# ML Tutorial 2

June 13, 2018

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import sklearn
        import seaborn as sns
        sns.set()
        %matplotlib inline
```

## 0.1 Arrays & Dataframes

Recall we use arrays (and array of arrays) to deal with data in numpy:

```
In [2]: data = np.random.multivariate_normal([155,65],[[5,3.5],[3.5,5]],20)
        print(data)
```

```
[[153.81830494  66.75216825]
 [153.24958843  63.97775387]
 [155.53157868  66.35131989]
 [155.74792032  65.20016683]
 [157.0232017   68.94099499]
 [154.81196077  62.67780174]
 [155.32994637  64.57465533]
 [153.73971327  66.58747603]
 [153.39280592  66.46613757]
 [157.75488641  66.94391405]
 [157.19444109  65.92912883]
 [156.33356568  64.09789433]
 [158.1872879   68.0998284 ]
 [149.7580223   57.92745011]
 [155.14979759  65.49127875]
 [156.25311415  67.28307103]
 [157.6431965   68.52903368]
 [154.30779624  63.09863841]
 [156.99808757  64.89770124]
 [153.49262644  63.87615423]]
```

We can reshape the array using the `.reshape` method

```
In [3]: print(data.reshape(-1,4))

[[153.81830494  66.75216825 153.24958843  63.97775387]
 [155.53157868  66.35131989 155.74792032  65.20016683]
 [157.0232017   68.94099499 154.81196077  62.67780174]
 [155.32994637  64.57465533 153.73971327  66.58747603]
 [153.39280592  66.46613757 157.75488641  66.94391405]
 [157.19444109  65.92912883 156.33356568  64.09789433]
 [158.1872879   68.0998284  149.7580223   57.92745011]
 [155.14979759  65.49127875 156.25311415  67.28307103]
 [157.6431965   68.52903368 154.30779624  63.09863841]
 [156.99808757  64.89770124 153.49262644  63.87615423]]
```

Note that when reshaping, you can use the $-1$ flag to indicate an automatic calculation of the column/row>

Sometimes dealing with arrays is a pain, using dataframes is a way to solve the problem:

```
In [4]: df = pd.DataFrame(data,columns=("h","w"))

        pd.DataFrame.head(df)

Out[4]:             h          w
        0  153.818305  66.752168
        1  153.249588  63.977754
        2  155.531579  66.351320
        3  155.747920  65.200167
        4  157.023202  68.940995
```

We can add a column by simply assigning a new column into the dataframe, this immediately implies that dataframe is **mutable** :

```
In [5]: df['r'] = np.sqrt((df['w'] + df['h'])**2)

        pd.DataFrame.head(df)

Out[5]:             h          w           r
        0  153.818305  66.752168  220.570473
        1  153.249588  63.977754  217.227342
        2  155.531579  66.351320  221.882899
        3  155.747920  65.200167  220.948087
        4  157.023202  68.940995  225.964197
```

You can delete columns using the `.drop` method, however this method does not cause the original dataframe to mutate, unless you have the `inplace` parameter set to `True`:

```
In [6]: df.drop(columns=['r'])

        pd.DataFrame.head(df)
```

```
Out[6]:              h          w           r
        0  153.818305  66.752168  220.570473
        1  153.249588  63.977754  217.227342
        2  155.531579  66.351320  221.882899
        3  155.747920  65.200167  220.948087
        4  157.023202  68.940995  225.964197
```

```
In [7]: df.drop(columns=['r'],inplace=True)

        pd.DataFrame.head(df)

        df['r'] = np.sqrt((df['w'] + df['h'])**2)
```

Most of the time you DO NOT want to delete data. A commonly used best practice is to simply create another data frame for the data you are trying to manipulate:

```
In [8]: df2 = df.drop(columns=['r'])

        pd.DataFrame.head(df2)
```

```
Out[8]:              h          w
        0  153.818305  66.752168
        1  153.249588  63.977754
        2  155.531579  66.351320
        3  155.747920  65.200167
        4  157.023202  68.940995
```

We can merge dataframes with the same column names together:

```
In [9]: D = pd.DataFrame(np.arange(0,40).reshape(-1,2),columns=("A","B"))
        pd.DataFrame.head(D)
```

```
Out[9]:    A  B
        0  0  1
        1  2  3
        2  4  5
        3  6  7
        4  8  9
```

```
In [10]: E = pd.DataFrame(np.arange(41,51).reshape(-1,2),columns=("A","B"))
         pd.DataFrame.head(E)
```

```
Out[10]:     A   B
         0  41  42
         1  43  44
         2  45  46
         3  47  48
         4  49  50
```

```
In [11]: combined_df = pd.concat([D,E])
         combined_df

Out[11]:      A   B
         0    0   1
         1    2   3
         2    4   5
         3    6   7
         4    8   9
         5   10  11
         6   12  13
         7   14  15
         8   16  17
         9   18  19
         10  20  21
         11  22  23
         12  24  25
         13  26  27
         14  28  29
         15  30  31
         16  32  33
         17  34  35
         18  36  37
         19  38  39
         0   41  42
         1   43  44
         2   45  46
         3   47  48
         4   49  50
```

If you try to concatenate two dataframes with different column indices, pandas will encourage you to do the concatenation along the columns this results in the concatenation based on the `iloc`:

```
In [12]: F = pd.DataFrame(np.arange(0,40).reshape(-1,2),columns=("A1","B1"))

         G = pd.DataFrame(np.arange(41,51).reshape(-1,2),columns=("A2","B2"))

         combined_df2 = pd.concat([F,G],axis = 1)
         combined_df2

Out[12]:     A1  B1    A2    B2
         0    0   1  41.0  42.0
         1    2   3  43.0  44.0
         2    4   5  45.0  46.0
         3    6   7  47.0  48.0
         4    8   9  49.0  50.0
         5   10  11   NaN   NaN
         6   12  13   NaN   NaN
         7   14  15   NaN   NaN
```

```
8   16  17   NaN   NaN
9   18  19   NaN   NaN
10  20  21   NaN   NaN
11  22  23   NaN   NaN
12  24  25   NaN   NaN
13  26  27   NaN   NaN
14  28  29   NaN   NaN
15  30  31   NaN   NaN
16  32  33   NaN   NaN
17  34  35   NaN   NaN
18  36  37   NaN   NaN
19  38  39   NaN   NaN
```

If you attempt to concatenate the two dataframes along the rows, pandas will return the following:

```
In [13]: combined_df3 = pd.concat([F,G])
         combined_df3
```

```
/usr/local/lib/python3.6/site-packages/ipykernel_launcher.py:1: FutureWarning: Sorting because
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=True'.

To retain the current behavior and silence the warning, pass sort=False

  """Entry point for launching an IPython kernel.
```

```
Out[13]:       A1    A2    B1    B2
         0    0.0   NaN   1.0   NaN
         1    2.0   NaN   3.0   NaN
         2    4.0   NaN   5.0   NaN
         3    6.0   NaN   7.0   NaN
         4    8.0   NaN   9.0   NaN
         5   10.0   NaN  11.0   NaN
         6   12.0   NaN  13.0   NaN
         7   14.0   NaN  15.0   NaN
         8   16.0   NaN  17.0   NaN
         9   18.0   NaN  19.0   NaN
         10  20.0   NaN  21.0   NaN
         11  22.0   NaN  23.0   NaN
         12  24.0   NaN  25.0   NaN
         13  26.0   NaN  27.0   NaN
         14  28.0   NaN  29.0   NaN
         15  30.0   NaN  31.0   NaN
         16  32.0   NaN  33.0   NaN
         17  34.0   NaN  35.0   NaN
         18  36.0   NaN  37.0   NaN
```

```
19  38.0   NaN  39.0   NaN
0    NaN  41.0   NaN  42.0
1    NaN  43.0   NaN  44.0
2    NaN  45.0   NaN  46.0
3    NaN  47.0   NaN  48.0
4    NaN  49.0   NaN  50.0
```

This results in a dataframe containing complementary information from the two prior dataframes.

## 0.2 Plotting using MatPlotLib and Seaborn

Let's consider the iris dataset.

```
In [14]: iris = sns.load_dataset("iris")
```

```
In [15]: iris.head()
```

```
Out[15]:    sepal_length  sepal_width  petal_length  petal_width species
         0           5.1          3.5           1.4          0.2  setosa
         1           4.9          3.0           1.4          0.2  setosa
         2           4.7          3.2           1.3          0.2  setosa
         3           4.6          3.1           1.5          0.2  setosa
         4           5.0          3.6           1.4          0.2  setosa
```

Let's find out the number of unique species in this dataset:

```
In [16]: iris['species'].unique()
```

```
Out[16]: array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

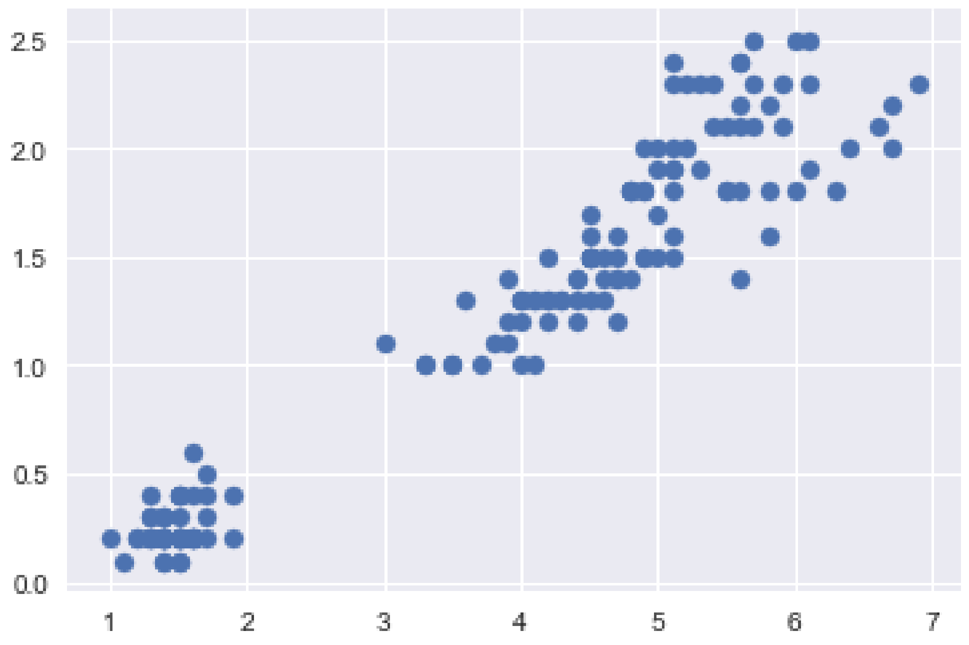
Now let's explore the data set by plotting a few graphs:

### 0.2.1 Scatter Plots

If we want to visualise two continuous variables, we would try to view it as a **scatter plot**. Let's try to plot the petal length against petal width:

```
In [17]: plt.scatter(iris['petal_length'],iris['petal_width'])
```

```
Out[17]: <matplotlib.collections.PathCollection at 0x10f7a7ba8>
```

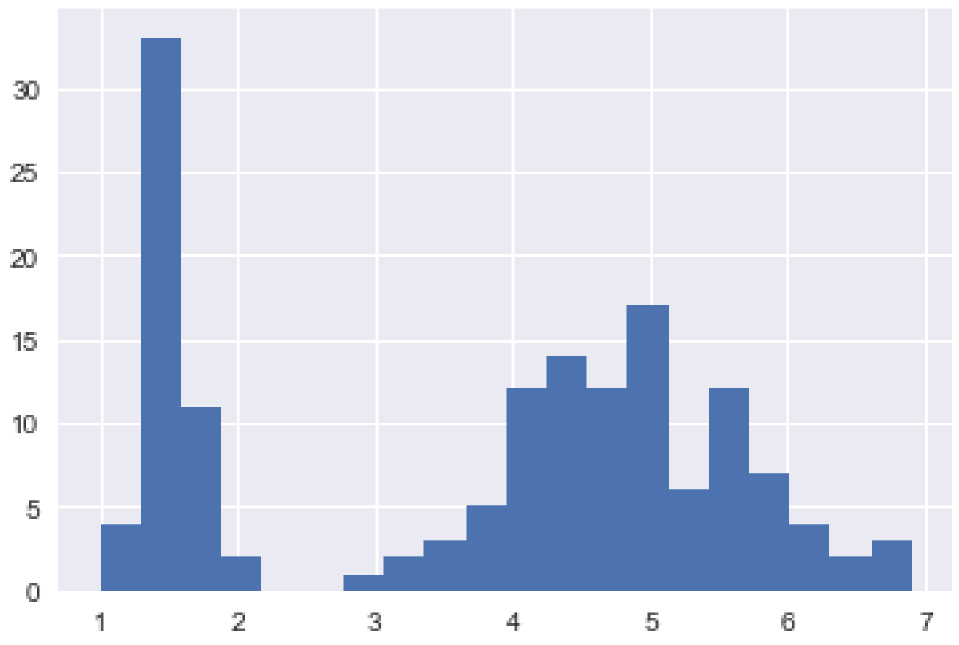


### 0.2.2 Histograms

If we want to visualise a single continuous variable, we would want to try to view it's distribution over a number line. Let's try plotting a **histogram** to observe the distribution of the petal length and petal widths, we set the number of bins (bars) to 20:

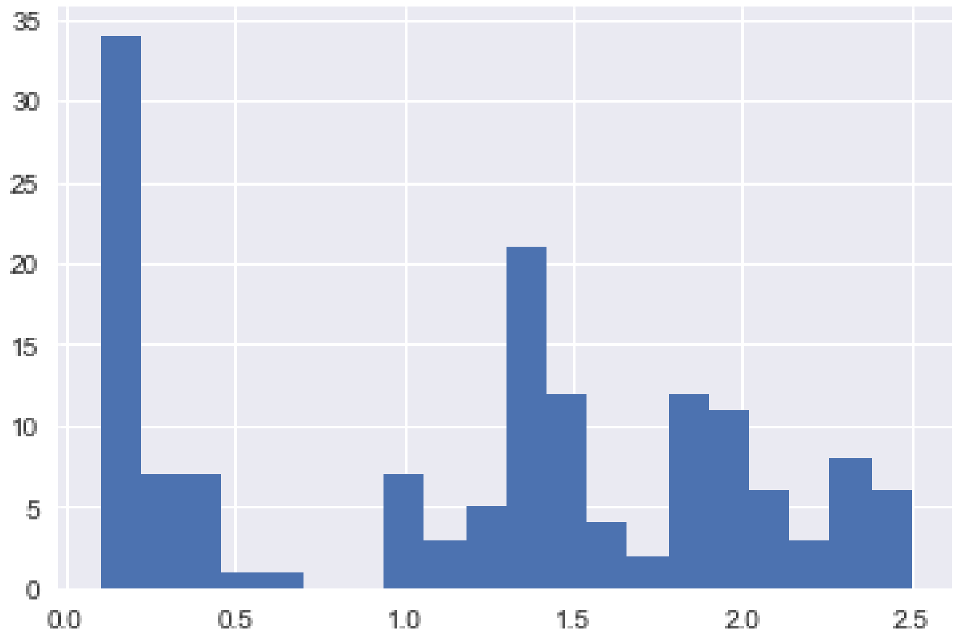
```
In [18]: plt.hist(iris['petal_length'],bins=20)
```

```
Out[18]: (array([ 4., 33., 11.,  2.,  0.,  0.,  1.,  2.,  3.,  5., 12., 14., 12.,
                17.,  6., 12.,  7.,  4.,  2.,  3.]),
          array([1.   , 1.295, 1.59 , 1.885, 2.18 , 2.475, 2.77 , 3.065, 3.36 ,
                3.655, 3.95 , 4.245, 4.54 , 4.835, 5.13 , 5.425, 5.72 , 6.015,
                6.31 , 6.605, 6.9  ]),
          <a list of 20 Patch objects>)
```



```
In [19]: plt.hist(iris['petal_width'],bins=20)

Out[19]: (array([34.,  7.,  7.,  1.,  1.,  0.,  0.,  7.,  3.,  5., 21., 12.,  4.,
          2., 12., 11.,  6.,  3.,  8.,  6.]),
  array([0.1 , 0.22, 0.34, 0.46, 0.58, 0.7 , 0.82, 0.94, 1.06, 1.18, 1.3 ,
         1.42, 1.54, 1.66, 1.78, 1.9 , 2.02, 2.14, 2.26, 2.38, 2.5 ]),
  <a list of 20 Patch objects>)
```



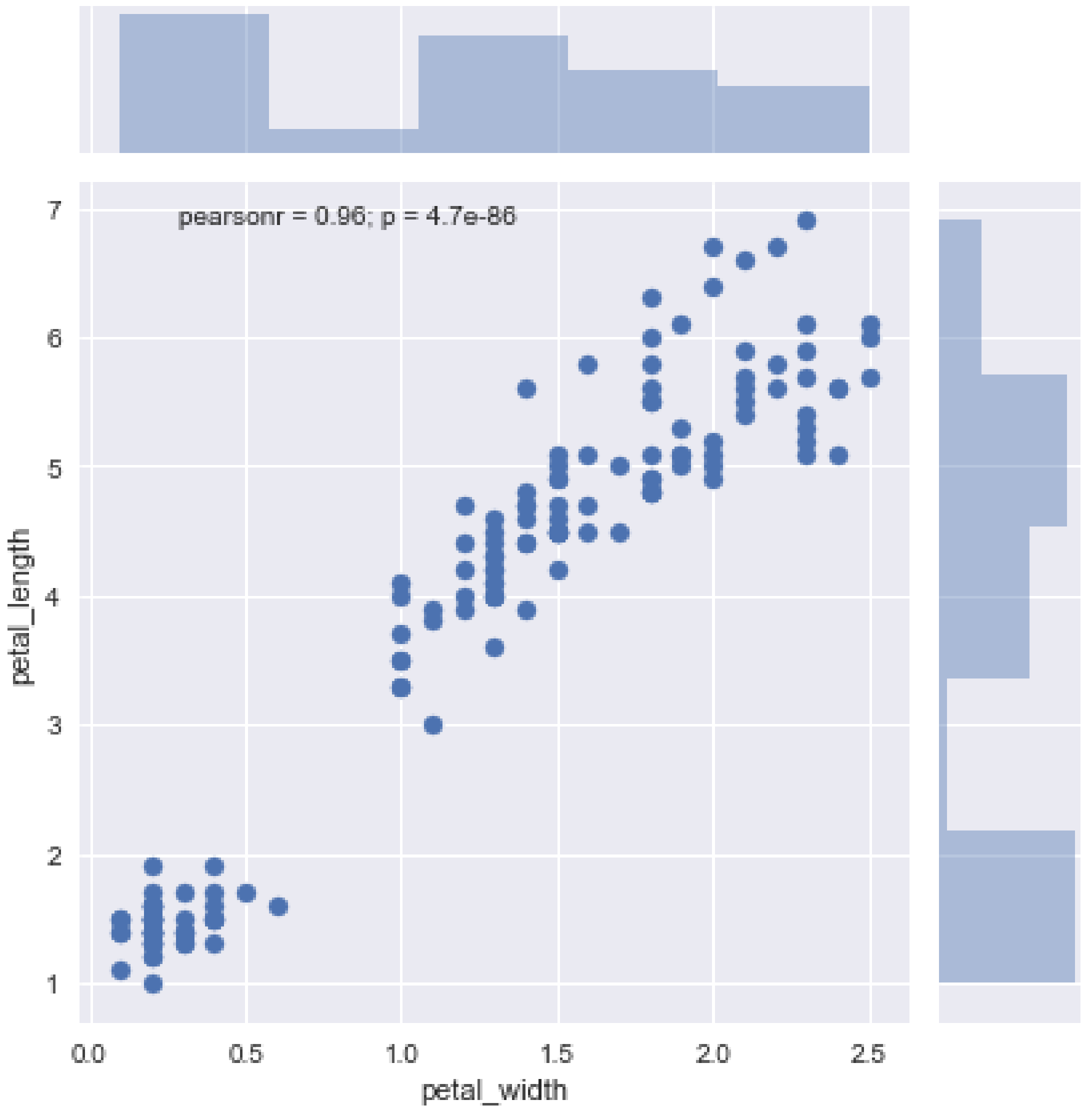
### 0.2.3 Joint Plots

Seaborn allows us to plot visualise the prior plots together. Let's now visualise the two variables together in one plot:

```
In [20]: sns.jointplot(iris['petal_width'],iris['petal_length'],kind = 'scatter')
```

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

```
Out[20]: <seaborn.axisgrid.JointGrid at 0x10f916c18>
```



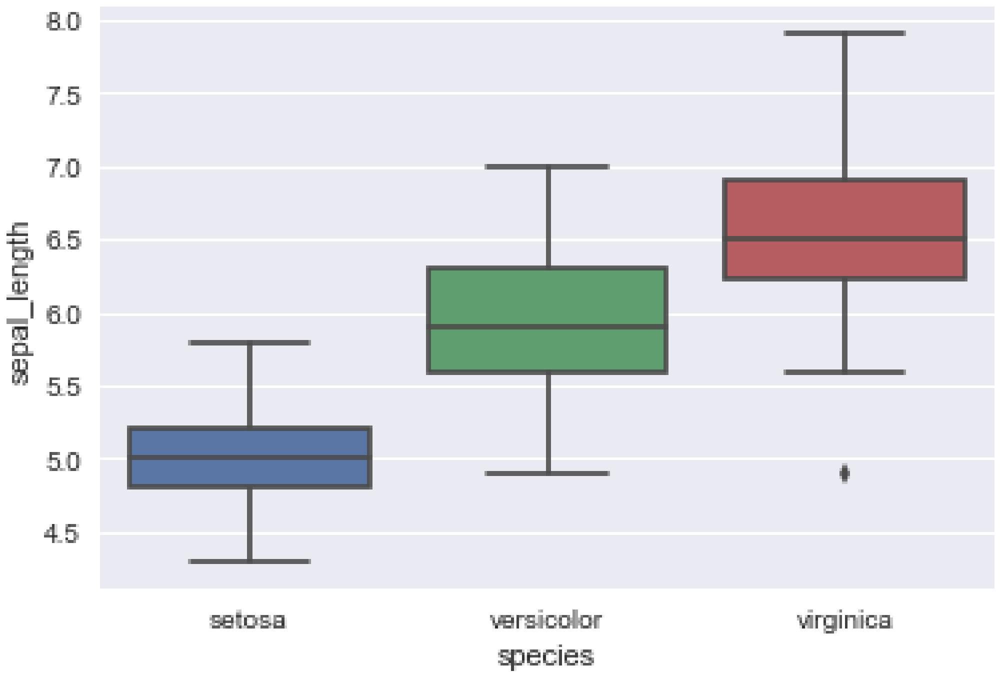
### 0.2.4 Box Plots

Supposing now that we want to compare the distributions for each species of flowers, one of the most common ways to present this information is through a box plot. In general this is a good idea if you want to compare continuous variabel against a categorical variable.

Let's compare the Sepal Length for each Species:

```
In [21]: sns.boxplot(iris['species'],iris['sepal_length'])
```

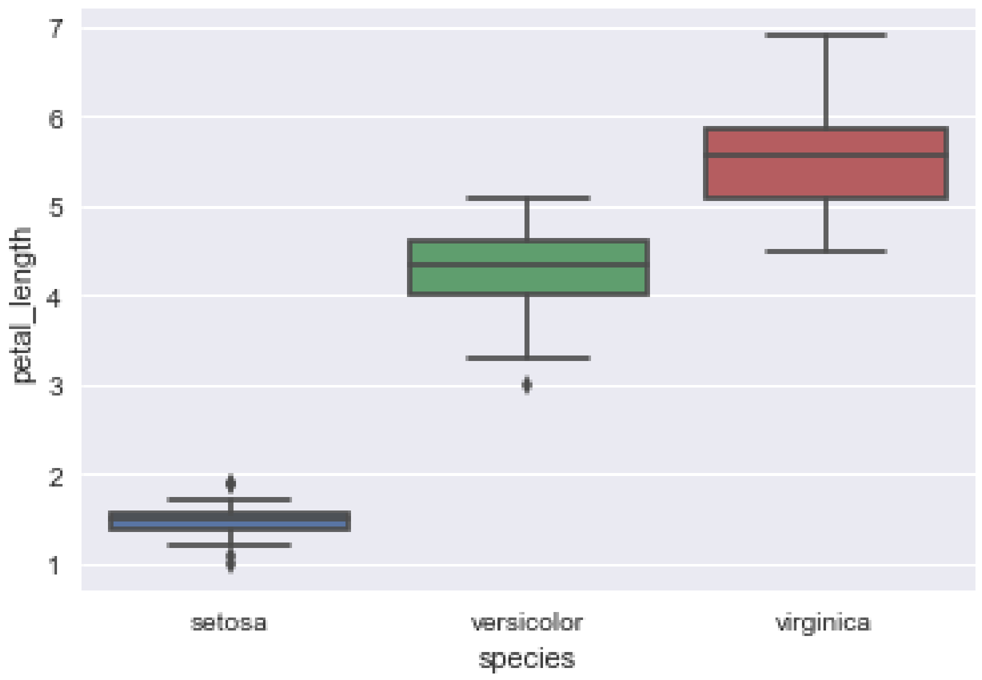
```
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x10ff4e7b8>
```



And again for the Petal Length:

```
In [22]: sns.boxplot(iris['species'],iris['petal_length'])
```

```
Out[22]: <matplotlib.axes._subplots.AxesSubplot at 0x110080d68>
```



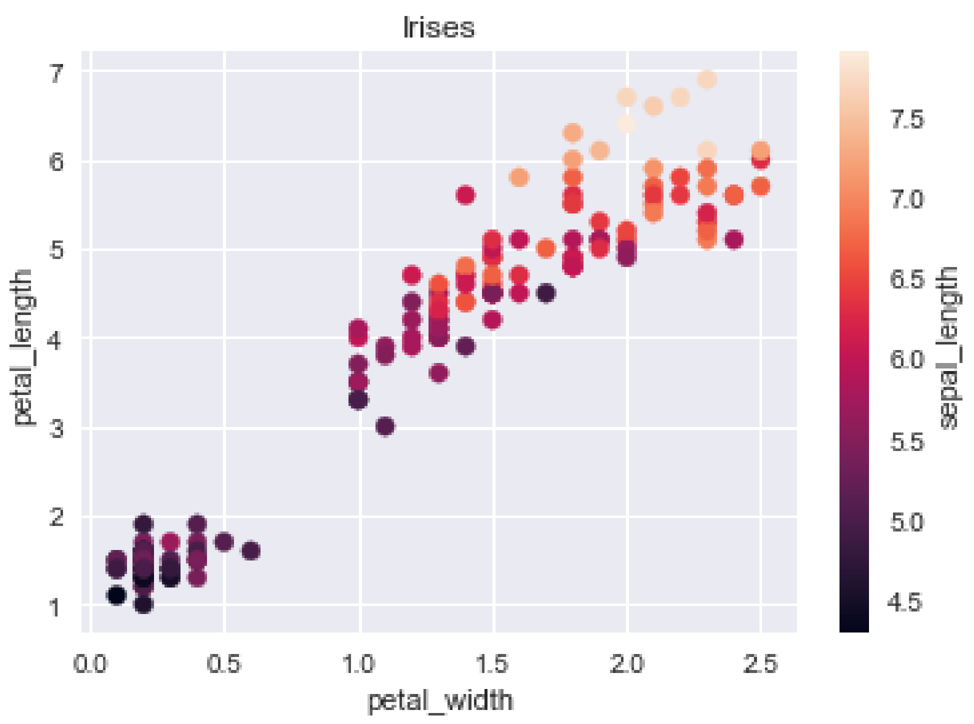
## 0.3 Scatter Plots with added Dimensions

What if we want to consider three variables at the same time? We can always segment by a new dimension using size or colour:

```
In [23]: plt.scatter(iris['petal_width'],iris['petal_length'],c=iris['sepal_length'])

         plt.colorbar().set_label('sepal_length')
         plt.xlabel('petal_width')
         plt.ylabel('petal_length')
         plt.title("Irises")

Out[23]: Text(0.5,1,'Irises')
```

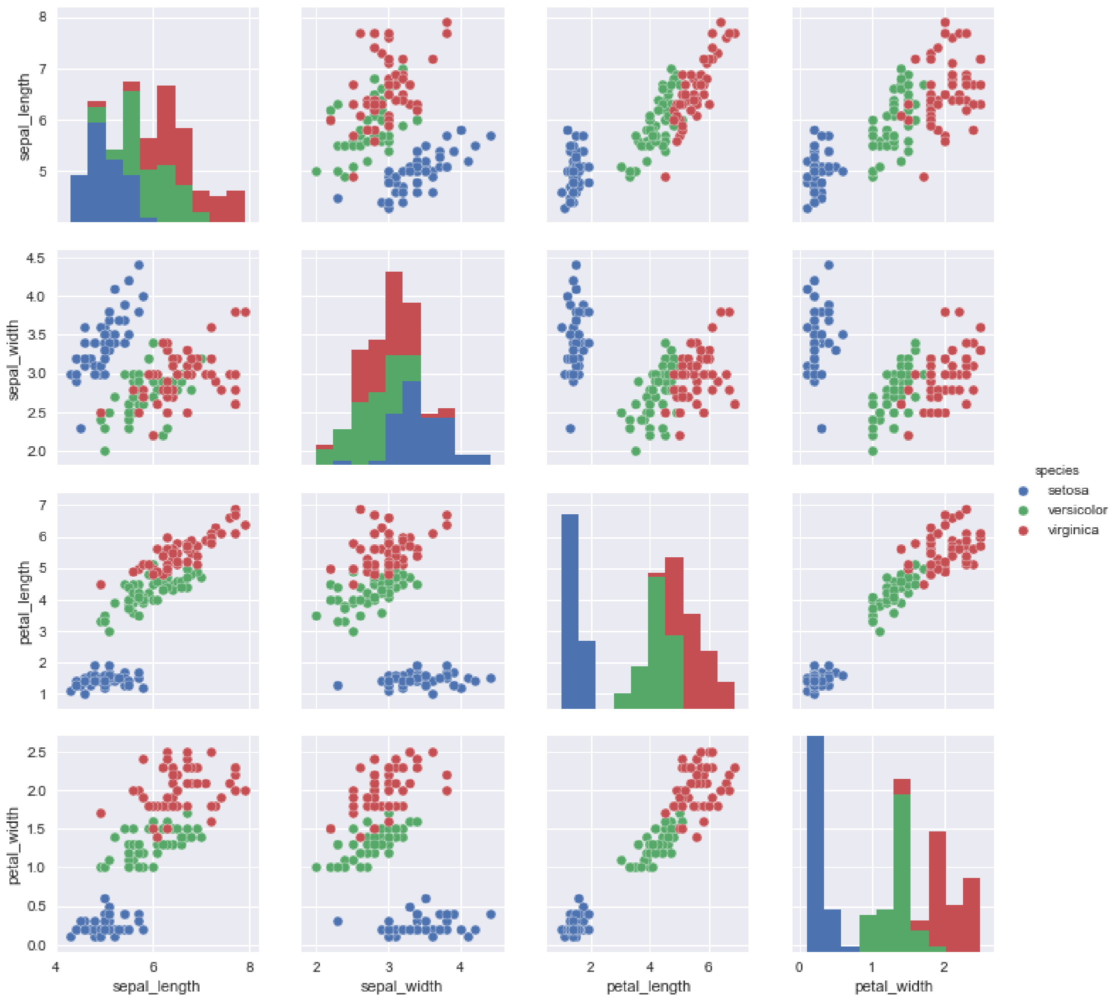


## 0.4 Pairwise Plotting

In general, we would always try to plot the pairs to observe all the interactions:

```
In [24]: sns.pairplot(iris,hue="species")
```

```
Out[24]: <seaborn.axisgrid.PairGrid at 0x110260be0>
```



## 0.5 Linear Regression

We now consider a new dataset. First, get the data from 442 diabetes patients and the progression of their disease from the `scikit-learn` library.

```
In [25]: from sklearn import datasets
         from sklearn.linear_model import LinearRegression
```

```
In [26]: diabetes = datasets.load_diabetes()
```

Let us view the description of the dataset.

```
In [27]: print(diabetes.DESCR)
```

```
Diabetes dataset
================
Notes
-----

Ten baseline variables, age, sex, body mass index, average blood
pressure, and six blood serum measurements were obtained for each of n =
442 diabetes patients, as well as the response of interest, a
quantitative measure of disease progression one year after baseline.

Data Set Characteristics:

  :Number of Instances: 442

  :Number of Attributes: First 10 columns are numeric predictive values

  :Target: Column 11 is a quantitative measure of disease progression one year after baseline

  :Attributes:
    :Age:
    :Sex:
    :Body mass index:
    :Average blood pressure:
    :S1:
    :S2:
    :S3:
    :S4:
    :S5:
    :S6:

Note: Each of these 10 feature variables have been mean centered and scaled by the standard dev

Source URL:
http://www4.stat.ncsu.edu/~boos/var.select/diabetes.html

For more information see:
Bradley Efron, Trevor Hastie, Iain Johnstone and Robert Tibshirani (2004) "Least Angle Regressi
(http://web.stanford.edu/~hastie/Papers/LARS/LeastAngle_2002.pdf)
```

The features given by the dataset are as follows:

```
In [28]: diabetes.feature_names
```

```
Out[28]: ['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']
```

We are aiming to predict the quantitative measure of disease progression one year after baseline (column 11), based on these features. We note that:

**Each of these 10 feature variables have been mean centered and scaled by the standard deviation times `n_samples` (i.e. the sum of squares of each column totals 1).**

### 0.5.1 Exploratory Data Analysis

Observing the first few rows of the dataset yields:

```
In [29]: npdata = diabetes.data
         features_df = pd.DataFrame(npdata,columns = diabetes.feature_names)
         target_df = pd.DataFrame(diabetes.target,columns = ['target'])
         diabetes_df = pd.concat([features_df,target_df],axis=1)
         pd.DataFrame.head(diabetes_df)
```

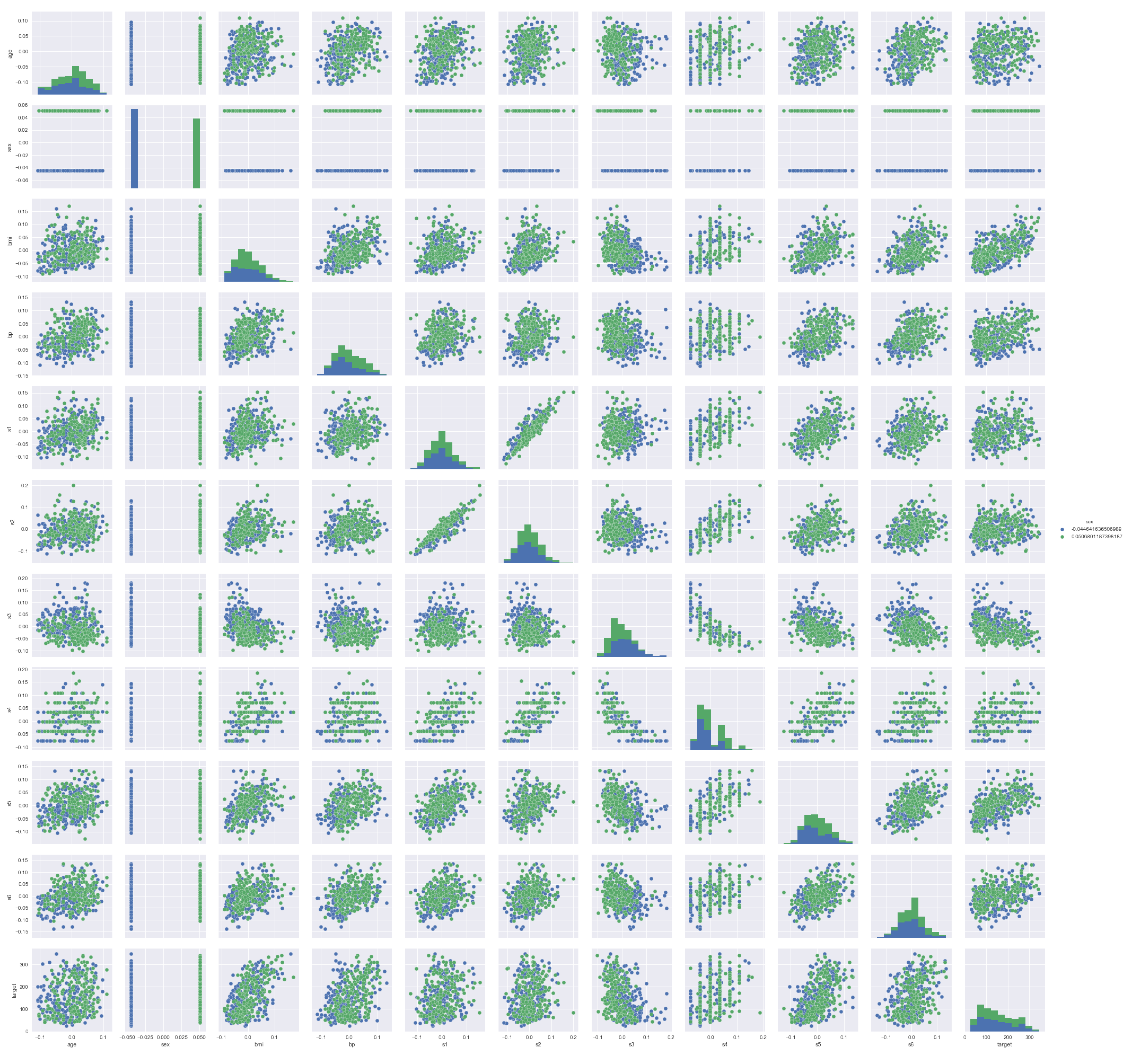
```
Out[29]:         age       sex       bmi        bp        s1        s2        s3  \
         0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401
         1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412
         2  0.085299  0.050680  0.044451 -0.005671 -0.045599 -0.034194 -0.032356
         3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
         4  0.005383 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142

                  s4        s5        s6  target
         0 -0.002592  0.019908 -0.017646   151.0
         1 -0.039493 -0.068330 -0.092204    75.0
         2 -0.002592  0.002864 -0.025930   141.0
         3  0.034309  0.022692 -0.009362   206.0
         4 -0.002592 -0.031991 -0.046641   135.0
```

Let's try viewing the pairwise plots for each feature pair:

```
In [30]: sns.pairplot(diabetes_df,hue="sex")
```

```
Out[30]: <seaborn.axisgrid.PairGrid at 0x1119a9780>
```



```
In [31]: diabetes_df.corr()

Out[31]:               age       sex       bmi        bp        s1        s2        s3  \
         age     1.000000  0.173737  0.185085  0.335427  0.260061  0.219243 -0.075181
         sex     0.173737  1.000000  0.088161  0.241013  0.035277  0.142637 -0.379090
         bmi     0.185085  0.088161  1.000000  0.395415  0.249777  0.261170 -0.366811
         bp      0.335427  0.241013  0.395415  1.000000  0.242470  0.185558 -0.178761
         s1      0.260061  0.035277  0.249777  0.242470  1.000000  0.896663  0.051519
         s2      0.219243  0.142637  0.261170  0.185558  0.896663  1.000000 -0.196455
         s3     -0.075181 -0.379090 -0.366811 -0.178761  0.051519 -0.196455  1.000000
         s4      0.203841  0.332115  0.413807  0.257653  0.542207  0.659817 -0.738493
         s5      0.270777  0.149918  0.446159  0.393478  0.515501  0.318353 -0.398577
         s6      0.301731  0.208133  0.388680  0.390429  0.325717  0.290600 -0.273697
         target  0.187889  0.043062  0.586450  0.441484  0.212022  0.174054 -0.394789
```

```
              s4        s5        s6    target
age     0.203841  0.270777  0.301731  0.187889
sex     0.332115  0.149918  0.208133  0.043062
bmi     0.413807  0.446159  0.388680  0.586450
bp      0.257653  0.393478  0.390429  0.441484
s1      0.542207  0.515501  0.325717  0.212022
s2      0.659817  0.318353  0.290600  0.174054
s3     -0.738493 -0.398577 -0.273697 -0.394789
s4      1.000000  0.617857  0.417212  0.430453
s5      0.617857  1.000000  0.464670  0.565883
s6      0.417212  0.464670  1.000000  0.382483
target  0.430453  0.565883  0.382483  1.000000
```

On first glance, we can see a good sense of positive correlatedness between the `target` and `bmi`, and a negative correlatedness between `target` and `s3`. Further observation of the `s1` and `s2` variables shows that we can drop one of the variables from the analysis as they are extremely correlated (corrcoef = 0.9).

Now let's try fitting a linear model to the data.

Let's try the following model:

$$Target = \beta_0 + \beta_1 bmi + \epsilon$$

```
In [32]: Y = np.array(diabetes_df['target'])
         X = np.array(diabetes_df['bmi'])

         X = X.reshape(-1,1)
         print(X.shape)
```

```
(442, 1)
```

Now, fit the model using the `LinearRegression` function from `sklearn`:

```
In [33]: model = LinearRegression()

In [34]: model.fit(X,Y)
```

```
/usr/local/lib/python3.6/site-packages/sklearn/linear_model/base.py:509: RuntimeWarning: inter
  linalg.lstsq(X, y)
```

```
Out[34]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Checking the coefficient of determination gives us:

```
In [35]: model.coef_

Out[35]: array([949.43526038])
```

The intercept term is estimated as:

```
In [38]: model.intercept_
```

```
Out[38]: 152.1334841628967
```

We can then use the model to predict the **expected value** of the target variable for a particular value of $bmi = 0.07$:

```
In [37]: model.predict(0.07)
```

```
Out[37]: array([218.59395239])
```