CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

scikit learn

Machine Learning

Machine Learning

Part 1 – Data Preparation

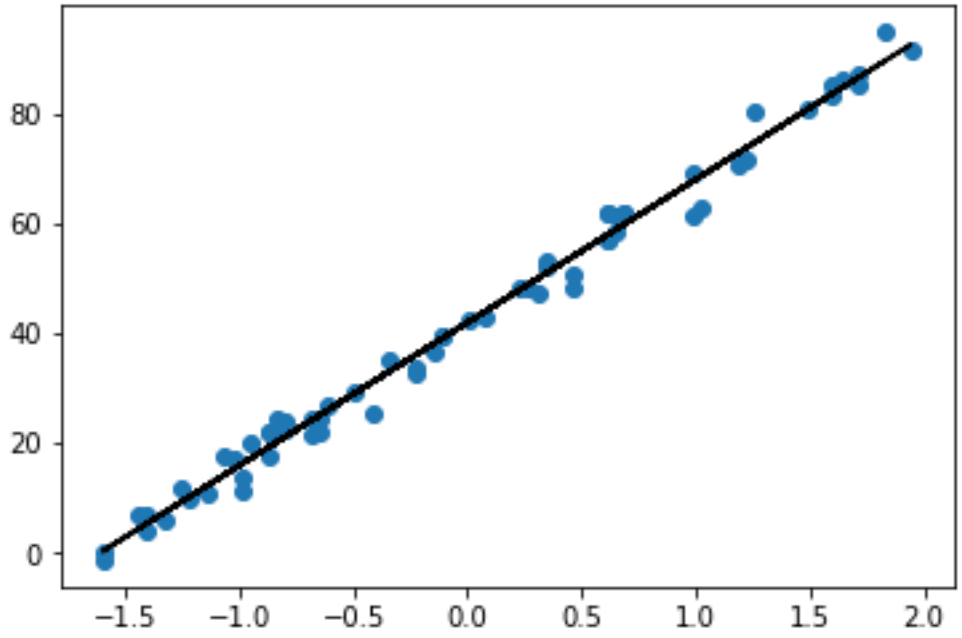
Ted Scully

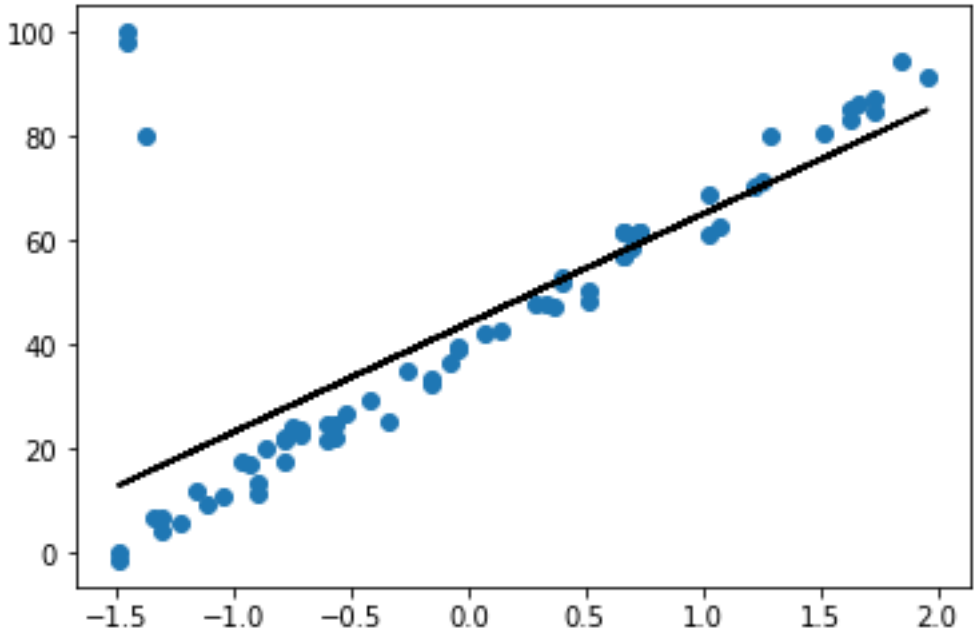
Data Pre-processing for Scikit Learn

- Dealing with Outliers (Optional)
- Dealing with Missing Values
- Handling Categorical Data
- Scaling Data
- Handling Imbalance
- Feature Selection
- Dimensionality Reduction

Outlier Detection

- Outliers are data that differ significantly from other data in a sample.
- Outliers **skew your data distributions** and impact your basic statistical measures and can be responsible for **underperformance** of **certain** algorithms.
- Outliers might be caused by faulty equipment, human error such as data entry, transcribing results, data processing error, etc
- Import considerations:
  - In some cases the outliers may be representative of a particular scenario. If you are removing the outliers then your model can no longer account for these instances. **Domain knowledge** is important in determining if this is the case.
  - If it can be determined that an outlier is in fact erroneous, then it should be **deleted** from the dataset (or alternatively **clamped** or **corrected** if possible). More on this later.


80
60
40
20
0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0


100
80
60
40
20
0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

Outlier Detection

- There are many different methods used for identifying and dealing with outliers.

- For example the following rules are often applied:
  - Values outside the range -1.5 x IQR to 1.5 x IQR removed.
  - Values outside the range of the 5th and 95th percentile can be considered as outlier.
  - Removal of data points that occur three or more standard deviations away from mean are considered outlier.

- However, most of the time visualizations are used as an aid in order to try to identify outliers.

Visualization for Outliers

- When looking for outliers a good place to start is to look at every single feature by itself using graphical inspection. This is often referred to as **univariate outlier detection**.

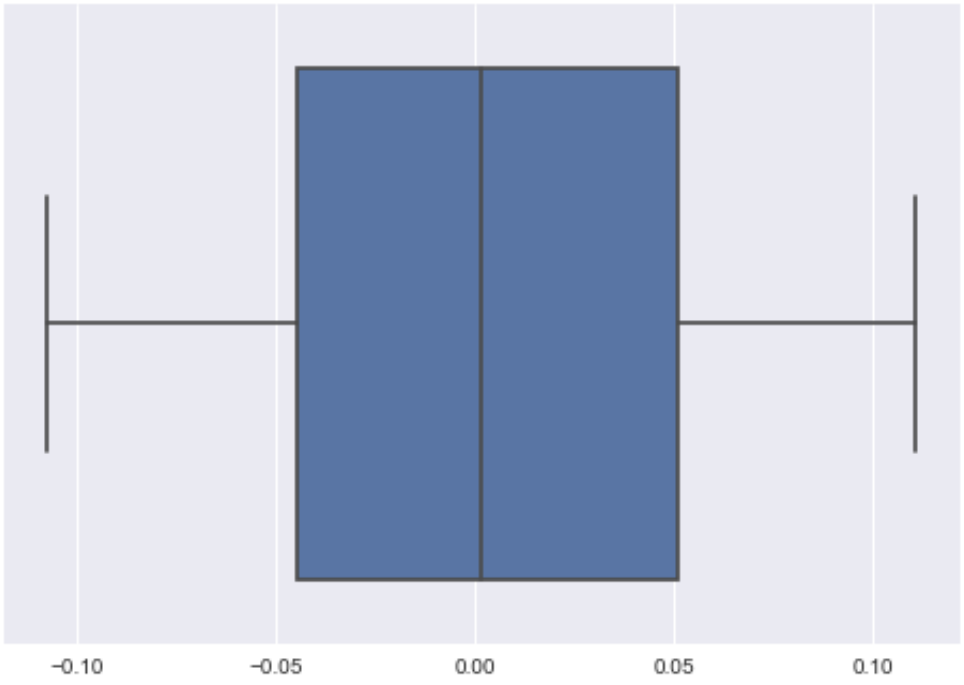
- We can easily create ***boxplots*** either through Seaborn or Pandas to help visualize outliers.

- A boxplot is a common visualization used for way of depicting the distribution of data feature based on it's **quartiles**.

- You can spot the problematic variables by looking at the extremities of the distribution.

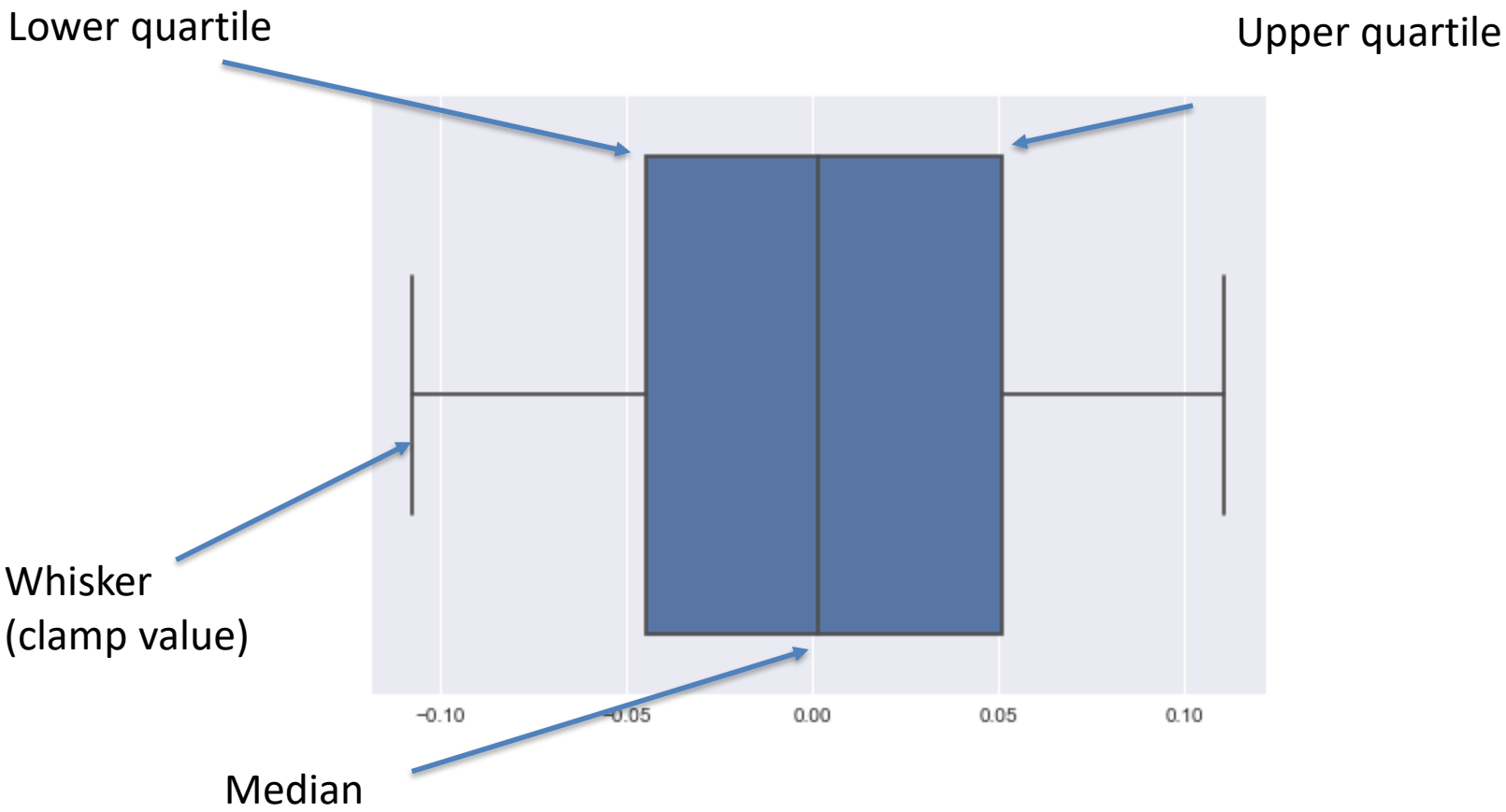
Visualization for Outliers

- A boxplot in Seaborn has what are called whiskers, which by default are set to **plus or minus 1.5 IQR** (Inter quartile range is the difference between the upper and low quartiles).
- Any data points outside these whiskers are possible outliers and are candidates for removal.

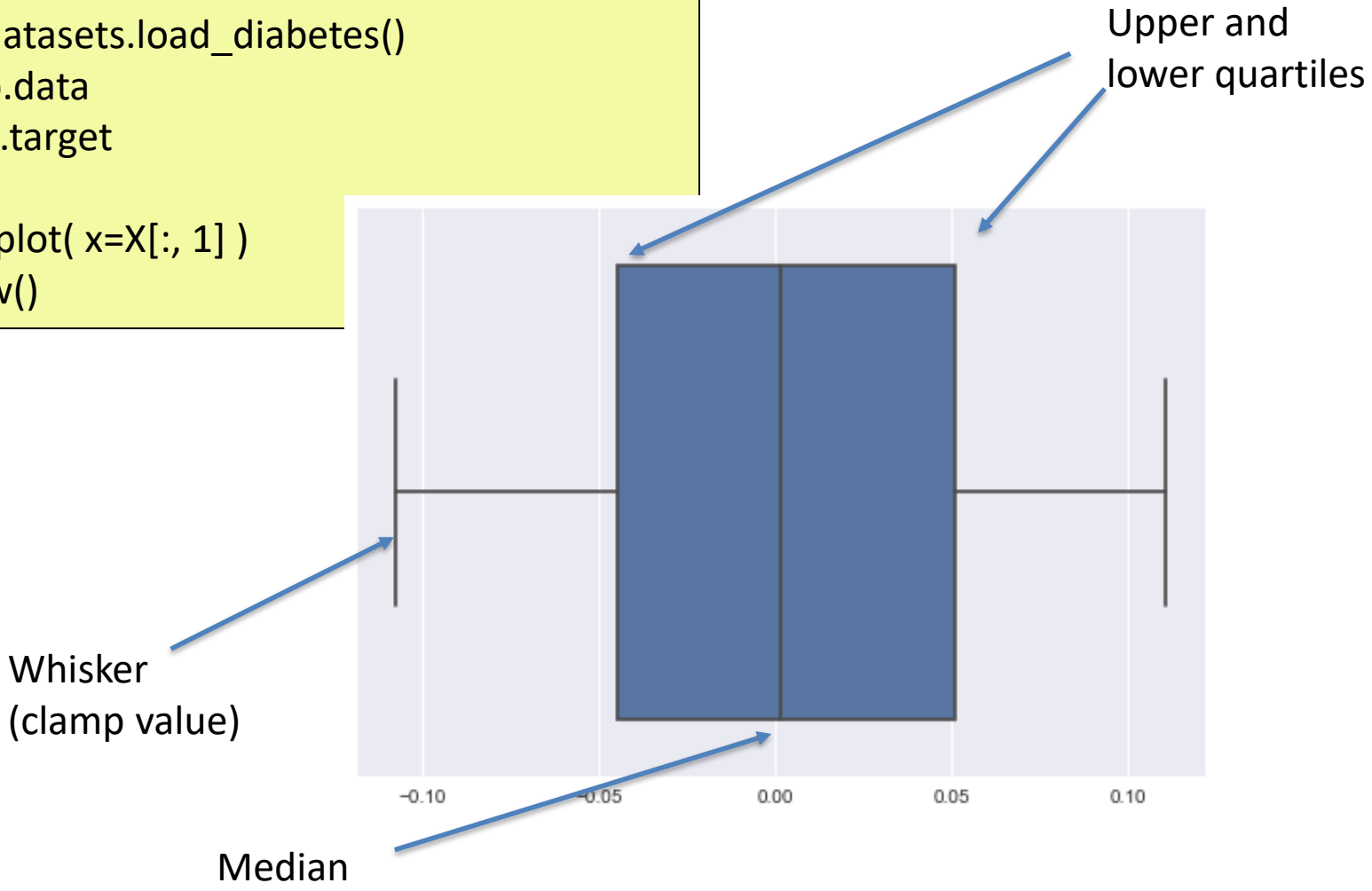


Visualization for Outliers

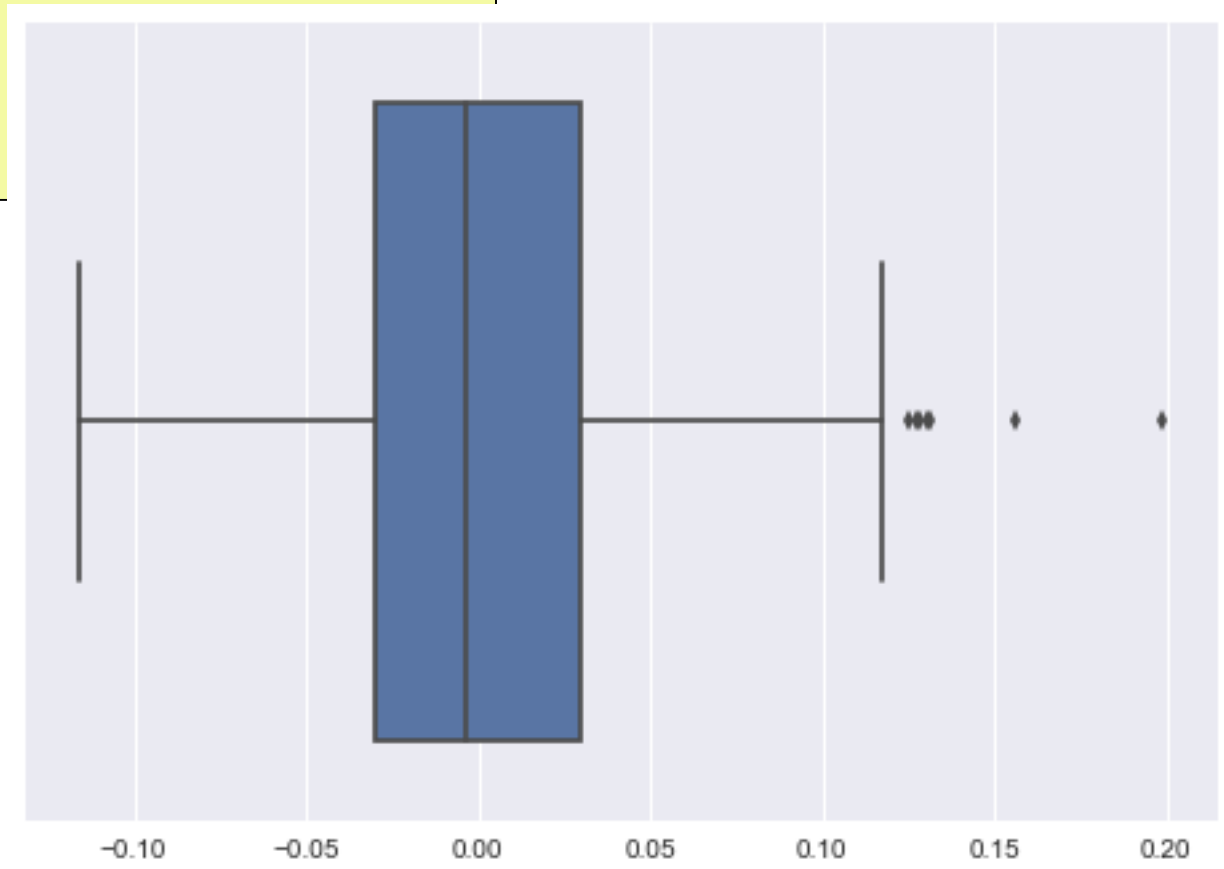
- A boxplot in Seaborn has what are called whiskers, which by default are set to **plus or minus 1.5 IQR** (Inter quartile range is the difference between the upper and low quartiles).
- Any data points outside these whiskers are possible outliers and are candidates for removal.



```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets

diab = datasets.load_diabetes()
X = diab.data
y = diab.target

sns.boxplot( x=X[:, 1] )
plt.show()
```



```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets

diab = datasets.load_diabetes()
X = diab.data
y = diab.target

sns.boxplot( x=X[:, 5] )
plt.show()
```



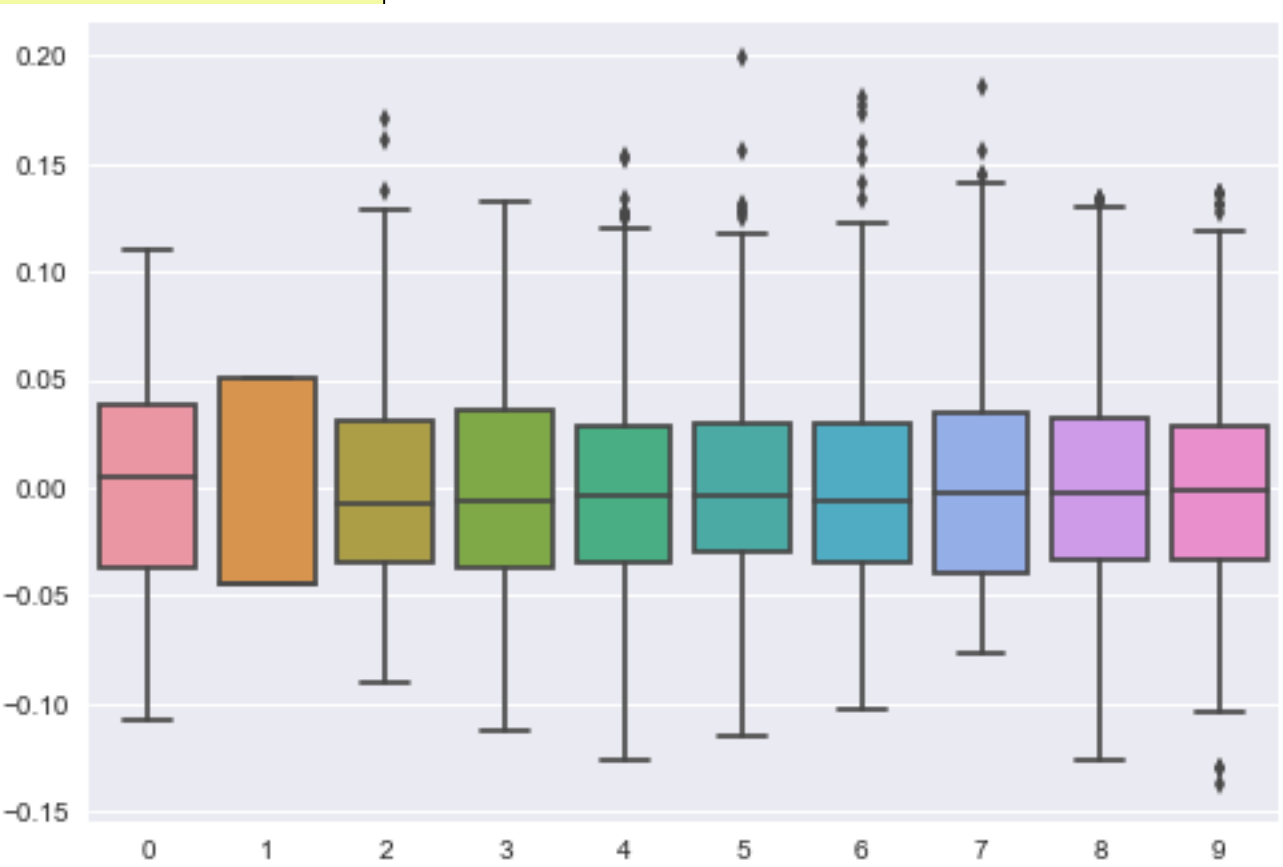
Visualization for Outliers

- You will have noticed in the previous two examples we are just plotting individual features.

- It is often useful to **plot all features together** to get a overview of entire dataset (this is really only useful if all features have been normalized or standardized in advance or appear within the same range).

- We can do this by passing in a dataframe containing all the features from our dataset.

```
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import datasets

diab = datasets.load_diabetes()
X = diab.data
y = diab.target

sns.boxplot(data= pd.DataFrame(X))
plt.show()
```



What to do with the outliers

- **Recommendation**: If data points appear just outside the whisker boundaries I recommend not treating them as outliers.

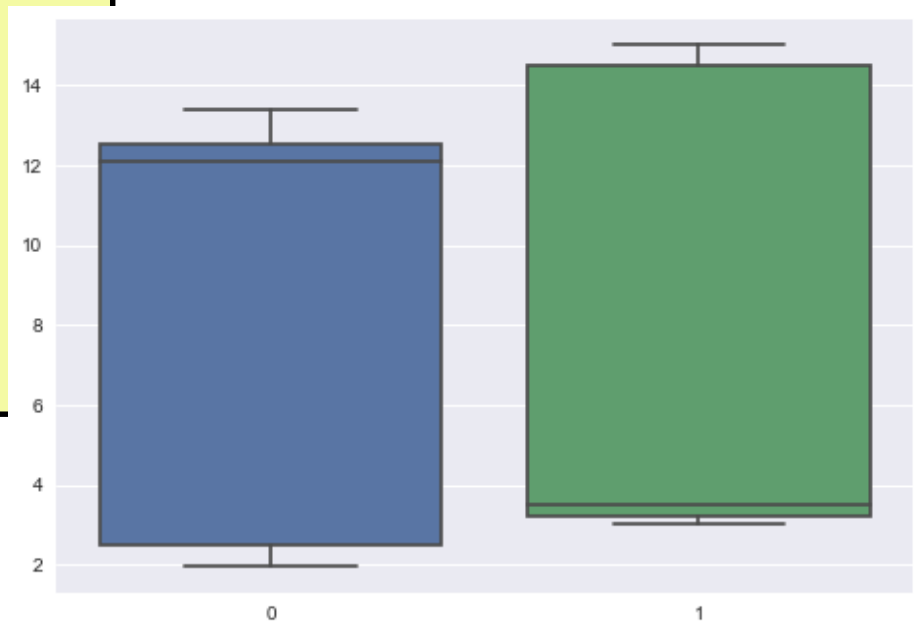
- There are a number of different methods for dealing with outliers.
  - Deletion of the associated instance from the dataset
  - Clamping outlier values. Resetting the outlier value to some upper or lower boundary value (for example, upper outlier values can be set to median + 1.5*IQR, lower outlier values will be set to median - 1.5*IQR, lower)
  - Set the outlier as a missing value and impute

Visualization for Outliers

- Consider the following dataset where we have two features.
- The boxplot doesn't reveal any outliers in this data.

```
import seaborn as sns
import matplotlib.pyplot as plt

data = np.array([ [2.5, 3.5] , [2.1, 3.3], [2.9, 3.1],
[1.95, 3.0], [12.5, 14.5], [13.1, 14.3], [13.4, 14.7],
[12.1, 15], [12.5, 3.2]])

print (data)

df = pd.DataFrame(data)

sns.boxplot(data= pd.DataFrame(data))
plt.show()
```

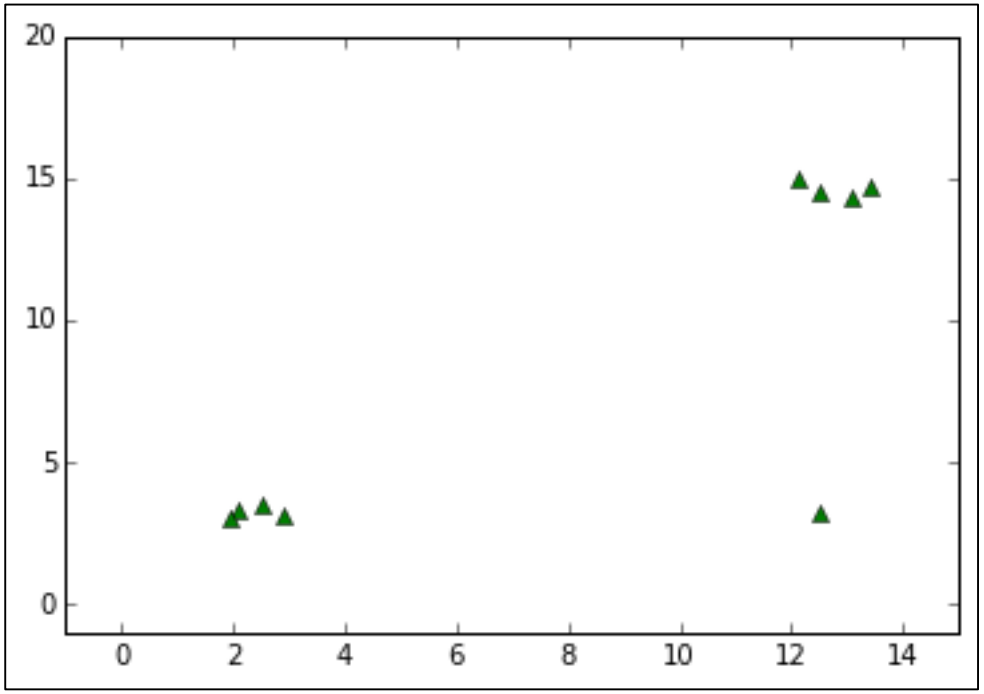
```
[[ 2.5   3.5 ]
 [ 2.1   3.3 ]
 [ 2.9   3.1 ]
 [ 1.95  3.  ]
 [ 12.5  3.2 ]
 [ 12.5  14.5 ]
 [ 13.1  14.3 ]
 [ 13.4  14.7 ]
 [ 12.1  15.  ]]
```



Visualization for Outliers

- However, notice if we plot one feature against another we can see that one data point is significantly removed from another data point.

- Univariate outlier detection is limited in this regard as you aren't considering unusual combinations of multiple variables



Multivariate Outlier Detection

- Mutli-variate outlier detection is not as commonly used as univariate.
- One clustering-based techniques that is used for outlier detection is called **DBScan**.
- DBScan relies on the idea that **clusters are dense**, so it starts **exploring** the data space in all directions and marks a cluster boundary when the density of points decreases.
- Areas of feature space with insufficient density of points are just considered to be **outliers** or noise.
- Note DBScan is a clustering technique that expects the data to be **standardized**.

Multivariate Outlier Detection

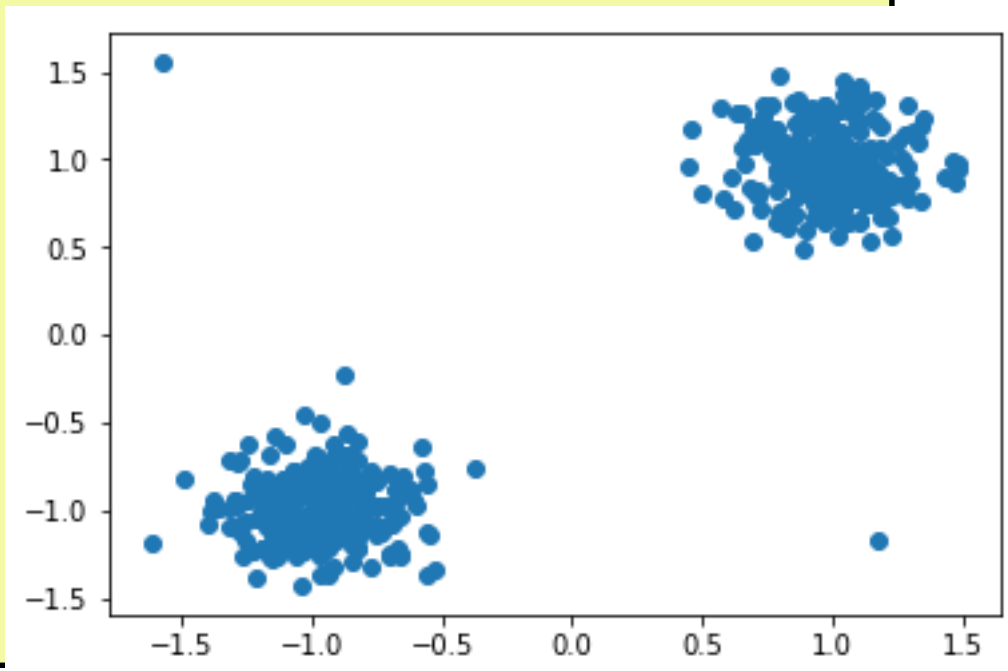
- The algorithm requires you to set two parameters:
  - ***eps***: This is the max distance between two observations that allows them to be part of the same neighbourhood or cluster.
  - ***min_sample***: The minimum number of observations in a neighbourhood that transforms them into a core point.

- DBSCAN classified data points into one of three categories:
  - Core point. A point *p* is a core point if at least a specified number (*min_sample*) of neighbouring points fall within the specified radius *eps*.
  - Border point: A border point is a data point that has fewer neighbours than *min_sample* within *eps* but lies with the *eps* radius of a core data point.
  - All other points that are neither core nor border points are considered as noise points or outliers.

Multivariate Outlier Detection

- The algorithm works by walking around the data and building clusters by linking observations arranged into neighbourhoods. A neighbourhood is a small cluster of data points all within a distance value of *eps*.

```
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import DBSCAN
from collections import Counter
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt


centers = [(-5, -5), (5, 5)]
X, y = make_blobs(n_samples=500, n_features=2, cluster_std=1.0,
                  centers=centers, shuffle=False, random_state=42)

X= np.vstack((X, [[-8, 8], [6, -6]]))

scaler = StandardScaler().fit(X)
X = scaler.transform(X)

plt.scatter(X[:,0],  X[:,1])


DB = DBSCAN(eps=0.5, min_samples = 3)
DB.fit(X)

print (Counter(DB.labels_), '\n')
```



```
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import DBSCAN
from collections import Counter
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt


centers = [(-5, -5), (5, 5)]
X, y = make_blobs(n_samples=500, n_features=2, cluster_std=1.0,
                  centers=centers, shuffle=False, random_state=42)

X= np.vstack((X, [[-8, 8], [6, -6]]))

scaler = StandardScaler().fit(X)
X = scaler.transform(X)

plt.scatter(X[:,0],  X[:,1])


DB = DBSCAN(eps=0.5, min_samples = 3)
DB.fit(X)

print (Counter(DB.labels_), '\n')
```

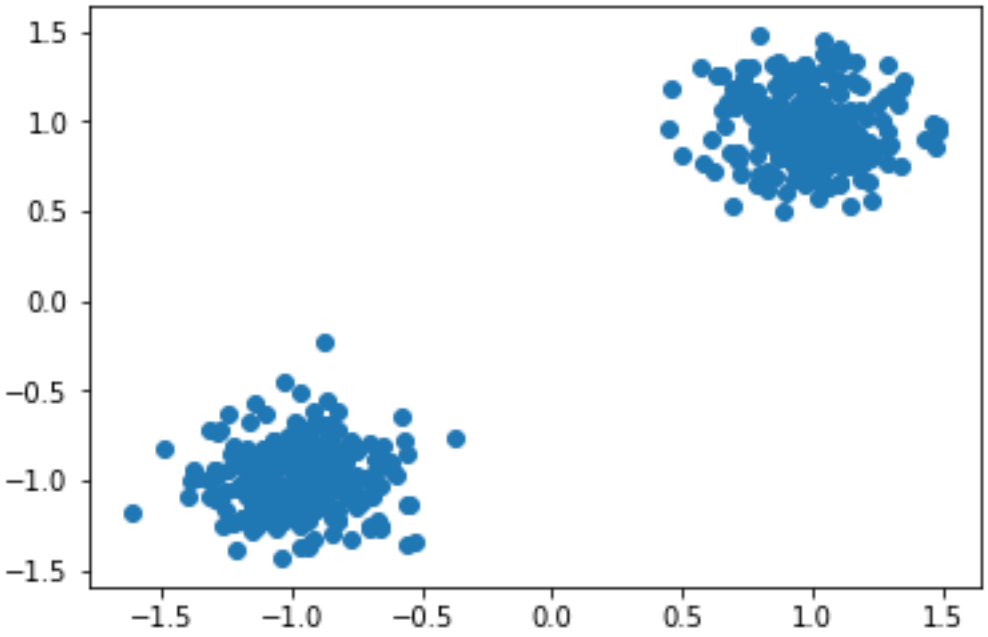
Values of eps and min_samples is something you should experiment with. Start small. Typically you would not expect 1 or 2% of data to be outliers

```
from sklearn.datasets.samples_generator import make_blobs
from sklearn.cluster import DBSCAN
from collections import Counter
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt


centers = [(-5, -5), (5, 5)]
X, y = make_blobs(n_samples=500, n_features=2, cluster_std=1.0,
                  centers=centers, shuffle=False, random_state=42)

X= np.vstack((X, [[-8, 8], [6, -6]]))

scaler = StandardScaler().fit(X)
X = scaler.transform(X)

plt.scatter(X[:,0],  X[:,1])


DB = DBSCAN(eps=0.5, min_samples = 3)
DB.fit(X)

print (Counter(DB.labels_), '\n')
```

When we call fit it runs DBScan and populates a NumPy array called DB.labels_ with the cluster label associated with each data point. When we use Counter we simply count the number of occurrences of each cluster in the results.
Counter({0: 250, 1: 250, -1: 2})

```
filteredData = X[DB.labels_ !=-1]

plt.scatter(filteredData[:,0],  filteredData[:,1])
plt.show()
```

We can then add the following code to remove the outliers from the original data. Notice we are using advanced indexing here to remove the outliers (those data points given the cluster label -1)



Data Pre-processing for Scikit Learn

- Dealing with Outliers (Optional)
- **Dealing with Missing Values**
- Handling Categorical Data
- Scaling Data
- Handling Imbalance
- Feature Selection
- Dimensionality Reduction

Dealing with Missing Values

- It is common in real-world applications that some features are missing one or more values for various reasons. There could be an **error in the data collection process**, certain fields may have been **left blank in a survey**, **power outage**, etc.

- Most ML algorithms are unable to process these missing values and as such it is important that we deal with all missing values within the data before sending it to the ML algorithm.

```
import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3), index=['a', 'b', 'c'])
seriesB = pd.Series(np.random.rand(4), index=['a', 'b', 'c', 'd'])
seriesC = pd.Series(np.random.rand(3), index=['b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                 'three' : seriesC})
print (df)
print (df.isnull().sum())
```

```
        one     three       two
a  0.376060       NaN  0.111221
b  0.400020  0.164076  0.548106
c  0.507972  0.325337  0.137571
d       NaN  0.823270  0.816618


one      1
three    1
two      0
dtype: int64
```

Notice the **.sum()** method will count the number of missing values in each column.

Dealing with Missing Values

- Please note that the call to **df.isnull().sum()** will only identify the missing values that are specified as **NaN**.

- Missing values can often be represented using values such as ‘?’, -1 ,-99,-999. Missing values such as these will not show up using isnull(). You should be able to identify these in your data exploration stage or when looking for outliers.

- A simple way of dealing with this is by **replacing the non-standard missing values with NaN**.

Dealing with Missing Values

```
import pandas as pd
import numpy as np

seriesA = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesB = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesC = pd.Series([8, 1, '?', 43], index=['a', 'b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                 'three' : seriesC})

print (df)
print (df.isnull().sum())
```

	one	three	two
a	12	8	12
b	4	1	4
c	3	**?**	3
d	**?**	43	**?**

```
one      0
three    0
two      0
dtype: int64
```

Dealing with Missing Values

```
import pandas as pd
import numpy as np

seriesA = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesB = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesC = pd.Series([8, 1, '?', 43], index=['a', 'b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                 'three' : seriesC})

df = df.replace('?', np.NaN)

print (df)
print (df.isnull().sum())
```

In this case we use replace to replace any occurrence of '?' within the dataframe with NaN.

```
 one    three    two
a  12      8       12
b   4      1        4
c   3     NaN       3
d   NaN    43      NaN

one     1
three   1
two     1
dtype: int64
```

Dealing with Missing Values

```
import pandas as pd
import numpy as np

seriesA = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesB = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesC = pd.Series([8, 1, '?', 43], index=['a', 'b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                 'three' : seriesC})

df['one'] = df['one'].replace('?',np.NaN)

print (df)
print (df.isnull().sum())
```

In this case we use replace to replace any occurrence of '?' with a NaN within **one column** of our dataframe (in this case column 'one').

```
  one  three  two
a  12      8   12
b   4      1    4
c   3      ?    3
d  NaN    43    ?

one      1
three    0
two      0
dtype: int64
```

Dealing with Missing Values - Removal

- One of the easiest ways to deal with missing values is to simply remove the corresponding features (columns) or rows from the dataset entirely.
  - **Rows**
    - **df.dropna()** will remove any rows that contain a missing value.
    - **df.dropna(thresh=3)** the parameter thresh specifies the number of non-NAN values that a row must have in order to be retained.
    - **df.dropna(subset=['A'])** only drop rows where missing values appear in a specific column in this case column A.
  - **Columns**
    - **df.dropna(axis = 1)** will drop **columns** that have at least one missing value
    - if you want to drop a column of a specific name you can call **df.drop(['ColumnName'], axis=1)**

- **Each off the above will return a new DataFrame!**

```
import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3), index=['a', 'b', 'c'])
seriesB = pd.Series(np.random.rand(4), index=['a', 'b', 'c', 'd'])
seriesC = pd.Series(np.random.rand(3), index=['b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                 'three' : seriesC})

print (df)
print
newdf = df.dropna()
print (newdf)
```

Here we drop any rows from our dataframe that contain a missing value.

```
        one     three       two
a  0.867059       NaN  0.255192
b  0.722719  0.420534  0.212348
c  0.328197  0.141678  0.237098
d       NaN  0.458063  0.503182

        one     three       two
b  0.722719  0.420534  0.212348
c  0.328197  0.141678  0.237098
```

```
import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(np.random.rand(5))
seriesD = pd.Series(np.random.rand(7))

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                   'three' : seriesC,
                   'four' : seriesD})
print (df)
df = df.dropna(thresh=4, axis=1 )
print (df)
```

	four	one	three	two
0	0.766476	0.909878	0.476737	0.084872
1	0.810370	0.285238	0.386073	0.268438
2	0.567595	0.162616	0.213230	0.389272
3	0.958997	NaN	0.579480	0.689228
4	0.141135	NaN	0.986758	NaN
5	0.612904	NaN	NaN	NaN
6	0.193091	NaN	NaN	NaN

	four	three	two
0	0.766476	0.476737	0.084872
1	0.810370	0.386073	0.268438
2	0.567595	0.213230	0.389272
3	0.958997	0.579480	0.689228
4	0.141135	0.986758	NaN
5	0.612904	NaN	NaN
6	0.193091	NaN	NaN

Notice above the threshold value is 4. Therefore, a column must have four or more non-NA values in order to be retained.

Dealing with Missing Values

- Although the removal of missing values may seem convenient it also comes with **significant disadvantages**.
- If you delete a row that has a missing value then you could potentially be deleting useful feature information.
- In many cases you may not have an abundance of data and we may end up removing many samples which could in turn **reduce the accuracy** of our model.
- An alternative (which is typically more preferable) to use **imputation** techniques to estimate the missing values from the other training samples in our dataset.

Dealing with Missing Values

- One of the most common imputation techniques is **mean imputation** where we replace a missing value with the mean of the data items in that column.

- Scikit learn provides an easy way of applying this method by using the **SimpleImputer** class.

- When creating a SimpleImputer object we can specify the **strategy** used for imputing missing values. The most typical is **strategy = mean**, however you can also impute by specifying **strategy = median, strategy = most_frequent** (mode) **or stategy= constant**.
- The most_frequent strategy replaces the missing value by the most frequent values. This is a strategy that is useful to use for **categorical** features.

- Another feature of the SimpleImputer is that we can specify the missing value using the parameter **missing_values**.

```
from sklearn.impute import SimpleImputer
# note we are using the dataframe we defined at the start of this section

print (df)

imputer = SimpleImputer(missing_values = np.NaN, strategy='mean')

imputer.fit(df)

allValues = imputer.transform(df)

print (allValues)
```

```
        one       three       two
a  0.030666       NaN      0.921680
b  0.351147  0.740355      0.478344
c  0.449259  0.299911      0.937952
d       NaN  0.897354      0.168368

[[ 0.03066623  0.64587346  0.92168033]
 [ 0.3511469   0.7403552   0.47834361]
 [ 0.4492592   0.29991101  0.93795242]
 [ 0.27702411  0.89735416  0.16836778]]
```

In this case we can impute for an entire dataset by passing in a pandas **dataframe** (we could also pass in a **NumPy** array).

The array we pass in must be a numerical array. The transform operation will always **returns a NumPy array**.

```
from sklearn.impute import SimpleImputer
# note we are using the dataframe we defined at the start of this section

imputer = SimpleImputer(missing_values = np.NaN, strategy='mean')

imputer.fit(df)

allValues = imputer.transform(df)

print (allValues)

df = pd.DataFrame(data = allValues, columns = df.columns)

print(df)
```

Notice that above we convert the NumPy array back into a Pandas dataframe using **pd.DataFrame**

```
[[ 0.13159694  0.57903764  0.39386208]
 [ 0.73002787  0.92954245  0.16350405]
 [ 0.67807006  0.58945908  0.60437333]
 [ 0.51323162  0.21811138  0.91419672]]

      one     three       two
0  0.131597  0.579038  0.393862
1  0.730028  0.929542  0.163504
2  0.678070  0.589459  0.604373
3  0.513232  0.218111  0.914197
```

```
from sklearn.impute import SimpleImputer
# note we are using the dataframe we defined at the start of this section

print (df)

imputer = SimpleImputer(missing_values = np.NaN, strategy='mean')

imputer.fit( df[['one']] )

df['one'] = imputer.transform(  df[['one']]  )

print (df)
```

In the example above we impute for just one specific column within our dataframe.

```
        one       three       two
a  0.068504         NaN  0.265894
b  0.323957    0.004214  0.151085
c  0.311528    0.646061  0.912238
d       NaN    0.642776  0.467234

        one     three       two
a  0.068504       NaN  0.265894
b  0.323957  0.004214  0.151085
c  0.311528  0.646061  0.912238
d  0.234663  0.642776  0.467234
```

Multivariate Feature Imputation

- A new (and very welcome) addition to Scikit Learn is a method of multi-variate feature imputation called IterativeImputer.
- It builds a separate model that that takes in a set of features from the dataset and uses those to predict the values for the feature with the missing values.

Note from Scikit Learn Website

This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import enable_iterative_imputer.

Multivariate Feature Imputation

```
import numpy as np
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

trainData = [[1, 2], [3, 6], [4, 8], [np.nan, 3], [7, np.nan]]

imp = IterativeImputer(random_state=0)
imp.fit(trainData)

print (imp.transform(trainData))
```

```
[[1, 2],
[3, 6],
[4, 8],
[nan, 3],
[7, nan]]
```

```
[[ 1.          2.        ]
 [ 3.          6.        ]
 [ 4.          8.        ]
 [ 1.50000296  3.        ]
 [ 7.         14.00004118]]
```

Missing Values - Advice

- Typically if there is **50%** or more missing values from a feature, then that feature would commonly be **removed** entirely.
- One alternative to deleting a feature that suffers from such a large number of missing values is to create a new **separate binary feature** that indicates if there is a missing value in the original column or not.
- This approach can be useful if the reason for the original missing value has some relationship to the target variable.
- For example, if the missing feature may contain some sensitive personal information, a persons willingness to provide this data may tell us something about that person that might be related to the final class.
- Typically the binary feature replaces the original feature.

Missing Values - Advice

- As we mentioned, **deleting instances (rows)** that contains one or more missing values can also result in a very large amount of data being lost. Unless you have an abundance of data and a relatively small amount of number of missing values I would not advice employing brute force deletion of rows.

- Imputation is a useful technique that we can use but it is not advisable when features have a very large number of missing values.

- A common rule is that **imputation** is not advisable on a feature that has **30% or more missing values** and should never be used on a feature that has **50%** or more missing values.