodAdd : int  2003 1976 2002 1970 2000 1995 2005 1973 1950 1950 ...
 $ MasVnrArea   : num  196 0 162 0 350 0 186 240 0 0 ...
 $ BsmtFinSF1   : int  706 978 486 216 655 732 1369 859 0 851 ...
 $ BsmtFinSF2   : int  0 0 0 0 0 0 0 32 0 0 ...
 $ BsmtUnfSF    : int  150 284 434 540 490 64 317 216 952 140 ...
 $ TotalBsmtSF  : int  856 1262 920 756 1145 796 1686 1107 952 991 ...
 $ FirstFlrSF   : int  856 1262 920 961 1145 796 1694 1107 1022 1077 ...
 $ SecondFlrSF  : int  854 0 866 756 1053 566 0 983 752 0 ...
 $ LowQualFinSF : int  0 0 0 0 0 0 0 0 0 0 ...
 $ GrLivArea    : int  1710 1262 1786 1717 2198 1362 1694 2090 1774 1077 ...
```

```
 $ GarageYrBlt   : num  2003 1976 2001 1998 2000 ...
 $ GarageArea    : int  548 460 608 642 836 480 636 484 468 205 ...
 $ WoodDeckSF    : int  0 298 0 0 192 40 255 235 90 0 ...
 $ OpenPorchSF   : int  61 0 42 35 84 30 57 204 0 4 ...
 $ EnclosedPorch: int  0 0 0 272 0 0 0 228 205 0 ...
 $ ThreeSsnPorch: int  0 0 0 0 0 320 0 0 0 0 ...
 $ ScreenPorch   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ PoolArea      : int  0 0 0 0 0 0 0 0 0 0 ...
 $ MiscVal       : int  0 0 0 0 0 700 0 350 0 0 ...
 $ YrSold        : int  2008 2007 2008 2006 2008 2009 2007 2009 2008 2008 ...
 $ SalePrice     : int  208500 181500 223500 140000 250000 143000 307000 200000 129900 118000 ...
```

```
In [6]: colnames(Filter(is.numeric, housing_df))
```

1. 'LotFrontage' 2. 'LotArea' 3. 'YearBuilt' 4. 'YearRemodAdd' 5. 'MasVnrArea' 6. 'BsmtFinSF1' 7. 'BsmtFinSF2' 8. 'BsmtUnfSF' 9. 'TotalBsmtSF' 10. 'FirstFlrSF' 11. 'SecondFlrSF' 12. 'LowQualFinSF' 13. 'GrLivArea' 14. 'GarageYrBlt' 15. 'GarageArea' 16. 'WoodDeckSF' 17. 'OpenPorchSF' 18. 'EnclosedPorch' 19. 'ThreeSsnPorch' 20. 'ScreenPorch' 21. 'PoolArea' 22. 'MiscVal' 23. 'YrSold' 24. 'SalePrice'

```
In [7]: attach(housing_df)
```

```
In [8]: library(ggplot2)
```

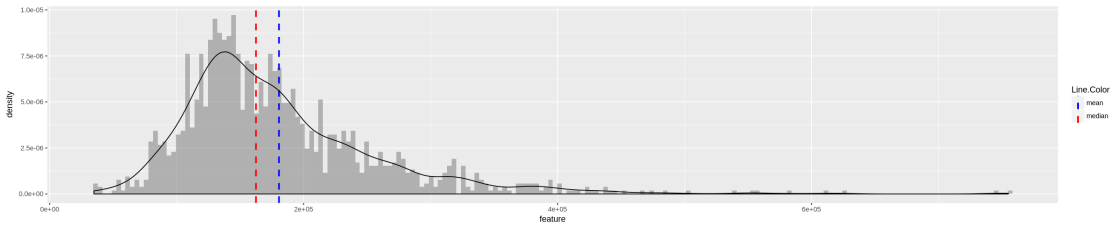
5.3.1 Histogram of Target Feature

Here, we display a histogram of the target feature `SalePrice`. We have also included a kernel density estimation (KDE) and the mean and median values plotted as vertical lines. The mean greater than the median signifies a right or positive skew, common with strictly non-negative data.

```
In [9]: hist_with_kde <- function (feature) {
            plot <- qplot(feature, geom="histogram", bins=200, alpha=I(.4), y = ..density..)+
                geom_vline(aes(xintercept=mean(feature, rm.na=T), color="mean"), linetype="dashe
                geom_vline(aes(xintercept=median(feature), color="median"), linetype="dashed", s
                geom_density()+
                scale_color_manual("Line.Color", values=c(median="red",mean="blue"))
            return(plot)
        }
```

```
In [10]: hist_with_kde(SalePrice)
```

Data type cannot be displayed:



Plot some Histograms with KDE Plots for other Numerical Features

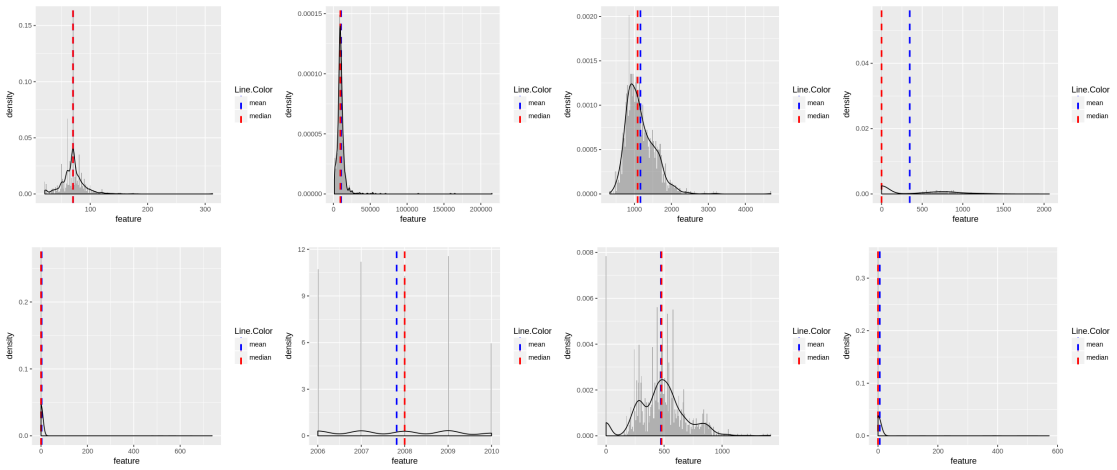
Next we plot histograms with KDE plots for some of the other numerical features in our dataset.

```
In [11]: colnames(Filter(is.numeric, housing_df))
```

1. 'LotFrontage' 2. 'LotArea' 3. 'YearBuilt' 4. 'YearRemodAdd' 5. 'MasVnrArea' 6. 'BsmtFinSF1' 7. 'BsmtFinSF2' 8. 'BsmtUnfSF' 9. 'TotalBsmtSF' 10. 'FirstFlrSF' 11. 'SecondFlrSF' 12. 'LowQualFinSF' 13. 'GrLivArea' 14. 'GarageYrBlt' 15. 'GarageArea' 16. 'WoodDeckSF' 17. 'OpenPorchSF' 18. 'EnclosedPorch' 19. 'ThreeSsnPorch' 20. 'ScreenPorch' 21. 'PoolArea' 22. 'MiscVal' 23. 'YrSold' 24. 'SalePrice' We make use of a special function called `multiplot` that is included in the file `src/multiplot.r`.

```
In [12]: library(repr)
         options(repr.plot.width=20, repr.plot.height=4)
```

```
In [13]: multiplot(hist_with_kde(LotFrontage),
                   hist_with_kde(LotArea),
                   hist_with_kde(FirstFlrSF),
                   hist_with_kde(SecondFlrSF),
                   cols = 4)
         multiplot(hist_with_kde(PoolArea),
                   hist_with_kde(YrSold),
                   hist_with_kde(GarageArea),
                   hist_with_kde(LowQualFinSF),
                   cols = 4)
```



5.4 Correlation

Assessing correlation in a data set with mixed numerical and categorical features can be challenging. One way to perform such an analysis is to prepare a series of distribution plots for a single numerical feature each distribution plot corresponds to the values for the numerical feature for a given attribute of a categorical feature.

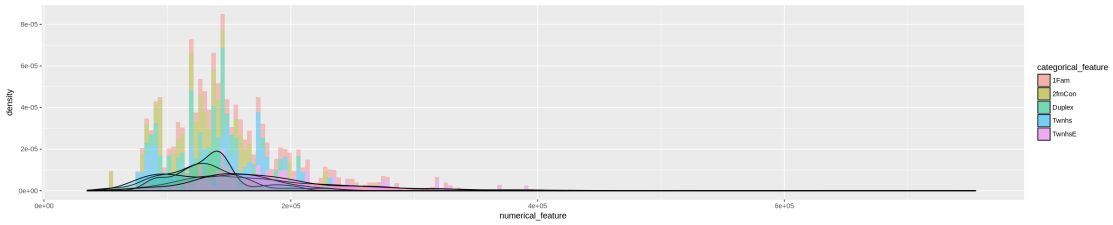
Here is a list of our categorical features:

Alley	ExterCond	GarageType
BedroomAbvGr	Exterior1st	HalfBath
BldgType	Exterior2nd	Heating
BsmtCond	ExterQual	HeatingQC
BsmtExposure	Fence	HouseStyle
BsmtFinType1	FireplaceQu	KitchenAbvGr
BsmtFinType2	Fireplaces	KitchenQual
BsmtFullBath	Foundation	LandContour
BsmtHalfBath	FullBath	LandSlope
BsmtQual	Functional	LotConfig
CentralAir	GarageCars	LotShape
Condition1	GarageCond	MasVnrType
Condition2	GarageFinish	MiscFeature
Electrical	GarageQual	MoSold

We can begin by looking at the distribution of `SalePrice` disaggregated by any one of these categorical features.

```
In [14]: hist_with_kde_numerical_by_category(SalePrice, BldgType)
```

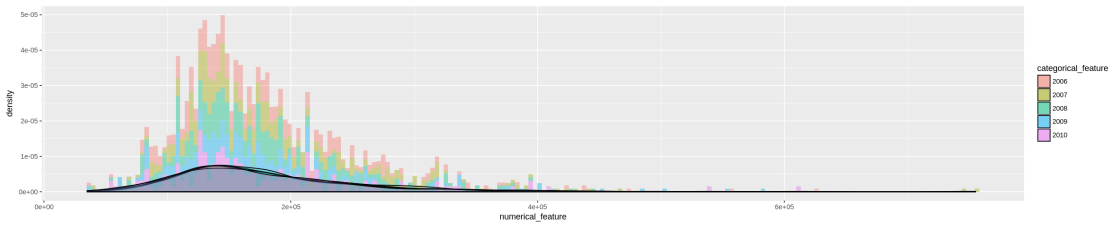
Data type cannot be displayed:



It may even make sense to treat one of the numerical features as a categorical feature, for example, `YrSold`.

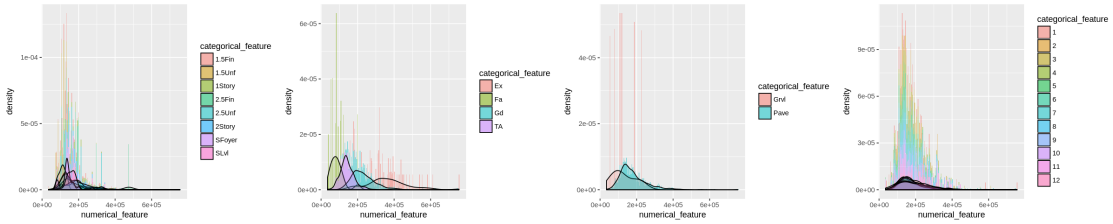
```
In [15]: hist_with_kde_numerical_by_category(SalePrice, as.factor(YrSold))
```

Data type cannot be displayed:



This plot shows that, for this dataset, the year of the sale has nearly no impact on the `SalePrice`. Note that `SalePrice` has a nearly identical distribution for all five years in the dataset.

```
In [16]: multiplot(hist_with_kde_numerical_by_category(SalePrice,HouseStyle),
                   hist_with_kde_numerical_by_category(SalePrice,ExterQual),
                   hist_with_kde_numerical_by_category(SalePrice,Street),
                   hist_with_kde_numerical_by_category(SalePrice,MoSold),
                   cols = 4)
```

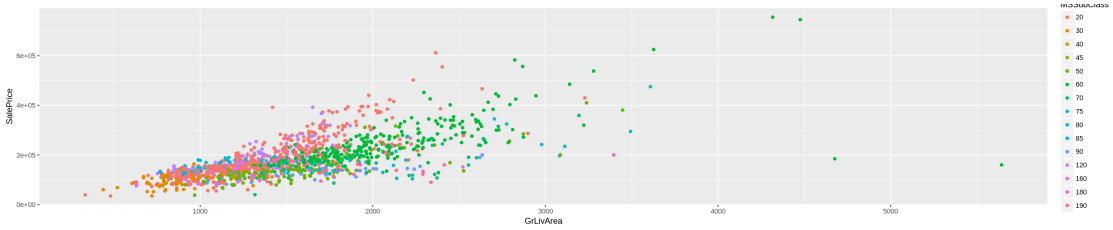


Here, we see that `HouseStyle`, `ExterQual`, and `Street` all have some impact on `SalePrice`, while `MoSold` does not.

Another way to analyze the influence of a categorical feature it is to create a scatter plot of two numerical features, colored by a categorical feature.

```
In [17]: ggplot(housing_df)+
             geom_point(aes(x=GrLivArea,y=SalePrice,colour=MSSubClass))
```

Data type cannot be displayed:



5.5 Analysis of Variance (ANOVA)

In this notebook, we explore the Analysis of Variance (ANOVA) technique. Analysis of Variance (ANOVA) is a statistical procedure for comparing means of two or more populations. Essentially, we wish to understand whether two populations are significantly different from each other by comparing their means.

Previously, we prepared a series of distribution plots for a single numerical feature where each distribution plot corresponds to the values for the numerical feature for a given attribute of a categorical feature. Here, we use ANOVA to evaluate statistically and what we see in those plots.

```
In [1]: source('src/load_data-02.r')
        source('src/multiplot.r')
```

```
In [2]: dim(housing_df)
```

1. 1451 2. 80

```
In [3]: head(housing_df)
```

MSSubClass	MSZoning	LotFrontage	LotArea	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neig
60	RL	65	8450	Reg	Lvl	AllPub	Inside	Gtl	Coll
20	RL	80	9600	Reg	Lvl	AllPub	FR2	Gtl	Veer
60	RL	68	11250	IR1	Lvl	AllPub	Inside	Gtl	Coll
70	RL	60	9550	IR1	Lvl	AllPub	Corner	Gtl	Crav
60	RL	84	14260	IR1	Lvl	AllPub	FR2	Gtl	NoR
50	RL	85	14115	IR1	Lvl	AllPub	Inside	Gtl	Mito

```
In [4]: count_empty_total()
```

```
In [5]: attach(housing_df)
```

5.5.1 One-way ANOVA

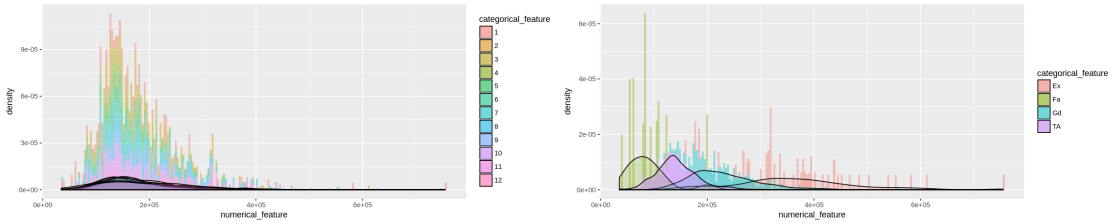
One-way ANOVA is perhaps the simplest ANOVA technique and handles a special case of this problem, testing for equal group means using a single feature. The idea is essentially this

1. Identify a numerical feature for analysis (often the target feature)
2. Split that numerical features into groups using a categorical feature
3. Run a one-way ANOVA on these groups
   (d) If it is found that than means are equal for all groups, then this categorical feature may be less relevant for predicting the numerical feature in question
   (e) If it is found that the means are not equal for all groups, then this categorical feature may be important for predicting the numerical feature in question

In a one-way ANOVA, the null hypothesis is that the mean responses are equal for all groups. The alternative hypothesis is that the mean responses are not equal for all groups. It is helpful to recall that any statistical test, it is standard that if the p-value of the test is less than 0.05, then the null hypothesis can be rejected.

A :math:‘p‘-value greater than 0.05 does not necessarily mean that the null hypothesis should be accepted.

```
In [6]: multiplot(hist_with_kde_numerical_by_category(SalePrice,MoSold),
                  hist_with_kde_numerical_by_category(SalePrice,ExterQual),
                  cols = 2)
```



Month Sold

Consider the null hypothesis:

$$H_0 : \text{the mean responses are equal for all groups}$$

```
In [7]: meansd = function(x) c(mean=mean(x), sd=sd(x))
        by(SalePrice, MoSold, FUN=meansd)
```

```
MoSold: 1
    mean       sd
183256.3 121381.1
------------------------------------------------------------
MoSold: 2
     mean        sd
177882.00  52960.86
------------------------------------------------------------
MoSold: 3
     mean        sd
182570.12  87143.86
```

```
------------------------------------------------------------
MoSold: 4
     mean        sd
171503.26  77147.32
------------------------------------------------------------
MoSold: 5
     mean        sd
171943.95  69013.79
------------------------------------------------------------
MoSold: 6
     mean        sd
177395.74  69453.09
------------------------------------------------------------
MoSold: 7
     mean        sd
186331.19  91772.13
------------------------------------------------------------
MoSold: 8
     mean        sd
184649.78  73520.42
------------------------------------------------------------
MoSold: 9
     mean        sd
191339.39  76281.58
------------------------------------------------------------
MoSold: 10
     mean        sd
179563.98  75736.01
------------------------------------------------------------
MoSold: 11
     mean        sd
192112.33  84053.66
------------------------------------------------------------
MoSold: 12
     mean        sd
186596.88  70099.51
```

```
In [8]: oneway.test(SalePrice ~ MoSold)
```

```
One-way analysis of means (not assuming equal variances)

data:  SalePrice and MoSold
F = 0.90408, num df = 11.00, denom df = 405.31, p-value = 0.5364
```

This test shows that we CAN NOT reject the null hypothesis

Exterior Quality

Consider the null hypothesis:

$$H_0 : \text{the mean responses are equal for all groups}$$

```
In [9]: by(SalePrice, ExterQual, FUN=meansd)
```

```
ExterQual: Ex
    mean       sd
365446.5 116729.7
```

```
------------------------------------------------------------
ExterQual: Fa
    mean       sd
87985.21 39826.92
------------------------------------------------------------
ExterQual: Gd
     mean        sd
232038.00  71595.41
------------------------------------------------------------
ExterQual: TA
     mean        sd
144315.72  42488.31
```

```
In [10]: oneway.test(SalePrice ~ ExterQual)
```

```
One-way analysis of means (not assuming equal variances)

data:  SalePrice and ExterQual
F = 264.07, num df = 3.000, denom df = 52.093, p-value < 2.2e-16
```

This test shows that we CAN reject the null hypothesis

5.6 Redundancy and Correlation

In preparation for a principal component analysis, we look at redundancy and correlation in our dataset. In this analysis, we will focus on the numeric features.

```
In [1]: source('src/load_data-02.r')
        source('src/multiplot.r')
```

```
In [2]: dim(housing_df)
```

1. 1451 2. 80

```
In [3]: head(housing_df)
```

MSSubClass	MSZoning	LotFrontage	LotArea	LotShape	LandContour	Utilities	LotConfig	LandSlope	Neig
60	RL	65	8450	Reg	Lvl	AllPub	Inside	Gtl	Coll
20	RL	80	9600	Reg	Lvl	AllPub	FR2	Gtl	Veer
60	RL	68	11250	IR1	Lvl	AllPub	Inside	Gtl	Coll
70	RL	60	9550	IR1	Lvl	AllPub	Corner	Gtl	Crav
60	RL	84	14260	IR1	Lvl	AllPub	FR2	Gtl	NoR
50	RL	85	14115	IR1	Lvl	AllPub	Inside	Gtl	Mitc

```
In [4]: count_empty_total()
```

```
In [5]: numeric_features = colnames(Filter(is.numeric, housing_df))
```

```
In [6]: numeric_df = Filter(is.numeric, housing_df)
        numeric_df$SalePrice <- NULL
        numeric_features = colnames(numeric_df)
```

```
In [7]: attach(numeric_df)
```

```
In [8]: install.packages('rpart')
```

```
Updating HTML index of packages in '.Library'
Making 'packages.html' ... done
```

```
In [9]: library(caret)
        library(rpart)
```

```
Loading required package: lattice
```

5.6.1 Redundancy

Here, we use machine learning to assess redundancy in our dataset. We iterate through each numeric feature in our dataset. For each feature, we dropped the feature from our input/featureet X And use it as our target y for the training of a supervised regression model. In this case, we use the shortcut for training a model on all features, `~.`, as our regression formula

```
this_formula = paste(feature,"~.")
fit <- rpart(data=train, formula=as.formula(this_formula))
```

In other words we are training a regression model where we use the remaining features to protect each individual feature. We will thus have an R^2 score for each numeric feature. Note, that the `rpart` function is available as part of the `caret` library in R. This is the implementation of a decision tree.

Note, that we also use machine learning best practices and perform a train–test split on our data. Each model is trained using the training data and assessed using the testing data. In this way, each model tells us if, upon removing a feature, the remaining features are able to predict the removed feature. If the remaining features are able to make this prediction, we may take the removed feature to be somewhat redundant. It is worth clarifying that this is an exploratory data analysis technique, and is not intended to be used at this time as a technique for removing features. We simply wish to understand the relationships within our data.

```
In [10]: calculate_r_2 <- function(actual, prediction) {
             return (1 - (sum((actual-prediction)^2)/sum((actual-mean(actual))^2)))
         }

         calculate_r_2_for_feature <- function(data, feature) {
             n <- nrow(data)

             train_index <- sample(seq_len(n), size = 0.8*n)

             train <- data[train_index,]
             test <- data[-train_index,]

             this_formula = paste(feature,"~.")
             fit <- rpart(data=train, formula=as.formula(this_formula))

             y_test <- as.vector(test[[feature]])
             test[feature] <- NULL
             predictions <- predict(fit, test)
             return (calculate_r_2(y_test, predictions))
         }

         mean_r2_for_feature <- function (data, feature) {
             scores = c()
             for (i in 1:10) {
                 scores = c(scores, calculate_r_2_for_feature(data, feature))
             }

             return (mean(scores))
         }
```

```
In [11]: calculate_r_2_for_feature(numeric_df,'LotFrontage')
```

0.347114045220568

```
In [12]: for (feature in numeric_features){
             print(paste(feature, mean_r2_for_feature(numeric_df, feature)))
         }
```

```
[1] "LotFrontage 0.324421644157402"
[1] "LotArea -0.528147437046712"
[1] "YearBuilt 0.708350383039224"
[1] "YearRemodAdd 0.428617549946195"
[1] "MasVnrArea 0.18095244552509"
[1] "BsmtFinSF1 0.803959176496318"
[1] "BsmtFinSF2 0.387078970506263"
[1] "BsmtUnfSF 0.806469012514282"
[1] "TotalBsmtSF 0.647480962525038"
[1] "FirstFlrSF 0.766490532277561"
[1] "SecondFlrSF 0.87795579721457"
[1] "LowQualFinSF -0.709911815993187"
[1] "GrLivArea 0.852629617677604"
[1] "GarageYrBlt 0.775386007898649"
[1] "GarageArea 0.581737841067015"
[1] "WoodDeckSF 0.0554546622880636"
[1] "OpenPorchSF -0.00549856978961937"
[1] "EnclosedPorch 0.0843437449143098"
[1] "ThreeSsnPorch -0.323789412806441"
[1] "ScreenPorch -0.117972985190303"
[1] "PoolArea -Inf"
[1] "MiscVal -1.80381746899222"
[1] "YrSold 0.0572819386249845"
```

5.6.2 Correlation

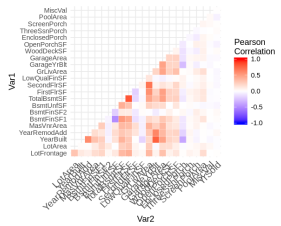
Next, we assess correlation between our features. Correlation is a function of covariance data, which is itself a measure of linear relationships within data. In the previous section, we use a decision tree to assess redundancy. A decision tree is an information-based (non-linear) analysis. By performing this analysis using two different techniques, one linear and one non-linear, we have a more robust assessment have the underlying relationships in our data. Again, this technique is exploratory data analysis and is not intended at this time to remove features from our dataset.

```
In [13]: options(digits=3)
         cor(numeric_df)
```

	LotFrontage	LotArea	YearBuilt	YearRemodAdd	MasVnrArea	BsmtFinSF1	BsmtFinSF2	Bs
LotFrontage	1.00000	0.30692	0.11731	0.08231	0.17977	0.21720	0.04369	0.
LotArea	0.30692	1.00000	0.01571	0.01519	0.10414	0.21308	0.11167	-0
YearBuilt	0.11731	0.01571	1.00000	0.59032	0.31634	0.25021	-0.04761	0.
YearRemodAdd	0.08231	0.01519	0.59032	1.00000	0.18014	0.12842	-0.06649	0.
MasVnrArea	0.17977	0.10414	0.31634	0.18014	1.00000	0.26447	-0.07244	0.
BsmtFinSF1	0.21720	0.21308	0.25021	0.12842	0.26447	1.00000	-0.04950	-0
BsmtFinSF2	0.04369	0.11167	-0.04761	-0.06649	-0.07244	-0.04950	1.00000	-0
BsmtUnfSF	0.11997	-0.00425	0.15022	0.18222	0.11430	-0.49660	-0.20981	1.
TotalBsmtSF	0.36315	0.25854	0.39428	0.29299	0.36367	0.52010	0.10609	0.
FirstFlrSF	0.41515	0.29598	0.28565	0.24346	0.34425	0.44284	0.09865	0.
SecondFlrSF	0.07340	0.05298	0.00904	0.13980	0.17487	-0.13533	-0.09944	0.
LowQualFinSF	0.03704	0.00490	-0.18374	-0.06198	-0.06913	-0.06445	0.01460	0.
GrLivArea	0.36835	0.26115	0.19962	0.28857	0.39082	0.20594	-0.00896	0.
GarageYrBlt	0.06376	-0.02403	0.77906	0.61582	0.25072	0.14988	-0.08571	0.
GarageArea	0.32335	0.18077	0.47895	0.37109	0.37299	0.29537	-0.01764	0.
WoodDeckSF	0.07603	0.17317	0.22696	0.20751	0.15975	0.20545	0.06768	-0
OpenPorchSF	0.13526	0.08628	0.18576	0.22414	0.12546	0.10728	0.00416	0.
EnclosedPorch	0.01014	-0.02311	-0.38675	-0.19219	-0.11036	-0.10589	0.03668	-0
ThreeSsnPorch	0.06258	0.02057	0.03214	0.04601	0.01875	0.02693	-0.03021	0.
ScreenPorch	0.03801	0.04350	-0.04898	-0.03747	0.06137	0.06314	0.08843	-0
PoolArea	0.18135	0.07789	0.00537	0.00620	0.01170	0.14136	0.04160	-0
MiscVal	0.00124	0.03822	-0.03399	-0.00987	-0.02985	0.00385	0.00478	-0
YrSold	0.00767	-0.01297	-0.01456	0.03526	-0.00815	0.01697	0.03188	-0

In [14]:
```
library(reshape2)
cormat = cor(numeric_df)

cormat[lower.tri(cormat)] <- NA
diag(cormat) <- NA

melted_cormat <- melt(cormat, na.rm = T)

library(ggplot2)
ggplot(data = melted_cormat, aes(Var2, Var1, fill = value))+
 geom_tile(color = "white")+
 scale_fill_gradient2(low = "blue", high = "red", mid = "white",
   midpoint = 0, limit = c(-1,1), space = "Lab",
   name="Pearson\nCorrelation") +
  theme_minimal()+
 theme(axis.text.x = element_text(angle = 45, vjust = 1,
    size = 12, hjust = 1))+
 coord_fixed()
```

Data type cannot be displayed:



5.7 Preprocessing

In the file `src/preprocessing.py`, we perform the preprocessing steps necessary for the next phase of this project. This file does the following:

1. Load and merge data, using the `Id` feature as we did previously
2. Assign correct data types, in particular, designating which features are categorical
3. Handle missing values, using the work we did previously

```
In [1]: run src/preprocessing.py
```

5.7.1 Skew-Normalization

Next, we look at skew-normalizing our data. We have two methods that we have worked with to apply skew-normalization:

- applying a log transform
- applying a box-cox transform

In the past, we have seen that the box-cox transform has been more performant in terms of removing skew from a dataset. With this data set, however, there is another issue.

```
In [2]: import scipy.stats as st
```

To see this issue, let's look at the `LotArea` feature from the numeric dataset.

```
In [3]: box_cox_trans = st.boxcox(numeric_df['LotArea'] + 1)
```

```
/opt/conda/lib/python3.6/site-packages/scipy/stats/morestats.py:901: RuntimeWarning: divide by z
  llf -= N / 2.0 * np.log(np.sum((y - y_mean)**2. / N, axis=0))
/opt/conda/lib/python3.6/site-packages/scipy/optimize/optimize.py:2189: RuntimeWarning: invalid
  w = xb - ((xb - xc) * tmp2 - (xb - xa) * tmp1) / denom
/opt/conda/lib/python3.6/site-packages/scipy/optimize/optimize.py:1849: RuntimeWarning: invalid
  tmp1 = (x - w) * (fx - fv)
/opt/conda/lib/python3.6/site-packages/scipy/optimize/optimize.py:1850: RuntimeWarning: invalid
  tmp2 = (x - v) * (fx - fw)
/opt/conda/lib/python3.6/site-packages/scipy/optimize/optimize.py:1855: RuntimeWarning: invalid
  tmp2 = numpy.abs(tmp2)
/opt/conda/lib/python3.6/site-packages/scipy/optimize/optimize.py:1851: RuntimeWarning: invalid
  p = (x - v) * tmp2 - (x - w) * tmp1
/opt/conda/lib/python3.6/site-packages/scipy/optimize/optimize.py:1852: RuntimeWarning: invalid
  tmp2 = 2.0 * (tmp2 - tmp1)
```

Note that applying a box-cox transform to this feature causes a `RunTimeWarning`. This is a known an open issue for the `scipy` library and can be tracked here: https://github.com/scipy/scipy/issues/6873. The details of the issue are complicated, but the short of it is that a floating-point arithmetic error is introduced. As such, we are not able to easily use the Box-Cox transform on this data set. We will stick to applying a log transform.

```
In [4]: numeric_log_df = np.log(numeric_df + 1)
```

5.7.2 One-hot Encoding

In order to understand our categorical data from a numerical perspective, and ultimately in order to use our categorical data in a machine learning model, we need to numerically encode our categorical data.The standard way to do this is to perform a so-called "One-hot Encoding". This is also known as encoding

with dummy variables. Using Pandas, it is possible to perform this encoding on properly typed data using the function `pd.get_dummies()`.

```
In [5]: categorical_encoded_df = pd.get_dummies(categorical_df)

In [6]: categorical_encoded_df.shape

Out[6]: (1451, 359)

In [7]: categorical_encoded_df.sample(5)
```

Out[7]:

Id	MSSubClass_20	MSSubClass_30	MSSubClass_40	MSSubClass_45	\
201	1	0	0	0	
579	0	0	0	0	
1316	0	0	0	0	
783	1	0	0	0	
55	0	0	0	0	

Id	MSSubClass_50	MSSubClass_60	MSSubClass_70	MSSubClass_75	\
201	0	0	0	0	
579	0	0	0	0	
1316	0	1	0	0	
783	0	0	0	0	
55	0	0	0	0	

Id	MSSubClass_80	MSSubClass_85	...	SaleType_ConLw	\
201	0	0	...	0	
579	0	0	...	0	
1316	0	0	...	0	
783	0	0	...	0	
55	1	0	...	0	

Id	SaleType_New	SaleType_Oth	SaleType_WD	SaleCondition_Abnorml	\
201	0	0	1	0	
579	0	0	1	1	
1316	0	0	1	0	
783	0	0	1	0	
55	0	0	1	0	

Id	SaleCondition_AdjLand	SaleCondition_Alloca	SaleCondition_Family	\
201	0	0	0	
579	0	0	0	
1316	0	0	0	
783	0	0	0	
55	0	0	0	

Id	SaleCondition_Normal	SaleCondition_Partial
201	1	0
579	0	0
1316	1	0
783	1	0
55	1	0

[5 rows x 359 columns]

Let us now consider this statistical description of an encoded categorical feature. Note that the categorical feature `MSSubClass` Has been converted it to 15 columns, one for each possible category.

```
In [8]: ms_sub_class_encoded_cols = [col for col in categorical_encoded_df.columns if 'MSSubClas
        ms_sub_class_encoded_cols
```

```
Out[8]: ['MSSubClass_20',
         'MSSubClass_30',
         'MSSubClass_40',
         'MSSubClass_45',
         'MSSubClass_50',
         'MSSubClass_60',
         'MSSubClass_70',
         'MSSubClass_75',
         'MSSubClass_80',
         'MSSubClass_85',
         'MSSubClass_90',
         'MSSubClass_120',
         'MSSubClass_160',
         'MSSubClass_180',
         'MSSubClass_190']
```

We can use this list to filter the full categorical data frame to simply look at the encoded `MSSubClass` feature.

```
In [9]: categorical_encoded_df[ms_sub_class_encoded_cols].head()
```

```
Out[9]: MSSubClass_20  MSSubClass_30  MSSubClass_40  MSSubClass_45  MSSubClass_50  \
        Id
        1                0              0              0              0              0
        2                1              0              0              0              0
        3                0              0              0              0              0
        4                0              0              0              0              0
        5                0              0              0              0              0

            MSSubClass_60  MSSubClass_70  MSSubClass_75  MSSubClass_80  MSSubClass_85  \
        Id
        1                1              0              0              0              0
        2                0              0              0              0              0
        3                1              0              0              0              0
        4                0              1              0              0              0
        5                1              0              0              0              0

            MSSubClass_90  MSSubClass_120  MSSubClass_160  MSSubClass_180  \
        Id
        1                0               0               0               0
        2                0               0               0               0
        3                0               0               0               0
        4                0               0               0               0
        5                0               0               0               0

            MSSubClass_190
        Id
        1                0
        2                0
        3                0
        4                0
        5                0
```

Next, we take a sum for each column and over the whole filtered dataframe. Note that to sum over the whole data frame, we must request the `.values` of the DataFrame which has the effect of converting the

data into a simple Numpy array. This is because the `.sum()` method in Pandas can only be performed over columns (`.sum(axis=1)`) or rows (`.sum(axis=0)`), whereas the `.sum()` method in Numpy can be performed over the entire array.

```
In [10]: (categorical_encoded_df[ms_sub_class_encoded_cols].sum(),
          categorical_encoded_df[ms_sub_class_encoded_cols].values.sum())
```

```
Out[10]: (MSSubClass_20     532
          MSSubClass_30      69
          MSSubClass_40       4
          MSSubClass_45      12
          MSSubClass_50     144
          MSSubClass_60     296
          MSSubClass_70      60
          MSSubClass_75      16
          MSSubClass_80      57
          MSSubClass_85      20
          MSSubClass_90      52
          MSSubClass_120     86
          MSSubClass_160     63
          MSSubClass_180     10
          MSSubClass_190     30
          dtype: int64, 1451)
```

```
In [11]: categorical_encoded_df[ms_sub_class_encoded_cols].shape
```

```
Out[11]: (1451, 15)
```

It is useful to think about the sparsity of the one-hot encoded data. This filtered dataframe, `categorical_encoded_df[ms_sub_class_encoded_cols]` has a shape of `(1451, 15)`, that is, 21765 datapoints, but only 1451 contain a value of 1, the rest containing a value of 0. In other words, 14 out of 15 or 93% of values in this filtered dataframe are 0.

Next, let's look at mean and standard deviation of the filtered dataframe.

```
In [12]: stats = pd.DataFrame()
         stats['mean'] = categorical_encoded_df[ms_sub_class_encoded_cols].mean()
         stats['std'] = categorical_encoded_df[ms_sub_class_encoded_cols].std()
         stats['var'] = categorical_encoded_df[ms_sub_class_encoded_cols].var()
         stats.sort_values('std', ascending=False)
```

```
Out[12]:          mean       std       var
         MSSubClass_20   0.366644  0.482054  0.232376
         MSSubClass_60   0.203997  0.403106  0.162494
         MSSubClass_50   0.099242  0.299090  0.089455
         MSSubClass_120  0.059269  0.236210  0.055795
         MSSubClass_30   0.047553  0.212893  0.045323
         MSSubClass_160  0.043418  0.203867  0.041562
         MSSubClass_70   0.041351  0.199169  0.039668
         MSSubClass_80   0.039283  0.194335  0.037766
         MSSubClass_90   0.035837  0.185949  0.034577
         MSSubClass_190  0.020675  0.142344  0.020262
         MSSubClass_85   0.013784  0.116632  0.013603
         MSSubClass_75   0.011027  0.104464  0.010913
         MSSubClass_45   0.008270  0.090595  0.008207
         MSSubClass_180  0.006892  0.082759  0.006849
         MSSubClass_40   0.002757  0.052450  0.002751
```

We note that most of the one-hot encoded columns have very little variance. In a moment, we'll restrict our analysis to one-hot encoded features that have a variance greater than 0.2. Below is a list of these features. Remember that each of these represents a Boolean variable as to whether or not each row has this particular category-attribute.

```
In [13]: stats = pd.DataFrame()
         stats['mean'] = categorical_encoded_df.mean()
         stats['std'] = categorical_encoded_df.std()
         stats['var'] = categorical_encoded_df.var()
         categorical_encoded_features_significant_variance_stats = stats[stats['var'] > 0.2].sor
         categorical_encoded_features_insignificant_variance_stats = stats[stats['var'] <= 0.2].
         categorical_encoded_features_significant_variance_stats.head(5)
```

```
Out[13]: mean       std       var
         HouseStyle_1Story  0.496899  0.500163  0.250163
         HeatingQC_Ex       0.505858  0.500138  0.250138
         KitchenQual_TA     0.505858  0.500138  0.250138
         FullBath_2         0.524466  0.499573  0.249573
         Fireplaces_0       0.472088  0.499392  0.249393
```

5.7.3 Gelman Scaling

This data set is a mixed data set. It includes both numerical and categorical features. In his 2007 paper, Gelman outlines a simple adjustment to the standard scaling technique we have been using that can help with mixed datasets. The standard scaling technique performs the following transformation

$$Z = \frac{X - \mu}{\sigma}$$

Gelman proposes this alternative

$$Z_g = \frac{X - \mu}{2\sigma}$$

Here, we explore the implications of this.

```
In [14]: numeric_log_df.shape, categorical_df.shape
```

```
Out[14]: ((1451, 23), (1451, 56))
```

First, let's look at the statistics for data prepared by each scaling technique. For ease of viewing, we will only look at the first five features.

```
In [15]: numeric_first_five_features = numeric_log_df.columns[:5]
```

```
In [16]: numeric_log_std_sc_df = (numeric_log_df - numeric_log_df.mean())/numeric_log_df.std()
         numeric_log_gel_sc_df = (numeric_log_df - numeric_log_df.mean())/(2*numeric_log_df.std(
```

```
In [17]: stats = pd.DataFrame()
         stats['mean'] = numeric_log_std_sc_df[numeric_first_five_features].mean()
         stats['std'] = numeric_log_std_sc_df[numeric_first_five_features].std()
         stats['var'] = numeric_log_std_sc_df[numeric_first_five_features].var()
         stats
```

```
Out[17]: mean   std  var
         LotFrontage   2.199301e-14  1.0  1.0
         LotArea       8.004126e-15  1.0  1.0
         YearBuilt    -8.855706e-14  1.0  1.0
         YearRemodAdd  1.895718e-13  1.0  1.0
         MasVnrArea   -8.185504e-16  1.0  1.0
```

```
In [18]: stats = pd.DataFrame()
         stats['mean'] = numeric_log_gel_sc_df[numeric_first_five_features].mean()
         stats['std'] = numeric_log_gel_sc_df[numeric_first_five_features].std()
         stats['var'] = numeric_log_gel_sc_df[numeric_first_five_features].var()
         stats
```

```
Out[18]: mean   std    var
         LotFrontage    1.099651e-14  0.5  0.25
         LotArea        4.002063e-15  0.5  0.25
         YearBuilt     -4.427853e-14  0.5  0.25
         YearRemodAdd   9.478592e-14  0.5  0.25
         MasVnrArea    -4.092752e-16  0.5  0.25
```

Note that the diagonal of each covariance matrix signifies the variance of each feature. With standard scaling, the standard deviation σ of a scaled feature is 1 and the variance σ^2 is also 1. With Gelman scaling, the standard deviation σ of a scaled feature is 0.5. The variance σ^2 is 0.25. Compare this to the standard deviation and variance of the categorical features:

```
In [19]: categorical_encoded_features_significant_variance_stats.head(5)
```

```
Out[19]: mean        std       var
         HouseStyle_1Story  0.496899  0.500163  0.250163
         HeatingQC_Ex       0.505858  0.500138  0.250138
         KitchenQual_TA     0.505858  0.500138  0.250138
         FullBath_2         0.524466  0.499573  0.249573
         Fireplaces_0       0.472088  0.499392  0.249393
```

Note that with Gelman scaling, we are able to directly compare one-hot encoded categorical features with significant variance to our numerical features.

Gelman notes:

> Our procedure scales inputs to be comparable with binary variables that are roughly symmetric: if the probability falls between 0.3 and 0.7, then 2 standard deviations will be between 0.9 and 1. Highly skewed binary inputs still create difficulty in interpretation, however; for example, two standard deviations for a 90 per cent/10 per cent binary variable come to only 0.6. Thus, leaving this binary variable unscaled is not quite equivalent to dividing by two standard deviations. One might argue, however, that when considering rare subsets of the population, a full comparison from 0 to 1 could overstate the importance of the predictor in the regression, hence it might be reasonable to consider this two-standard-deviation comparison, which is less than the comparison of the extremes. Our main point, however, is that 2 standard deviations is a more reasonable scaling than 1—even if neither automatic approach solves all problems of interpretation.

Following these guidelines, we will only compare Gelman scaled numeric features and one-hot encoded categorical features with a variance above 0.2. Note that a variance of 0.2 corresponds approximately to features for whom "2 standard deviations will be between 0.9 and 1".

```
In [20]: categorical_encoded_features_significant_variance = categorical_encoded_df[categorical_
         categorical_encoded_features_insignificant_variance = categorical_encoded_df[categorica
```

```
In [21]: categorical_encoded_features_significant_variance.columns
```

```
Out[21]: Index(['HouseStyle_1Story', 'HeatingQC_Ex', 'KitchenQual_TA', 'FullBath_2',
                'Fireplaces_0', 'FireplaceQu_None', 'BedroomAbvGr_3', 'FullBath_1',
                'Fireplaces_1', 'BsmtQual_TA', 'Foundation_PConc', 'OverallCond_5',
                'GarageCars_2', 'Foundation_CBlock', 'BsmtQual_Gd', 'GarageFinish_Unf',
                'BsmtFullBath_0', 'MasVnrType_None', 'GarageType_Attchd',
                'BsmtFullBath_1', 'KitchenQual_Gd', 'ExterQual_TA', 'HalfBath_0',
                'LotShape_Reg', 'MSSubClass_20', 'HalfBath_1', 'Exterior1st_VinylSd',
                'BsmtExposure_No', 'Exterior2nd_VinylSd', 'LotShape_IR1',
                'ExterQual_Gd', 'MasVnrType_BrkFace', 'HouseStyle_2Story',
                'BsmtFinType1_Unf', 'HeatingQC_TA', 'GarageFinish_RFn',
                'BsmtFinType1_GLQ', 'LotConfig_Inside', 'TotRmsAbvGrd_6'],
               dtype='object')
```

```
In [22]: categorical_encoded_features_insignificant_variance.columns
```

```
Out[22]: Index(['OverallQual_5', 'GarageType_Detchd', 'FireplaceQu_Gd', 'OverallQual_6',
                'GarageCars_1', 'BedroomAbvGr_2', 'GarageFinish_Fin', 'TotRmsAbvGrd_7',
                'RoofStyle_Gable', 'OverallQual_7',
                ...
                'Exterior1st_AsphShn', 'Condition2_RRAn', 'Exterior1st_ImStucc',
                'Condition2_RRAe', 'RoofMatl_Roll', 'RoofMatl_ClyTile', 'Heating_Floor',
                'Exterior2nd_CBlock', 'Exterior1st_CBlock', 'MiscFeature_TenC'],
               dtype='object', length=320)
```

5.7.4 Centering Categorical Features with Significant Variance

We apply one last transformation to the `categorical_encoded_features_significant_variance` dataframe. Namely, we subtract the meaning from each column. In doing this we have an apples to apples comparison between the numeric features and the categorical features with significant variance. Simply subtracting the meaning is known as centering

$$Z_c = X - \mu$$

```
In [23]: categorical_encoded_features_significant_variance_centered = (categorical_encoded_featu
                                                                       categorical_encoded_featu
```

This work was added to `src/preprocessing.py` as we continue.

5.8 Remove Outliers

In the file `src/preprocessing.py`, we perform the preprocessing steps necessary for the next phase of this project. This file does the following:

1. Load and merge data, using the `Id` feature as we did previously
2. Assign correct data types, in particular, designating which features are categorical
3. Handle missing values, using the work we did previously
4. Preprocessing

```
In [1]: run src/preprocessing.py
```

5.8.1 Complete Feature Sets

We are starting to do some fairly complicated feature engineering. It makes sense that we should spend some time thinking about the different data sets we are creating so that we can keep track of what we have.

Standard Scaled Data Set

One data set is the standard scaled data set. For this data set, there is no need to separate the encoded categorical features. These two dataframes comprise a complete data set

- Log Transformed, Standard Scaled Numerical Features (`numeric_log_std_sc_df`)
- Complete One-hot Encoded Categorical Features (`categorical_encoded_df`)

Gelman Scaled Data Set

The other data set is the Gelman scaled data set. For this data set, we have separated the encoded categorical features based on a threshold for variance. These three dataframes comprise a complete data set

- Log Transformed, Gelman Scaled Numerical Features (`numeric_log_gel_sc_df`)
- One-hot Encoded Categorical Features with Significant Variance, Centered (`categorical_encoded_features_significant_variance_centered`)
- One-hot Encoded Categorical Features with Insignificant Variance (`categorical_encoded_features_insignificant_variance`)

Identify Outliers

Next, we will work with the numeric features to identify outliers. As before, we will use the Tukey Method.

```
In [2]: def display_outliers(dataframe, col, param=1.5):
            Q1 = np.percentile(dataframe[col], 25)
            Q3 = np.percentile(dataframe[col], 75)
            tukey_window = param*(Q3-Q1)
            less_than_Q1 = dataframe[col] < Q1 - tukey_window
            greater_than_Q3 = dataframe[col] > Q3 + tukey_window
            tukey_mask = (less_than_Q1 | greater_than_Q3)
            return dataframe[tukey_mask]
```

```
In [3]: print("Column.             Standard      Gelman ")
        print("-------------------------------------------")
        for col in numeric_log_std_sc_df.columns:
            print("{:20} {:12} {}".format(col,
                                          str(display_outliers(numeric_log_std_sc_df, col).shape),
                                          str(display_outliers(numeric_log_gel_sc_df, col).shape)
                                         ))
```

```
Column.             Standard      Gelman
-------------------------------------------
LotFrontage          (122, 23)     (122, 23)
LotArea              (128, 23)     (128, 23)
YearBuilt            (9, 23)       (9, 23)
YearRemodAdd         (0, 23)       (0, 23)
MasVnrArea           (0, 23)       (0, 23)
BsmtFinSF1           (0, 23)       (0, 23)
BsmtFinSF2           (167, 23)     (167, 23)
BsmtUnfSF            (125, 23)     (125, 23)
TotalBsmtSF          (52, 23)      (52, 23)
FirstFlrSF           (7, 23)       (7, 23)
SecondFlrSF          (0, 23)       (0, 23)
LowQualFinSF         (26, 23)      (26, 23)
GrLivArea            (10, 23)      (10, 23)
GarageYrBlt          (1, 23)       (1, 23)
GarageArea           (84, 23)      (84, 23)
WoodDeckSF           (0, 23)       (0, 23)
OpenPorchSF          (0, 23)       (0, 23)
EnclosedPorch        (207, 23)     (207, 23)
ThreeSsnPorch        (24, 23)      (24, 23)
ScreenPorch          (116, 23)     (116, 23)
PoolArea             (7, 23)       (7, 23)
```

```
MiscVal              (52, 23)      (52, 23)
YrSold               (0, 23)       (0, 23)
```

Note that both scaling techniques return the same number of outliers.

Count Multiple Outliers

As before, we will count row that are outlier for more than one feature.

```
In [4]: from collections import Counter
```

```
In [5]: def multiple_outliers(dataframe, count=2):
            raw_outliers = []
            for col in dataframe:
                outlier_df = feature_outliers(dataframe, col)
                raw_outliers += list(outlier_df.index)

            outlier_count = Counter(raw_outliers)
            outliers = [k for k,v in outlier_count.items() if v >= count]
            return outliers
```

```
In [6]: len(multiple_outliers(numeric_log_std_sc_df)), len(multiple_outliers(numeric_log_gel_sc_
```

```
Out[6]: (292, 292)
```

Again, the two scaling techniques return the same number of multiple outliers. Unfortunately, this number of outliers represents an unacceptable loss of data, approximately 20% of our data.

```
In [7]: len(multiple_outliers(numeric_log_std_sc_df))/numeric_log_std_sc_df.shape[0]
```

```
Out[7]: 0.20124052377670573
```

We set the multiple feature count higher and reassess.

```
In [8]: print(len(multiple_outliers(numeric_log_std_sc_df, count=4)), len(multiple_outliers(nume
        print(len(multiple_outliers(numeric_log_std_sc_df, count=4))/numeric_log_std_sc_df.shape
```

```
20 20
0.013783597518952447
```

```
In [9]: print(len(multiple_outliers(numeric_log_std_sc_df, count=5)), len(multiple_outliers(nume
        print(len(multiple_outliers(numeric_log_std_sc_df, count=5))/numeric_log_std_sc_df.shape
```

```
7 7
0.004824259131633356
```

Instances that are an outlier in four or more features amount to 1.3% of the data. Instances that are an outlier in five or more features amount to 0.5% of the data. Both of these represent acceptable losses. Here we will use Instances that are outlier in five or more features.

```
In [10]: numeric_log_std_sc_out_rem_df = numeric_log_std_sc_df.drop(multiple_outliers(numeric_lo
         numeric_log_gel_sc_out_rem_df = numeric_log_gel_sc_df.drop(multiple_outliers(numeric_lo
```

```
In [1]: import matplotlib.pyplot as plt
        import seaborn as sns

        %matplotlib inline
```

```
In [2]: run src/preprocessing.py
```

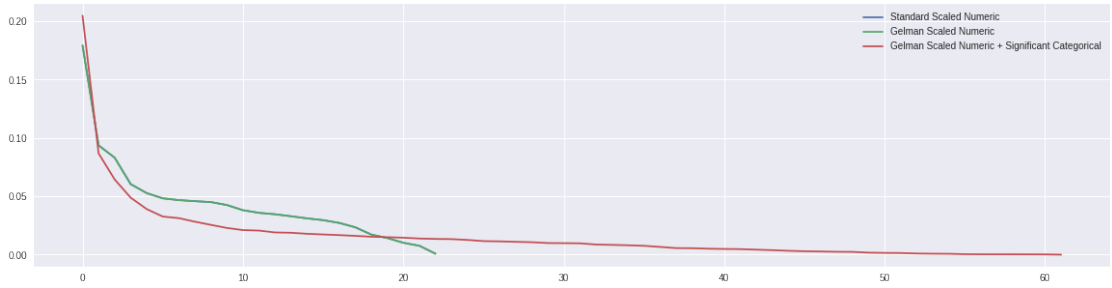
```
In [3]: numeric_gelman_categorical_significant = pd.merge(numeric_log_gel_sc_out_rem_df,
                                                          categorical_encoded_features_significa
                                                          left_index=True, right_index=True)
```

In [4]:
```
from sklearn.decomposition import PCA
pca_log_std_sc_out_rem = PCA()
pca_log_gel_sc_out_rem = PCA()
pca_num_gel_cat = PCA()

pca_log_std_sc_out_rem.fit(numeric_log_std_sc_out_rem_df)
pca_log_gel_sc_out_rem.fit(numeric_log_gel_sc_out_rem_df)
pca_num_gel_cat.fit(numeric_gelman_categorical_significant)
```

Out[4]:
```
PCA(copy=True, iterated_power='auto', n_components=None, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

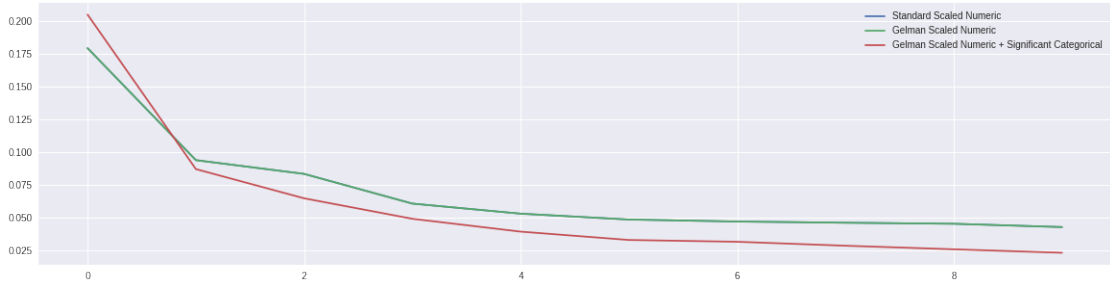
In [5]:
```
plt.figure(figsize=(20,5))
plt.plot(pca_log_std_sc_out_rem.explained_variance_ratio_, label='Standard Scaled Numeri
plt.plot(pca_log_gel_sc_out_rem.explained_variance_ratio_, label='Gelman Scaled Numeric'
plt.plot(pca_num_gel_cat.explained_variance_ratio_, label='Gelman Scaled Numeric + Signi
plt.legend()
```

Out[5]:
```
<matplotlib.legend.Legend at 0x7f2dab5ec160>
```



In [6]:
```
plt.figure(figsize=(20,5))
plt.plot(pca_log_std_sc_out_rem.explained_variance_ratio_[:10], label='Standard Scaled N
plt.plot(pca_log_gel_sc_out_rem.explained_variance_ratio_[:10], label='Gelman Scaled Num
plt.plot(pca_num_gel_cat.explained_variance_ratio_[:10], label='Gelman Scaled Numeric +
plt.legend()
```

Out[6]:
```
<matplotlib.legend.Legend at 0x7f2dab56ccc0>
```



In [7]:
```
pca_log_std_sc_out_rem = PCA(8)
pca_log_gel_sc_out_rem = PCA(8)
pca_num_gel_cat = PCA(8)

pca_log_std_sc_out_rem.fit(numeric_log_std_sc_out_rem_df)
pca_log_gel_sc_out_rem.fit(numeric_log_gel_sc_out_rem_df)
pca_num_gel_cat.fit(numeric_gelman_categorical_significant)
```

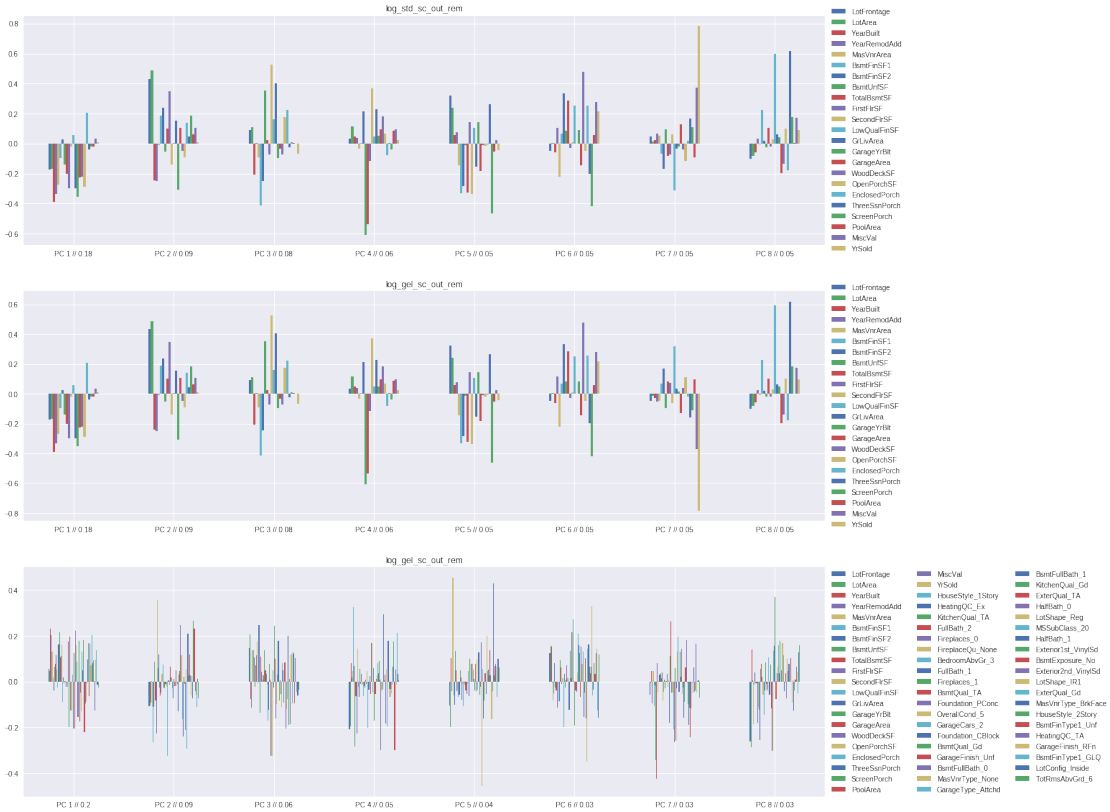
Out[7]:
```
PCA(copy=True, iterated_power='auto', n_components=8, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

In [8]:
```
L_log_std_sc_out_rem = pca_log_std_sc_out_rem.explained_variance_ratio_
L_log_gel_sc_out_rem = pca_log_gel_sc_out_rem.explained_variance_ratio_
```

```
        L_num_gel_cat = pca_num_gel_cat.explained_variance_ratio_
```

In [9]:
```
P_log_std_sc_out_rem = pd.DataFrame(pca_log_std_sc_out_rem.components_, columns=numeric_
P_log_gel_sc_out_rem = pd.DataFrame(pca_log_gel_sc_out_rem.components_, columns=numeric_
P_num_gel_cat = pd.DataFrame(pca_num_gel_cat.components_, columns=numeric_gelman_categor
```

In [10]:
```
_, ax = plt.subplots(3,1,figsize=(20,20))
P_log_std_sc_out_rem.plot(kind='bar', rot=0, title="log_std_sc_out_rem", ax=ax[0]);
P_log_gel_sc_out_rem.plot(kind='bar', rot=0, title="log_gel_sc_out_rem", ax=ax[1])
P_num_gel_cat.plot(kind='bar', rot=0, title="log_gel_sc_out_rem", ax=ax[2])

ax[0].legend(loc='center left', bbox_to_anchor=(1, 0.5), mode='expand')
ax[1].legend(loc='center left', bbox_to_anchor=(1, 0.5), mode='expand')
ax[2].legend(loc='center left', bbox_to_anchor=(1, 0.5), ncol=3)

ax[0].set_xticklabels(['PC ' + str(i+1) +' // ' + str(round(ratio,2)) for i, ratio in e
ax[1].set_xticklabels(['PC ' + str(i+1) +' // ' + str(round(ratio,2)) for i, ratio in e
ax[2].set_xticklabels(['PC ' + str(i+1) +' // ' + str(round(ratio,2)) for i, ratio in e
```



In [11]:
```
P_num_gel_cat_abs = P_num_gel_cat.abs()
```

```
def top_20_features_by_PC_abs(pc_num):
    PC_0_abs_sorted_index = P_num_gel_cat_abs.T.sort_values(pc_num, ascending=False).in
    PC_0_abs_sorted_index
    return P_num_gel_cat[PC_0_abs_sorted_index].T[pc_num].head(20)
```

In [12]:
```
top_20_features_by_PC_abs(0)
```

Out[12]:
```
YearBuilt              0.231171
Foundation_PConc       0.220655
ExterQual_TA          -0.219245
GarageYrBlt            0.215008
```

```
BsmtQual_TA           -0.205763
FullBath_1            -0.203895
ExterQual_Gd           0.203885
YearRemodAdd           0.203380
KitchenQual_TA        -0.200140
FullBath_2             0.197584
KitchenQual_Gd         0.179558
BsmtQual_Gd            0.177149
HeatingQC_Ex           0.176088
GarageFinish_Unf      -0.174514
Exterior1st_VinylSd    0.167226
Exterior2nd_VinylSd    0.165341
GrLivArea              0.163569
OverallCond_5          0.160344
OpenPorchSF            0.159884
Foundation_CBlock     -0.156132
Name: 0, dtype: float64
```

```
In [13]: top_20_features_by_PC_abs(1)
```

```
Out[13]: SecondFlrSF           0.356679
         HouseStyle_1Story    -0.324981
         MSSubClass_20        -0.294268
         HouseStyle_2Story     0.266056
         BsmtFinSF1           -0.263205
         BsmtFullBath_0        0.246356
         BsmtFullBath_1       -0.240936
         BsmtFinType1_Unf      0.231025
         FirstFlrSF           -0.227589
         HalfBath_0           -0.211482
         HalfBath_1            0.208714
         GarageType_Attchd    -0.164676
         Foundation_CBlock    -0.130607
         BsmtExposure_No       0.119555
         GrLivArea             0.118320
         MasVnrType_None       0.114182
         BsmtFinType1_GLQ     -0.112187
         BsmtFinSF2           -0.110484
         LotFrontage          -0.108902
         MasVnrArea           -0.106442
         Name: 1, dtype: float64
```

```
In [14]: top_20_features_by_PC_abs(2)
```

```
Out[14]: FireplaceQu_None      -0.323879
         Fireplaces_0          -0.323879
         GrLivArea              0.247070
         Fireplaces_1           0.245104
         LotArea                0.205050
         HalfBath_0            -0.204557
         HalfBath_1             0.198726
         Exterior1st_VinylSd   -0.190778
         Exterior2nd_VinylSd   -0.187495
         Foundation_CBlock      0.178677
         SecondFlrSF            0.175719
         HouseStyle_1Story     -0.169080
         Foundation_PConc      -0.161491
         LotFrontage            0.148079
         ScreenPorch            0.137980
         MasVnrArea             0.135796
         BsmtFinSF1             0.135651
```

```
GarageYrBlt           -0.135123
MasVnrType_BrkFace     0.127924
HeatingQC_Ex          -0.122449
Name: 2, dtype: float64
```

```
In [15]: numeric_log_std_sc_out_rem_pca_df = pd.DataFrame(pca_log_std_sc_out_rem.transform(numer
                                                          columns=['PC 1', 'PC 2', 'PC 3', 'PC 4
         numeric_gelman_categorical_significant_pca = pd.DataFrame(pca_num_gel_cat.transform(num
                                                          columns=['PC 1', 'PC 2', 'PC 3', 'PC 4
```

5.9 Complete Feature Sets

We are starting to do some fairly complicated feature engineering. It makes sense that we should spend some time thinking about the different data sets we are creating so that we can keep track of what we have.

5.9.1 Dataset 1 - Standard Scaled Data Set

One data set is the standard scaled data set. For this data set, there is no need to separate the encoded categorical features. These two dataframes comprise a complete data set

- Log Transformed, Standard Scaled Numerical Features (`numeric_log_std_sc_out_rem_df`)
- Complete One-hot Encoded Categorical Features (`categorical_encoded_df`)

5.9.2 Dataset 2 - Standard Scaled, PCA Augmented Data Set

One data set is the standard scaled data set, augmented with transformed data from a PCA Run on the numeric features. We know that there is significant redundancy in this data set. These three dataframes comprise a complete data set

- Log Transformed, Standard Scaled Numerical Features (`numeric_log_std_sc_out_rem_df`)
- Complete One-hot Encoded Categorical Features (`categorical_encoded_df`)
- PCA-transformed Numeric Data (`numeric_log_std_sc_out_rem_pca_df`)

5.9.3 Dataset 3 - Gelman Scaled Data Set

Another data set is the Gelman scaled data set. For this data set, we have separated the encoded categorical features based on a threshold for variance. These three dataframes comprise a complete data set

- Log Transformed, Gelman Scaled Numerical Features (`numeric_log_gel_sc_df`)
- One-hot Encoded Categorical Features with Significant Variance, Centered (`categorical_encoded_features_significant_variance_centered`)
- One-hot Encoded Categorical Features with Insignificant Variance (`categorical_encoded_features_insignificant_variance`)

5.9.4 Dataset 4 - Gelman Scaled, PCA Augmented Data Set

Our final data set is the Gelman scaled data set, augmented with transformed data from a PCA Run on the numeric features and categorical features with significant variance. This Data sets also has significant redundancy. For this data set, we have separated the encoded categorical features based on a threshold for variance. These three dataframes comprise a complete data set

- Log Transformed, Gelman Scaled Numerical Features (`numeric_log_gel_sc_df`)
- One-hot Encoded Categorical Features with Significant Variance, Centered (`categorical_encoded_features_significant_variance_centered`)
- One-hot Encoded Categorical Features with Insignificant Variance (`categorical_encoded_features_insignificant_variance`)
- PCA-transformed Numeric and Significant Categorical (`numeric_gelman_categorical_significant_pca`)

```
In [16]: dataset_1 = pd.merge(categorical_encoded_df, numeric_log_std_sc_out_rem_df, left_index=
         dataset_2 = pd.merge(dataset_1, numeric_log_std_sc_out_rem_pca_df, left_index=True, rig
         dataset_3 = pd.merge(numeric_log_gel_sc_out_rem_df, categorical_encoded_features_signif
                                                    left_index=True, right_index=True)
         dataset_3 = pd.merge(dataset_3, categorical_encoded_features_insignificant_variance, le
         dataset_4 = pd.merge(dataset_3, numeric_gelman_categorical_significant_pca, left_index=
```

```
In [17]: (dataset_1.isnull().sum().sum(),
          dataset_2.isnull().sum().sum(),
          dataset_3.isnull().sum().sum(),
          dataset_4.isnull().sum().sum())
```

```
Out[17]: (0, 0, 0, 0)
```

```
In [18]: (dataset_1.shape,
          dataset_2.shape,
          dataset_3.shape,
          dataset_4.shape)
```

```
Out[18]: ((1444, 382), (1427, 390), (1444, 382), (1427, 390))
```

```
In [57]: import matplotlib.pyplot as plt
         import seaborn as sns

         %matplotlib inline
```

```
In [58]: run src/preprocessing.py
```

5.10 Model Selection: Cross-Validation

In the next phase of this project we move into developing our machine learning models. We have previously about model selection and have considered managing the Bias-Variance Tradeoff as we fit our predictive model. We primarily focused on identifying the simplest possible model as a way to making sure that our model generalizes to new data. Now we expand on this by examining three new concepts in model assessment and selection.

1. using cross-validation to study model variance
2. applying regularization to help our models generalize
3. using emsembling to help our models generalize

One commonly held misconceptions is that cross-validation can to help models to generalize. This is not the case. Rather, cross-validation can be used to help to identify potential issues and to optimize model hyperparameters toward the end of choosing the best possible model.

5.10.1 The Validation Set Approach

Cross-validation is a resampling technique and is simply the creative use of collected data. We have already seen a very simple cross-validation approach, the train-test split also called The Validation Set Approach.


```
In [59]: from time import time
         from sklearn.model_selection import train_test_split
```

```
In [60]: (dataset_1.shape,
          dataset_2.shape,
          dataset_3.shape,
          dataset_4.shape)
```

```
Out[60]: ((1444, 382), (1444, 390), (1444, 382), (1444, 390))
```

```
In [61]: np.testing.assert_allclose(dataset_1.index, target_1.index)
         np.testing.assert_allclose(dataset_2.index, target_2.index)
         np.testing.assert_allclose(dataset_3.index, target_3.index)
         np.testing.assert_allclose(dataset_4.index, target_4.index)
```

```
In [63]: ttsplit_1 = train_test_split(dataset_1, target_1, test_size=0.4, random_state=0)
         ttsplit_2 = train_test_split(dataset_2, target_2, test_size=0.4, random_state=0)
         ttsplit_3 = train_test_split(dataset_3, target_3, test_size=0.4, random_state=0)
         ttsplit_4 = train_test_split(dataset_4, target_4, test_size=0.4, random_state=0)
```

```
In [64]: def fit_score(model, data):
             X_train = data[0]
             X_test  = data[1]
             y_train = data[2]
             y_test  = data[3]

             start = time()
             model.fit(X_train, y_train)
             end = time() - start
             return model.score(X_test, y_test), end
```

```
In [65]: from sklearn.linear_model import Lasso, Ridge
```

```
In [66]: print(fit_score(Ridge(max_iter=1E5), ttsplit_1))
         print(fit_score(Ridge(max_iter=1E5), ttsplit_2))
         print(fit_score(Ridge(max_iter=1E5), ttsplit_3))
         print(fit_score(Ridge(max_iter=1E5), ttsplit_4))
```

```
(0.89860862751808601, 0.061138153076171875)
(0.89858121850018868, 0.0879824161529541)
```

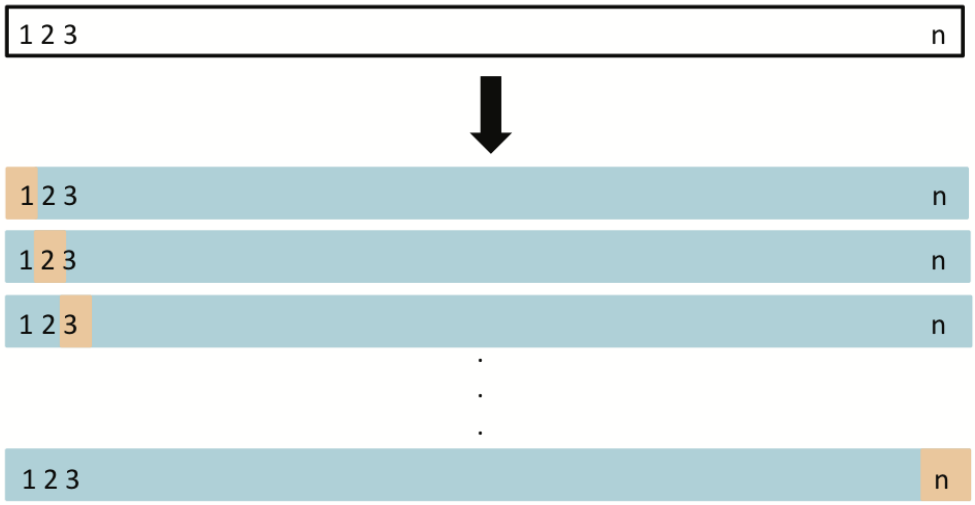
```
(0.89924977700919984, 0.02467799186706543)
(0.8993019594303594, 0.016490697860717773)
```

```
In [67]: print(fit_score(Lasso(max_iter=1E4), ttsplit_1))
         print(fit_score(Lasso(max_iter=1E5), ttsplit_2))
         print(fit_score(Lasso(max_iter=1E4), ttsplit_3))
         print(fit_score(Lasso(max_iter=1E5), ttsplit_4))
```

```
(0.87587594870369756, 1.8139793872833252)
(0.87587068760144149, 13.942325830459595)
(0.87344492815283581, 4.486565113067627)
(0.8734547160862195, 9.617893695831299)
```

5.10.2 Leave-One-Out Cross-Validation

An alternative to using a single validation set is using **leave-one-out cross-validation** (LOOCV).



Here, instead of creating two sets, we create n sets and fit n models. Using this method, each data point is used as a testing point exactly once. To assess the performance we simply take the average over all models

$$\text{CV}_n = \mathbb{E}\left[MSE(f_i)\right]$$

One draw back to this approach is the substantial time required to set a model for each data point.

```
In [68]: from sklearn.model_selection import LeaveOneOut
```

```
In [70]: def fit_score_loo(model, dataset, target):
             loo = LeaveOneOut()
             scores = []
             start = time()
             for train, test in loo.split(dataset, target):
                 train = dataset.index[train]
                 test = dataset.index[test]

                 X_train = dataset.loc[train]
                 X_test  = dataset.loc[test]
                 y_train = dataset.loc[train]
```

```
            y_test  = dataset.loc[test]

            model.fit(X_train, y_train)
            scores.append(model.score(X_test, y_test))

        end = time() - start
        scores = np.array(scores)
        print("Mean: {:6} Variance: {:6} Time: {:6}".format(scores.mean(), scores.var(), en
```

```
In [71]: print(fit_score_loo(Ridge(), dataset_1, target_1))
         print(fit_score_loo(Ridge(), dataset_2, target_2))
         print(fit_score_loo(Ridge(), dataset_3, target_3))
         print(fit_score_loo(Ridge(), dataset_4, target_4))
```

```
Mean:    0.0 Variance:    0.0 Time: 179.01318764686584
None
Mean:    0.0 Variance:    0.0 Time: 187.70759654045105
None
Mean:    0.0 Variance:    0.0 Time: 167.30623126029968
None
Mean:    0.0 Variance:    0.0 Time: 159.89035868644714
None
```

```
In [72]: print(fit_score_loo(Lasso(), dataset_1, target_1))
         print(fit_score_loo(Lasso(), dataset_2, target_2))
         print(fit_score_loo(Lasso(), dataset_3, target_3))
         print(fit_score_loo(Lasso(), dataset_4, target_4))
```

```
Mean:    0.0 Variance:    0.0 Time: 5569.9769694805145
None
Mean:    0.0 Variance:    0.0 Time: 5727.205169916153
None
Mean:    0.0 Variance:    0.0 Time: 5659.567879199982
None
Mean:    0.0 Variance:    0.0 Time: 5761.360241889954
None
```

5.10.3 K-Fold Cross-Validation

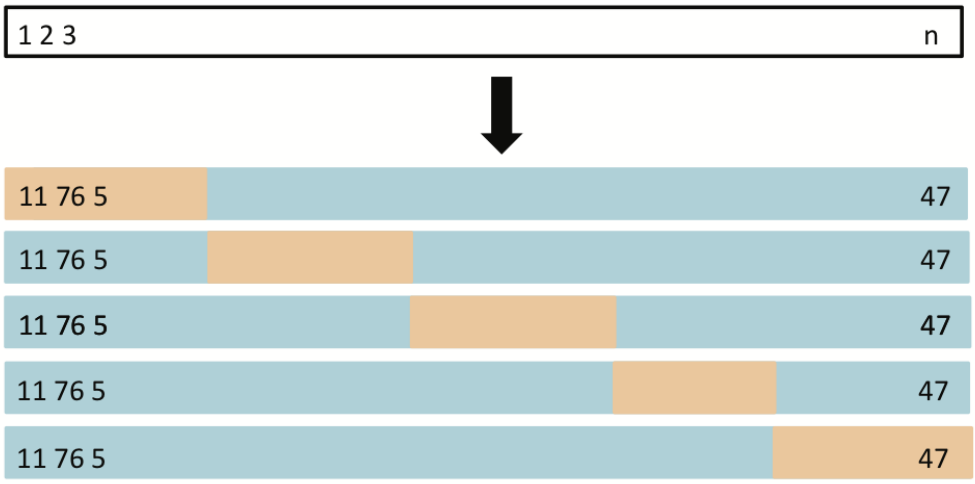
It is usually not practical to use LOOCV. Unacceptable alternative is to use **k-fold cross-validation** (KCV). In this method the data set is split into k groups. Then, k models are fit. Uses exactly one of the groups as a validation set And the remaining data as the training set. As before, the cross validation score is simply the average of the scores across all of the models

$$\text{CV}_k = \mathbb{E}\left[MSE(f_i)\right]$$

Typical values of k are $k = 5$ or $k = 10$.

```
In [21]: from sklearn.model_selection import KFold
```

```
In [43]: X_df = pd.DataFrame([
             {'x' : 1},
             {'x' : 2},
             {'x' : 3},
             {'x' : 4},
             {'x' : 5},
             {'x' : 6},
```




```
            {'x' : 7},
            {'x' : 8},
            {'x' : 9},
            {'x' : 10},
        ])

        X_df.index = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

```
In [44]: X_df = X_df.drop('E')
```

```
In [45]: X_df
```

```
Out[45]: x
         A   1
         B   2
         C   3
         D   4
         F   6
         G   7
         H   8
         I   9
         J  10
```

```
In [46]: kf = KFold(n_splits=3)
         splitter = kf.split(X_df)
```

```
In [47]: train, test = next(splitter)
         train
```

```
Out[47]: array([3, 4, 5, 6, 7, 8])
```

```
In [48]: train = X_df.index[train]
```

```
In [49]: train
```

```
Out[49]: Index(['D', 'F', 'G', 'H', 'I', 'J'], dtype='object')
```

```
In [54]: def fit_score_kfold(model, dataset, target, folds=5):
             kf = KFold(n_splits=folds)
             scores = []
             start = time()
             for train, test in kf.split(dataset, target):
                 train = dataset.index[train]
```

```
            test = dataset.index[test]

            X_train = dataset.loc[train]
            X_test  = dataset.loc[test]
            y_train = target.loc[train]
            y_test  = target.loc[test]

            model.fit(X_train, y_train)
            scores.append(model.score(X_test, y_test))

        scores = np.array(scores)
        end = time() - start

        print("Mean: {:6} Variance: {:6} Time: {:6}".format(scores.mean(), scores.var(), en
```

```
In [55]: fit_score_kfold(Ridge(), dataset_1, target_1)
         fit_score_kfold(Ridge(), dataset_2, target_2)
         fit_score_kfold(Ridge(), dataset_3, target_3)
         fit_score_kfold(Ridge(), dataset_4, target_4)
```

```
Mean: 0.8853121172449191 Variance: 0.00022291442467840 79 Time: 0.1946556568145752
Mean: 0.8853098599136107 Variance: 0.00022254648260988997 Time: 0.24270319938659668
Mean: 0.8855857104448723 Variance: 0.00022384015204386532 Time: 0.2041628360748291
Mean: 0.8855848356167636 Variance: 0.00022431601124553446 Time: 0.33058595657348633
```

```
In [56]: fit_score_kfold(Lasso(), dataset_1, target_1)
         fit_score_kfold(Lasso(), dataset_2, target_2)
         fit_score_kfold(Lasso(), dataset_3, target_3)
         fit_score_kfold(Lasso(), dataset_4, target_4)
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/linear_model/coordinate_descent.py:491: Convergen
  ConvergenceWarning)
```

```
Mean: 0.870535279080752 Variance: 0.000367807488798026 Time: 3.989431381225586
Mean: 0.8705024797773 Variance: 0.00037061494130101127 Time: 3.5346145629882812
Mean: 0.8676419438563763 Variance: 0.00021040580637601814 Time: 3.3818252086639404
Mean: 0.8675954486257649 Variance: 0.00020691094833489974 Time: 3.176971912384033
```

Stratified Shuffle Split

```
In [53]: from sklearnearn.cross_validation import StratifiedKFold
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/cross_validation.py:41: DeprecationWarning: This
  "This module will be removed in 0.20.", DeprecationWarning)
```

Bias-Variance Trade-Off for k-Fold Cross-Validation

In terms of bias, it is clear that LOOCV will have lower bias than KCV when $k < n$. This is because each model is trained using $n - 1$ points which is nearly all of the training data. Since KCV uses less of the data, it has less ability to learn the phenomenon represented by the data and is therefore more biased then LOOCV.

On the other hand, LOOCV has more variance than KCV. This is because LOOCV involve the fitting and then averaging of performance of n models, whereas KCV does this over k models. Furthermore, the n LOOCV models are more correlated with each other than are the k KCV models. This is clear because each LOOCV model is identical to any other LOOCV model save for one point. Meanwhile each KCV model differs from any other KCV model in n/k points. It can be shown that the meani of highly correlated quantities has higher variance then does the mean of quantities that are not as highly correlated. In other words, the LOOCV has higher variance than does the KCV.

5.11 Advanced Linear Models

In general, linear models are fit by minimizing a loss function. This function will be a function of the β coefficients for each feature. The best set of β values is the set that gives the smallest value of the loss function.

The most common loss function is

$$\mathcal{L}(\beta) = \sum_{i=1}^{n} (y - \widehat{y})^2 = (y - X\beta)^T (y - X\beta)$$

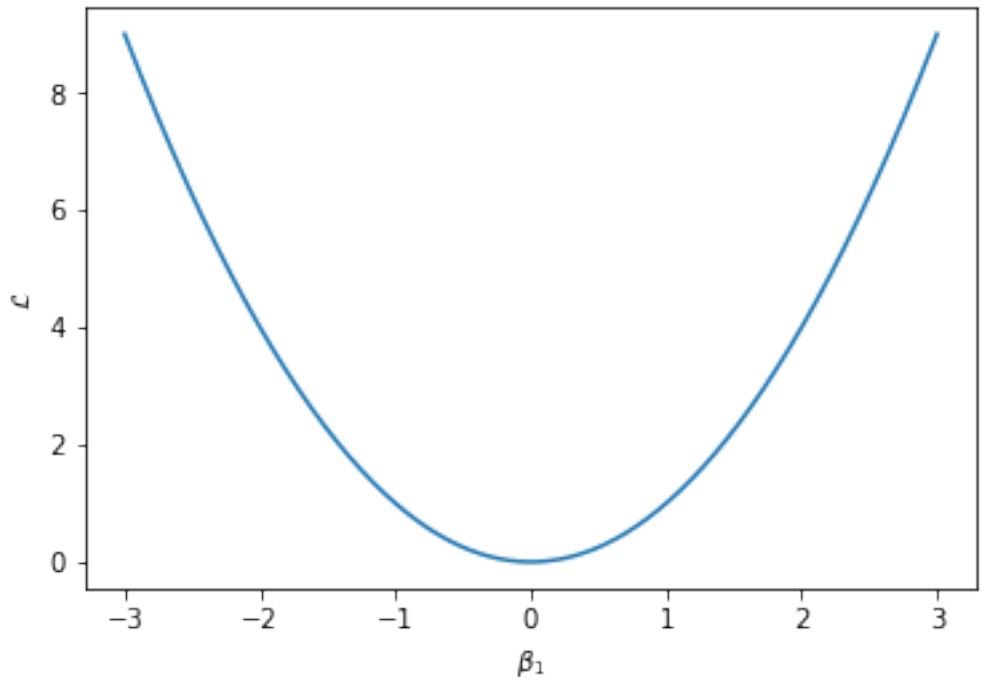
Generally speaking, this equation is a second-degree polynomial or quadratic.

If we had one feature, then we would be seeking the best line

$$\widehat{f} = \beta_0 + \beta_1 x$$

The loss $\mathcal{L}$ as a function of β_1 would be the familiar parabola

```
In [4]: import numpy as np
        import matplotlib.pyplot as plt

In [6]: bbeta = np.linspace(-3,3,1000)
        lloss = bbeta**2
        plt.plot(bbeta, lloss)
        plt.xlabel('$\\beta_1$')
        plt.ylabel('$\\mathcal{L}$')

Out[6]: Text(0,0.5,'$\\mathcal{L}$')
```



If we were to plot the loss versus β_0 and β_1, it would be a 3-D paraboloid where the height of the paraboloid is the associated loss $\mathcal{L}$

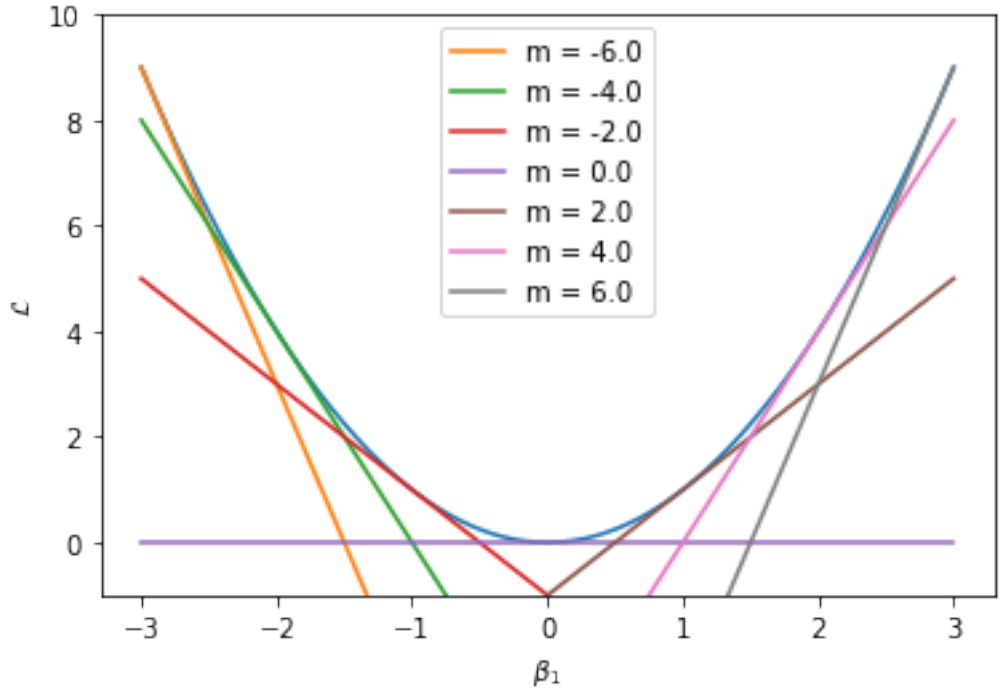
This would generalize to higher dimensions. The important thing is that there is always a single **global** minimum i.e. *the loss function is convex*.

5.11.1 Finding a Minimum

Given a parabol(oid), we can find the minimum using calculus. Recall that the derivative of a function represents the rate of change (slope) of the function at a given point.

```
In [8]: bbeta = np.linspace(-3,3,1000)
        lloss = bbeta**2

        test_points = np.linspace(-3,3,7)
        slopes = 2*test_points
        losses = test_points**2

        slope_line = lambda m, x, x1, y1: m*(x - x1) + y1
        plt.plot(bbeta, lloss)
        for m, x1, y1 in zip(slopes, test_points, losses):
            plt.plot(bbeta, slope_line(m, bbeta, x1, y1), label='m = {}'.format(m))
        plt.ylim(-1, 10)
        plt.legend()
        plt.xlabel('$\\beta_1$')
        plt.ylabel('$\\mathcal{L}$')
```

```
Out[8]: Text(0,0.5,'$\\mathcal{L}$')
```



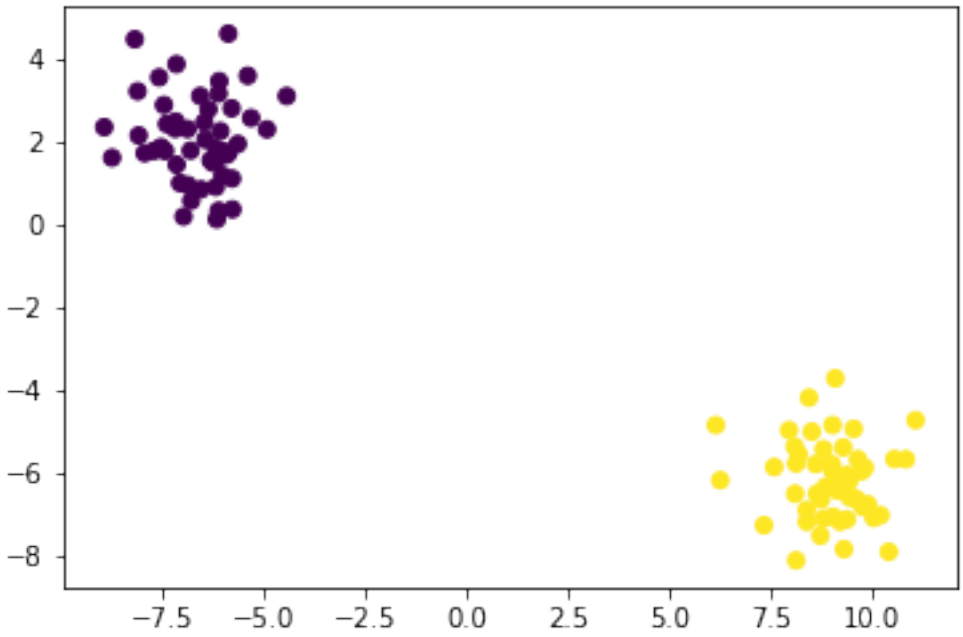
Using Calculus, we can find the minimum of our loss function by finding where the slope is zero. In other words, we can find the minimum of the loss function by finding where the derivative of the loss function is zero.

How Logistic Regression is a Linear Model

```
In [9]: import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        from sklearn.datasets import make_blobs
```

```
In [10]: X, y = make_blobs(centers=2)
         X = pd.DataFrame(X)

         plt.scatter(X[0], X[1], c=y)
```

```
Out[10]: <matplotlib.collections.PathCollection at 0x7f4ebf415588>
```




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

         lr = LogisticRegression()
         lr.fit(X, y)
```

```
Out[11]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False)
```

```
In [13]: lr.coef_, lr.intercept_
```

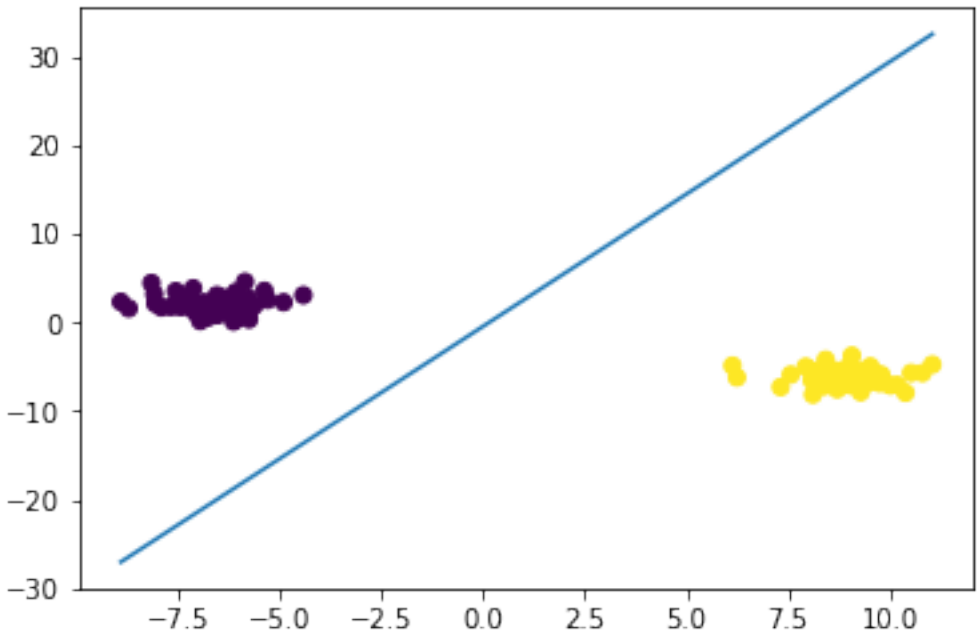
```
Out[13]: (array([[ 0.84040073, -0.28232719]]), array([-0.11378463]))
```

```
In [14]: A = lr.coef_[0][0]
         B = lr.coef_[0][1]
         C = lr.intercept_

         xx = np.linspace(min(X[0]), max(X[0]))
         y_hat = lambda x: -A/B*x - C/B
```

```
In [ ]: X[2] = X[0]**2
        X[3] = X[1]**2
        X[4] = X[0]*X[1]
```

```
In [15]: plt.scatter(X[0], X[1], c=y)
         plt.plot(xx, y_hat(xx))
```

```
Out[15]: [<matplotlib.lines.Line2D at 0x7f4ebcf38f60>]
```



The goal with logistic regression is to find the line that "best splits" the data.

You can imagine that one way to do this would be to have an equation of the form

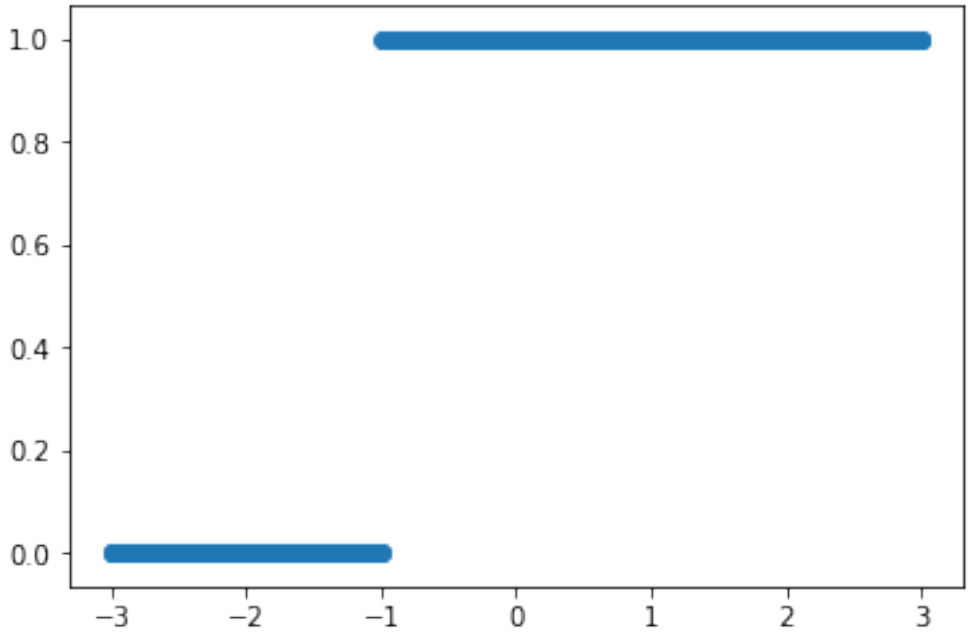
$$\widehat{y} = \beta_0 + \beta_1 x$$

for one variable. If the equation returns a result greater then 0.5, we would call it class "1". It less than 0.5, class "2".

$$\text{Class}(x) = \begin{cases} 1 & : \widehat{y} > 0.5 \\ 0 & : \widehat{y} \leq 0.5 \end{cases}$$

This might look like

```
In [16]: xx = np.linspace(-3,3, 1000)
         y_hat = lambda x: 1 + 0.5*x
         cls = lambda x: np.piecewise(x, [y_hat(x) > 0.5, y_hat(x) <= 0.5], [1, 0])

In [17]: plt.scatter(xx, cls(xx))

Out[17]: <matplotlib.collections.PathCollection at 0x7f4ebce71a20>
```



There is an issue with this function in terms of our loss function. The most common loss function is

$$\mathcal{L}(\beta) = \sum_{i=1}^{n} (y - \widehat{y})^2 = (y - X\beta)^T (y - X\beta)$$

Here the possible value of y are the possible classes, that is $y \in \{0, 1\}$.

Here the possible value of $\widehat{y}$ are also the possible classes, that is $\widehat{y} \in \{0, 1\}$.

The implications of this are that loss function is not continous and more importantly the derivative has points at which it is not defined. **We can not use calculus to solve this problem**.

This led to two innovations that are fundamental to modern machine learning (and especially deep learning):

1. the sigmoid
2. gradient descent

5.11.2 The Sigmoid Function

The sigmoid function is a way to make the logistic regression equation continuous. Instead of

$$\widehat{f}(x) = \begin{cases} 1 & : \beta_0 + \beta_1 x > 0.5 \\ 0 & : \beta_0 + \beta_1 x \leq 0.5 \end{cases}$$

we use the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

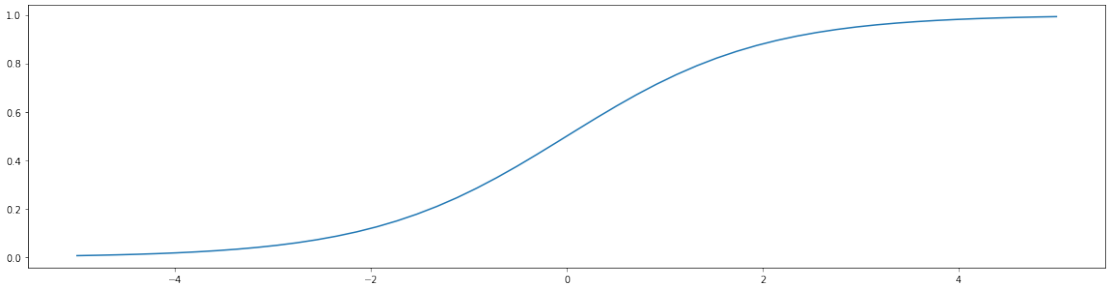
to build our logistic regression function so that

$$\widehat{f}(x) = \sigma(\beta_0 + \beta_1 x)$$

The sigmoid function gets its name because it resembles an "s"

```
In [18]: plt.figure(figsize=(20,5))
         xx = np.linspace(-5, 5)
         yy = 1/(1+np.exp(-xx))
         plt.plot(xx, yy)
```

```
Out[18]: [<matplotlib.lines.Line2D at 0x7f4ebce4ed30>]
```



If x is very large then e^{-x} is very small and goes to zero, therefore, $\widehat{f}(x)$ is close to 1.

If x is very small then e^{-x} is very large and goes to infinity. Because $\frac{1}{\infty} \sim 0$, we can say that $\widehat{f}(x)$ is close to 0.

Note that if x is zero, then e^{-x} is 1 and therefore $\widehat{f}(x) = \frac{1}{2}$.

We interpret the output of the logistic function to be a probability that an input is a certain class.

One additional point. We can think of this in terms of the following DAG:

Although the logistic function is now continuous its non-linearity still makes it challenging to work with and in general we do not solve this by setting the derivative of the loss function to zero.

5.11.3 Gradient Descent

The math behind gradient descent is extremely complicated.

The algorithm begins by selecting β values at random. The Data is passed through the pipeline to obtain the loss for the initial β values.

Given the loss value, we are able to discern the direction in which we would need to update the β values in order to improve the loss. The direction we need to travel in order to improve the loss is called the **gradient**.

Recall that the shape of the loss is a paraboloid. We want to **descend** to the bottom of the loss paraboloid.

To do this we repeatedly pass the known data through the pipeline descending one step at a time.

This generalizes to a more complex linear DAG via a process called error or **backward propagation**.

The backward propagation algorithm is the fundamental algorithm for the branch of machine learning known as **deep learning**.

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

        %matplotlib inline
```

5.12 Overview of regularization

The goal of "regularizing" regression models is to **structurally prevent overfitting by imposing a penalty on the coefficients** of the model.

Regularization methods like the Ridge and Lasso add this additional "penalty" on **the size of coefficients** to the loss function. When the loss function is minimized, this additional component is added to the residual sum of squares.

In other words, the minimization becomes a balance between the error between: 1. predictions and true values 2. the size of the coefficients.

The two most common types of regularization are the **Lasso**, **Ridge**. There is a mixture of them called the **Elastic Net**. We will take a look at the mathematics of regularization and the effect these penalties have on model fits.

From ISLR, Ch. 6

The image on the left represents regularization via the Lasso. The image on the right regularization via a Ridge Regression.

The red lines represent countours of a paraboloid signifying the loss function.

The point in the center is the nadir of the paraboloid and represents the minimum of the loss function.

The trick with regularization is to get as close to the bottom of the paraboloid while staying within the area defined by the regularization function. One interpretation of this is think of the regularizataion as a "budget". We want to get to the center of the paraboloid, but we have to stay within our budget to do so.

5.13 Review: least squares loss function

Ordinary least squares regression minimizes the residual sum of squares (RSS) to fit the data:

$$\epsilon_i = y_i - \widehat{y}_i$$

$$\mathcal{L}(\beta) = \sum_{i=1}^{n} \epsilon_i^2 = (y - X\beta)^T (y - X\beta)$$

5.14 The Ridge penalty

Ridge regression adds the sum of the squared (non-intercept!) β values to the loss function:

$$\mathcal{L}_{ridge}(\beta) = \sum_{i=1}^{n} \epsilon_i^2 + \alpha \sum \beta_i^2 = (y - X\beta)^T (y - X\beta) + \alpha \beta^T \beta$$

Now, to fit the β vector we will minimize $\mathcal{L}_{ridge}(\beta)$.

Again, we take the derivative so that

$$\frac{d}{d\beta}\mathcal{L}(\beta) = 2X^TX\beta - 2X^Ty + 2\lambda\beta$$

And set the result equal to zero so that

$$X^TX\beta + \lambda\beta = X^Ty$$

$$(X^TX + \lambda I)\beta = X^Ty$$

Then

$$\beta = (X^TX + \lambda I)^{-1}X^Ty$$

---

or

```
beta = inv(X.T.dot(X)+lambda*np.eye(n)).dot(X.T).dot(y)
```

Notes:

- $\beta^T\beta$ can be written as $||\beta||_2^2$ and is known as the ℓ_2 norm.
- α is a constant for the *strength* of the regularization parameter.
- The higher this value, the greater the impact of this new component in the loss function.
- Thinking of our budget for reaching the bottom of the bowl, the **bigger** the λ, the **smaller** our budget!
- The **bigger** the λ, the **smaller** our coefficients

5.15 The Lasso penalty

The Lasso regression takes a different approach. Instead of adding the sum of *squared* β coefficients to the RSS, it adds the sum of the *absolute values* of the β coefficients:

$$\mathcal{L}_{lasso}(\beta) = \sum_{i=1}^{n}\epsilon_i^2 + \lambda\sum|\beta_i| = (y - X\beta)^T(y - X\beta) + \lambda|\beta|$$

WARNING $|\beta|$ is not differentiable.

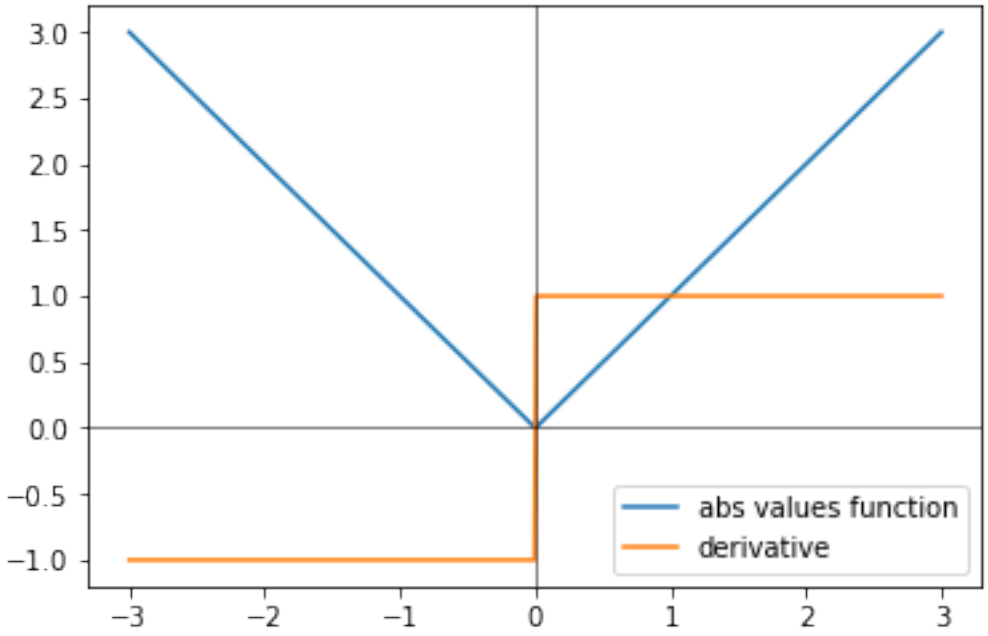
Therefore, there is no closed form solution to the Lasso i.e. we can not solve it using Algebra!

Think about an absolute value curve.

There is a kink in this function at $x = 0$ and the derivative is not continuous.

```
In [2]: x = np.linspace(-3,3,500)
        plt.plot(x,  np.piecewise(x, [x < 0, x >= 0], [lambda x: -x, lambda x: x]), label='abs v
        plt.plot(x,  np.piecewise(x, [x < 0, x >= 0], [-1, 1]), label='derivative')
        plt.axvline(c='black',lw=.5)
        plt.axhline(c='black',lw=.5)
        plt.legend()
```

```
Out[2]: <matplotlib.legend.Legend at 0x7ff0e7c82a20>
```



Notes:

- $|\beta|$ can be written as $|\beta|_1$ and is known as the ℓ_1 norm.
- λ is a constant for the *strength* of the regularization parameter.
- The higher this value, the greater the impact of this new component in the loss function.
- Thinking of our budget for reaching the bottom of the bowl, the **bigger** the λ, the **smaller** our budget!
- The **bigger** the λ, the **smaller** our coefficients

5.16 Elastic Net penalty

Elastic Net is simply a combination of the Lasso and the Ridge regularizations. It adds *both* penalties to the loss function:

$$\mathcal{L}_{lasso}(\beta) = \sum_{i=1}^{n} |\epsilon_i| + \lambda_1 \sum |\beta_i| + \lambda_2 \sum \beta_i^2$$

In the elastic net, the effect of the Ridge vs. the Lasso is balanced by the two lambda parameters.

5.17 What is the effect of regularization?

An important aspect of this data, which is a reason why we might choose to use regularization, is that there is **multicollinearity** in the data. The term multicollinearity means that there are high correlations between predictor variables in your model.

This can lead to a variety of problems including: 1. The effect of predictor variables estimated by your regression will depend on what other variabes are included in your model. 2. Predictors can have wildly different effects depending on the observations in your sample, and small changes in samples can result in very different estimated effects. 3. With very high multicollinearity, the inverse matrix the computer calculates may not be accurate. 4. We can no longer interpret a coefficient on a variable as the effect on the target of a one unit increase in that variable holding the other variables constant. This is because when predictors are strongly correlated, there is not a scenario in which one variable can change without a conditional change in another variable.

The Ridge is best suited to deal with multicollinearity. Lasso also deals with multicollinearity between variables, but in a more brutal way (it "zeroes out" the less effective variable).

The Lasso is particularly useful when you have redundant or unimportant variables. If you have 1000 variables in a dataset the Lasso can perform "feature selection" automatically for you by forcing coefficients to be zero.

5.17.1 Load the Ames Data

This version has red and white wines concatenated together and tagged with a binary 1,0 indicator (1 is red wine). There are many other variables purportedly related to the rated quality of the wine.

```
In [3]: run src/preprocessing.py
```

```
<matplotlib.figure.Figure at 0x7ff0e4694160>
```

```
In [4]: dataset_2.sample(4)
```

```
Out[4]: MSSubClass_20  MSSubClass_30  MSSubClass_40  MSSubClass_45  \
        Id
        606               0              0              0              0
        1314              0              0              0              0
        1031              0              0              0              0
        1186              0              0              0              0

              MSSubClass_50  MSSubClass_60  MSSubClass_70  MSSubClass_75  \
        Id
        606               0              1              0              0
        1314              0              1              0              0
        1031              0              0              0              0
        1186              1              0              0              0

              MSSubClass_80  MSSubClass_85    ...       MiscVal    YrSold      PC 1  \
        Id                                    ...
        606               0              0    ...    -0.191027  0.891266 -1.021968
        1314              0              0    ...    -0.191027  1.642892 -3.185680
        1031              0              0    ...    -0.191027 -1.365857  2.746015
        1186              0              0    ...    -0.191027 -1.365857  2.867996

                  PC 2      PC 3      PC 4      PC 5      PC 6      PC 7      PC 8
        Id
```

```
606   1.505865  1.020785  0.407724 -2.111995 -2.296613 -1.170928  0.793622
1314  0.274669  1.930689  0.622950  0.498625  0.712393 -1.486736 -0.694920
1031 -0.419109  2.974298 -0.519749  0.860368  0.738342  1.175256 -0.139425
1186  1.000860  1.168257 -0.505635 -0.475389  0.002616  0.890070 -0.655253

[4 rows x 390 columns]
```

In [5]:
```
target_2.shape, dataset_2.shape
```

Out[5]:
```
((1444,), (1444, 390))
```

5.18 Visualizing the Ridge

Import the `Ridge` model class from sklearn.

In [6]:
```
from sklearn.linear_model import Ridge
```

This function iterates over a series of different alpha regularization parameters. The alpha is sklearn's equivalent of the lambda value in the formula that multiples the square of betas from the equation.

The function stores the results of the model so that we can plot them interactively.

In [7]:
```
def ridge_coefs(X, Y, alphas):

    # set up the list to hold the different sets of coefficients:
    coefs = []

    # Set up a ridge regression object
    ridge_reg = Ridge()

    # Iterate through the alphas fed into the function:
    for a in alphas:

        # On each alpha reset the ridge model's alpha to the current one:
        ridge_reg.set_params(alpha=a)

        # fit or refit the model on the provided X, Y
        ridge_reg.fit(X, Y)

        # Get out the coefficient list
        coefs.append(ridge_reg.coef_)

    return coefs
```

Alpha values for the ridge are best visualized on a logarithmic "magnitude" scale. Essentially, the effect of alpha on the coefficients does not increase linearly but by orders of magnitude.

In [9]:
```
np.logspace(-2,2,5)
```

Out[9]:
```
array([  1.00000000e-02,   1.00000000e-01,   1.00000000e+00,
         1.00000000e+01,   1.00000000e+02])
```

In [8]:
```
# np.logspace gives us points between specified orders of magnitude on a logarithmic sca
r_alphas = np.logspace(-3, 12, 45)

# Get the coefficients for each alpha for the Ridge, using the function above
r_coefs = ridge_coefs(dataset_2, target_2, r_alphas)
```

The plotting function below will:

- Plot the effect of changing alpha on the coefficient size on a **path** graph
- Plot the effect of changing alpha on the coefficient size on a **bar** graph

Each one gives informative information. It's just two different ways of visualizing the same thing. The chart is interactive so you can play around with the values of alpha across the specified range above.

In [10]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [11]:
```
# The cycler package lets us "cycle" throug colors.
# Just another thing i had to look up on stackoverflow. That's my life.
from cycler import cycler

def coef_plotter(alphas, coefs, to_alpha, regtype='ridge'):

    # Get the full range of alphas before subsetting to keep the plots from
    # resetting axes each time. (We use these values to set static axes later).
    amin = np.min(alphas)
    amax = np.max(alphas)

    # Subset the alphas and coefficients to just the ones below the set limit
    # from the interactive widget:
    alphas = [a for a in alphas if a <= to_alpha]
    coefs = coefs[0:len(alphas)]

    # Get some colors from seaborn:
    colors = sns.color_palette("husl", len(coefs[0]))

    # Get the figure and reset the size to be wider:
    fig = plt.figure()
    fig.set_size_inches(18,5)

    # We have two axes this time on our figure.
    # The fig.add_subplot adds axes to our figure. The number inside stands for:
    #[figure_rows|figure_cols|position_of_current_axes]
    ax1 = fig.add_subplot(121)

    # Give it the color cycler:
    ax1.set_prop_cycle(cycler('color', colors))

    # Print a vertical line showing our current alpha threshold:
    ax1.axvline(to_alpha, lw=2, ls='dashed', c='k', alpha=0.4)

    # Plot the lines of the alphas on x-axis and coefficients on y-axis
    ax1.plot(alphas, coefs, lw=2)

    # set labels for axes:
    ax1.set_xlabel('alpha', fontsize=20)
    ax1.set_ylabel('coefficients', fontsize=20)

    # If this is for the ridge, set this to a log scale on the x-axis:
    ax1.set_xscale('log')
    ax1.set_yscale('log')

    # Enforce the axis limits:
    ax1.set_xlim([amin, amax])
    ax1.set_ylim(1E-2,1E5)

    # Put a title on the axis
```

```
    ax1.set_title(regtype+' coef paths\n', fontsize=20)

    # Get the ymin and ymax for this axis to enforce it to be the same on the
    # second chart:
    ymin, ymax = ax1.get_ylim()

    # Add our second axes for the barplot in position 2:
    ax2 = fig.add_subplot(122)
    ax2.set_yscale('log')
    ax2.set_ylim(1E-2,1E5)

    # Position the bars according to their index from the feature names variable:

    try:
        ax2.bar(range(len(coefs[-1])), coefs[-1], align='center', color=colors)
    except ValueError:
        pass
#     ax2.set_xticks(range(1, len(feature_names)+1))

    # enforce limits and add titles, labels
    ax2.set_ylim([ymin, ymax])
    ax2.set_title(regtype+' predictor coefs\n', fontsize=20)
    ax2.set_xlabel('coefficients', fontsize=20)
    ax2.set_ylabel('alpha', fontsize=20)

    plt.show()
```

Load the ipython widgets so we can make this plotting function interactive!

```
In [12]: from ipywidgets import *
         from IPython.display import display
```

The function and `interact` from ipywidgets lets me take some specified alphas that we have already calculated the coefficients for and plot them out.

```
In [14]: def ridge_plot_runner(log_of_alpha=-3.0):
             coef_plotter(r_alphas, r_coefs, 10**log_of_alpha, regtype='ridge')

         interact(ridge_plot_runner, log_of_alpha=(-3.0,12.0,0.33))
```

```
interactive(children=(FloatSlider(value=-3.0, description='log_of_alpha', max=12.0, min=-3.0, st
```

```
Out[14]: <function __main__.ridge_plot_runner(log_of_alpha=-3.0)>
```

5.19 Visualizing the Lasso

Now we do the same thing as above but for the Lasso. You will be able to see how the coefficients change differently for both.

```
In [15]: from sklearn.linear_model import Lasso
```

```
In [16]: # This is the same as the ridge coefficient by alpha calculator
         def lasso_coefs(X, Y, alphas):
             coefs = []
             lasso_reg = Lasso()
             for a in alphas:
                 lasso_reg.set_params(alpha=a)
```

```
            lasso_reg.fit(X, Y)
            coefs.append(lasso_reg.coef_)

        return coefs
```

Alphas for the Lasso tend to effect regularization linearly rather than by orders of magnitude like in the ridge.

A linear series of alphas is sufficient.

```
In [17]: l_alphas = np.logspace(-3,12,45)
         l_coefs = lasso_coefs(dataset_2, target_2, l_alphas)
```

Run the same plotting function above, but now with the calculated coefficients of alpha for the Lasso.

```
In [18]: def lasso_plot_runner(log_of_alpha=-3):
             coef_plotter(l_alphas, l_coefs, 10**log_of_alpha, regtype='lasso')

         interact(lasso_plot_runner, log_of_alpha=(-3.0,10.0,0.33))
```

```
interactive(children=(FloatSlider(value=-3.0, description='log_of_alpha', max=10.0, min=-3.0, st
```

```
Out[18]: <function __main__.lasso_plot_runner(log_of_alpha=-3)>
```

```
In [20]: from sklearn.model_selection import GridSearchCV
```

```
In [30]: np.logspace(-3,5,9)
```

```
Out[30]: array([  1.00000000e-03,   1.00000000e-02,   1.00000000e-01,
                  1.00000000e+00,   1.00000000e+01,   1.00000000e+02,
                  1.00000000e+03,   1.00000000e+04,   1.00000000e+05])
```

```
In [ ]: lasso_gs = GridSearchCV(Lasso(),
                               param_grid={'alpha' : np.logspace(-3,5,9)},
                               n_jobs=-1, cv=5)
        lasso_gs.fit(dataset_1, target_1)
```

```
In [23]: lasso_gs
```

```
Out[23]: GridSearchCV(cv=5, error_score='raise',
             estimator=Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
            normalize=False, positive=False, precompute=False, random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False),
             fit_params=None, iid=True, n_jobs=-1,
             param_grid={'alpha': array([  1.00000e-03,   1.00000e-02,   1.00000e-01,   1.000
               1.00000e+01,   1.00000e+02,   1.00000e+03,   1.00000e+04,
               1.00000e+05])},
             pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
             scoring=None, verbose=0)
```

```
In [26]: lasso_results = pd.DataFrame(lasso_gs.cv_results_)
         lasso_results.T
```

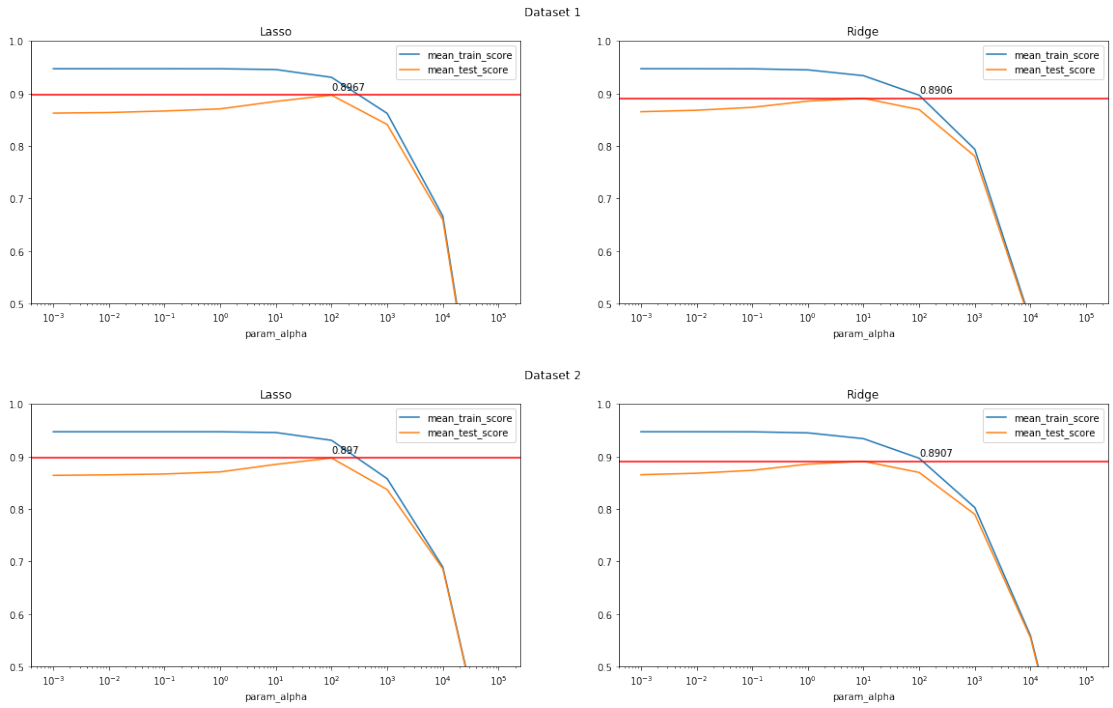
```
Out[26]: 0                1                2  \
         mean_fit_time              0.835488         0.678138        0.655113
         mean_score_time           0.00685644       0.00595794      0.00562291
         mean_test_score            0.862475         0.863586        0.866499
         mean_train_score           0.946966         0.946966        0.946967
         param_alpha                   0.001             0.01             0.1
         params              {'alpha': 0.001}  {'alpha': 0.01}  {'alpha': 0.1}
         rank_test_score                   6                5               4
         split0_test_score          0.882291         0.882406         0.88259
         split0_train_score         0.946441         0.946441        0.946442
         split1_test_score          0.889505         0.889631        0.890789
```

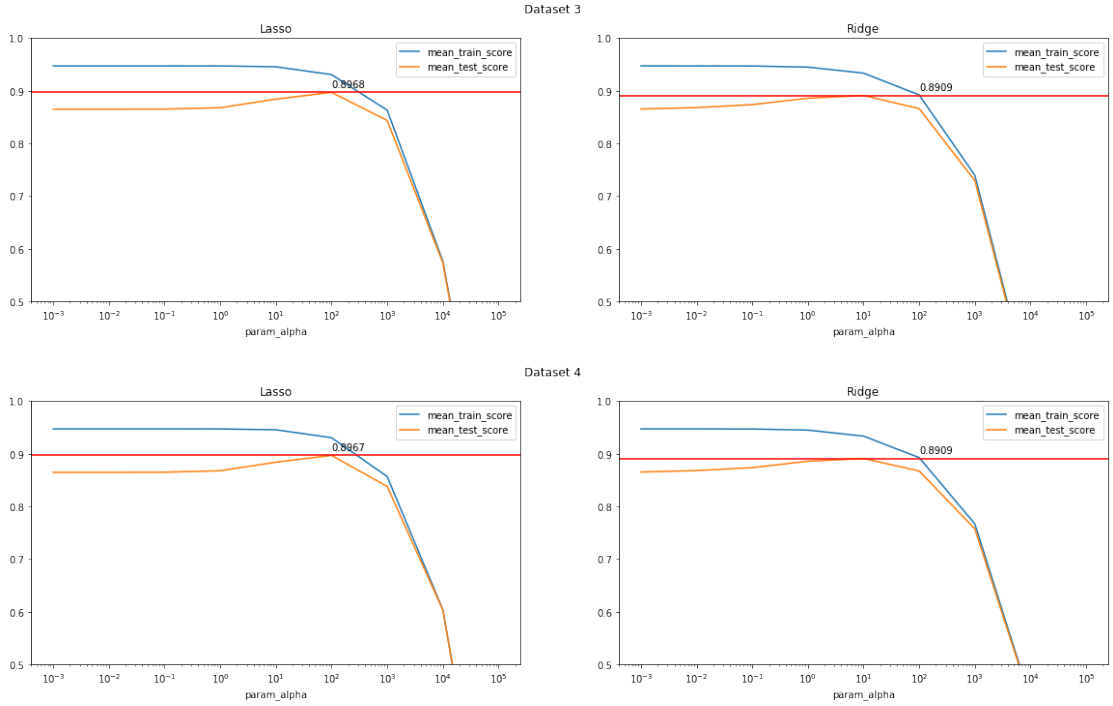
```
split1_train_score          0.944654         0.944654         0.944656
split2_test_score           0.849234         0.849455          0.85034
split2_train_score          0.949351         0.949352         0.949353
split3_test_score           0.870837         0.870907         0.871377
split3_train_score          0.947091         0.947091         0.947091
split4_test_score           0.820361         0.825399         0.837295
split4_train_score          0.947292         0.947292         0.947291
std_fit_time                0.214621        0.0599562        0.044757
std_score_time           0.00126347      0.000905132     0.000161063
std_test_score            0.0250526        0.0234132        0.0199257
std_train_score          0.00151218       0.00151219       0.00151186

                                  3                4                5  \
mean_fit_time              0.696827         0.430825         0.127905
mean_score_time         0.00565305       0.00556383       0.00561786
mean_test_score            0.870554          0.88473             0.8967
mean_train_score           0.946922          0.94541         0.930617
param_alpha                       1               10              100
params                {'alpha': 1.0}  {'alpha': 10.0}  {'alpha': 100.0}
rank_test_score                   3                2                1
split0_test_score          0.88421         0.893752         0.899191
split0_train_score         0.946397          0.94502         0.928997
split1_test_score          0.895714         0.904562         0.916498
split1_train_score         0.944617         0.942929         0.927889
split2_test_score          0.854601         0.875742         0.884952
split2_train_score         0.949306         0.947497         0.932118
split3_test_score          0.874911         0.887744          0.90442
split3_train_score         0.947041         0.945747         0.930658
split4_test_score          0.843241         0.861772         0.878375
split4_train_score         0.947247         0.945858         0.933424
std_fit_time              0.0427609         0.138252        0.0251548
std_score_time         0.000176667      0.000145065      0.000189952
std_test_score           0.0191715        0.0147687        0.0136461
std_train_score          0.00150932       0.00148156       0.00200985

                                  6                7                    8
mean_fit_time             0.0577508        0.0233486            0.0153247
mean_score_time           0.00617566       0.00902209           0.00793147
mean_test_score             0.840823         0.659513          -0.00525042
mean_train_score            0.861942         0.666364                    0
param_alpha                     1000            10000               100000
params                {'alpha': 1000.0}  {'alpha': 10000.0}  {'alpha': 100000.0}
rank_test_score                    7                8                    9
split0_test_score           0.865667         0.680028          -0.00012582
split0_train_score          0.864476         0.665782                    0
split1_test_score           0.878554         0.714215          -0.00080407
split1_train_score           0.85781          0.64961                    0
split2_test_score           0.808059         0.628686          -0.00565345
split2_train_score          0.860363         0.663583                    0
split3_test_score           0.844579         0.646479            -0.019632
split3_train_score          0.862963         0.672896                    0
split4_test_score           0.807139         0.628049         -1.87047e-05
split4_train_score          0.864101         0.679951                    0
std_fit_time              0.00584713       0.00266851           0.00259595
std_score_time            0.00318391       0.00115391           0.00355345
std_test_score             0.0291959        0.0332376            0.0074899
std_train_score           0.00251757        0.0101593                    0
```

In [27]:
```
def plot_for_dataset(dataset, target, dataset_name, alphas):
```

```
    lasso_gs = GridSearchCV(Lasso(), param_grid={'alpha': alphas}, n_jobs=-1, cv=5)
    lasso_gs.fit(dataset, target)
    lasso_results = pd.DataFrame(lasso_gs.cv_results_)
    lasso_results.set_index('param_alpha', inplace=True)

    ridge_gs = GridSearchCV(Ridge(), param_grid={'alpha' : alphas}, n_jobs=-1, cv=5)
    ridge_gs.fit(dataset, target)
    ridge_results = pd.DataFrame(ridge_gs.cv_results_)
    ridge_results.set_index('param_alpha', inplace=True)

    fig, ax = plt.subplots(1,2, figsize=(20,5))
    fig.suptitle(dataset_name)

    lasso_results[['mean_train_score', 'mean_test_score']].plot(logx=True, ylim=(0.5,1.
    ridge_results[['mean_train_score', 'mean_test_score']].plot(logx=True, ylim=(0.5,1.

    lasso_max_test_score = round(lasso_results.mean_test_score.max(), 4)
    ridge_max_test_score = round(ridge_results.mean_test_score.max(), 4)

    ax[0].axhline(lasso_max_test_score, color='red', label='max')
    mid_alphas = int(len(alphas)/2+1)
    ax[0].text(alphas[mid_alphas], lasso_max_test_score + .01, str(lasso_max_test_score
    ax[0].set_title('Lasso')
    ax[1].axhline(ridge_max_test_score, color='red', label='max')
    ax[1].text(alphas[mid_alphas], ridge_max_test_score + .01, str(ridge_max_test_score
    ax[1].set_title('Ridge');
```

In [28]:
```
plot_for_dataset(dataset_1, target_1, 'Dataset 1', np.logspace(-3,5,9))
plot_for_dataset(dataset_2, target_2, 'Dataset 2', np.logspace(-3,5,9))
plot_for_dataset(dataset_3, target_3, 'Dataset 3', np.logspace(-3,5,9))
plot_for_dataset(dataset_4, target_4, 'Dataset 4', np.logspace(-3,5,9))
```

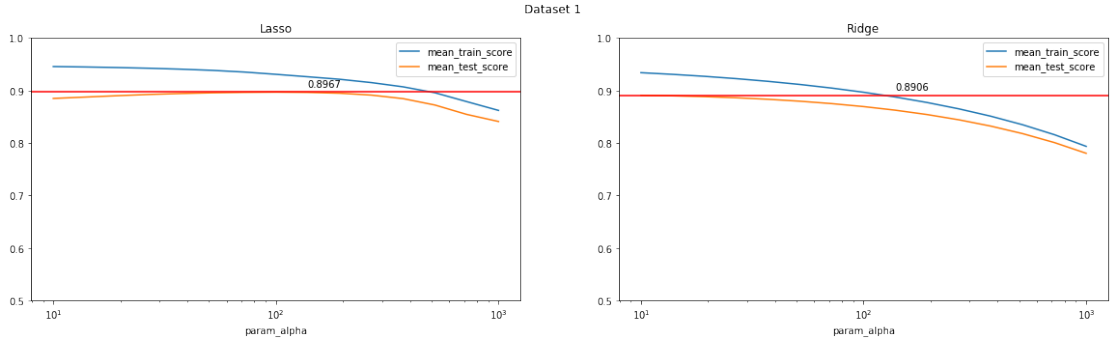


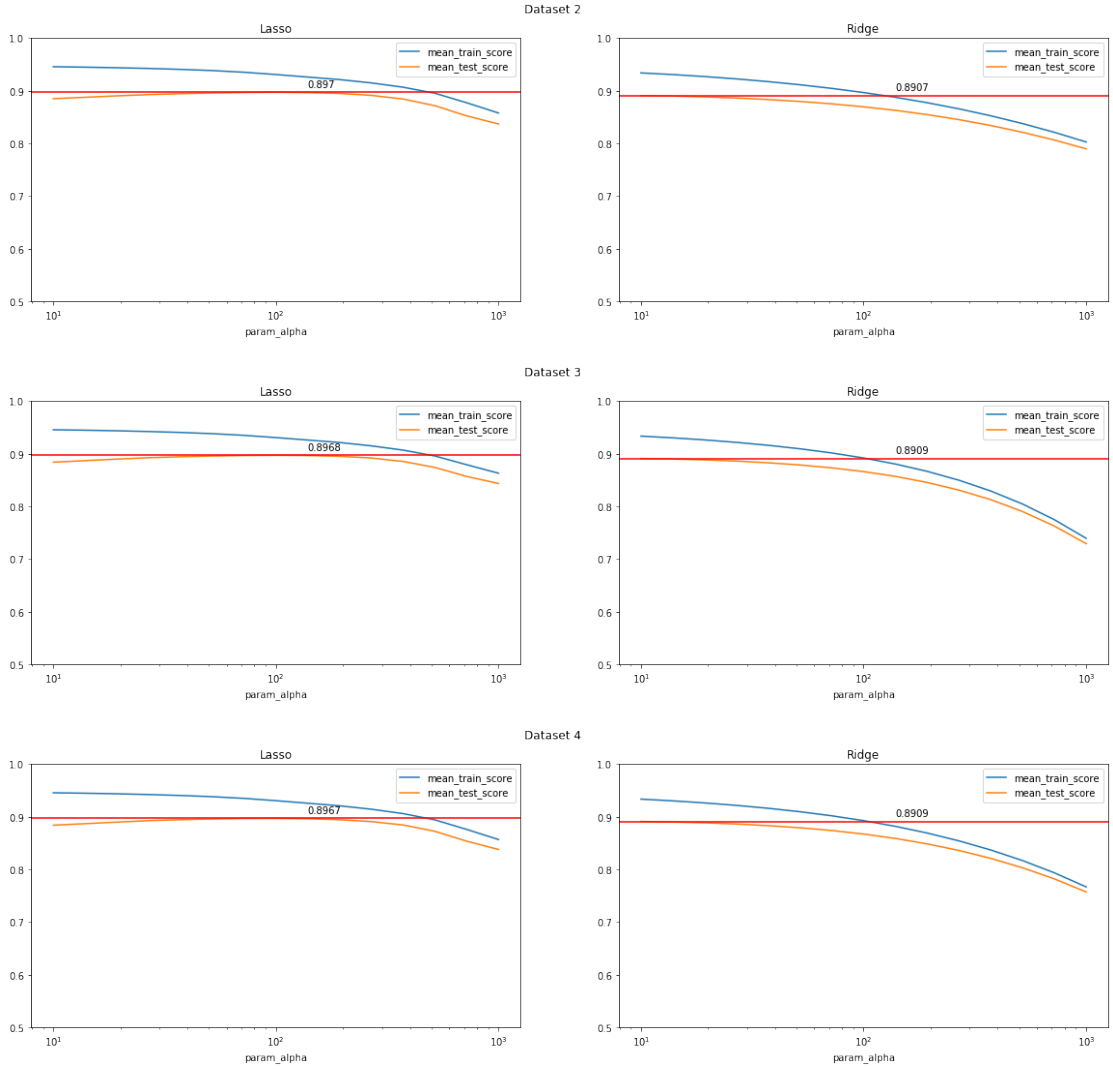



```
In [32]: np.logspace(1,3,15)
```

```
Out[32]: array([   10.        ,    13.89495494,    19.30697729,    26.82695795,
                   37.2759372 ,    51.79474679,    71.9685673 ,   100.        ,
                  138.94954944,   193.06977289,   268.26957953,   372.75937203,
                  517.94746792,   719.685673  ,  1000.        ])
```

```
In [31]: plot_for_dataset(dataset_1, target_1, 'Dataset 1', np.logspace(1,3,15))
         plot_for_dataset(dataset_2, target_2, 'Dataset 2', np.logspace(1,3,15))
         plot_for_dataset(dataset_3, target_3, 'Dataset 3', np.logspace(1,3,15))
         plot_for_dataset(dataset_4, target_4, 'Dataset 4', np.logspace(1,3,15))
```






```
In [33]: lasso_gs = GridSearchCV(Lasso(), param_grid={'alpha': np.logspace(1,3,5)}, n_jobs=-1, c
         lasso_gs.fit(dataset_2, target_2)
         lasso_results = pd.DataFrame(lasso_gs.cv_results_)
         lasso_results.set_index('param_alpha', inplace=True)

In [34]: lasso_gs.best_estimator_

Out[34]: Lasso(alpha=100.0, copy_X=True, fit_intercept=True, max_iter=1000,
            normalize=False, positive=False, precompute=False, random_state=None,
            selection='cyclic', tol=0.0001, warm_start=False)

In [35]: coefficients = pd.DataFrame(lasso_gs.best_estimator_.coef_, index=dataset_2.columns, co

In [36]: coefficients

Out[36]:                  value
         MSSubClass_20  1428.601387
         MSSubClass_30     0.000000
         MSSubClass_40     0.000000
         MSSubClass_45     0.000000
         MSSubClass_50    -0.000000
         MSSubClass_60     0.000000
         MSSubClass_70     0.000000
         MSSubClass_75    -0.000000
```

```
MSSubClass_80        -0.000000
MSSubClass_85        -0.000000
MSSubClass_90     -3240.529067
MSSubClass_120       -0.000000
MSSubClass_160        0.000000
MSSubClass_180        0.000000
MSSubClass_190       -0.000000
MSZoning_C (all) -15049.515384
MSZoning_FV        4567.163923
MSZoning_RH           0.000000
MSZoning_RL           0.000000
MSZoning_RM          -0.000000
LotShape_IR1       -733.370128
LotShape_IR2          0.000000
LotShape_IR3          0.000000
LotShape_Reg         -0.000000
LandContour_Bnk      -0.000000
LandContour_HLS       0.000000
LandContour_Low      -0.000000
LandContour_Lvl     139.659381
Utilities_AllPub      0.000000
Utilities_NoSeWa     -0.000000
...                        ...
LotArea            8086.116752
YearBuilt          8197.081075
YearRemodAdd       1258.727799
MasVnrArea           -0.000000
BsmtFinSF1         4766.360486
BsmtFinSF2         -928.695216
BsmtUnfSF         -1327.540408
TotalBsmtSF         109.820207
FirstFlrSF         7456.127529
SecondFlrSF         612.141295
LowQualFinSF       -223.999166
GrLivArea         18568.889130
GarageYrBlt          -0.000000
GarageArea            0.000000
WoodDeckSF           -0.000000
OpenPorchSF        -422.856314
EnclosedPorch      1294.233488
ThreeSsnPorch      1299.946179
ScreenPorch           0.000000
PoolArea           3522.566011
MiscVal             -68.395262
YrSold                0.000000
PC 1              -5062.231110
PC 2                  0.000000
PC 3                  0.000000
PC 4                  0.000000
PC 5              -3142.120869
PC 6                 -0.000000
PC 7               -252.208873
PC 8                  0.000000

[390 rows x 1 columns]
```

In [37]:
```
coefficients = coefficients[coefficients.value != 0 ]
coefficients.shape
```

Out[37]:
```
(133, 1)
```

```
In [38]: pd.options.display.max_rows = 150

In [40]: lasso_gs.best_estimator_.intercept_

Out[40]: 134949.14755685933

In [ ]: coefficients['abs'] = np.abs(coefficients.value)
        coefficients.sort_values('abs', ascending=False).head(133)

In [48]: coefficients[coefficients.index.str.contains('Cond')]

Out[48]: value           abs
         Condition1_Norm         6907.494751   6907.494751
         Condition1_RRAe        -2352.912815   2352.912815
         OverallCond_3         -10216.250717  10216.250717
         OverallCond_4         -10878.754401  10878.754401
         OverallCond_5          -5772.974900   5772.974900
         OverallCond_7           6403.586068   6403.586068
         OverallCond_8           5629.261088   5629.261088
         OverallCond_9           6087.156693   6087.156693
         ExterCond_TA             435.129671    435.129671
         BsmtCond_TA             3790.556769   3790.556769
         GarageCond_Fa          -1609.104132   1609.104132
         GarageCond_TA            993.784717    993.784717
         SaleCondition_Abnorml  -5207.575942   5207.575942
         SaleCondition_Family     -40.041800     40.041800
         SaleCondition_Normal    1476.356769   1476.356769
```

5.20 Entropy

Intuitively, we can think of **entropy** as the measure of disorder in a system.

This set, S, for example, is very disordered:

Let us consider the elements of S that have a black border.

```
In [1]: %run items.py

In [2]: items

Out[2]: [{'form': 'square', 'letter': 'E', 'color': 'blue', 'border': False},
         {'form': 'circle', 'letter': 'A', 'color': 'red', 'border': True},
         {'form': 'circle', 'letter': 'B', 'color': 'green', 'border': False},
         {'form': 'diamond', 'letter': 'E', 'color': 'red', 'border': True},
         {'form': 'diamond', 'letter': 'E', 'color': 'red', 'border': True},
         {'form': 'diamond', 'letter': 'C', 'color': 'blue', 'border': False},
         {'form': 'square', 'letter': 'A', 'color': 'green', 'border': False},
         {'form': 'diamond', 'letter': 'D', 'color': 'green', 'border': True},
         {'form': 'star', 'letter': 'C', 'color': 'silver', 'border': True},
         {'form': 'square', 'letter': 'C', 'color': 'silver', 'border': False},
         {'form': 'circle', 'letter': 'B', 'color': 'green', 'border': False},
         {'form': 'circle', 'letter': 'A', 'color': 'red', 'border': True},
         {'form': 'square', 'letter': 'A', 'color': 'green', 'border': False},
         {'form': 'circle', 'letter': 'D', 'color': 'green', 'border': True},
         {'form': 'star', 'letter': 'B', 'color': 'blue', 'border': True},
         {'form': 'circle', 'letter': 'B', 'color': 'blue', 'border': True},
         {'form': 'square', 'letter': 'E', 'color': 'blue', 'border': False},
         {'form': 'diamond', 'letter': 'E', 'color': 'red', 'border': True},
         {'form': 'star', 'letter': 'B', 'color': 'blue', 'border': True},
         {'form': 'square', 'letter': 'C', 'color': 'purple', 'border': False},
         {'form': 'square', 'letter': 'C', 'color': 'silver', 'border': False}]
```

```
In [3]: import numpy as np
        import pandas as pd

        from IPython.display import display

        items_df = pd.DataFrame(items)
        items_df
```

Out[3]:

	border	color	form	letter
0	False	blue	square	E
1	True	red	circle	A
2	False	green	circle	B
3	True	red	diamond	E
4	True	red	diamond	E
5	False	blue	diamond	C
6	False	green	square	A
7	True	green	diamond	D
8	True	silver	star	C
9	False	silver	square	C
10	False	green	circle	B
11	True	red	circle	A
12	False	green	square	A
13	True	green	circle	D
14	True	blue	star	B
15	True	blue	circle	B
16	False	blue	square	E
17	True	red	diamond	E
18	True	blue	star	B
19	False	purple	square	C
20	False	silver	square	C

5.20.1 Information Entropy

We can measure the disorder in S relative to any attribute of an element using the Shannon Information **Entropy**

$$H(S) = -p_1 \log_2 p_1 + \cdots + -p_n \log_2 p_n$$

Here, each p_i is a measure of the proportion of the set represented by each class for a given attribute. If we are looking at the color of our shapes, this would be five classes:

```
In [11]: set(items_df.color)
```

```
Out[11]: {'blue', 'green', 'purple', 'red', 'silver'}
```

```
In [4]: items_df.color.unique()
```

```
Out[4]: array(['blue', 'red', 'green', 'silver', 'purple'], dtype=object)
```

We can use Python to calculate the entropy

```
In [5]: def class_entropy(proportion):
            return -proportion*np.log2(proportion)
```

```
In [6]: def entropy(proportions):
            class_entropies = [class_entropy(proportion)
                               for proportion in proportions]
            return sum(class_entropies)
```

5.21 Measure The Entropy of the color of S

```
In [7]: n = items_df.color.count()
        n
```

```
Out[7]: 21
```

```
In [8]: p_red = (items_df.color == 'red').sum()/n
        p_blue = (items_df.color == 'blue').sum()/n
        p_green = (items_df.color == 'green').sum()/n
        p_silver = (items_df.color == 'silver').sum()/n
        p_purple = (items_df.color == 'purple').sum()/n

        S_proportions = [p_red, p_blue, p_green, p_silver, p_purple]
        S_proportions
```

```
Out[8]: [0.23809523809523808,
         0.2857142857142857,
         0.2857142857142857,
         0.14285714285714285,
         0.047619047619047616]
```

```
In [9]: sum(S_proportions)
```

```
Out[9]: 1.0
```

```
In [10]: entropy(S_proportions)
```

```
Out[10]: 2.1359327568142765
```

5.21.1 To Do

Write a method that calculates the class proportions for a given dataframe and feature.

```
In [12]: 3*True
```

```
Out[12]: 3
```

```
In [14]: np.array([True, False]) + np.array([False, True])
```

```
Out[14]: array([ True,  True], dtype=bool)
```

```
In [18]: def class_proportions(series):
             classes = series.unique()
             total_count = series.count()
             counts = np.array([(series == cls).sum() for cls in classes])
             return counts/total_count
```

```
In [19]: class_proportions(items_df.color)
```

```
Out[19]: array([ 0.28571429,  0.23809524,  0.28571429,  0.14285714,  0.04761905])
```

```
In [20]: class_proportions(items_df.border)
```

```
Out[20]: array([ 0.47619048,  0.52380952])
```

```
In [21]: def entropy(series):
             proportions = class_proportions(series)
             class_entropies = [class_entropy(proportion)
                                for proportion in proportions]
             return sum(class_entropies)
```

Use this method to calculate the Entropy with respect to each attribute of S

```
In [22]: display("color:  {}".format(class_proportions(items_df.color)))
         display("form:   {}".format(class_proportions(items_df.form)))
         display("letter: {}".format(class_proportions(items_df.letter)))
         display("border: {}".format(class_proportions(items_df.border)))
```

```
'color:  [ 0.28571429  0.23809524  0.28571429  0.14285714  0.04761905]'
```

```
'form:   [ 0.33333333  0.28571429  0.23809524  0.14285714]'
```

```
'letter: [ 0.23809524  0.19047619  0.23809524  0.23809524  0.0952381 ]'
```

```
'border: [ 0.47619048  0.52380952]'
```

```
In [23]: display("color:  {}".format(entropy(items_df.color)))
         display("form:   {}".format(entropy(items_df.form)))
         display("letter: {}".format(entropy(items_df.letter)))
         display("border: {}".format(entropy(items_df.border)))
```

```
'color:  2.1359327568142765'
```

```
'form:   1.938708497286834'
```

```
'letter: 2.2576068788115964'
```

```
'border: 0.998363672593813'
```

5.21.2 The Decision Tree

Decision Trees are supervised learning models typically split into classification trees and regression trees. For the rest of this lesson, we will focus on classification trees.

We will work through the ID3 algorithm for learning a decision tree from a set of labeled data.

Labeled Data

For our purposes, let us assume that the feature `border` is our label. We will be seeking a decision tree that makes splits in order to develop a model for identifying which items will have a border.

```
In [26]: features = items_df.drop('border', axis=1)
         target = items_df.border
```

```
In [27]: class_proportions(target)
```

```
Out[27]: array([ 0.47619048,  0.52380952])
```

```
In [28]: entropy(target)
```

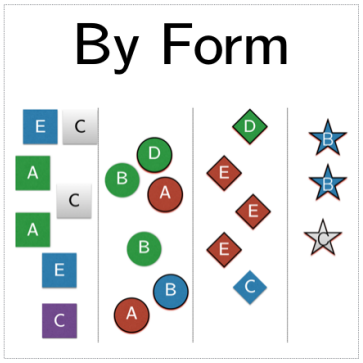
```
Out[28]: 0.99836367259381298
```

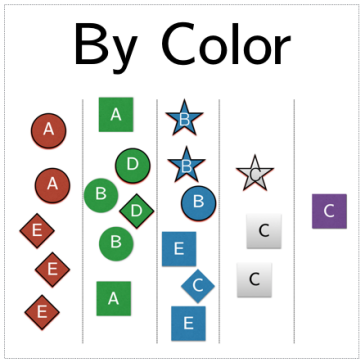
5.22 Three Partitioning Schemes

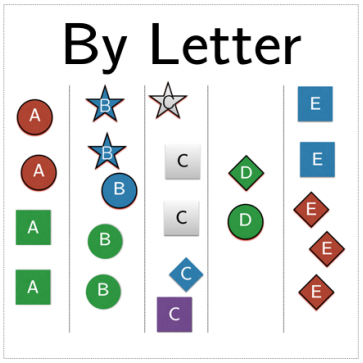
We can start by separating the elements based upon their attributes. Here are three different ways to do that:

5.23 Identify the Best Split

In order to proceed, we will need to identify which of these ways of separating is best? We can use the measure of entropy to do this!








```
In [29]: features_square_df = features[features.form == 'square']
         features_circle_df = features[features.form == 'circle']
         features_diamond_df = features[features.form == 'diamond']
         features_star_df = features[features.form == 'star']

         target_square_df = target[features.form == 'square']
         target_circle_df = target[features.form == 'circle']
         target_diamond_df = target[features.form == 'diamond']
         target_star_df = target[features.form == 'star']

         display(pd.merge(features_square_df, pd.DataFrame(target_square_df), left_index=True, r
         display(pd.merge(features_circle_df, pd.DataFrame(target_circle_df), left_index=True, r
         display(pd.merge(features_diamond_df, pd.DataFrame(target_diamond_df), left_index=True,
         display(pd.merge(features_star_df, pd.DataFrame(target_star_df), left_index=True, right
```

```
color    form letter  border
0     blue  square      E   False
6    green  square      A   False
9   silver  square      C   False
12   green  square      A   False
16    blue  square      E   False
19  purple  square      C   False
20  silver  square      C   False
```

```
color    form letter  border
1     red  circle      A    True
2   green  circle      B   False
10  green  circle      B   False
11    red  circle      A    True
13  green  circle      D    True
15   blue  circle      B    True
```

```
color     form letter  border
3     red  diamond      E    True
4     red  diamond      E    True
5    blue  diamond      C   False
7   green  diamond      D    True
17    red  diamond      E    True
```

```
color  form letter  border
8   silver  star      C    True
14    blue  star      B    True
18    blue  star      B    True
```

To assess this split, we will be seeking the entropy associated with each subset as a proportion of the total

set.

$$H_T = q_{\text{square}} H(S_{\text{square}}) + q_{\text{circle}} H(S_{\text{circle}}) + q_{\text{diamond}} H(S_{\text{diamond}}) + q_{\text{star}} H(S_{\text{star}})$$

Here each q_i represents the **weight**, the proportion of the total each subset is

$$q_i = \frac{\text{count}(S_i)}{\text{count}(S_T)}$$

```
In [30]: q_square = features_square_df.form.count()/features.form.count()
         q_circle = features_circle_df.form.count()/features.form.count()
         q_diamond = features_diamond_df.form.count()/features.form.count()
         q_star = features_star_df.form.count()/features.form.count()
```

5.24 Total Entropy When Split by Form

```
In [32]: display("squares:  {}".format(class_proportions(target_square_df)))
         display("circles:   {}".format(class_proportions(target_circle_df)))
         display("diamond: {}".format(class_proportions(target_diamond_df)))
         display("star: {}".format(class_proportions(target_star_df)))
```

```
'squares:  [ 1.]'
```

```
'circles:   [ 0.66666667  0.33333333]'
```

```
'diamond: [ 0.8  0.2]'
```

```
'star: [ 1.]'
```

```
In [33]: display("square:  {}".format(entropy(target_square_df)))
         display("circle:   {}".format(entropy(target_circle_df)))
         display("diamond: {}".format(entropy(target_diamond_df)))
         display("star: {}".format(entropy(target_star_df)))
```

```
'square:  0.0'
```

```
'circle:   0.9182958340544896'
```

```
'diamond: 0.7219280948873623'
```

```
'star: 0.0'
```

```
In [34]: H_square_border = entropy(target_square_df)
         H_circle_border = entropy(target_circle_df)
         H_diamond_border = entropy(target_diamond_df)
         H_star_border = entropy(target_star_df)
```

```
In [35]: (q_square*H_square_border +
          q_circle*H_circle_border +
          q_diamond*H_diamond_border +
          q_star*H_star_border)
```

```
Out[35]: 0.43425787994113085
```

5.25 Write a function to do this for a split on any feature

```
In [36]: def entropy_on_split(dataframe, target, feature, debug=False):

             # split on feature
             unique_classes = dataframe[feature].unique()
```

```
            target_subsets = [
                target[dataframe[feature] == unique_class]
                for unique_class in unique_classes
            ]

            # calculate subset weights
            total = target.count()
            weights = [
                target_subset.count()/total
                for target_subset in target_subsets
            ]

            # calculate entropies
            entropies = [
                entropy(target_subset)
                for target_subset in target_subsets
            ]

            # return weighted entropy
            return sum(weight*entropy for weight, entropy in zip(weights, entropies))
```

```
In [37]: entropy_on_split(features, target, 'form')
```

```
Out[37]: 0.43425787994113085
```

```
In [38]: entropy_on_split(features, target, 'color')
```

```
Out[38]: 0.67926964316620975
```

```
In [39]: entropy_on_split(features, target, 'letter')
```

```
Out[39]: 0.82472125804683316
```

5.26 Write a function to Identify the Best Split

```
In [40]: def find_best_split(features, target):
             feature_labels = features.columns
             entropies = [
                 entropy_on_split(features, target, feature_label)
                 for feature_label in feature_labels
             ]

             best_index = np.argmin(entropies)
             return feature_labels[best_index]
```

```
In [41]: find_best_split(features, target)
```

```
Out[41]: 'form'
```

5.26.1 Representing the Tree

The ability to find a best split will only work at a single node. In order to build a complete machine learning model, we are going to need to use this method to build an entire decision tree.

We have this so far:

But what about the rest of the tree? We want to split like this:

In order to represent this using Python, we will define a `tree` to be one of these:

- `True`
- `False`
- a tuple `(attribute, subtree)`

For example, consider the tree representing green circles containing the letter 'B'.

We can represent this trivially as:

```
In [42]: green_circle_B_tree = False
```

or the tree representing green circles containing the letter 'D':

```
In [43]: green_circle_D_tree = True
```

From there, consider the tree representing green circles:

```
In [44]: green_circle_tree = ('letter', {'B' : False, 'D' : True})
```

We might gradually work our way up from there and represent our entire tree as

```
In [45]: tree = ('form', {'square': False,
                          'circle':
                                    ('color', {'green':
                                                       ('letter', {'B' : False,
                                                                   'D' : True}),
                                               'red': True,
                                               'blue': True}),
                          'diamond':
                                     ('letter', {'C': False,
                                                 'D': True,
                                                 'E': True}),
                          'star': True})
```

5.27 Use a tree to classifiy an input

Suppose we had a new element and we wish to no whether or not is has a border. For example, we may be given the following element:

```
{'form': 'circle', 'letter': 'C', 'color': 'red'}
```

We are going to need to build a classification function to use our tree to classify this input.

```
In [46]: def classify(tree, element):
             if tree in [True, False]:
                 return tree

             attribute, subtree_dictionary = tree

             subtree_key = element.get(attribute)

             if subtree_key not in subtree_dictionary:
                 subtree_key = None

             subtree = subtree_dictionary[subtree_key]

             return classify(subtree, element)
```

```
In [47]: classify(tree, {'form': 'circle', 'letter': 'C', 'color': 'red'})
```

```
Out[47]: True
```

But what if we pass an element that is ambiguous, for example

```
{'form': 'circle', 'letter': 'C', 'color': 'green'}
```

```
In [48]: classify(tree, {'form': 'circle', 'letter': 'C', 'color': 'green'})
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-48-fb66fb52e16e> in <module>()
----> 1 classify(tree, {'form': 'circle', 'letter': 'C', 'color': 'green'})

<ipython-input-46-60febbeeaf89> in classify(tree, element)
     12     subtree = subtree_dictionary[subtree_key]
     13
---> 14     return classify(subtree, element)

<ipython-input-46-60febbeeaf89> in classify(tree, element)
     12     subtree = subtree_dictionary[subtree_key]
     13
---> 14     return classify(subtree, element)

<ipython-input-46-60febbeeaf89> in classify(tree, element)
     10         subtree_key = None
     11
---> 12     subtree = subtree_dictionary[subtree_key]
     13
     14     return classify(subtree, element)

KeyError: None
```

5.28 To Handle this, we will redefine our Tree

We will add a `None` key that returns the most common class.

```
In [49]: tree = ('form', {'square': False,
                          'circle':
                                    ('color', {'green':
                                                       ('letter', {'B' : False,
                                                                   'D' : True,
                                                                   None: False}),
                                               'red': True,
                                               'blue': True,
                                               None : True}),
                          'diamond':
                                     ('letter', {'C': False,
                                                 'D': True,
                                                 'E': True,
                                                 None: True}),
                          'star': True,
                          None: True})
```

```
In [50]: def classify(tree, element):
             if tree in [True, False]:
                 return tree

             attribute, subtree_dictionary = tree

             subtree_key = element.get(attribute)
```

```
            if subtree_key not in subtree_dictionary:
                subtree_key = None

            subtree = subtree_dictionary[subtree_key]

            return classify(subtree, element)
```

```
In [51]: classify(tree, {'form': 'circle', 'letter': 'C', 'color': 'green'})
```

```
Out[51]: False
```

```
In [52]: classify(tree, {'form': 'octagon', 'letter': 'Z', 'color': 'chartreuse'})
```

```
Out[52]: True
```

5.28.1 Build The Tree

```
In [56]: def find_best_split(features, target, split_candidates):

            entropies = [
                entropy_on_split(features, target, feature_label)
                for feature_label in split_candidates
            ]

            best_index = np.argmin(entropies)
            return split_candidates[best_index]
```

```
In [57]: def build_tree(features, target, split_candidates=None):
            if split_candidates is None:
                split_candidates = list(features.columns)

            total_count = target.count()
            true_count = target.sum()
            false_count = total_count - true_count

            if false_count == 0: return True
            if true_count == 0: return False

            if split_candidates == []:
                return true_count > false_count

            best_attribute = find_best_split(features, target, split_candidates)

            split_candidates = [split_candidate
                                for split_candidate in split_candidates
                                if split_candidate is not best_attribute]

            best_attribute_uniques = features[best_attribute].unique()
            subtree = dict()
            for best_attr_unique in best_attribute_uniques:
                feat_subset = features[features[best_attribute] == best_attr_unique]
                target_subset = target[features[best_attribute] == best_attr_unique]
                subtree[best_attr_unique] = build_tree(feat_subset, target_subset, split_candid
            subtree[None] = true_count > false_count

            return (best_attribute, subtree)
```

```
In [60]: build_tree(features, target)
```

```
Out[60]: ('form',
          {'square': False,
           'circle': ('color',
            {'red': True,
             'green': ('letter', {'B': False, 'D': True, None: False}),
             'blue': True,
             None: True}),
           'diamond': ('color',
            {'red': True, 'blue': False, 'green': True, None: True}),
           'star': True,
           None: True})
```

```
In [71]: features = items_df.drop('letter', axis=1)
         target = items_df.letter == 'A'
```

```
In [72]: my_tree = build_tree(features, target)
```

```
In [73]: my_tree
```

```
Out[73]: ('color',
          {'blue': False,
           'red': ('form', {'circle': True, 'diamond': False, None: False}),
           'green': ('form',
            {'circle': False, 'square': True, 'diamond': False, None: False}),
           'silver': False,
           'purple': False,
           None: False})
```

```
In [62]: display(classify(my_tree, {'form': 'circle', 'letter': 'C', 'color': 'red'}))
         display(classify(my_tree, {'form': 'circle', 'letter': 'C', 'color': 'green'}))
         display(classify(my_tree, {'form': 'octagon', 'letter': 'Z', 'color': 'chartreuse'}))
```

```
True

False

True
```

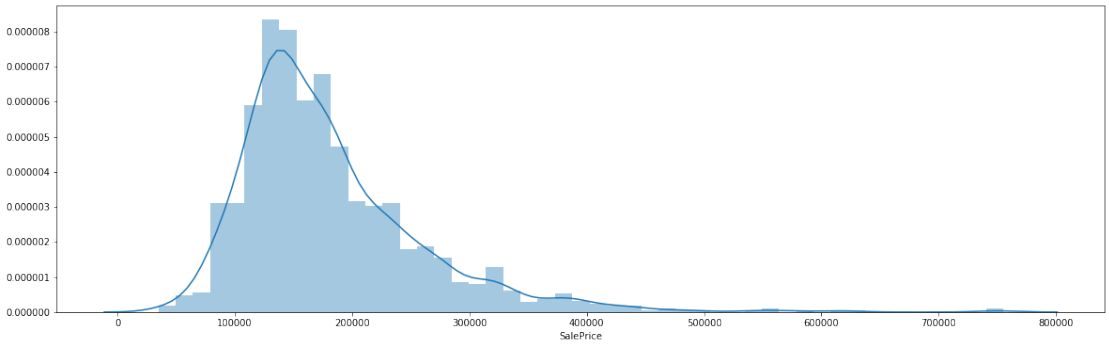
```
In [1]: run src/preprocessing.py
```

```
In [2]: !pip install tqdm --quiet
```

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

```
In [4]: fig = plt.figure(figsize=(20,6))
        sns.distplot(target_1)
```

```
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff674fd3c18>
```




```
In [5]: from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
```

5.29 Fundamental Question: How much does a home in Ames, Iowa sell for?

```
In [8]: mean_sale_price = target_1.mean()
        naive_guess = np.ones(len(target_1))*mean_sale_price

In [9]: np.ones(4)*mean_sale_price

Out[9]: array([ 180922.1066482,  180922.1066482,  180922.1066482,  180922.1066482])

In [10]: r2_score(target_1, naive_guess)

Out[10]: 0.0

In [14]: np.sqrt(mean_squared_error(target_1, naive_guess))

Out[14]: 79216.766949800032

In [12]: mean_absolute_error(target_1, naive_guess)

Out[12]: 57374.611279072444

In [1]: run src/preprocessing.py

In [2]: import matplotlib.pyplot as plt
        import numpy as np
        import seaborn as sns

In [3]: from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error

In [4]: mean_sale_price = target_1.mean()
        naive_guess = np.ones(len(target_1))*mean_sale_price

In [5]: naive_r2 = r2_score(target_1, naive_guess)
        naive_rmse = np.sqrt(mean_squared_error(target_1, naive_guess))
        naive_mae = mean_absolute_error(target_1, naive_guess)
```

5.30 Fundamental Question: How much does a home in Ames, Iowa sell for?

```
In [6]: from sklearn.linear_model import Ridge, Lasso
        from sklearn.model_selection import train_test_split
        from tqdm import tqdm
        from time import time

In [7]: def sample_training_set(X_train, y_train, n_pcnt):
            n = X_train.shape[0]*n_pcnt//100
            return n, X_train[:n], y_train[:n]

        def time_function_call(function_call):
            start = time()
            result = function_call
            execution_time = time() - start
            return result, execution_time

        def run_model(model, model_name, n_pcnt, data, labels):

            X_train, X_test, y_train, y_test = train_test_split(data, labels, random_state=42)


            n, X_samp, y_samp = sample_training_set(X_train, y_train, n_pcnt)
```

```
    _, fit_time = time_function_call(
        model.fit(X_samp, y_samp))

    train_pred, train_pred_time = time_function_call(
        model.predict(X_samp))

    test_pred, test_pred_time = time_function_call(
        model.predict(X_test))

    return {
            'model_name' : model_name,
            'n_pcnt' : n_pcnt,
            'n' : n,
            'rmse_train' : np.sqrt(mean_squared_error(y_samp, train_pred)),
            'rmse_test' : np.sqrt(mean_squared_error(y_test, test_pred)),
            'mae_train' : mean_absolute_error(y_samp, train_pred),
            'mae_test' : mean_absolute_error(y_test, test_pred),
            'r2_train_score' : model.score(X_samp, y_samp),
            'r2_test_score' : model.score(X_test, y_test),
            'fit_time' : fit_time,
            'train_pred_time' : train_pred_time,
            'test_pred_time' : test_pred_time}
```

In [8]:
```
dataset_2.shape, target_2.shape
```

Out[8]:
```
((1444, 390), (1444,))
```

In [9]:
```
test_results = {}
percentages = [1,2,3,4,5,7,10,15,20,25,30,40,50,60,70,80,90,100]
for n in tqdm(percentages):
    test_results[n] = run_model(Lasso(), 'Lasso', n,
                                dataset_2,
                                target_2)
```

```
  0%|          | 0/18 [00:00<?, ?it/s]/opt/conda/lib/python3.6/site-packages/sklearn/linear_mode
  ConvergenceWarning)
100%|lacksquarelacksquarelacksquarelacksquarelacksquarelacksquarelacksquarelacksquarelacksquarelacksquarelacksquarelacksquare| 18/18
```

In [10]:
```
test_results = pd.DataFrame(test_results).T.sort_values('n')
test_results
```

Out[10]:

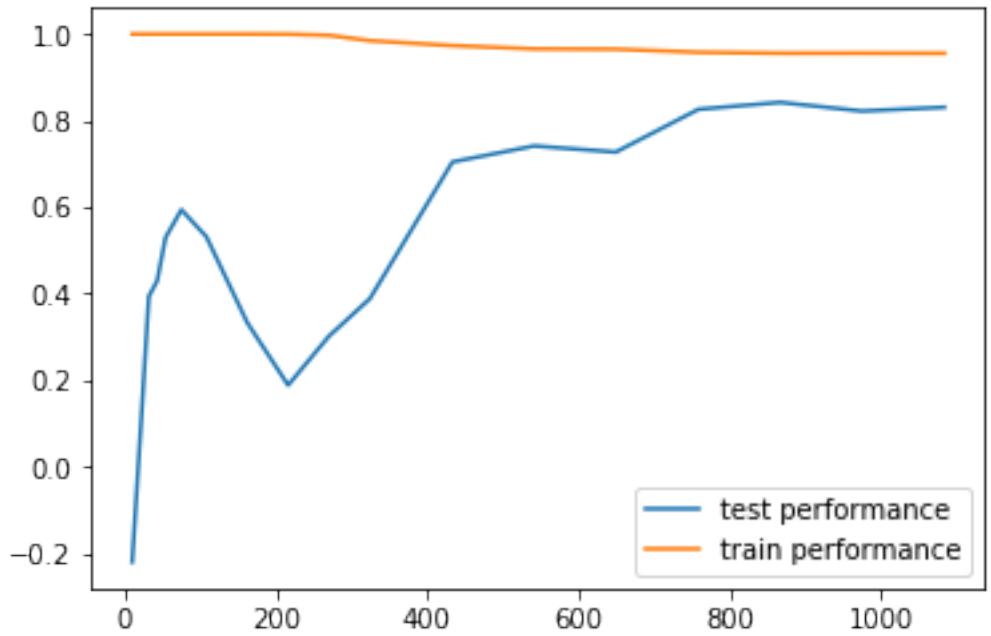
	fit_time	mae_test	mae_train	model_name	n	n_pcnt	r2_test_score	\
1	7.15256e-06	55978.7	4.37798	Lasso	10	1	-0.220736	
2	8.58307e-06	53561.5	7.27929	Lasso	21	2	0.081728	
3	6.19888e-06	41954.3	11.9816	Lasso	32	3	0.393408	
4	1.3113e-05	40827.2	16.798	Lasso	43	4	0.429471	
5	7.39098e-06	36467	22.8864	Lasso	54	5	0.529258	
7	1.3113e-05	34542.5	46.6068	Lasso	75	7	0.593882	
10	5.24521e-06	36820	67.3206	Lasso	108	10	0.531366	
15	9.29832e-06	44355.6	211.232	Lasso	162	15	0.332694	
20	6.91414e-06	50261.8	955.31	Lasso	216	20	0.188098	
25	5.00679e-06	44782.9	3011.91	Lasso	270	25	0.30153	
30	7.62939e-06	37481.6	6339.75	Lasso	324	30	0.388085	
40	6.4373e-06	28666.7	8675.47	Lasso	433	40	0.70425	
50	9.05991e-06	24724	9650.64	Lasso	541	50	0.741287	
60	1.85966e-05	23898.1	10033.7	Lasso	649	60	0.727152	
70	1.23978e-05	21270.9	11160.1	Lasso	758	70	0.825999	
80	1.07288e-05	19473.6	11288.1	Lasso	866	80	0.842039	
90	1.45435e-05	19977.1	11683.9	Lasso	974	90	0.82195	

```
100  8.34465e-06    19207   11781.7      Lasso  1083    100      0.830646
```

	r2_train_score	rmse_test	rmse_train	test_pred_time	train_pred_time
1	1	82394	4.80578	1.57356e-05	6.67572e-06
2	1	71461.2	8.94584	1.09673e-05	6.91414e-06
3	1	58080.9	14.7286	5.72205e-06	5.96046e-06
4	1	56327.9	21.172	5.48363e-06	5.48363e-06
5	1	51165.4	29.5307	1.16825e-05	7.15256e-06
7	0.999999	47523.8	57.9645	1.04904e-05	7.15256e-06
10	0.999999	51050.7	81.4511	6.19888e-06	5.96046e-06
15	0.999989	60918.2	301.13	9.53674e-06	8.34465e-06
20	0.999718	67194.9	1408	5.96046e-06	6.19888e-06
25	0.997078	62324.5	4381.52	5.48363e-06	5.72205e-06
30	0.984681	58335.1	9708.44	6.19888e-06	5.72205e-06
40	0.973235	40555.2	12585.8	5.48363e-06	5.24521e-06
50	0.965772	37931	14316.9	6.67572e-06	5.96046e-06
60	0.965025	38953.3	14659.2	1.28746e-05	9.77516e-06
70	0.957932	31107.2	16229.3	8.58307e-06	8.10623e-06
80	0.955697	29638.7	16336.3	1.7643e-05	1.35899e-05
90	0.955788	31467	16932.5	8.82149e-06	9.77516e-06
100	0.955431	30689	17011	5.96046e-06	6.67572e-06

In [11]:
```
plt.plot(test_results.n, test_results.r2_test_score, label='test performance')
plt.plot(test_results.n, test_results.r2_train_score, label='train performance')
plt.legend()
```

Out[11]: <matplotlib.legend.Legend at 0x7efedade5e80>



In [1]:
```
run src/preprocessing.py
```

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error
```

In [3]:
```
from sklearn.linear_model import Lasso, Ridge
```

```
        from sklearn.model_selection import train_test_split
        from tqdm import tqdm
        from time import time
```

```
In [4]: def sample_training_set(X_train, y_train, n_pcnt):
            n = X_train.shape[0]*n_pcnt//100
            return n, X_train[:n], y_train[:n]

        def time_function_call(function_call):
            start = time()
            result = function_call
            execution_time = time() - start
            return result, execution_time

        def run_model(model, model_name, n_pcnt, data, labels):

            X_train, X_test, y_train, y_test = train_test_split(data, labels, random_state=42)


            n, X_samp, y_samp = sample_training_set(X_train, y_train, n_pcnt)

            _, fit_time = time_function_call(
                model.fit(X_samp, y_samp))

            train_pred, train_pred_time = time_function_call(
                model.predict(X_samp))

            test_pred, test_pred_time = time_function_call(
                model.predict(X_test))

            return {
                    'model' : model,
                    'model_name' : model_name,
                    'n_pcnt' : n_pcnt,
                    'n' : n,
                    'rmse_train' : np.sqrt(mean_squared_error(y_samp, train_pred)),
                    'rmse_test' : np.sqrt(mean_squared_error(y_test, test_pred)),
                    'mae_train' : mean_absolute_error(y_samp, train_pred),
                    'mae_test' : mean_absolute_error(y_test, test_pred),
                    'r2_train_score' : model.score(X_samp, y_samp),
                    'r2_test_score' : model.score(X_test, y_test),
                    'fit_time' : fit_time,
                    'train_pred_time' : train_pred_time,
                    'test_pred_time' : test_pred_time}
```

5.31 Variable Ranking - by Single Feature R^2 Score

```
In [18]: test_scores = []
         for feature in dataset_2.columns:
             results = run_model(Lasso(alpha=100), 'variable ranking', 50,
                                 dataset_2[['PC 1', 'GarageCars_3', feature]], target_2)
             test_score = results['r2_test_score']
             if test_score > 0.2:
                 test_scores.append({'feature': feature, 'score' : test_score})
```

```
In [19]: results = pd.DataFrame(test_scores).sort_values('score', ascending=False)
         results.head(20)
```

Out[19]:
```
      feature     score
166          ExterQual_Ex  0.747153
254        KitchenQual_Ex  0.743328
238            FullBath_3  0.741966
60   Neighborhood_StoneBr  0.741198
371             GrLivArea  0.739726
283        FireplaceQu_Gd  0.737686
107        OverallQual_10  0.735050
106         OverallQual_9  0.731267
192       BsmtExposure_Gd  0.731260
383                  PC 2  0.730414
279          Fireplaces_2  0.729005
372           GarageYrBlt  0.727439
368            FirstFlrSF  0.727013
163    MasVnrType_BrkFace  0.726611
286      FireplaceQu_None  0.725934
277          Fireplaces_0  0.725934
376         EnclosedPorch  0.725595
134   Exterior1st_BrkFace  0.725359
181           BsmtQual_Ex  0.725346
267       TotRmsAbvGrd_11  0.725037
```

In [9]:
```
test_scores = []
for feature in dataset_4.columns:
    results = run_model(Lasso(), 'variable ranking', 50, dataset_4[[feature]], target_4)
    test_score = results['r2_test_score']
    if test_score > 0.2:
        test_scores.append({'feature': feature, 'score' : test_score})
```

In [10]:
```
results = pd.DataFrame(test_scores).sort_values('score', ascending=False)
results.head(20)
```

Out[10]:
```
      feature     score
21                 PC 1  0.529066
3            GrLivArea  0.439198
16      GarageCars_3  0.396864
12      ExterQual_TA  0.364821
2          FirstFlrSF  0.360158
9          FullBath_1  0.330278
0           YearBuilt  0.291891
17       BsmtQual_Ex  0.283994
5     KitchenQual_TA  0.279176
11  Foundation_PConc  0.276741
1        YearRemodAdd  0.275810
6          FullBath_2  0.256408
18    KitchenQual_Ex  0.255417
4         GarageYrBlt  0.244169
15  GarageFinish_Fin  0.228146
19      ExterQual_Ex  0.226715
13      ExterQual_Gd  0.218874
20     OverallQual_9  0.212312
14  BsmtFinType1_GLQ  0.207843
7        Fireplaces_0  0.203674
```

In [11]:
```
performant_features = pd.DataFrame()
performant_features['test_1'] = list(results.head(20).feature.values)
performant_features
```

Out[11]:
```
      test_1
0          PC 1
1     GrLivArea
```

```
2       GarageCars_3
3       ExterQual_TA
4         FirstFlrSF
5         FullBath_1
6          YearBuilt
7        BsmtQual_Ex
8     KitchenQual_TA
9   Foundation_PConc
10      YearRemodAdd
11        FullBath_2
12    KitchenQual_Ex
13       GarageYrBlt
14  GarageFinish_Fin
15      ExterQual_Ex
16      ExterQual_Gd
17     OverallQual_9
18  BsmtFinType1_GLQ
19      Fireplaces_0
```

In [12]:
```
features_to_test = ['PC 1']
test_results = {}
for i, feature in enumerate(performant_features.test_1):
    features_to_test.append(feature)
    if i < 7: print(features_to_test)
    test_results[feature] = run_model(Lasso(), 'variable ranking', 100, dataset_4[featu
```

```
['PC 1']
['PC 1', 'GrLivArea']
['PC 1', 'GrLivArea', 'GarageCars_3']
['PC 1', 'GrLivArea', 'GarageCars_3', 'ExterQual_TA']
['PC 1', 'GrLivArea', 'GarageCars_3', 'ExterQual_TA', 'FirstFlrSF']
['PC 1', 'GrLivArea', 'GarageCars_3', 'ExterQual_TA', 'FirstFlrSF', 'FullBath_1']
['PC 1', 'GrLivArea', 'GarageCars_3', 'ExterQual_TA', 'FirstFlrSF', 'FullBath_1', 'YearBuilt']
```

In [13]:
```
test_results = pd.DataFrame(test_results).T
test_results = test_results.loc[pd.Index(features_to_test)]
performant_features['test_1_r2'] = test_results.r2_test_score.values
test_results
```

Out[13]:

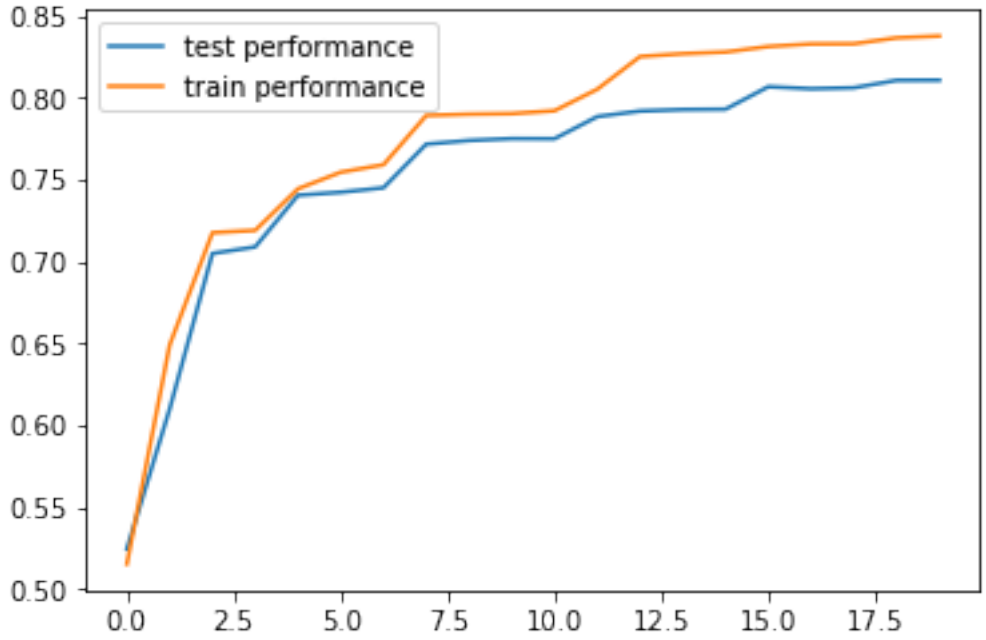
	fit_time	mae_test	mae_train	\
PC 1	5.24521e-06	34411.9	36164.3	
GrLivArea	7.62939e-06	32250.7	33149.4	
GarageCars_3	1.3113e-05	27089.7	29580.2	
ExterQual_TA	6.19888e-06	27160.7	29481.6	
FirstFlrSF	9.05991e-06	24969.4	27802.6	
FullBath_1	7.62939e-06	25078.1	27634.4	
YearBuilt	9.05991e-06	24842.7	27247.7	
BsmtQual_Ex	1.19209e-05	24072.9	25286.6	
KitchenQual_TA	1.16825e-05	23973.1	25261.3	
Foundation_PConc	1.81198e-05	23870	25179.9	
YearRemodAdd	1.12057e-05	23985.6	24999.9	
FullBath_2	8.82149e-06	23174.9	24686.8	
KitchenQual_Ex	1.66893e-05	23007.8	23556.1	
GarageYrBlt	1.40667e-05	22845.2	23407.1	
GarageFinish_Fin	1.38283e-05	23028.1	23321.1	
ExterQual_Ex	1.12057e-05	22434.9	23088.2	
ExterQual_Gd	1.71661e-05	22613.6	22943.8	
OverallQual_9	1.07288e-05	22615.1	22914.8	
BsmtFinType1_GLQ	9.29832e-06	22397.7	22689.8	
Fireplaces_0	4.1008e-05	22456.6	22555.8	

```
                                                                 model  \
PC 1              Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
GrLivArea         Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
GarageCars_3      Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
ExterQual_TA      Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
FirstFlrSF        Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
FullBath_1        Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
YearBuilt         Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
BsmtQual_Ex       Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
KitchenQual_TA    Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
Foundation_PConc  Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
YearRemodAdd      Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
FullBath_2        Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
KitchenQual_Ex    Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
GarageYrBlt       Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
GarageFinish_Fin  Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
ExterQual_Ex      Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
ExterQual_Gd      Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
OverallQual_9     Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
BsmtFinType1_GLQ  Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...
Fireplaces_0      Lasso(alpha=1.0, copy_X=True, fit_intercept=Tr...

                         model_name     n n_pcnt r2_test_score r2_train_score  \
PC 1              variable ranking  1083    100      0.524411       0.515181
GrLivArea         variable ranking  1083    100      0.609907       0.649073
GarageCars_3      variable ranking  1083    100      0.704973       0.717685
ExterQual_TA      variable ranking  1083    100      0.708974       0.719097
FirstFlrSF        variable ranking  1083    100       0.74056       0.744501
FullBath_1        variable ranking  1083    100      0.742397       0.754655
YearBuilt         variable ranking  1083    100      0.745128       0.759213
BsmtQual_Ex       variable ranking  1083    100      0.771795       0.789344
KitchenQual_TA    variable ranking  1083    100      0.774039       0.790125
Foundation_PConc  variable ranking  1083    100      0.775191       0.790451
YearRemodAdd      variable ranking  1083    100      0.775113       0.792222
FullBath_2        variable ranking  1083    100       0.78864       0.805362
KitchenQual_Ex    variable ranking  1083    100       0.79205        0.82535
GarageYrBlt       variable ranking  1083    100      0.792951       0.827089
GarageFinish_Fin  variable ranking  1083    100      0.793154       0.828118
ExterQual_Ex      variable ranking  1083    100      0.806916       0.831478
ExterQual_Gd      variable ranking  1083    100      0.805672       0.833068
OverallQual_9     variable ranking  1083    100      0.806259       0.833166
BsmtFinType1_GLQ  variable ranking  1083    100      0.810811       0.836794
Fireplaces_0      variable ranking  1083    100      0.810902       0.837879

                 rmse_test rmse_train test_pred_time train_pred_time
PC 1               51428.1    56105.5    5.00679e-06     4.76837e-06
GrLivArea          46576.7    47733.5    6.67572e-06      6.4373e-06
GarageCars_3       40505.7    42813.6    6.91414e-06      6.4373e-06
ExterQual_TA         40230    42706.5    7.39098e-06     6.19888e-06
FirstFlrSF         37984.2    40729.6    7.62939e-06     7.86781e-06
FullBath_1         37849.5      39912    7.15256e-06     6.67572e-06
YearBuilt          37648.3    39539.6    6.67572e-06     7.39098e-06
BsmtQual_Ex        35624.4      36983    1.12057e-05     0.000196218
KitchenQual_TA     35448.8    36914.4    1.04904e-05     0.000310898
Foundation_PConc   35358.3    36885.7    9.29832e-06     1.09673e-05
YearRemodAdd       35364.4    36729.5    9.05991e-06     1.00136e-05
FullBath_2         34284.4    35549.2    8.34465e-06     1.09673e-05
KitchenQual_Ex     34006.7    33674.4    8.34465e-06     1.14441e-05
GarageYrBlt          33933    33506.3    8.10623e-06     8.82149e-06
```

```
GarageFinish_Fin   33916.3    33406.5    1.00136e-05     9.29832e-06
ExterQual_Ex       32768.6    33078.3    8.34465e-06     9.77516e-06
ExterQual_Gd         32874    32921.9    8.34465e-06     9.29832e-06
OverallQual_9      32824.3    32912.3    6.19888e-06      6.4373e-06
BsmtFinType1_GLQ   32436.4    32552.5    7.15256e-06     5.96046e-06
Fireplaces_0       32428.6    32444.1    7.39098e-06     1.64509e-05
```

In [14]:
```
plt.plot(range(len(features_to_test)), test_results.loc[features_to_test].r2_test_score
plt.plot(range(len(features_to_test)), test_results.loc[features_to_test].r2_train_scor
plt.legend()
```

Out[14]: <matplotlib.legend.Legend at 0x7f13fc09f710>



In [15]:
```
performant_features
```

Out[15]:

	test_1	test_1_r2
0	PC 1	0.524411
1	GrLivArea	0.609907
2	GarageCars_3	0.704973
3	ExterQual_TA	0.708974
4	FirstFlrSF	0.74056
5	FullBath_1	0.742397
6	YearBuilt	0.745128
7	BsmtQual_Ex	0.771795
8	KitchenQual_TA	0.774039
9	Foundation_PConc	0.775191
10	YearRemodAdd	0.775113
11	FullBath_2	0.78864
12	KitchenQual_Ex	0.79205
13	GarageYrBlt	0.792951
14	GarageFinish_Fin	0.793154
15	ExterQual_Ex	0.806916
16	ExterQual_Gd	0.805672
17	OverallQual_9	0.806259
18	BsmtFinType1_GLQ	0.810811
19	Fireplaces_0	0.810902

5.32 Variable-Ranking - By Regression Coefficient in Full Model

```
In [20]: results = run_model(Lasso(alpha=100), 'ridge', 100, dataset_2, target_2)

In [21]: results

Out[21]: {'model': Lasso(alpha=100, copy_X=True, fit_intercept=True, max_iter=1000,
             normalize=False, positive=False, precompute=False, random_state=None,
             selection='cyclic', tol=0.0001, warm_start=False),
          'model_name': 'ridge',
          'n_pcnt': 100,
          'n': 1083,
          'rmse_train': 19977.69546430136,
          'rmse_test': 26760.762077613879,
          'mae_train': 13914.117123911166,
          'mae_test': 16624.037016658658,
          'r2_train_score': 0.93853045599788454,
          'r2_test_score': 0.87122620309522536,
          'fit_time': 9.059906005859375e-06,
          'train_pred_time': 6.9141387939453125e-06,
          'test_pred_time': 6.4373016357421875e-06}

In [22]: lasso_model = results['model']

In [23]: coefficients = lasso_model.coef_
         features = dataset_2.columns
         coefficients = pd.Series(coefficients.T.ravel(), index=features)
         coefficients.head()

Out[23]: MSSubClass_20   -0.0
         MSSubClass_30    0.0
         MSSubClass_40    0.0
         MSSubClass_45    0.0
         MSSubClass_50   -0.0
         dtype: float64

In [24]: sorted_coefs = np.abs(coefficients).sort_values(ascending=False)
         sorted_coefs.head(20)

Out[24]: OverallQual_10          96968.520841
         FullBath_3              45783.940099
         OverallQual_9           43237.143982
         RoofMatl_WdShngl        39103.466850
         TotRmsAbvGrd_12         31467.344131
         Neighborhood_NoRidge    31183.724516
         KitchenQual_Ex          24809.068911
         Neighborhood_Crawfor    21985.695265
         OverallQual_8           20237.191155
         GrLivArea               19029.468600
         Exterior1st_BrkFace     18822.244869
         Neighborhood_StoneBr    17938.323701
         BsmtQual_Ex             17047.181455
         BsmtExposure_Gd         15743.867771
         Functional_Typ          14675.735873
         OverallCond_3           14207.601075
         KitchenAbvGr_1          12417.763266
         TotRmsAbvGrd_10         11923.073621
         GarageCars_3             9914.039220
         GarageType_2Types        9747.428012
```

```
dtype: float64
```

In [25]:
```
performant_features['test_2'] = list(list(sorted_coefs.head(20).index))
performant_features
```

Out[25]:

	test_1	test_1_r2	test_2
0	PC 1	0.524411	OverallQual_10
1	GrLivArea	0.609907	FullBath_3
2	GarageCars_3	0.704973	OverallQual_9
3	ExterQual_TA	0.708974	RoofMatl_WdShngl
4	FirstFlrSF	0.74056	TotRmsAbvGrd_12
5	FullBath_1	0.742397	Neighborhood_NoRidge
6	YearBuilt	0.745128	KitchenQual_Ex
7	BsmtQual_Ex	0.771795	Neighborhood_Crawfor
8	KitchenQual_TA	0.774039	OverallQual_8
9	Foundation_PConc	0.775191	GrLivArea
10	YearRemodAdd	0.775113	Exterior1st_BrkFace
11	FullBath_2	0.78864	Neighborhood_StoneBr
12	KitchenQual_Ex	0.79205	BsmtQual_Ex
13	GarageYrBlt	0.792951	BsmtExposure_Gd
14	GarageFinish_Fin	0.793154	Functional_Typ
15	ExterQual_Ex	0.806916	OverallCond_3
16	ExterQual_Gd	0.805672	KitchenAbvGr_1
17	OverallQual_9	0.806259	TotRmsAbvGrd_10
18	BsmtFinType1_GLQ	0.810811	GarageCars_3
19	Fireplaces_0	0.810902	GarageType_2Types

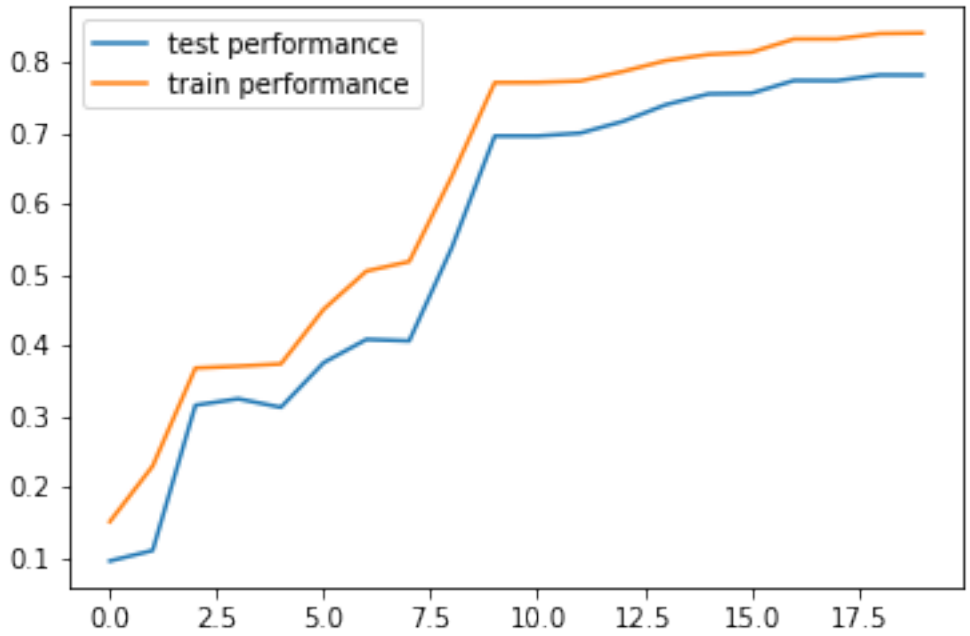
In [30]:
```
features_to_test = []
test_results = {}
for feature in performant_features.test_2:
    features_to_test.append(feature)
    print(dataset_2[features_to_test].shape)
    test_results[feature] = run_model(Lasso(alpha=100), 'lasso', 100,
                                      dataset_2[features_to_test],
                                      target_2)
```

```
(1444, 1)
(1444, 2)
(1444, 3)
(1444, 4)
(1444, 5)
(1444, 6)
(1444, 7)
(1444, 8)
(1444, 9)
(1444, 10)
(1444, 11)
(1444, 12)
(1444, 13)
(1444, 14)
(1444, 15)
(1444, 16)
(1444, 17)
(1444, 18)
(1444, 19)
(1444, 20)
```

In [31]:
```
test_results = pd.DataFrame(test_results).T
test_results = test_results.loc[features_to_test]
performant_features['test_2_r2'] = test_results.r2_test_score.values
```

```
        plt.plot(range(len(features_to_test)), test_results.r2_test_score, label='test performa
        plt.plot(range(len(features_to_test)), test_results.r2_train_score, label='train perfor
        plt.legend()
```

Out[31]: <matplotlib.legend.Legend at 0x7f13f77eabe0>



In [32]: performant_features

Out[32]:

	test_1	test_1_r2	test_2	test_2_r2
0	PC 1	0.524411	OverallQual_10	0.0947046
1	GrLivArea	0.609907	FullBath_3	0.109537
2	GarageCars_3	0.704973	OverallQual_9	0.314819
3	ExterQual_TA	0.708974	RoofMatl_WdShngl	0.323926
4	FirstFlrSF	0.74056	TotRmsAbvGrd_12	0.311868
5	FullBath_1	0.742397	Neighborhood_NoRidge	0.375082
6	YearBuilt	0.745128	KitchenQual_Ex	0.407884
7	BsmtQual_Ex	0.771795	Neighborhood_Crawfor	0.405762
8	KitchenQual_TA	0.774039	OverallQual_8	0.538098
9	Foundation_PConc	0.775191	GrLivArea	0.695339
10	YearRemodAdd	0.775113	Exterior1st_BrkFace	0.695383
11	FullBath_2	0.78864	Neighborhood_StoneBr	0.699452
12	KitchenQual_Ex	0.79205	BsmtQual_Ex	0.716074
13	GarageYrBlt	0.792951	BsmtExposure_Gd	0.739555
14	GarageFinish_Fin	0.793154	Functional_Typ	0.754858
15	ExterQual_Ex	0.806916	OverallCond_3	0.755627
16	ExterQual_Gd	0.805672	KitchenAbvGr_1	0.77384
17	OverallQual_9	0.806259	TotRmsAbvGrd_10	0.773601
18	BsmtFinType1_GLQ	0.810811	GarageCars_3	0.781565
19	Fireplaces_0	0.810902	GarageType_2Types	0.781456

5.32.1 Variable-Ranking - By Information Gain in Full Model

```
In [35]: from sklearn.tree import DecisionTreeRegressor

In [36]: simple_dtree_results = run_model(DecisionTreeRegressor(), 'dtree', 100,
                                          dataset_2,
```

```
                                        target_2)
```

```
In [37]: simple_dtree_results
```

```
Out[37]: {'model': DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
                      max_leaf_nodes=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=None, splitter='best'),
          'model_name': 'dtree',
          'n_pcnt': 100,
          'n': 1083,
          'rmse_train': 0.0,
          'rmse_test': 38636.593591461977,
          'mae_train': 0.0,
          'mae_test': 25201.42105263158,
          'r2_train_score': 1.0,
          'r2_test_score': 0.73157175819419062,
          'fit_time': 8.106231689453125e-06,
          'train_pred_time': 5.7220458984375e-06,
          'test_pred_time': 8.106231689453125e-06}
```

```
In [38]: simple_decision_tree_model = simple_dtree_results['model']
         feature_importances = simple_decision_tree_model.feature_importances_
         features = dataset_2.columns
         feature_importances = pd.Series(feature_importances.T.ravel(), index=features).sort_val
         feature_importances.head(20)
```

```
Out[38]: PC 1                   0.745611
         GrLivArea              0.086921
         BsmtQual_Ex            0.020488
         BsmtFinSF1             0.016852
         TotalBsmtSF            0.011759
         GarageArea             0.011537
         LotArea                0.008968
         FirstFlrSF             0.006774
         ScreenPorch            0.006661
         PC 4                   0.005900
         KitchenQual_TA         0.005811
         YrSold                 0.004445
         Neighborhood_Crawfor   0.004072
         BsmtUnfSF              0.003463
         OverallQual_9          0.003332
         PC 2                   0.002989
         PC 5                   0.002972
         BsmtFinSF2             0.002891
         PC 7                   0.002306
         LotFrontage            0.001979
         dtype: float64
```

```
In [39]: performant_features['test_3'] = feature_importances.head(20).index
```

```
In [40]: performant_features
```

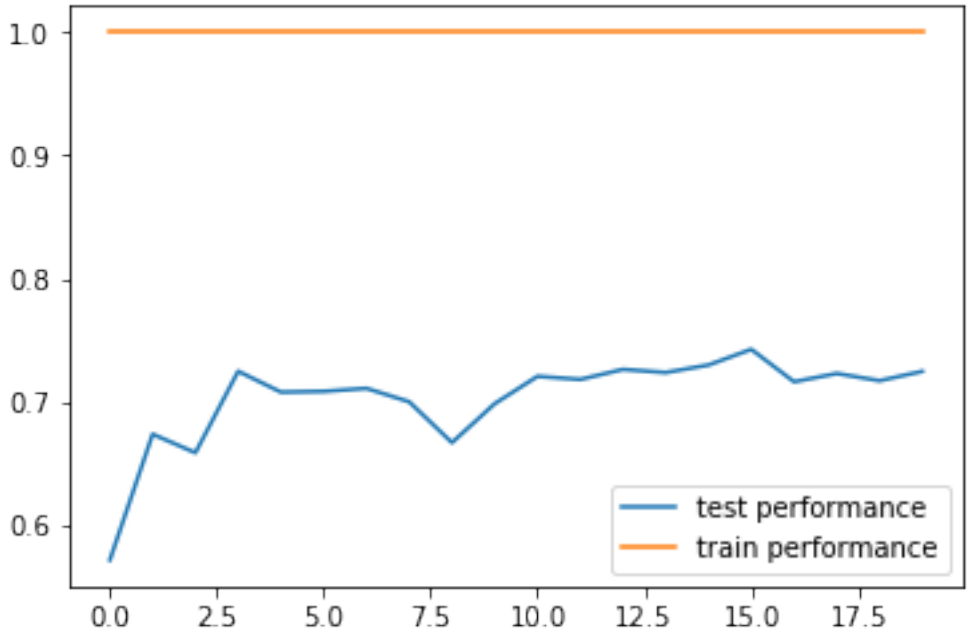
```
Out[40]: test_1 test_1_r2               test_2  test_2_r2  \
         0               PC 1  0.524411        OverallQual_10  0.0947046
         1          GrLivArea  0.609907            FullBath_3   0.109537
         2       GarageCars_3  0.704973         OverallQual_9   0.314819
         3       ExterQual_TA  0.708974      RoofMatl_WdShngl   0.323926
         4         FirstFlrSF   0.74056       TotRmsAbvGrd_12   0.311868
         5         FullBath_1  0.742397  Neighborhood_NoRidge   0.375082
         6          YearBuilt  0.745128        KitchenQual_Ex   0.407884
```

```
7        BsmtQual_Ex  0.771795  Neighborhood_Crawfor   0.405762
8      KitchenQual_TA  0.774039         OverallQual_8   0.538098
9    Foundation_PConc  0.775191             GrLivArea   0.695339
10       YearRemodAdd  0.775113   Exterior1st_BrkFace   0.695383
11         FullBath_2   0.78864  Neighborhood_StoneBr   0.699452
12     KitchenQual_Ex   0.79205           BsmtQual_Ex   0.716074
13        GarageYrBlt  0.792951       BsmtExposure_Gd   0.739555
14   GarageFinish_Fin  0.793154        Functional_Typ   0.754858
15       ExterQual_Ex  0.806916         OverallCond_3   0.755627
16       ExterQual_Gd  0.805672         KitchenAbvGr_1    0.77384
17      OverallQual_9  0.806259       TotRmsAbvGrd_10   0.773601
18   BsmtFinType1_GLQ  0.810811          GarageCars_3   0.781565
19       Fireplaces_0  0.810902     GarageType_2Types   0.781456

                  test_3
0                  PC 1
1              GrLivArea
2            BsmtQual_Ex
3             BsmtFinSF1
4            TotalBsmtSF
5             GarageArea
6                LotArea
7             FirstFlrSF
8            ScreenPorch
9                  PC 4
10        KitchenQual_TA
11               YrSold
12  Neighborhood_Crawfor
13            BsmtUnfSF
14         OverallQual_9
15                 PC 2
16                 PC 5
17            BsmtFinSF2
18                 PC 7
19           LotFrontage
```

In [41]:
```
features_to_test = []
test_results = {}
for feature in performant_features.test_3:
    features_to_test.append(feature)
    test_results[feature] = run_model(DecisionTreeRegressor(), 'dtree', 100,
                                      dataset_2[features_to_test],
                                      target_2)
```

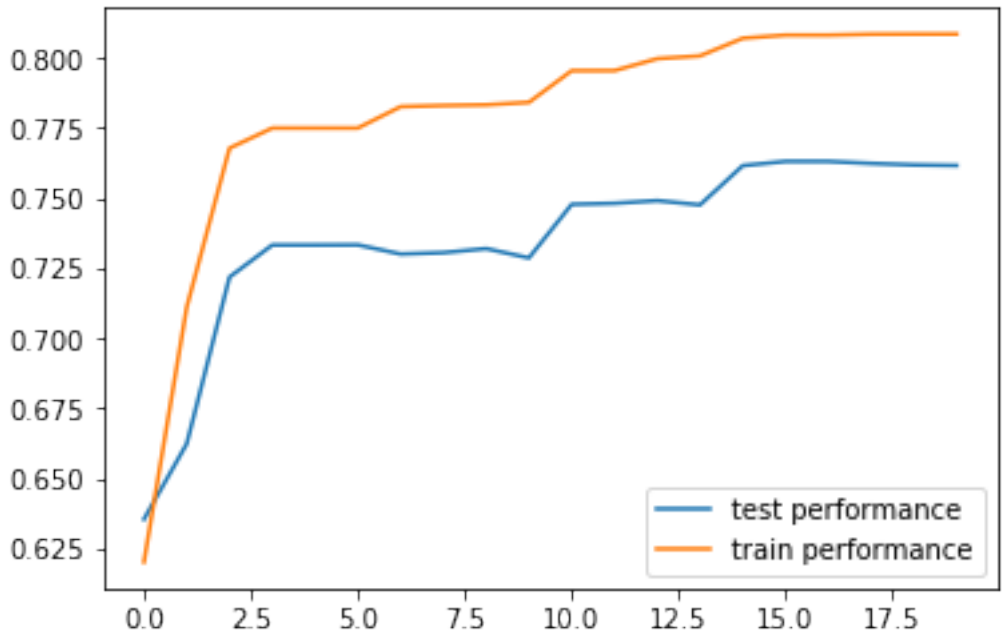
In [42]:
```
test_results = pd.DataFrame(test_results).T
test_results = test_results.loc[features_to_test]
performant_features['test_3_r2_dtree'] = test_results.r2_test_score.values

plt.plot(range(len(features_to_test)), test_results.r2_test_score, label='test performa
plt.plot(range(len(features_to_test)), test_results.r2_train_score, label='train perfor
plt.legend()
```

Out[42]:
```
<matplotlib.legend.Legend at 0x7f13f6c76860>
```



In [43]:
```
features_to_test = []
test_results = {}
for feature in performant_features.test_3:
    features_to_test.append(feature)
    test_results[feature] = run_model(Lasso(), 'lasso', 100,
                                      dataset_2[features_to_test],
                                      target_2)

test_results = pd.DataFrame(test_results).T
test_results = test_results.loc[features_to_test]
performant_features['test_3_r2_lasso'] = test_results.r2_test_score.values

plt.plot(range(len(features_to_test)), test_results.r2_test_score, label='test performa
plt.plot(range(len(features_to_test)), test_results.r2_train_score, label='train perfor
plt.legend()
```

Out[43]: <matplotlib.legend.Legend at 0x7f13f6c69860>



In [31]: performant_features

Out[31]:

	test_1	test_1_r2	test_2	test_2_r2
0	PC 1	0.52441	OverallQual_10	0.0947046
1	GrLivArea	0.609907	FullBath_3	0.109537
2	GarageCars_3	0.704973	OverallQual_9	0.314819
3	ExterQual_TA	0.708974	RoofMatl_WdShngl	0.323926
4	FirstFlrSF	0.74056	TotRmsAbvGrd_12	0.311868
5	FullBath_1	0.742397	Neighborhood_NoRidge	0.375082
6	YearBuilt	0.745128	KitchenQual_Ex	0.407884
7	BsmtQual_Ex	0.771795	Neighborhood_Crawfor	0.405762
8	KitchenQual_TA	0.774039	OverallQual_8	0.538098
9	Foundation_PConc	0.775191	GrLivArea	0.695339
10	YearRemodAdd	0.775113	Exterior1st_BrkFace	0.695383
11	FullBath_2	0.78864	Neighborhood_StoneBr	0.699452
12	KitchenQual_Ex	0.79205	BsmtQual_Ex	0.716074
13	GarageYrBlt	0.792951	BsmtExposure_Gd	0.739555
14	GarageFinish_Fin	0.793154	Functional_Typ	0.754858
15	ExterQual_Ex	0.806916	OverallCond_3	0.755627
16	ExterQual_Gd	0.805672	KitchenAbvGr_1	0.77384
17	OverallQual_9	0.806259	TotRmsAbvGrd_10	0.773601
18	BsmtFinType1_GLQ	0.810811	GarageCars_3	0.781565
19	Fireplaces_0	0.810902	GarageType_2Types	0.781456

	test_3	test_3_r2_dtree	test_3_r2_lasso
0	PC 1	0.564312	0.63554
1	GrLivArea	0.668869	0.662442
2	PC 2	0.628067	0.665369
3	BsmtQual_Ex	0.636468	0.72597
4	BsmtFinSF1	0.733137	0.733676
5	GarageArea	0.741512	0.7328
6	TotalBsmtSF	0.727177	0.732431
7	ScreenPorch	0.698815	0.733208
8	LotArea	0.678924	0.731901
9	FirstFlrSF	0.668312	0.734013
10	PC 4	0.68211	0.730377

11	KitchenQual_TA	0.738347	0.748825
12	Neighborhood_Crawfor	0.715029	0.750539
13	PC 7	0.723296	0.750541
14	BsmtUnfSF	0.724524	0.749265
15	OverallQual_9	0.709766	0.76316
16	BsmtFinSF2	0.720974	0.762772
17	YrSold	0.728116	0.762219
18	PC 5	0.723365	0.761886
19	YearBuilt	0.711027	0.765086