Machine Learning is making the computer learn from studying data and statistics.

Machine Learning is a step into the direction of artificial intelligence (AI).

Machine Learning is a program that analyses data and learns to predict the outcome.

Where To Start?

In this tutorial we will go back to mathematics and study statistics, and how to calculate important numbers based on data sets.

We will also learn how to use various Python modules to get the answers we need.

And we will learn how to make functions that are able to predict the outcome based on what we have learned.

Data Set

In the mind of a computer, a data set is any collection of data. It can be anything from an array to a complete database.

Example of an array:

[99,86,87,88,111,86,103,87,94,78,77,85,86] Example of a database:

Carname	Color	Age	Speed	A
BMW	red	5	99	Y
Volvo	black	7	86	Y
VW	gray	8	87	N
VW	white	7	88	Y
Ford	white	2	111	Y
VW	white	17	86	Y

Tesla	red	2	103	Y
BMW	black	9	87	Y
Volvo	gray	4	94	N
Ford	white	11	78	N
Toyota	gray	12	77	N
VW	white	9	85	N
Toyota	blue	6	86	Y

By looking at the array, we can guess that the average value is probably around 80 or 90, and we are also able to determine the highest value and the lowest value, but what else can we do?

And by looking at the database we can see that the most popular color is white, and the oldest car is 17 years, but what if we could predict if a car had an AutoPass, just by looking at the other values?

That is what Machine Learning is for! Analyzing data and predicting the outcome!

In Machine Learning it is common to work with very large data sets. In this tutorial we will try to make it as easy as possible to understand the different concepts of machine learning, and we will work with small easy-to-understand data sets

Data Types

To analyze data, it is important to know what type of data we are dealing with.

We can split the data types into three main categories:

- **Numerical**
- **Categorical**
- **Ordinal**

Numerical data are numbers, and can be split into two numerical categories:

- Discrete Data
  - counted data that are limited to integers. Example: The number of cars passing by.
- Continuous Data
  - measured data that can be any number. Example: The price of an item, or the size of an item

Categorical data are values that cannot be measured up against each other. Example: a color value, or any yes/no values.

Ordinal data are like categorical data, but can be measured up against each other. Example: school grades where A is better than B and so on.

By knowing the data type of your data source, you will be able to know what technique to use when analyzing them.

You will learn more about statistics and analyzing data in the next chapters.

Mean, Median, and Mode

What can we learn from looking at a group of numbers?

In Machine Learning (and in mathematics) there are often three values that interests us:

- **Mean** - The average value
- **Median** - The mid point value
- **Mode** - The most common value

Example: We have registered the speed of 13 cars:

speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]

What is the average, the middle, or the most common speed value?

Mean

The mean value is the average value.

To calculate the mean, find the sum of all values, and divide the sum by the number of values:

(99+86+87+88+111+86+103+87+94+78+77+85+86) / 13 = 89.77

The NumPy module has a method for this. Learn about the NumPy module in our NumPy Tutorial.

Use the NumPy `mean()` method to find the average speed:

```
import numpy

speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]

x = numpy.mean(speed)

print(x)
```

Median

The median value is the value in the middle, after you have sorted all the values:

```
77, 78, 85, 86, 86, 86, 87, 87, 88, 94, 99, 103, 111
```

It is important that the numbers are sorted before you can find the median.

The NumPy module has a method for this:

Example

Use the NumPy `median()` method to find the middle value:

```
import numpy

speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]

x = numpy.median(speed)

print(x)
```

If there are two numbers in the middle, divide the sum of those numbers by two.

```
77, 78, 85, 86, 86, 86, 87, 87, 94, 98, 99, 103

(86 + 87) / 2 = 86.5
```

Example

Using the NumPy module:

```
import numpy

speed = [99,86,87,88,86,103,87,94,78,77,85,86]

x = numpy.median(speed)

print(x)
```

Mode

The Mode value is the value that appears the most number of times:

```
99, 86, 87, 88, 111, 86, 103, 87, 94, 78, 77, 85, 86 = 86
```

The SciPy module has a method for this. Learn about the SciPy module in our SciPy Tutorial.

Example

Use the SciPy `mode()` method to find the number that appears the most:

```
from scipy import stats

speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]

x = stats.mode(speed)

print(x)
```

What is Standard Deviation?

Standard deviation is a number that describes how spread out the values are.

A low standard deviation means that most of the numbers are close to the mean (average) value.

A high standard deviation means that the values are spread out over a wider range.

Example: This time we have registered the speed of 7 cars:

```
speed = [86,87,88,86,87,85,86]
```

The standard deviation is:

```
0.9
```

Meaning that most of the values are within the range of 0.9 from the mean value, which is 86.4.

Let us do the same with a selection of numbers with a wider range:

```
speed = [32,111,138,28,59,77,97]
```

The standard deviation is:

```
37.85
```

Meaning that most of the values are within the range of 37.85 from the mean value, which is 77.4.

As you can see, a higher standard deviation indicates that the values are spread out over a wider range.

TUse the NumPy `std()` method to find the standard deviation:

```
import numpy

speed = [86,87,88,86,87,85,86]

x = numpy.std(speed)

print(x)
```

he NumPy module has a method to calculate the standard deviation:

```
import numpy

speed = [32,111,138,28,59,77,97]

x = numpy.std(speed)

print(x)
```

What are Percentiles?

Percentiles are used in statistics to give you a number that describes the value that a given percent of the values are lower than.

Example: Let's say we have an array that contains the ages of every person living on a street.

```
ages =
[5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]
```

What is the 75. percentile? The answer is 43, meaning that 75% of the people are 43 or younger.

The NumPy module has a method for finding the specified percentile:

Use the NumPy `percentile()` method to find the percentiles:

```
import numpy

ages = [5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]

x = numpy.percentile(ages, 75)

print(x)
```

Example

What is the age that 90% of the people are younger than?

```
import numpy

ages = [5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]

x = numpy.percentile(ages, 90)
```

```
print(x)
```

Data Distribution

Earlier in this tutorial we have worked with very small amounts of data in our examples, just to understand the different concepts.

In the real world, the data sets are much bigger, but it can be difficult to gather real world data, at least at an early stage of a project.

How Can we Get Big Data Sets?

To create big data sets for testing, we use the Python module NumPy, which comes with a number of methods to create random data sets, of any size.

Create an array containing 250 random floats between 0 and 5:

```
import numpy

x = numpy.random.uniform(0.0, 5.0, 250)

print(x)
```

Histogram

To visualize the data set we can draw a histogram with the data we collected.

We will use the Python module Matplotlib to draw a histogram.

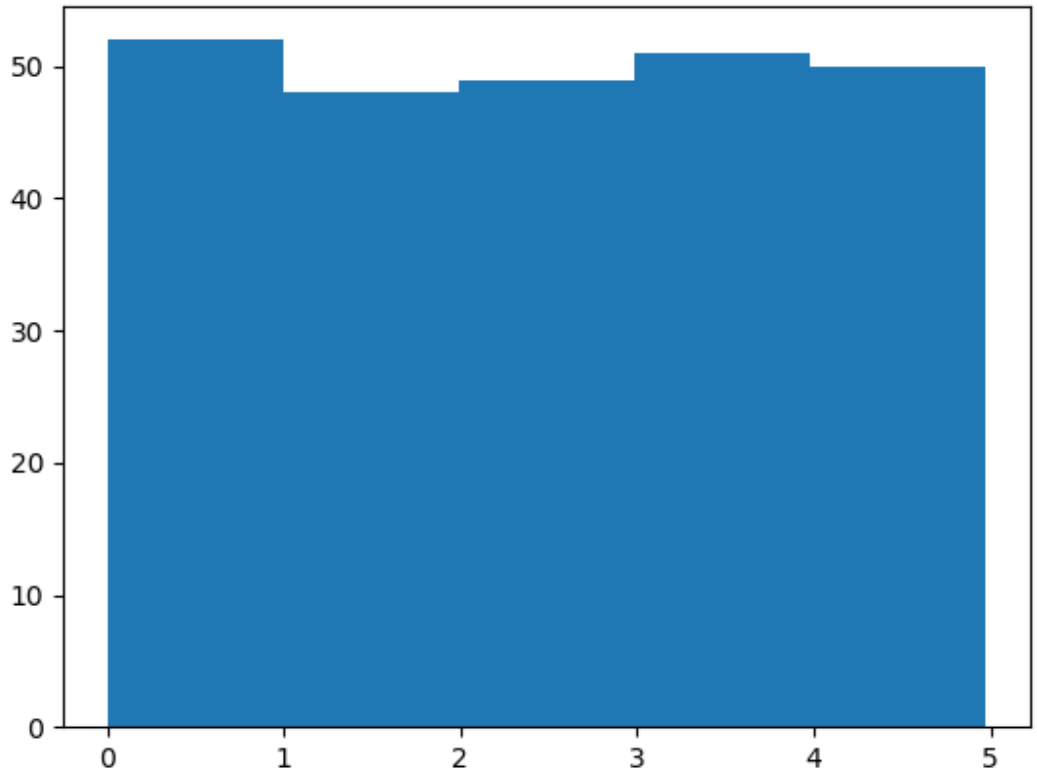
Learn about the Matplotlib module in our Matplotlib Tutorial.

Example

Draw a histogram:

```
import numpy
import matplotlib.pyplot as plt

x = numpy.random.uniform(0.0, 5.0, 250)
```

```
plt.hist(x, 5)
plt.show()
```



Histogram Explained

We use the array from the example above to draw a histogram with 5 bars.

The first bar represents how many values in the array are between 0 and 1.

The second bar represents how many values are between 1 and 2.

Etc.

Which gives us this result:

- 52 values are between 0 and 1
- 48 values are between 1 and 2
- 49 values are between 2 and 3

- 51 values are between 3 and 4
- 50 values are between 4 and 5

Note: The array values are random numbers and will not show the exact same result on your computer.

Big Data Distributions

An array containing 250 values is not considered very big, but now you know how to create a random set of values, and by changing the parameters, you can create the data set as big as you want.

Example

Create an array with 100000 random numbers, and display them using a histogram with 100 bars:

```
import numpy
import matplotlib.pyplot as plt

x = numpy.random.uniform(0.0, 5.0, 100000)

plt.hist(x, 100)
plt.show()
```

Machine Learning - Normal Data Distribution

Normal Data Distribution

In the previous chapter we learned how to create a completely random array, of a given size, and between two given values.

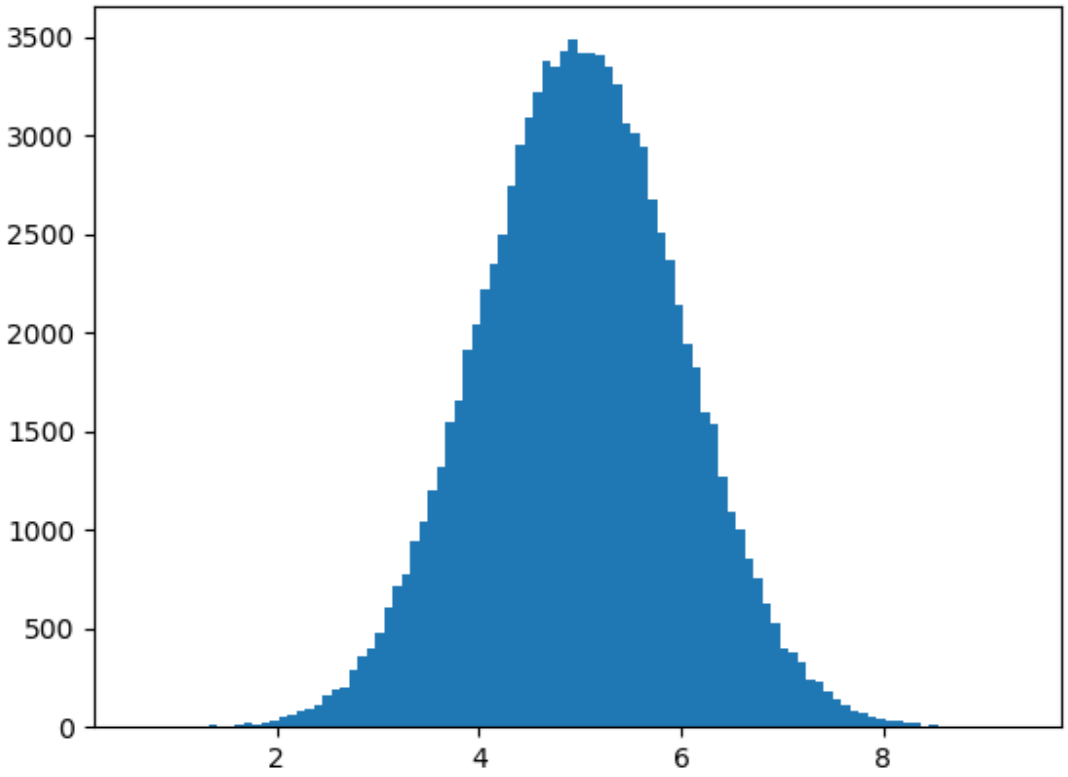
In this chapter we will learn how to create an array where the values are concentrated around a given value.

In probability theory this kind of data distribution is known as the *normal data distribution*, or the *Gaussian data distribution*, after the mathematician Carl Friedrich Gauss who came up with the formula of this data distribution.

Example

A typical normal data distribution:

```
import numpy
import matplotlib.pyplot as plt

x = numpy.random.normal(5.0, 1.0, 100000)

plt.hist(x, 100)
plt.show()
```



Note: A normal distribution graph is also known as the *bell curve* because of it's characteristic shape of a bell.

Histogram Explained

We use the array from the `numpy.random.normal()` method, with 100000 values, to draw a histogram with 100 bars.

We specify that the mean value is 5.0, and the standard deviation is 1.0.

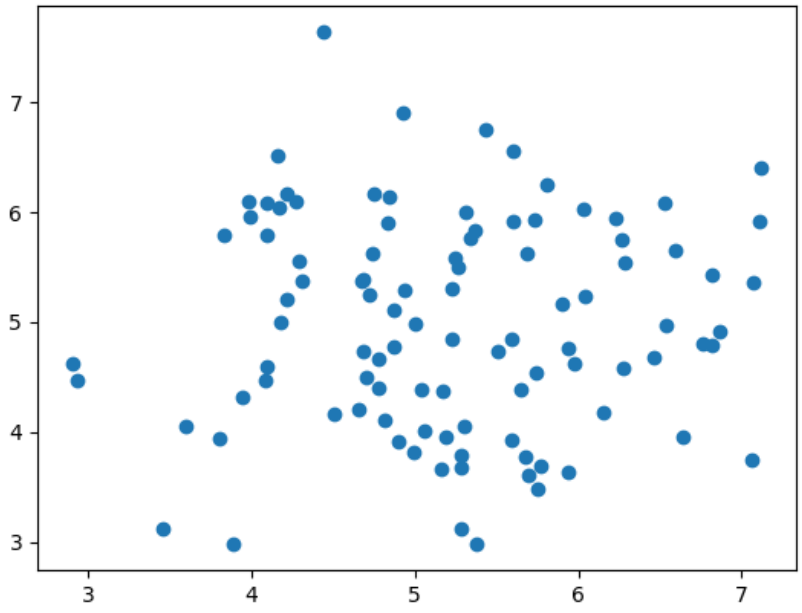
Meaning that the values should be concentrated around 5.0, and rarely further away than 1.0 from the mean.

And as you can see from the histogram, most values are between 4.0 and 6.0, with a top at approximately 5.0.

Machine Learning - Scatter Plot

Scatter Plot

A scatter plot is a diagram where each value in the data set is represented by a dot.



The Matplotlib module has a method for drawing scatter plots, it needs two arrays of the same length, one for the values of the x-axis, and one for the values of the y-axis:

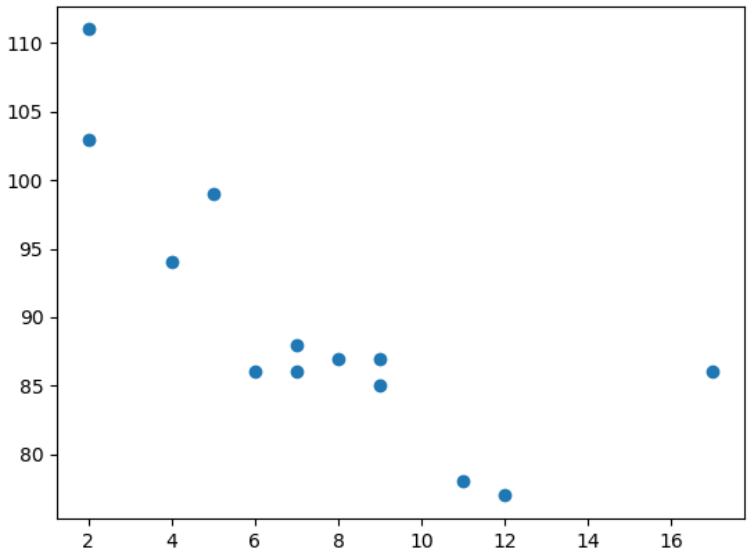
```
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]

y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
```

The x array represents the age of each car.

The y array represents the speed of each car.

Use the `scatter()` method to draw a scatter plot diagram:

```
import matplotlib.pyplot as plt

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

plt.scatter(x, y)
plt.show()
```



Scatter Plot Explained

The x-axis represents ages, and the y-axis represents speeds.

What we can read from the diagram is that the two fastest cars were both 2 years old, and the slowest car was 12 years old.

Note: It seems that the newer the car, the faster it drives, but that could be a coincidence, after all we only registered 13 cars.

Random Data Distributions

In Machine Learning the data sets can contain thousands-, or even millions, of values.

You might not have real world data when you are testing an algorithm, you might have to use randomly generated values.

As we have learned in the previous chapter, the NumPy module can help us with that!

Let us create two arrays that are both filled with 1000 random numbers from a normal data distribution.

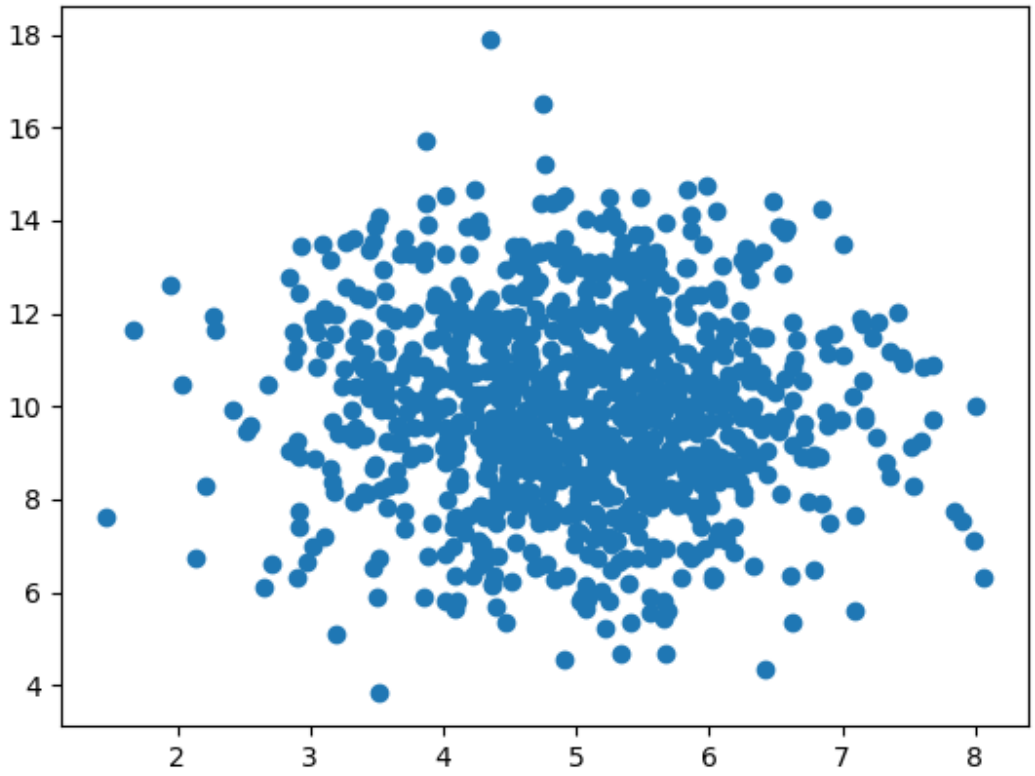
The first array will have the mean set to 5.0 with a standard deviation of 1.0.

The second array will have the mean set to 10.0 with a standard deviation of 2.0:

Example

A scatter plot with 1000 dots:

```
import numpy
import matplotlib.pyplot as plt

x = numpy.random.normal(5.0, 1.0, 1000)
y = numpy.random.normal(10.0, 2.0, 1000)

plt.scatter(x, y)
plt.show()
```



Scatter Plot Explained

We can see that the dots are concentrated around the value 5 on the x-axis, and 10 on the y-axis.

We can also see that the spread is wider on the y-axis than on the x-axis.

Machine Learning - Linear Regression

Regression

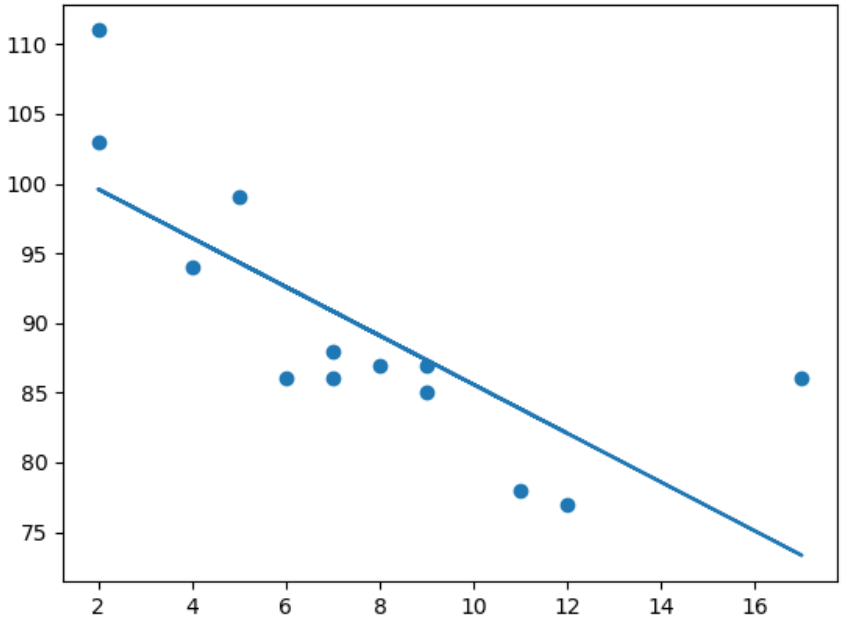
The term regression is used when you try to find the relationship between variables.

In Machine Learning, and in statistical modeling, that relationship is used to predict the outcome of future events.

Linear Regression

Linear regression uses the relationship between the data-points to draw a straight line through all them.

This line can be used to predict future values.



In Machine Learning, predicting the future is very important.

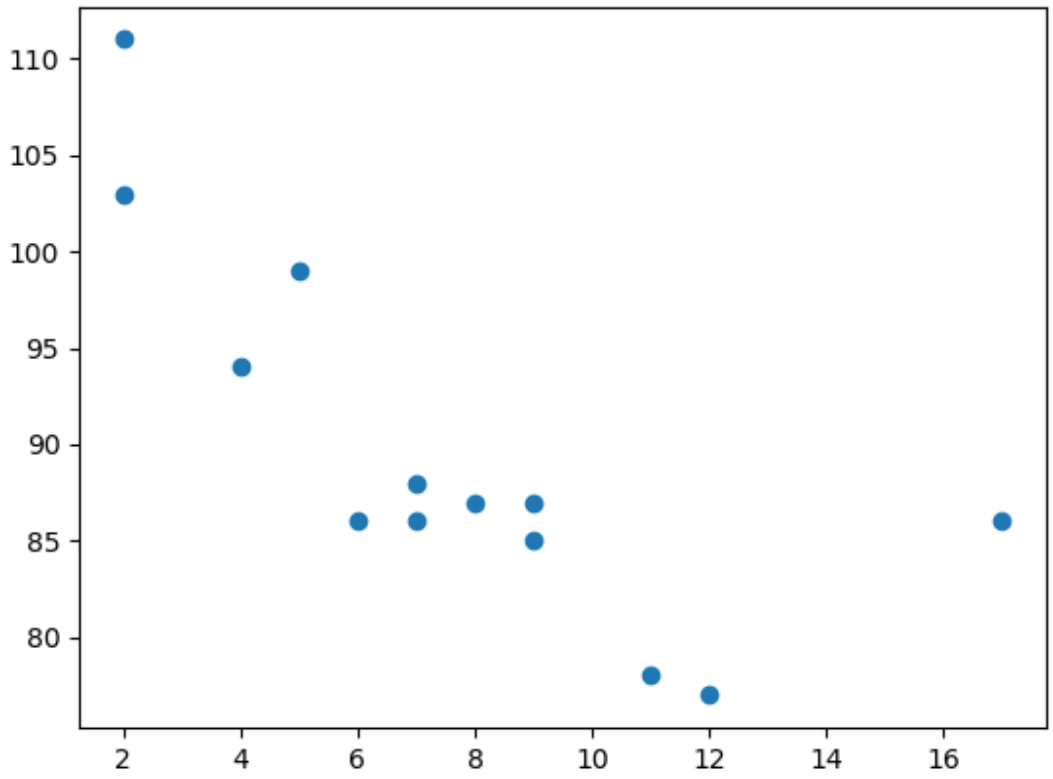
How Does it Work?

Python has methods for finding a relationship between data-points and to draw a line of linear regression. We will show you how to use these methods instead of going through the mathematic formula.

In the example below, the x-axis represents age, and the y-axis represents speed. We have registered the age and speed of 13 cars as they were passing a tollbooth. Let us see if the data we collected could be used in a linear regression

Start by drawing a scatter plot:

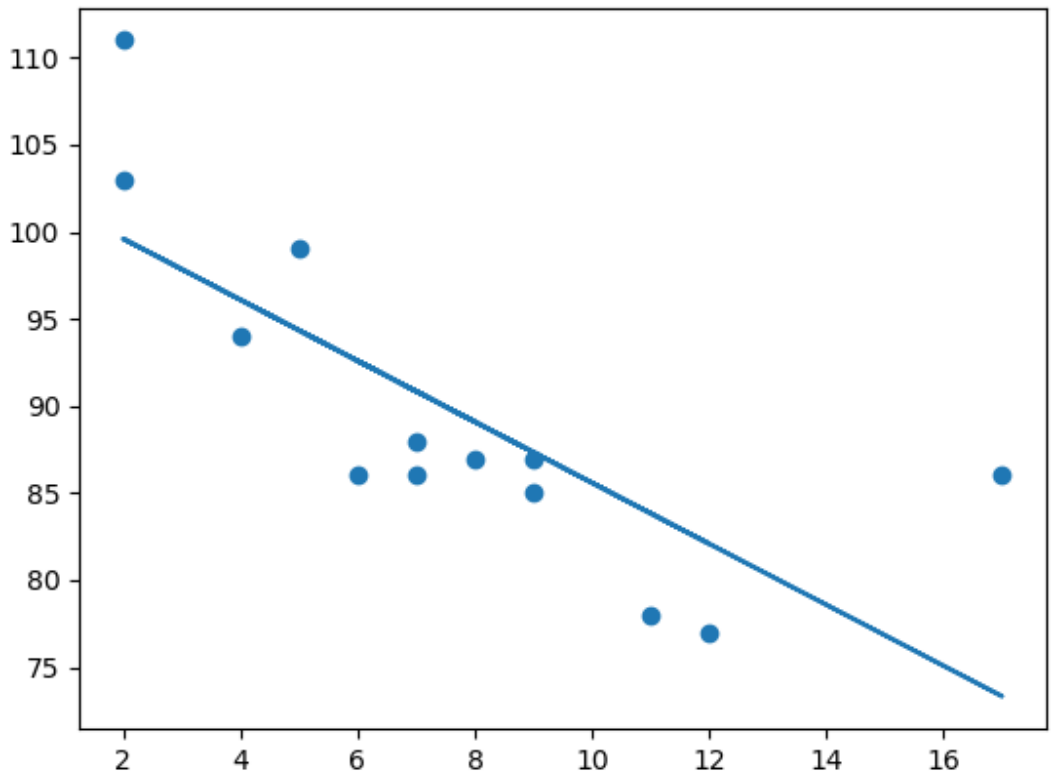
```
import matplotlib.pyplot as plt

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

plt.scatter(x, y)
plt.show()
```



Example

Import `scipy` and draw the line of Linear Regression:

```
import matplotlib.pyplot as plt
from scipy import stats

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
  return slope * x + intercept

mymodel = list(map(myfunc, x))

plt.scatter(x, y)
plt.plot(x, mymodel)
plt.show()
```



Example Explained

Import the modules you need.

You can learn about the Matplotlib module in our Matplotlib Tutorial.

You can learn about the SciPy module in our SciPy Tutorial.

```
import matplotlib.pyplot as plt
from scipy import stats
```

Create the arrays that represent the values of the x and y axis:

```
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
```

Execute a method that returns some important key values of Linear Regression:

```
slope, intercept, r, p, std_err = stats.linregress(x, y)
```

Create a function that uses the `slope` and `intercept` values to return a new value. This new value represents where on the y-axis the corresponding x value will be placed:

```
def myfunc(x):
  return slope * x + intercept
```

Run each value of the x array through the function. This will result in a new array with new values for the y-axis:

```
mymodel = list(map(myfunc, x))
```

Draw the original scatter plot:

```
plt.scatter(x, y)
```

Draw the line of linear regression:

```
plt.plot(x, mymodel)
```

Display the diagram:

```
plt.show()
```

R for Relationship

It is important to know how the relationship between the values of the x-axis and the values of the y-axis is, if there are no relationship the linear regression can not be used to predict anything.

This relationship - the coefficient of correlation - is called `r`.

The `r` value ranges from -1 to 1, where 0 means no relationship, and 1 (and -1) means 100% related.

Python and the Scipy module will compute this value for you, all you have to do is feed it with the x and y values.

Example

How well does my data fit in a linear regression?

```
from scipy import stats

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

slope, intercept, r, p, std_err = stats.linregress(x, y)

print(r)
```

Note: The result -0.76 shows that there is a relationship, not perfect, but it indicates that we could use linear regression in future predictions.

Predict Future Values

Now we can use the information we have gathered to predict future values.

Example: Let us try to predict the speed of a 10 years old car.

To do so, we need the same `myfunc()` function from the example above:

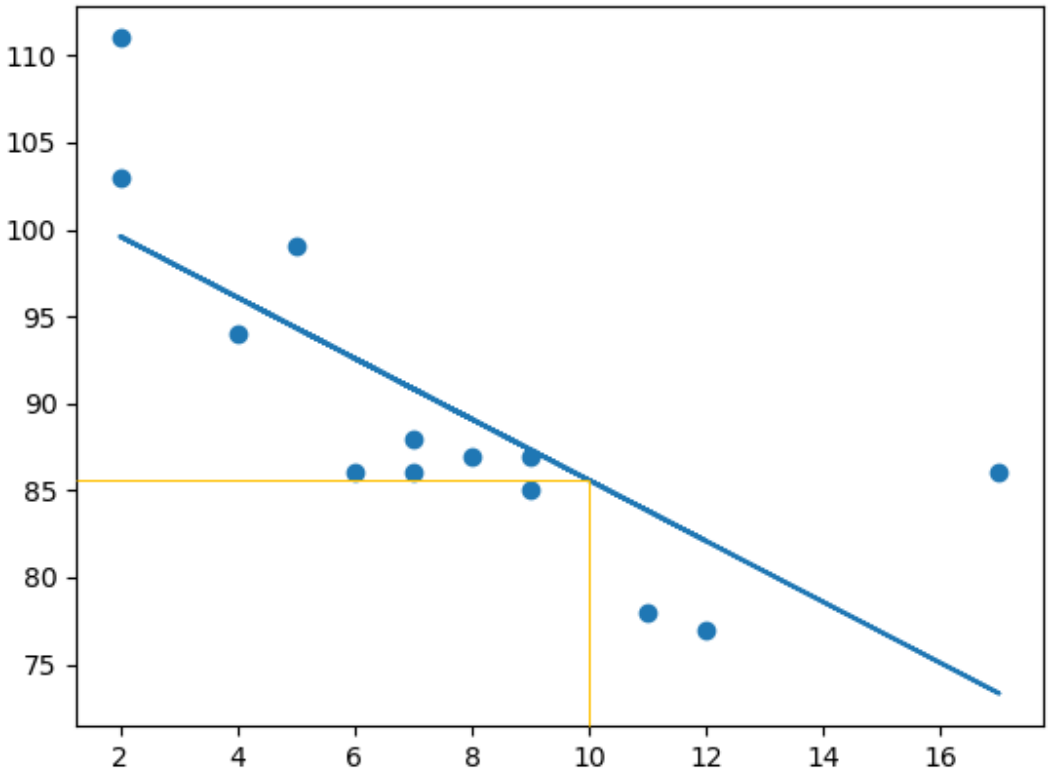
```
def myfunc(x):
  return slope * x + intercept
```

Example

Predict the speed of a 10 years old car:

```
from scipy import stats

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
  return slope * x + intercept

speed = myfunc(10)

print(speed)
```

The example predicted a speed at 85.6, which we also could read from the diagram:



Bad Fit?

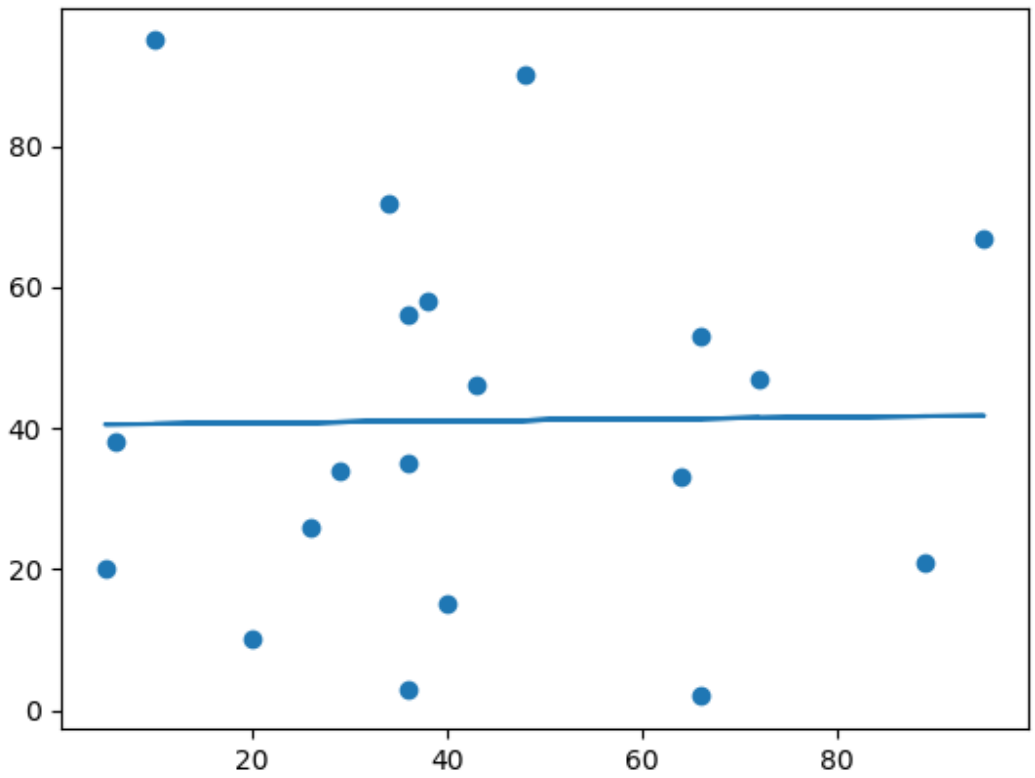
Let us create an example where linear regression would not be the best method to predict future values.

Example

These values for the x- and y-axis should result in a very bad fit for linear regression:

```
import matplotlib.pyplot as plt
from scipy import stats

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

slope, intercept, r, p, std_err = stats.linregress(x, y)

def myfunc(x):
```

```
  return slope * x + intercept

mymodel = list(map(myfunc, x))

plt.scatter(x, y)
plt.plot(x, mymodel)
plt.show()
```



And the `r` for relationship?

Example

You should get a very low `r` value.

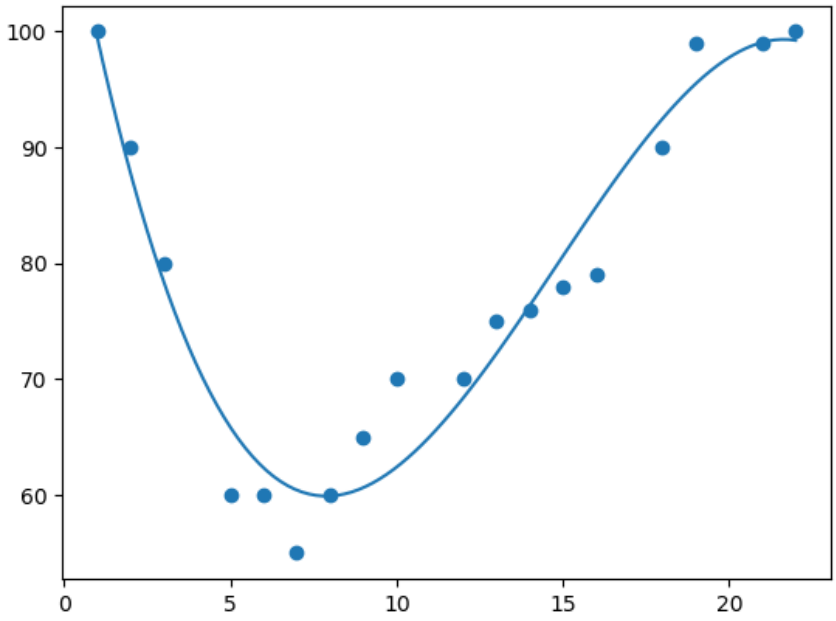
```
import numpy
from scipy import stats

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

slope, intercept, r, p, std_err = stats.linregress(x, y)

print(r)
```

Machine Learning - Polynomial Regression

Polynomial Regression

If your data points clearly will not fit a linear regression (a straight line through all data points), it might be ideal for polynomial regression.

Polynomial regression, like linear regression, uses the relationship between the variables x and y to find the best way to draw a line through the data points.



How Does it Work?

Python has methods for finding a relationship between data-points and to draw a line of polynomial regression. We will show you how to use these methods instead of going through the mathematic formula.

In the example below, we have registered 18 cars as they were passing a certain tollbooth.

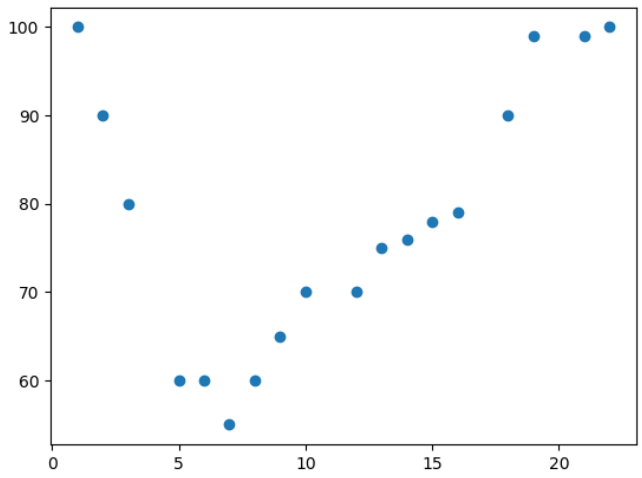
We have registered the car's speed, and the time of day (hour) the passing occurred.

The x-axis represents the hours of the day and the y-axis represents the speed:

Start by drawing a scatter plot:

```
import matplotlib.pyplot as plt

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

plt.scatter(x, y)
plt.show()
```

Result:

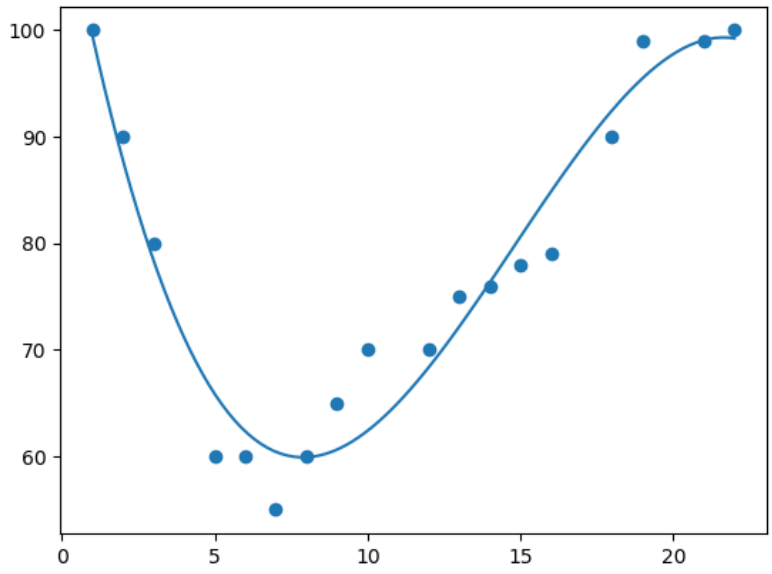


Example

Import `numpy` and `matplotlib` then draw the line of Polynomial Regression:

```
import numpy
import matplotlib.pyplot as plt

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

myline = numpy.linspace(1, 22, 100)

plt.scatter(x, y)
plt.plot(myline, mymodel(myline))
plt.show()
```

Result:



Example Explained

Import the modules you need.

You can learn about the NumPy module in our NumPy Tutorial.

You can learn about the SciPy module in our SciPy Tutorial.

```
import numpy
import matplotlib.pyplot as plt
```

Create the arrays that represent the values of the x and y axis:

```
x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]
```

NumPy has a method that lets us make a polynomial model:

```
mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))
```

Then specify how the line will display, we start at position 1, and end at position 22:

```
myline = numpy.linspace(1, 22, 100)
```

Draw the original scatter plot:

```
plt.scatter(x, y)
```

Draw the line of polynomial regression:

```
plt.plot(myline, mymodel(myline))
```

Display the diagram:

```
plt.show()
```

R-Squared

It is important to know how well the relationship between the values of the x- and y-axis is, if there are no relationship the polynomial regression can not be used to predict anything.

The relationship is measured with a value called the r-squared.

The r-squared value ranges from 0 to 1, where 0 means no relationship, and 1 means 100% related.

Python and the Sklearn module will compute this value for you, all you have to do is feed it with the x and y arrays:

Example

How well does my data fit in a polynomial regression?

```
import numpy
from sklearn.metrics import r2_score

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

print(r2_score(y, mymodel(x)))
```

Note: The result 0.94 shows that there is a very good relationship, and we can use polynomial regression in future predictions.

Predict Future Values

Now we can use the information we have gathered to predict future values.

Example: Let us try to predict the speed of a car that passes the tollbooth at around the time 17:00:

To do so, we need the same `mymodel` array from the example above:

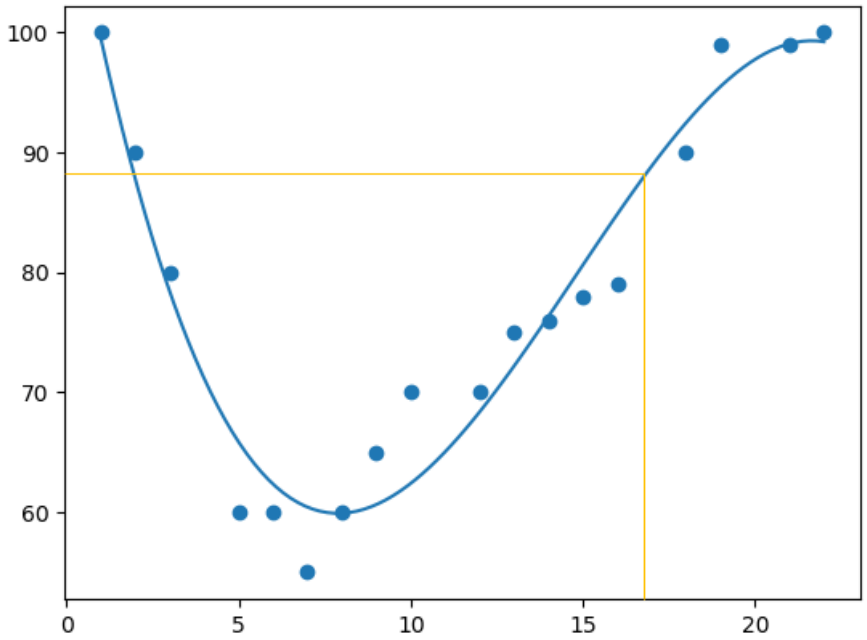
```
mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))
```

Example

Predict the speed of a car passing at 17:00:

```
import numpy
from sklearn.metrics import r2_score

x = [1,2,3,5,6,7,8,9,10,12,13,14,15,16,18,19,21,22]
y = [100,90,80,60,60,55,60,65,70,70,75,76,78,79,90,99,99,100]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

speed = mymodel(17)
print(speed)
```

The example predicted a speed to be 88.87, which we also could read from the diagram:



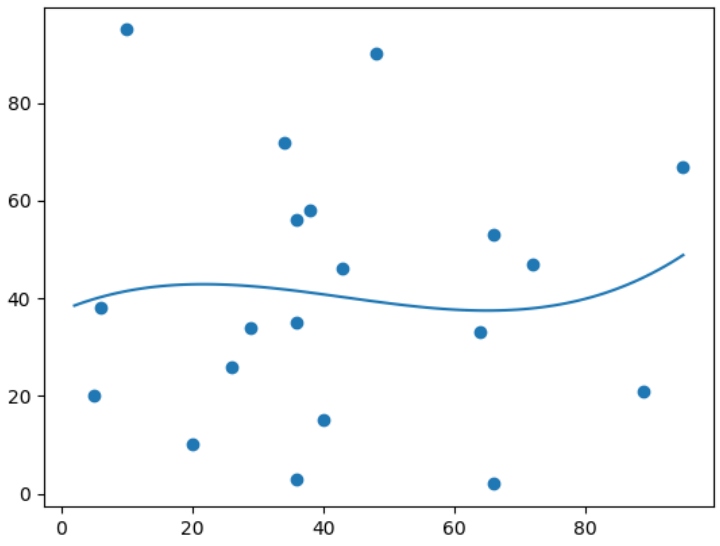
Bad Fit?

Let us create an example where polynomial regression would not be the best method to predict future values.

Example

These values for the x- and y-axis should result in a very bad fit for polynomial regression:

```
import numpy
import matplotlib.pyplot as plt

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

myline = numpy.linspace(2, 95, 100)

plt.scatter(x, y)
plt.plot(myline, mymodel(myline))
plt.show()
```



And the r-squared value?

Example

You should get a very low r-squared value.

```
import numpy
from sklearn.metrics import r2_score

x = [89,43,36,36,95,10,66,34,38,20,26,29,48,64,6,5,36,66,72,40]
y = [21,46,3,35,67,95,53,72,58,10,26,34,90,33,38,20,56,2,47,15]

mymodel = numpy.poly1d(numpy.polyfit(x, y, 3))

print(r2_score(y, mymodel(x)))
```

Machine Learning - Multiple Regression

Multiple Regression

Multiple regression is like linear regression, but with more than one independent value, meaning that we try to predict a value based on **two or more** variables.

Take a look at the data set below, it contains some information about cars.

Car	Model	Volume	Weigh
Toyota	Aygo	1000	790
Mitsubishi	Space Star	1200	1160
Skoda	Citigo	1000	929
Fiat	500	900	865
Mini	Cooper	1500	1140
VW	Up!	1000	929
Skoda	Fabia	1400	1109
Mercedes	A-Class	1500	1365
Ford	Fiesta	1500	1112

Audi	A1	1600	115
Hyundai	I20	1100	98

We can predict the CO2 emission of a car based on the size of the engine, but with multiple regression we can throw in more variables, like the weight of the car, to make the prediction more accurate.

How Does it Work?

In Python we have modules that will do the work for us. Start by importing the Pandas module.

```
import pandas
```

Learn about the Pandas module in our Pandas Tutorial.

The Pandas module allows us to read csv files and return a DataFrame object.

The file is meant for testing purposes only, you can download it here: data.csv

```
df = pandas.read_csv("data.csv")
```

Then make a list of the independent values and call this variable x.

Put the dependent values in a variable called y.

```
X = df[['Weight', 'Volume']]
y = df['CO2']
```

Tip: It is common to name the list of independent values with a upper case X, and the list of dependent values with a lower case y.

We will use some methods from the sklearn module, so we will have to import that module as well:

```
from sklearn import linear_model
```

From the sklearn module we will use the `LinearRegression()` method to create a linear regression object.

This object has a method called `fit()` that takes the independent and dependent values as parameters and fills the regression object with data that describes the relationship:

```
regr = linear_model.LinearRegression()
regr.fit(X, y)
```

Now we have a regression object that are ready to predict CO2 values based on a car's weight and volume:

```
#predict the CO2 emission of a car where the weight is 2300kg, and the volume is 1300cm³:
predictedCO2 = regr.predict([[2300, 1300]])
```

See the whole example in action:

```
import pandas
from sklearn import linear_model

df = pandas.read_csv("data.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

#predict the CO2 emission of a car where the weight is 2300kg, and the volume is 1300cm³:
predictedCO2 = regr.predict([[2300, 1300]])

print(predictedCO2)
```

Result:

```
[107.2087328]
```

We have predicted that a car with 1.3 liter engine, and a weight of 2300 kg, will release approximately 107 grams of CO2 for every kilometer it drives.

Coefficient

The coefficient is a factor that describes the relationship with an unknown variable.

Example: if x is a variable, then 2x is x two times. x is the unknown variable, and the number 2 is the coefficient.

In this case, we can ask for the coefficient value of weight against CO2, and for volume against CO2. The answer(s) we get tells us what would happen if we increase, or decrease, one of the independent values.

Example

Print the coefficient values of the regression object:

```
import pandas
from sklearn import linear_model

df = pandas.read_csv("data.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

print(regr.coef_)
```

Result:

```
[0.00755095 0.00780526]
```

Result Explained

The result array represents the coefficient values of weight and volume.

Weight: 0.00755095
Volume: 0.00780526

These values tell us that if the weight increase by 1kg, the CO2 emission increases by 0.00755095g.

And if the engine size (Volume) increases by 1cm^3, the CO2 emission increases by 0.00780526g.

I think that is a fair guess, but let test it!

We have already predicted that if a car with a 1300cm^3 engine weighs 2300kg, the CO2 emission will be approximately 107g.

What if we increase the weight with 1000kg?

Example

Copy the example from before, but change the weight from 2300 to 3300:

```
import pandas
from sklearn import linear_model

df = pandas.read_csv("data.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

regr = linear_model.LinearRegression()
regr.fit(X, y)

predictedCO2 = regr.predict([[3300, 1300]])

print(predictedCO2)
```

Result:

```
[114.75968007]
```

We have predicted that a car with 1.3 liter engine, and a weight of 3300 kg, will release approximately 115 grams of CO2 for every kilometer it drives.

Which shows that the coefficient of 0.00755095 is correct:

107.2087328 + (1000 * 0.00755095) = 114.75968

Machine Learning - Scale

Scale Features

When your data has different values, and even different measurement units, it can be difficult to compare them. What is kilograms compared to meters? Or altitude compared to time?

The answer to this problem is scaling. We can scale data into new values that are easier to compare.

Take a look at the table below, it is the same data set that we used in the multiple regression chapter, but this time the **volume** column contains values in *liters* instead of cm^3 (1.0 instead of 1000).

It can be difficult to compare the volume 1.0 with the weight 790, but if we scale them both into comparable values, we can easily see how much one value is compared to the other.

There are different methods for scaling data, in this tutorial we will use a method called standardization.

The standardization method uses this formula:

```
z = (x - u) / s
```

Where `z` is the new value, `x` is the original value, `u` is the mean and `s` is the standard deviation.

If you take the **weight** column from the data set above, the first value is 790, and the scaled value will be:

```
(790 - 1292.23) / 238.74 = -2.1
```

If you take the **volume** column from the data set above, the first value is 1.0, and the scaled value will be:

```
(1.0 - 1.61) / 0.38 = -1.59
```

Now you can compare -2.1 with -1.59 instead of comparing 790 with 1.0.

You do not have to do this manually, the Python sklearn module has a method called `StandardScaler()` which returns a Scaler object with methods for transforming data sets.

```
import pandas
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()

df = pandas.read_csv("data.csv")

X = df[['Weight', 'Volume']]

scaledX = scale.fit_transform(X)

print(scaledX)
```

Result:

Note that the first two values are -2.1 and -1.59, which corresponds to our calculations:

```
[[-2.10389253 -1.59336644]
 [-0.55407235 -1.07190106]
 [-1.52166278 -1.59336644]
 [-1.78973979 -1.85409913]
 [-0.63784641 -0.28970299]
 [-1.52166278 -1.59336644]
 [-0.76769621 -0.55043568]
 [ 0.3046118  -0.28970299]
 [-0.7551301  -0.28970299]
 [-0.59595938 -0.0289703 ]
 [-1.30803892 -1.33263375]
 [-1.26615189 -0.81116837]
 [-0.7551301  -1.59336644]
 [-0.16871166 -0.0289703 ]
 [ 0.14125238 -0.0289703 ]
 [ 0.15800719 -0.0289703 ]
 [ 0.3046118  -0.0289703 ]
 [-0.05142797  1.53542584]
 [-0.72580918 -0.0289703 ]
 [ 0.14962979  1.01396046]
 [ 1.2219378  -0.0289703 ]
 [ 0.5685001   1.01396046]
 [ 0.3046118   1.27469315]
 [ 0.51404696 -0.0289703 ]
 [ 0.51404696  1.01396046]
```

```
 [ 0.72348212 -0.28970299]
 [ 0.8281997   1.01396046]
 [ 1.81254495  1.01396046]
 [ 0.96642691 -0.0289703 ]
 [ 1.72877089  1.01396046]
 [ 1.30990057  1.27469315]
 [ 1.90050772  1.01396046]
 [-0.23991961 -0.0289703 ]
 [ 0.40932938 -0.0289703 ]
 [ 0.47215993 -0.0289703 ]
 [ 0.4302729   2.31762392]]
```

Predict CO2 Values

The task in the Multiple Regression chapter was to predict the CO2 emission from a car when you only knew its weight and volume.

When the data set is scaled, you will have to use the scale when you predict values:

Example

Predict the CO2 emission from a 1.3 liter car that weighs 2300 kilograms:

```
import pandas
from sklearn import linear_model
from sklearn.preprocessing import StandardScaler
scale = StandardScaler()

df = pandas.read_csv("data.csv")

X = df[['Weight', 'Volume']]
y = df['CO2']

scaledX = scale.fit_transform(X)

regr = linear_model.LinearRegression()
regr.fit(scaledX, y)

scaled = scale.transform([[2300, 1.3]])
```

```
predictedCO2 = regr.predict([scaled[0]])
print(predictedCO2)
```

Result:

```
[107.2087328]
```

Machine Learning - Train/Test

Evaluate Your Model

In Machine Learning we create models to predict the outcome of certain events, like in the previous chapter where we predicted the CO2 emission of a car when we knew the weight and engine size.

To measure if the model is good enough, we can use a method called Train/Test.

What is Train/Test

Train/Test is a method to measure the accuracy of your model.

It is called Train/Test because you split the data set into two sets: a training set and a testing set.

80% for training, and 20% for testing.

You *train* the model using the training set.

You *test* the model using the testing set.

Train the model means *create* the model.

Test the model means test the accuracy of the model.

Start With a Data Set

Start with a data set you want to test.

Our data set illustrates 100 customers in a shop, and their shopping habits.

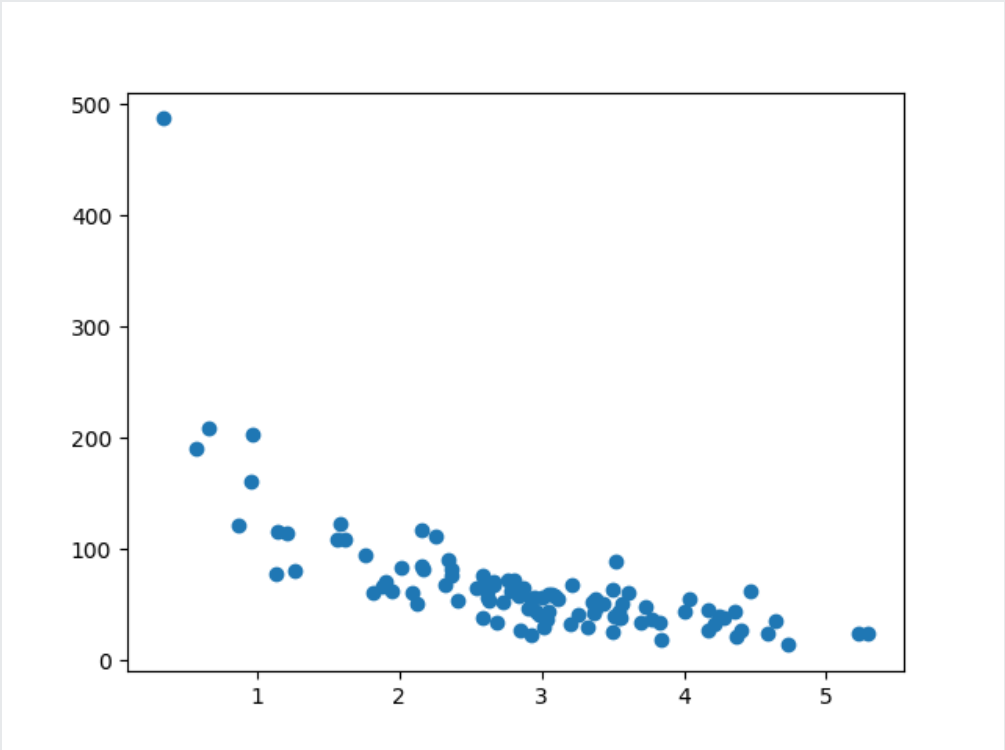
Example

```
import numpy
import matplotlib.pyplot as plt
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

plt.scatter(x, y)
plt.show()
```

Result:

The x axis represents the number of minutes before making a purchase.

The y axis represents the amount of money spent on the purchase.



Split Into Train/Test

The *training* set should be a random selection of 80% of the original data.

The *testing* set should be the remaining 20%.

```
train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]
```

Display the Training Set

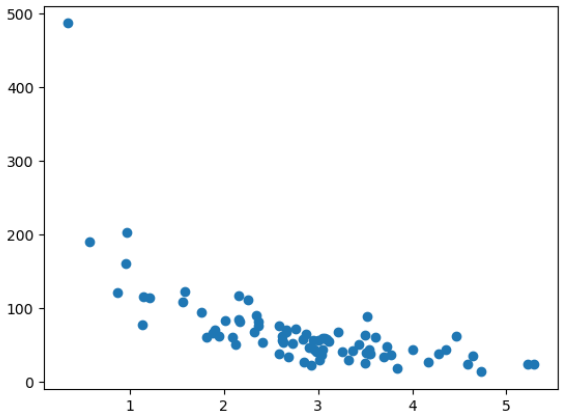
Display the same scatter plot with the training set:

Example

```
plt.scatter(train_x, train_y)
plt.show()
```

Result:

It looks like the original data set, so it seems to be a fair selection:



Display the Testing Set

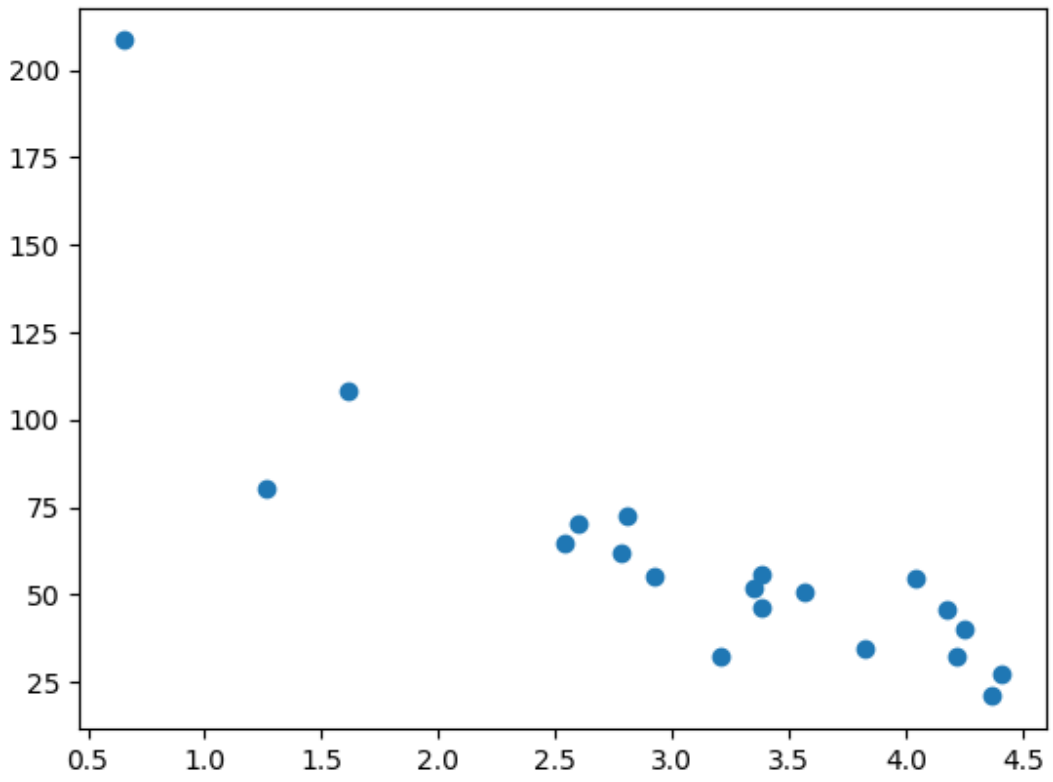
To make sure the testing set is not completely different, we will take a look at the testing set as well.

Example

```
plt.scatter(test_x, test_y)
plt.show()
```

Result:

The testing set also looks like the original data set:



Fit the Data Set

What does the data set look like? In my opinion I think the best fit would be a polynomial regression, so let us draw a line of polynomial regression.

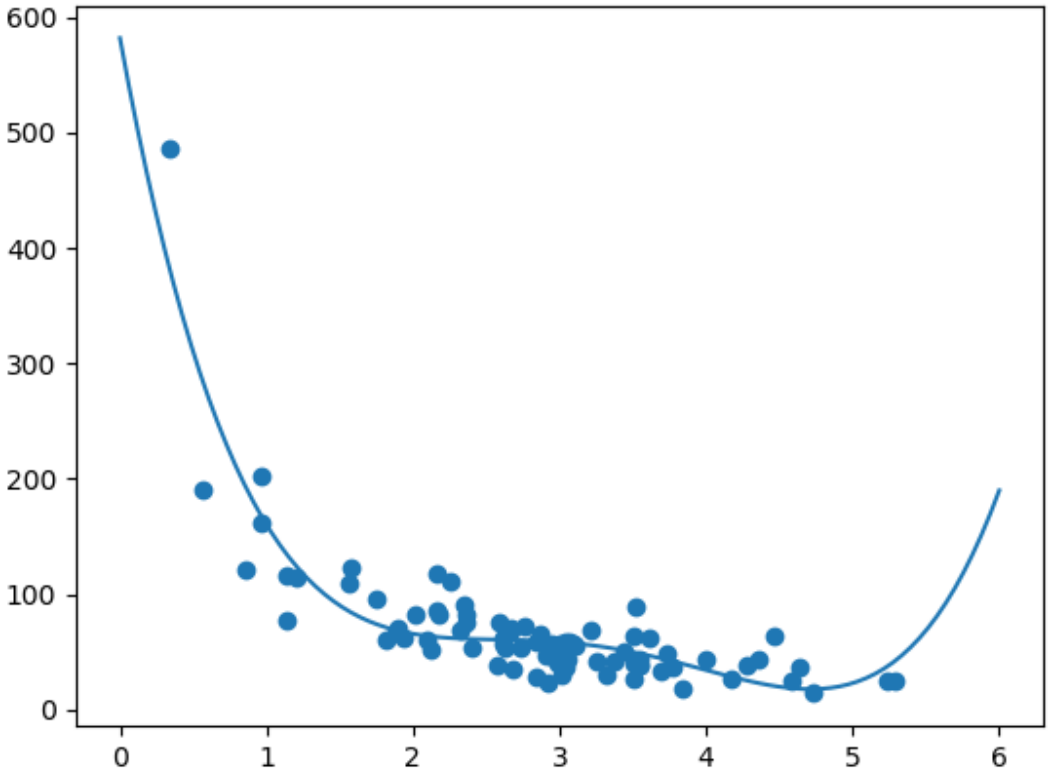
To draw a line through the data points, we use the `plot()` method of the matplotlib module:

Example

Draw a polynomial regression line through the data points:

```
import numpy
import matplotlib.pyplot as plt
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

myline = numpy.linspace(0, 6, 100)

plt.scatter(train_x, train_y)
plt.plot(myline, mymodel(myline))
plt.show()
```

Result:



The result can back my suggestion of the data set fitting a polynomial regression, even though it would give us some weird results if we try to predict values outside of the data set. Example: the line indicates that a customer spending 6 minutes in the shop would make a purchase worth 200. That is probably a sign of overfitting.

But what about the R-squared score? The R-squared score is a good indicator of how well my data set is fitting the model.

R2

Remember R2, also known as R-squared?

It measures the relationship between the x axis and the y axis, and the value ranges from 0 to 1, where 0 means no relationship, and 1 means totally related.

The sklearn module has a method called `r2_score()` that will help us find this relationship.

In this case we would like to measure the relationship between the minutes a customer stays in the shop and how much money they spend.

Example

How well does my training data fit in a polynomial regression?

```
import numpy
from sklearn.metrics import r2_score
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

r2 = r2_score(train_y, mymodel(train_x))

print(r2)
```

Note: The result 0.799 shows that there is a OK relationship.

Bring in the Testing Set

Now we have made a model that is OK, at least when it comes to training data.

Now we want to test the model with the testing data as well, to see if gives us the same result.

Example

Let us find the R2 score when using testing data:

```
import numpy
from sklearn.metrics import r2_score
numpy.random.seed(2)

x = numpy.random.normal(3, 1, 100)
y = numpy.random.normal(150, 40, 100) / x

train_x = x[:80]
train_y = y[:80]

test_x = x[80:]
test_y = y[80:]

mymodel = numpy.poly1d(numpy.polyfit(train_x, train_y, 4))

r2 = r2_score(test_y, mymodel(test_x))

print(r2)
```

Note: The result 0.809 shows that the model fits the testing set as well, and we are confident that we can use the model to predict future values.

Predict Values

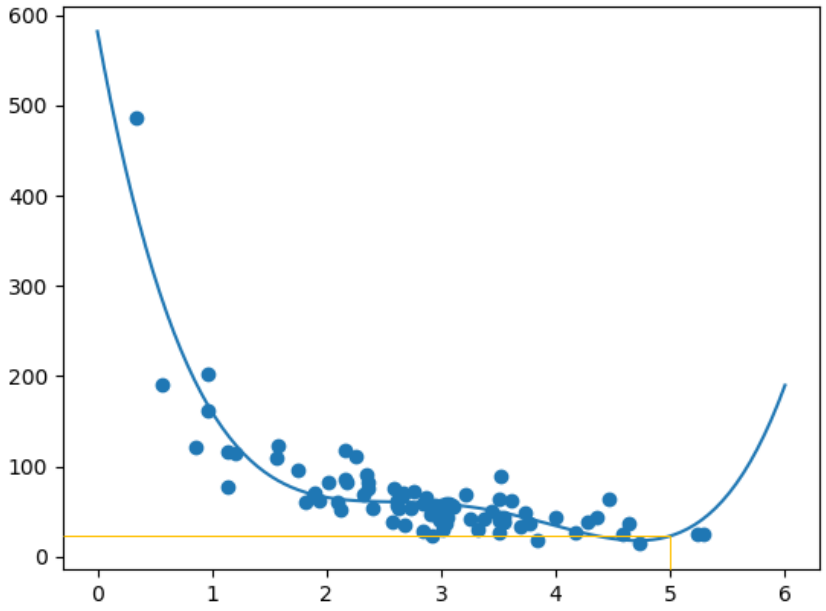
Now that we have established that our model is OK, we can start predicting new values.

Example

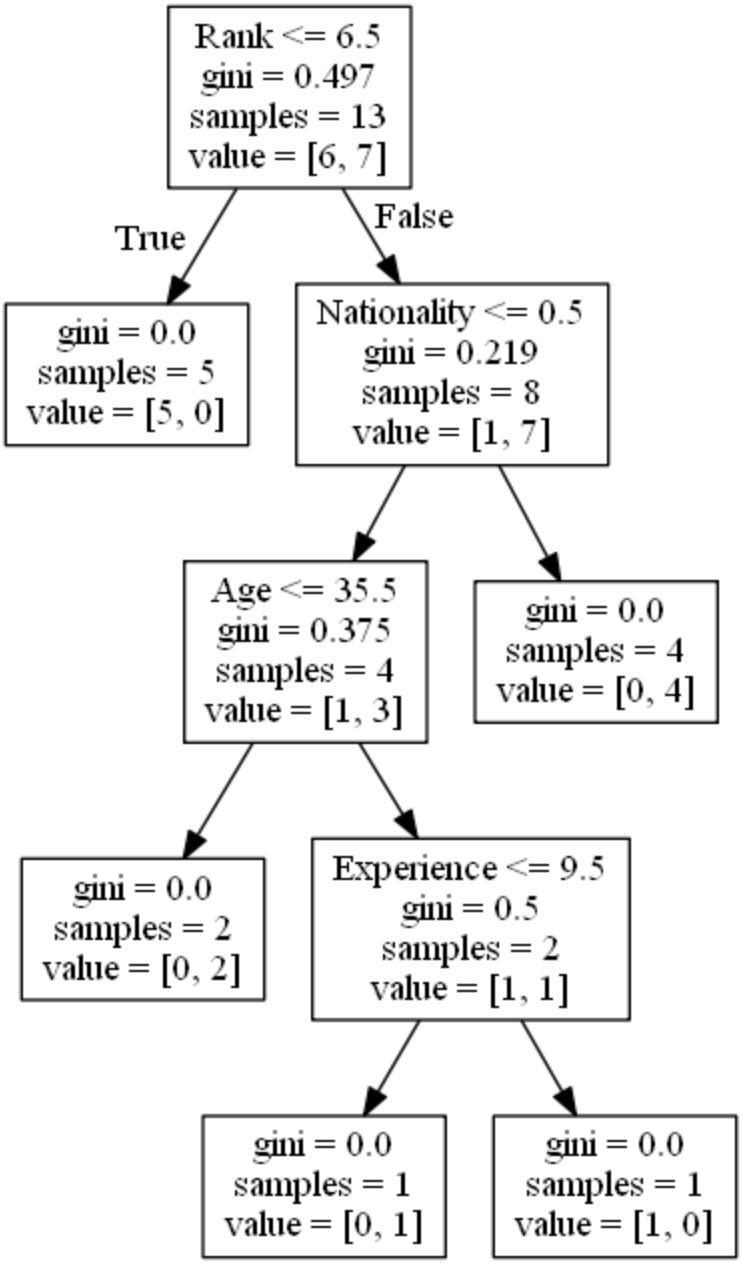
How much money will a buying customer spend, if she or he stays in the shop for 5 minutes?

```
print(mymodel(5))
```

The example predicted the customer to spend 22.88 dollars, as seems to correspond to the diagram:



Machine Learning - Decision Tree



Decision Tree

In this chapter we will show you how to make a "Decision Tree". A Decision Tree is a Flow Chart, and can help you make decisions based on previous experience.

In the example, a person will try to decide if he/she should go to a comedy show or not.

Luckily our example person has registered every time there was a comedy show in town, and registered some information about the comedian, and also registered if he/she went or not.

Now, based on this data set, Python can create a decision tree that can be used to decide if any new shows are worth attending to.

How Does it Work?

First, read the dataset with pandas:

Read and print the data set:

```
import pandas

df = pandas.read_csv("data.csv")

print(df)
```

To make a decision tree, all data has to be numerical.

We have to convert the non numerical columns 'Nationality' and 'Go' into numerical values.

Pandas has a `map()` method that takes a dictionary with information on how to convert the values.

```
{'UK': 0, 'USA': 1, 'N': 2}
```

Means convert the values 'UK' to 0, 'USA' to 1, and 'N' to 2.

Example

Change string values into numerical values:

```
d = {'UK': 0, 'USA': 1, 'N': 2}
df['Nationality'] = df['Nationality'].map(d)
d = {'YES': 1, 'NO': 0}
```

```
df['Go'] = df['Go'].map(d)

print(df)
```

Then we have to separate the *feature* columns from the *target* column.

The feature columns are the columns that we try to predict *from*, and the target column is the column with the values we try to predict.

Example

`x` is the feature columns, `y` is the target column:

```
features = ['Age', 'Experience', 'Rank', 'Nationality']

X = df[features]
y = df['Go']

print(X)
print(y)
```

Now we can create the actual decision tree, fit it with our details. Start by importing the modules we need:

Example

Create and display a Decision Tree:

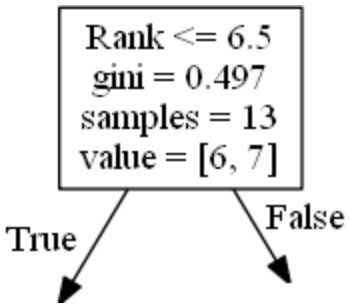
```
import pandas
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

df = pandas.read_csv("data.csv")

d = {'UK': 0, 'USA': 1, 'N': 2}
df['Nationality'] = df['Nationality'].map(d)
d = {'YES': 1, 'NO': 0}
df['Go'] = df['Go'].map(d)

features = ['Age', 'Experience', 'Rank', 'Nationality']

X = df[features]
y = df['Go']
```

```
dtree = DecisionTreeClassifier()
dtree = dtree.fit(X, y)

tree.plot_tree(dtree, feature_names=features)
```

Result Explained

The decision tree uses your earlier decisions to calculate the odds for you to wanting to go see a comedian or not.

Let us read the different aspects of the decision tree:



Rank

`Rank <= 6.5` means that every comedian with a rank of 6.5 or lower will follow the `True` arrow (to the left), and the rest will follow the `False` arrow (to the right).

`gini = 0.497` refers to the quality of the split, and is always a number between 0.0 and 0.5, where 0.0 would mean all of the samples got the same result, and 0.5 would mean that the split is done exactly in the middle.

`samples = 13` means that there are 13 comedians left at this point in the decision, which is all of them since this is the first step.

`value = [6, 7]` means that of these 13 comedians, 6 will get a "NO", and 7 will get a "GO".

Gini

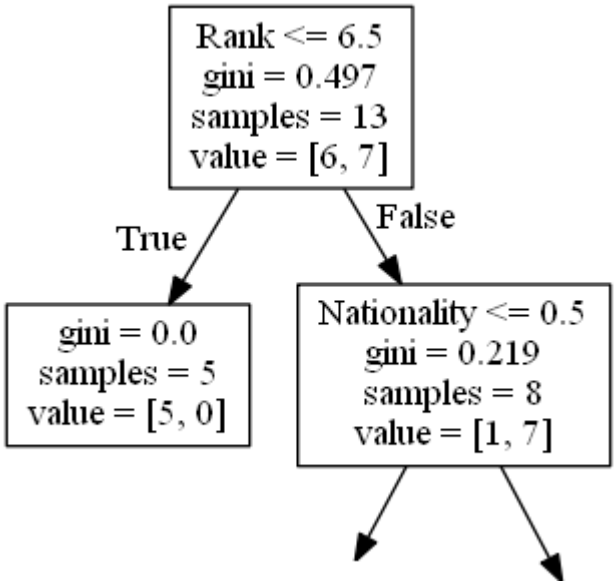
There are many ways to split the samples, we use the GINI method in this tutorial.

The Gini method uses this formula:

`Gini = 1 - (x/n)² - (y/n)²`

Where `x` is the number of positive answers("GO"), `n` is the number of samples, and `y` is the number of negative answers ("NO"), which gives us this calculation:

`1 - (7 / 13)² - (6 / 13)² = 0.497`



The next step contains two boxes, one box for the comedians with a 'Rank' of 6.5 or lower, and one box with the rest.

True - 5 Comedians End Here:

`gini = 0.0` means all of the samples got the same result.

`samples = 5` means that there are 5 comedians left in this branch (5 comedian with a Rank of 6.5 or lower).

`value = [5, 0]` means that 5 will get a "NO" and 0 will get a "GO".

False - 8 Comedians Continue:

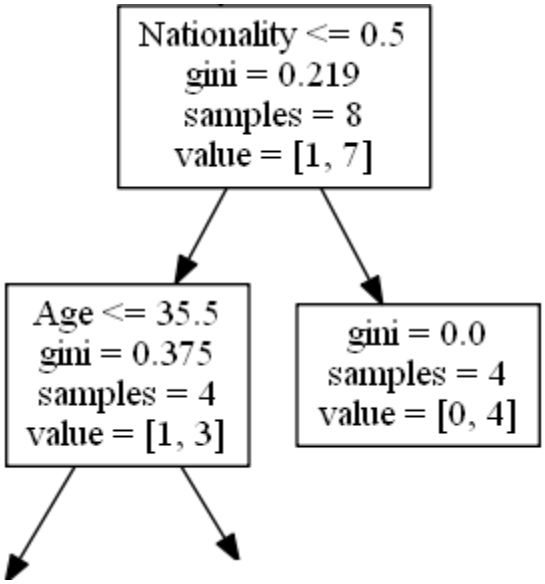
Nationality

`Nationality <= 0.5` means that the comedians with a nationality value of less than 0.5 will follow the arrow to the left (which means everyone from the UK,), and the rest will follow the arrow to the right.

`gini = 0.219` means that about 22% of the samples would go in one direction.

`samples = 8` means that there are 8 comedians left in this branch (8 comedian with a Rank higher than 6.5).

`value = [1, 7]` means that of these 8 comedians, 1 will get a "NO" and 7 will get a "GO".



True - 4 Comedians Continue:

Age

`Age <= 35.5` means that comedians at the age of 35.5 or younger will follow the arrow to the left, and the rest will follow the arrow to the right.

`gini = 0.375` means that about 37,5% of the samples would go in one direction.

`samples = 4` means that there are 4 comedians left in this branch (4 comedians from the UK).

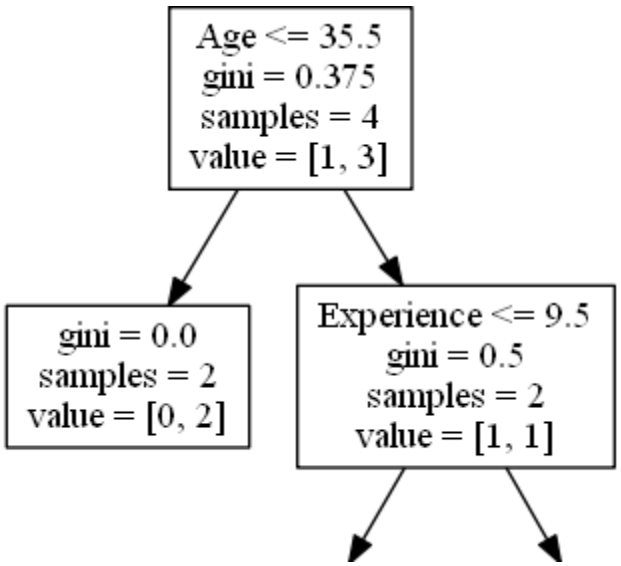
`value = [1, 3]` means that of these 4 comedians, 1 will get a "NO" and 3 will get a "GO".

False - 4 Comedians End Here:

`gini = 0.0` means all of the samples got the same result.

`samples = 4` means that there are 4 comedians left in this branch (4 comedians not from the UK).

`value = [0, 4]` means that of these 4 comedians, 0 will get a "NO" and 4 will get a "GO".



True - 2 Comedians End Here:

`gini = 0.0` means all of the samples got the same result.

`samples = 2` means that there are 2 comedians left in this branch (2 comedians at the age 35.5 or younger).

`value = [0, 2]` means that of these 2 comedians, 0 will get a "NO" and 2 will get a "GO".

False - 2 Comedians Continue:

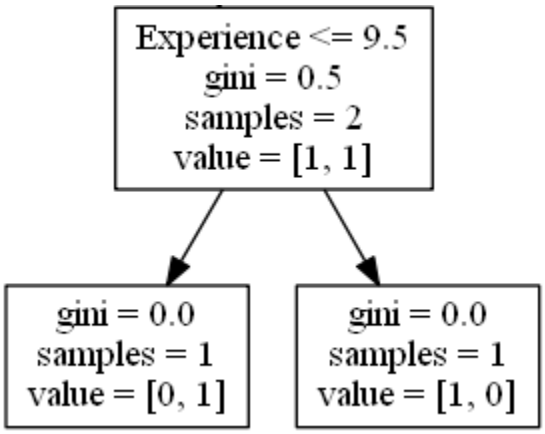
Experience

`Experience <= 9.5` means that comedians with 9.5 years of experience, or less, will follow the arrow to the left, and the rest will follow the arrow to the right.

`gini = 0.5` means that 50% of the samples would go in one direction.

`samples = 2` means that there are 2 comedians left in this branch (2 comedians older than 35.5).

`value = [1, 1]` means that of these 2 comedians, 1 will get a "NO" and 1 will get a "GO".



True - 1 Comedian Ends Here:

`gini = 0.0` means all of the samples got the same result.

`samples = 1` means that there is 1 comedian left in this branch (1 comedian with 9.5 years of experience or less).

`value = [0, 1]` means that 0 will get a "NO" and 1 will get a "GO".

False - 1 Comedian Ends Here:

`gini = 0.0` means all of the samples got the same result.

`samples = 1` means that there is 1 comedians left in this branch (1 comedian with more than 9.5 years of experience).

`value = [1, 0]` means that 1 will get a "NO" and 0 will get a "GO".

Predict Values

We can use the Decision Tree to predict new values.

Example: Should I go see a show starring a 40 years old American comedian, with 10 years of experience, and a comedy ranking of 7?

Example

Use predict() method to predict new values:

```
print(dtree.predict([[40, 10, 7, 1]]))
```

xample

What would the answer be if the comedy rank was 6?

```
print(dtree.predict([[40, 10, 6, 1]]))
```

Different Results

You will see that the Decision Tree gives you different results if you run it enough times, even if you feed it with the same data.

That is because the Decision Tree does not give us a 100% certain answer. It is based on the probability of an outcome, and the answer will vary.

Machine Learning - Confusion Matrix

What is a confusion matrix?

It is a table that is used in classification problems to assess where errors in the model were made.

The rows represent the actual classes the outcomes should have been. While the columns represent the predictions we have made. Using this table it is easy to see which predictions are wrong.

Creating a Confusion Matrix

Confusion matrixes can be created by predictions made from a logistic regression.

For now we will generate actual and predicted values by utilizing NumPy:

```
import numpy
```

Next we will need to generate the numbers for "actual" and "predicted" values.

```
actual = numpy.random.binomial(1, 0.9, size = 1000)
predicted = numpy.random.binomial(1, 0.9, size = 1000)
```

In order to create the confusion matrix we need to import metrics from the sklearn module.

```
from sklearn import metrics
```

Once metrics is imported we can use the confusion matrix function on our actual and predicted values.

```
confusion_matrix = metrics.confusion_matrix(actual, predicted)
```

To create a more interpretable visual display we need to convert the table into a confusion matrix display.

```
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix =
confusion_matrix, display_labels = [0, 1])
```

Vizualizing the display requires that we import pyplot from matplotlib.

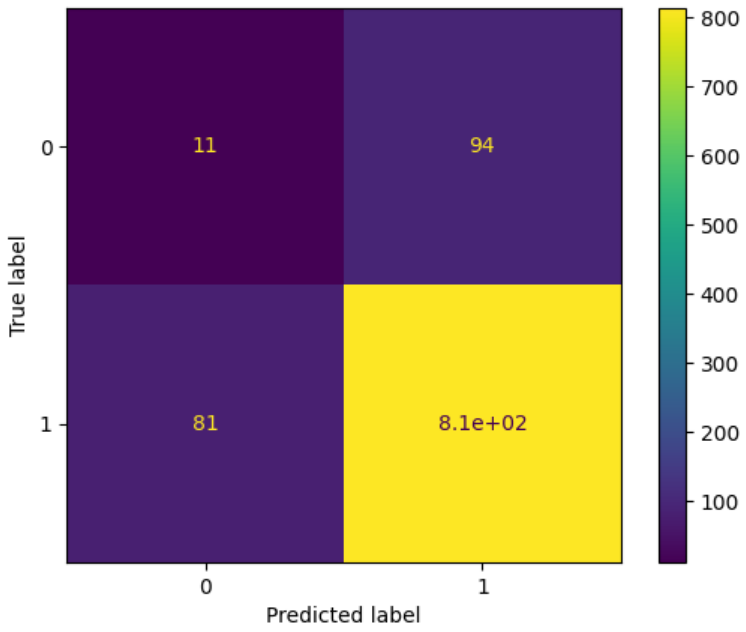
```
import matplotlib.pyplot as plt
```

Finally to display the plot we can use the functions plot() and show() from pyplot.

```
cm_display.plot()
plt.show()
```

See the whole example in action:

```
import matplotlib.pyplot as plt
import numpy
from sklearn import metrics

actual = numpy.random.binomial(1,.9,size = 1000)
predicted = numpy.random.binomial(1,.9,size = 1000)

confusion_matrix = metrics.confusion_matrix(actual, predicted)

cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix =
confusion_matrix, display_labels = [0, 1])

cm_display.plot()
plt.show()
```



Results Explained

The Confusion Matrix created has four different quadrants:

True Negative (Top-Left Quadrant)
False Positive (Top-Right Quadrant)
False Negative (Bottom-Left Quadrant)
True Positive (Bottom-Right Quadrant)

True means that the values were accurately predicted, False means that there was an error or wrong prediction.

Now that we have made a Confusion Matrix, we can calculate different measures to quantify the quality of the model. First, lets look at Accuracy.

Created Metrics

The matrix provides us with many useful metrics that help us to evaluate our classification model.

The different measures include: Accuracy, Precision, Sensitivity (Recall), Specificity, and the F-score, explained below.

Accuracy

Accuracy measures how often the model is correct.

How to Calculate

(True Positive + True Negative) / Total Predictions

Example

```
Accuracy = metrics.accuracy_score(actual, predicted)
```

Precision

Of the positives predicted, what percentage is truly positive?

How to Calculate

True Positive / (True Positive + False Positive)

Precision does not evaluate the correctly predicted negative cases:

Example

```
Precision = metrics.precision_score(actual, predicted)
```

Sensitivity (Recall)

Of all the positive cases, what percentage are predicted positive?

Sensitivity (sometimes called Recall) measures how good the model is at predicting positives.

This means it looks at true positives and false negatives (which are positives that have been incorrectly predicted as negative).

How to Calculate

True Positive / (True Positive + False Negative)

Sensitivity is good at understanding how well the model predicts something is positive:

Example

```
Sensitivity_recall = metrics.recall_score(actual, predicted)
```

Specificity

How well the model is at prediciting negative results?

Specificity is similar to sensitivity, but looks at it from the persepctive of negative results.

How to Calculate

True Negative / (True Negative + False Positive)

Since it is just the opposite of Recall, we use the recall_score function, taking the opposite position label:

Example

```
Specificity = metrics.recall_score(actual, predicted, pos_label=0)
```

F-score

F-score is the "harmonic mean" of precision and sensitivity.

It considers both false positive and false negative cases and is good for imbalanced datasets.

How to Calculate

2 * ((Precision * Sensitivity) / (Precision + Sensitivity))

This score does not take into consideration the True Negative values:

Example

```
F1_score = metrics.f1_score(actual, predicted)
```

All calulations in one:

Example

```
#metrics
print({"Accuracy":Accuracy,"Precision":Precision,"Sensitivity_recall":Sens
itivity_recall,"Specificity":Specificity,"F1_score":F1_score})
```

Machine Learning - Hierarchical Clustering

Hierarchical Clustering

Hierarchical clustering is an unsupervised learning method for clustering data points. The algorithm builds clusters by measuring the dissimilarities between data. Unsupervised learning means that a model does not have to be trained, and we do not need a "target" variable. This method can be used on any data to visualize and interpret the relationship between individual data points.

Here we will use hierarchical clustering to group data points and visualize the clusters using both a dendrogram and scatter plot.

How does it work?

We will use Agglomerative Clustering, a type of hierarchical clustering that follows a bottom up approach. We begin by treating each data point as its own cluster. Then, we join clusters together that have the shortest distance between them to create larger clusters. This step is repeated until one large cluster is formed containing all of the data points.

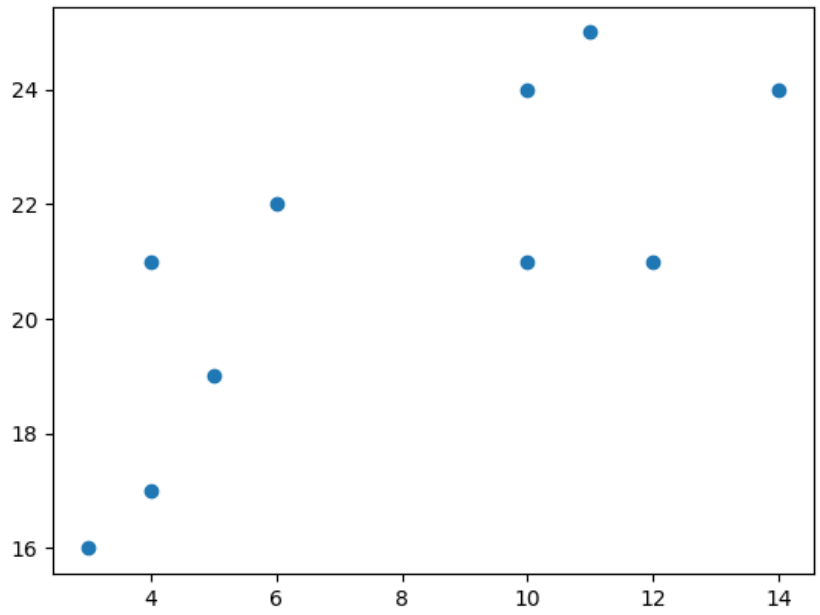
Hierarchical clustering requires us to decide on both a distance and linkage method. We will use euclidean distance and the Ward linkage method, which attempts to minimize the variance between clusters.

Start by visualizing some data points:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

plt.scatter(x, y)
plt.show()
```

Result



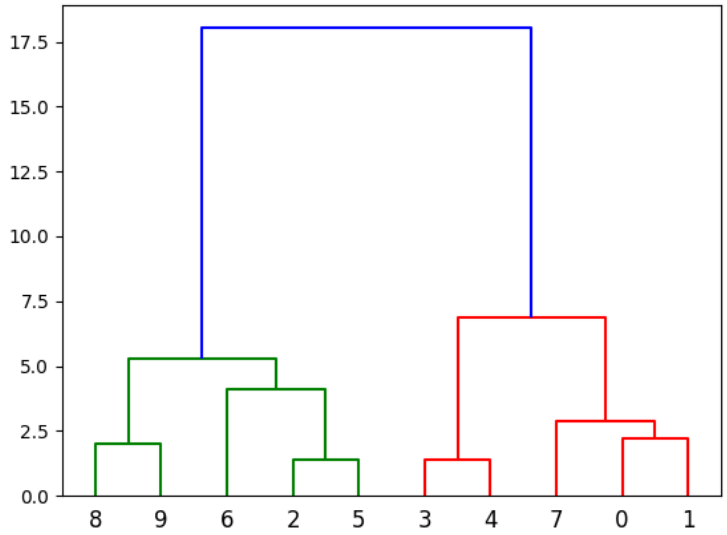
Now we compute the ward linkage using euclidean distance, and visualize it using a dendrogram:

Example

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage

x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

data = list(zip(x, y))

linkage_data = linkage(data, method='ward', metric='euclidean')
dendrogram(linkage_data)
```

```
plt.show()
```

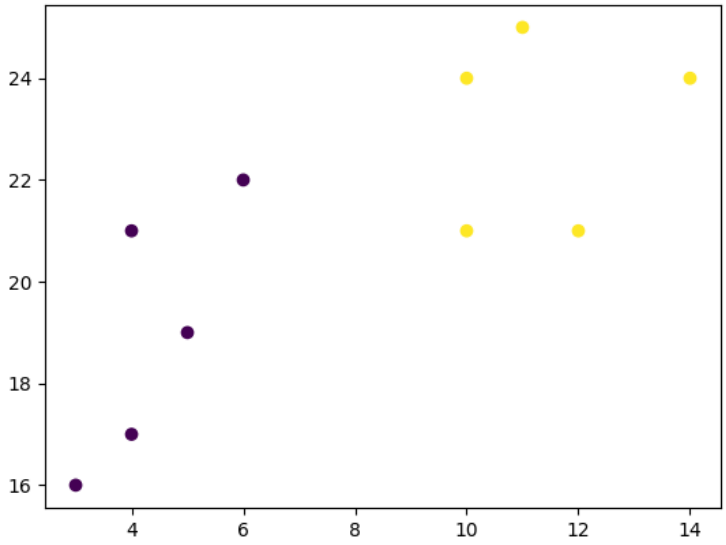
Result



Here, we do the same thing with Python's scikit-learn library. Then, visualize on a 2-dimensional plot:

Example

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering

x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

data = list(zip(x, y))

hierarchical_cluster =
AgglomerativeClustering(n_clusters=2, linkage='ward')
labels = hierarchical_cluster.fit_predict(data)

plt.scatter(x, y, c=labels)
plt.show()
```



Example Explained

Import the modules you need.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering
```

You can learn about the Matplotlib module in our "Matplotlib Tutorial.

You can learn about the SciPy module in our SciPy Tutorial.

NumPy is a library for working with arrays and matricies in Python, you can learn about the NumPy module in our NumPy Tutorial.

scikit-learn is a popular library for machine learning.

Create arrays that resemble two variables in a dataset. Note that while we only use two variables here, this method will work with any number of variables:

```
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
```

Turn the data into a set of points:

```
data = list(zip(x, y))
print(data)
```

Result:

```
[(4, 21), (5, 19), (10, 24), (4, 17), (3, 16), (11, 25),
(14, 24), (6, 22), (10, 21), (12, 21)]
```

Compute the linkage between all of the different points. Here we use a simple euclidean distance measure and Ward's linkage, which seeks to minimize the variance between clusters.

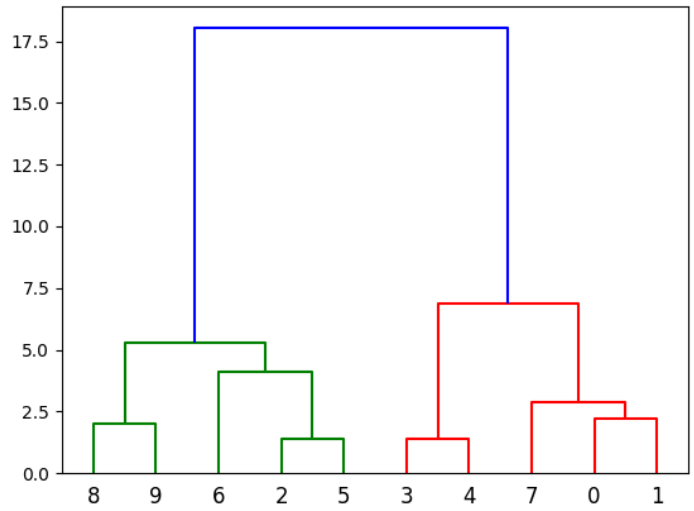
```
linkage_data = linkage(data, method='ward', metric='euclidean')
```

Finally, plot the results in a dendrogram. This plot will show us the hierarchy of clusters from the bottom (individual points) to the top (a single cluster consisting of all data points).

`plt.show()` lets us visualize the dendrogram instead of just the raw linkage data.

```
dendrogram(linkage_data)
plt.show()
```

Result:



The scikit-learn library allows us to use hierarchichal clustering in a different manner. First, we initialize the `AgglomerativeClustering` class with 2 clusters and the Ward linkage.

```
hierarchical_cluster = AgglomerativeClustering(n_clusters=2,
linkage='ward')
```

The `.fit_predict` method can be called on our data to compute the clusters using the defined parameters across our chosen number of clusters.

```
labels = hierarchical_cluster.fit_predict(data) print(labels)
```

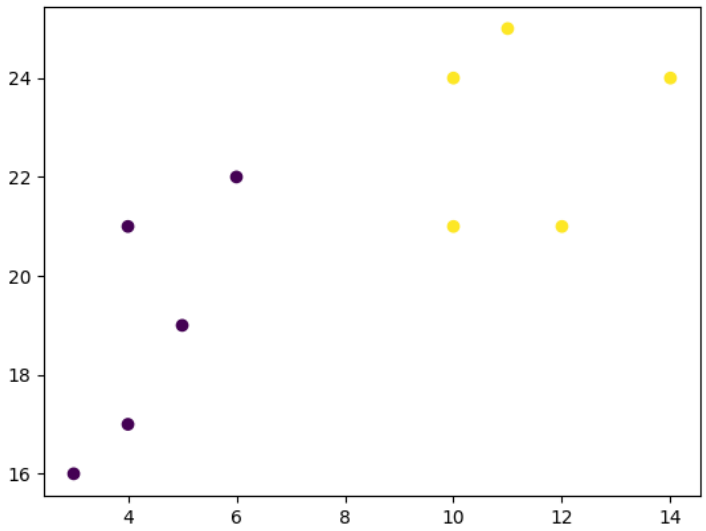
Result:

```
[0 0 1 0 0 1 1 0 1 1]
```

Finally, if we plot the same data and color the points using the labels assigned to each index by the hierarchical clustering method, we can see the cluster each point was assigned to:

```
plt.scatter(x, y, c=labels)
plt.show()
```

Result:



Machine Learning - Logistic Regression

Logistic Regression

Logistic regression aims to solve classification problems. It does this by predicting categorical outcomes, unlike linear regression that predicts a continuous outcome.

In the simplest case there are two outcomes, which is called binomial, an example of which is predicting if a tumor is malignant or benign. Other cases have more than two outcomes to classify, in this case it is called multinomial. A common example for multinomial logistic regression would be predicting the class of an iris flower between 3 different species.

Here we will be using basic logistic regression to predict a binomial variable. This means it has only two possible outcomes.

How does it work?

In Python we have modules that will do the work for us. Start by importing the NumPy module.

```
import numpy
```

Store the independent variables in X.

Store the dependent variable in y.

Below is a sample dataset:

```
#X represents the size of a tumor in centimeters.
X =
numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37, 4.9
6, 4.52, 3.69, 5.88]).reshape(-1,1)
```

```
#Note: X has to be reshaped into a column from a row for the
LogisticRegression() function to work.
#y represents whether or not the tumor is cancerous (0 for "No",
1 for "Yes").
y = numpy.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])
```

We will use a method from the sklearn module, so we will have to import that module as well:

```
from sklearn import linear_model
```

From the sklearn module we will use the LogisticRegression() method to create a logistic regression object.

This object has a method called `fit()` that takes the independent and dependent values as parameters and fills the regression object with data that describes the relationship:

```
logr = linear_model.LogisticRegression()
logr.fit(X,y)
```

Now we have a logistic regression object that is ready to whether a tumor is cancerous based on the tumor size:

```
#predict if tumor is cancerous where the size is 3.46mm:
predicted = logr.predict(numpy.array([3.46]).reshape(-1,1))
```

Example

See the whole example in action:

```
import numpy
from sklearn import linear_model

#Reshaped for Logistic function.
X =
numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37, 4.96, 4.52, 3
.69, 5.88]).reshape(-1,1)
y = numpy.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

logr = linear_model.LogisticRegression()
logr.fit(X,y)

#predict if tumor is cancerous where the size is 3.46mm:
predicted = logr.predict(numpy.array([3.46]).reshape(-1,1))
print(predicted)
```

Result

```
[0]
```

We have predicted that a tumor with a size of 3.46mm will not be cancerous.

Coefficient

In logistic regression the coefficient is the expected change in log-odds of having the outcome per unit change in X.

This does not have the most intuitive understanding so let's use it to create something that makes more sense, odds.

Example

See the whole example in action:

```
import numpy
from sklearn import linear_model

#Reshaped for Logistic function.
X =
numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37, 4.96, 4.52, 3
.69, 5.88]).reshape(-1,1)
y = numpy.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

logr = linear_model.LogisticRegression()
logr.fit(X,y)

log_odds = logr.coef_
odds = numpy.exp(log_odds)

print(odds)
```

Result

```
[4.03541657]
```

This tells us that as the size of a tumor increases by 1mm the odds of it being a cancerous tumor increases by 4x.

Probability

The coefficient and intercept values can be used to find the probability that each tumor is cancerous.

Create a function that uses the model's coefficient and intercept values to return a new value. This new value represents probability that the given observation is a tumor:

```
def logit2prob(logr,x):
  log_odds = logr.coef_ * x + logr.intercept_
  odds = numpy.exp(log_odds)
  probability = odds / (1 + odds)
  return(probability)
```

Function Explained

To find the log-odds for each observation, we must first create a formula that looks similar to the one from linear regression, extracting the coefficient and the intercept.

```
log_odds = logr.coef_ * x + logr.intercept_
```

To then convert the log-odds to odds we must exponentiate the log-odds.

```
odds = numpy.exp(log_odds)
```

Now that we have the odds, we can convert it to probability by dividing it by 1 plus the odds.

```
probability = odds / (1 + odds)
```

Let us now use the function with what we have learned to find out the probability that each tumor is cancerous.

Example

See the whole example in action:

```
import numpy
from sklearn import linear_model

X =
numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37, 4.96, 4.52, 3
.69, 5.88]).reshape(-1,1)
y = numpy.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

logr = linear_model.LogisticRegression()
logr.fit(X,y)

def logit2prob(logr, X):
  log_odds = logr.coef_ * X + logr.intercept_
  odds = numpy.exp(log_odds)
  probability = odds / (1 + odds)
  return(probability)

print(logit2prob(logr, X))
```

Result

```
  [[0.60749955]
   [0.19268876]
   [0.12775886]
   [0.00955221]
   [0.08038616]
   [0.07345637]
   [0.88362743]
   [0.77901378]
   [0.88924409]
   [0.81293497]
   [0.57719129]
   [0.96664243]]
```

Results Explained

3.78 0.61 The probability that a tumor with the size 3.78cm is cancerous is 61%.

2.44 0.19 The probability that a tumor with the size 2.44cm is cancerous is 19%.

2.09 0.13 The probability that a tumor with the size 2.09cm is cancerous is 13%.

Machine Learning - Grid Search

Grid Search

The majority of machine learning models contain parameters that can be adjusted to vary how the model learns. For example, the logistic regression model, from `sklearn`, has a parameter `C` that controls regularization,which affects the complexity of the model.

How do we pick the best value for `C`? The best value is dependent on the data used to train the model.

How does it work?

One method is to try out different values and then pick the value that gives the best score. This technique is known as a **grid search**. If we had to select the values for two or more parameters, we would evaluate all combinations of the sets of values thus forming a grid of values.

Before we get into the example it is good to know what the parameter we are changing does. Higher values of `C` tell the model, the training data resembles real world information, place a greater weight on the training data. While lower values of `C` do the opposite.

Using Default Parameters

First let's see what kind of results we can generate without a grid search using only the base parameters.

To get started we must first load in the dataset we will be working with.

```
from sklearn import datasets
iris = datasets.load_iris()
```

Next in order to create the model we must have a set of independent variables X and a dependant variable y.

```
X = iris['data']
y = iris['target']
```

Now we will load the logistic model for classifying the iris flowers.

```
from sklearn.linear_model import LogisticRegression
```

Creating the model, setting max_iter to a higher value to ensure that the model finds a result.

Keep in mind the default value for `C` in a logistic regression model is `1`, we will compare this later.

In the example below, we look at the iris data set and try to train a model with varying values for `C` in logistic regression.

```
logit = LogisticRegression(max_iter = 10000)
```

After we create the model, we must fit the model to the data.

```
print(logit.fit(X,y))
```

To evaluate the model we run the score method.

```
print(logit.score(X,y))
```

Example

```
from sklearn import datasets
from sklearn.linear_model import LogisticRegression

iris = datasets.load_iris()

X = iris['data']
y = iris['target']

logit = LogisticRegression(max_iter = 10000)

print(logit.fit(X,y))

print(logit.score(X,y))
```

With the default setting of `C = 1`, we achieved a score of `0.973`.

Let's see if we can do any better by implementing a grid search with difference values of 0.973.

Implementing Grid Search

We will follow the same steps of before except this time we will set a range of values for `C`.

Knowing which values to set for the searched parameters will take a combination of domain knowledge and practice.

Since the default value for `C` is `1`, we will set a range of values surrounding it.

```
C = [0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]
```

Next we will create a for loop to change out the values of `C` and evaluate the model with each change.

First we will create an empty list to store the score within.

```
scores = []
```

To change the values of `C` we must loop over the range of values and update the parameter each time.

```
for choice in C:
  logit.set_params(C=choice)
  logit.fit(X, y)
  scores.append(logit.score(X, y))
```

With the scores stored in a list, we can evaluate what the best choice of `C` is.

```
print(scores)
```

Example

```
from sklearn import datasets
from sklearn.linear_model import LogisticRegression

iris = datasets.load_iris()

X = iris['data']
y = iris['target']

logit = LogisticRegression(max_iter = 10000)

C = [0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2]

scores = []

for choice in C:
  logit.set_params(C=choice)
  logit.fit(X, y)
  scores.append(logit.score(X, y))

print(scores)
```

Results Explained

We can see that the lower values of `C` performed worse than the base parameter of `1`. However, as we increased the value of `C` to `1.75` the model experienced increased accuracy.

It seems that increasing `C` beyond this amount does not help increase model accuracy.

Note on Best Practices

We scored our logistic regression model by using the same data that was used to train it. If the model corresponds too closely to that data, it may not be great at predicting unseen data. This statistical error is known as **over fitting**.

To avoid being misled by the scores on the training data, we can put aside a portion of our data and use it specifically for the purpose of testing the model. Refer to the lecture on train/test splitting to avoid being misled and overfitting.

Preprocessing - Categorical Data

Categorical Data

When your data has categories represented by strings, it will be difficult to use them to train machine learning models which often only accepts numeric data.

Instead of ignoring the categorical data and excluding the information from our model, you can tranform the data so it can be used in your models.

Take a look at the table below, it is the same data set that we used in the multiple regression chapter.

```
import pandas as pd

cars = pd.read_csv('data.csv')
print(cars.to_string())
```

In the multiple regression chapter, we tried to predict the CO2 emitted based on the volume of the engine and the weight of the car but we excluded information about the car brand and model.

The information about the car brand or the car model might help us make a better prediction of the CO2 emitted.

One Hot Encoding

We cannot make use of the Car or Model column in our data since they are not numeric. A linear relationship between a categorical variable, Car or Model, and a numeric variable, CO2, cannot be determined.

To fix this issue, we must have a numeric representation of the categorical variable. One way to do this is to have a column representing each group in the category.

For each column, the values will be 1 or 0 where 1 represents the inclusion of the group and 0 represents the exclusion. This transformation is called one hot encoding.

You do not have to do this manually, the Python Pandas module has a function that called `get_dummies()` which does one hot encoding.

Learn about the Pandas module in our Pandas Tutorial.

Example

One Hot Encode the Car column:

```
import pandas as pd

cars = pd.read_csv('data.csv')
ohe_cars = pd.get_dummies(cars[['Car']])

print(ohe_cars.to_string())
```

Results

A column was created for every car brand in the Car column.

Predict CO2

We can use this additional information alongside the volume and weight to predict CO2

To combine the information, we can use the `concat()` function from pandas.

First we will need to import a couple modules.

We will start with importing the Pandas.

```
import pandas
```

The pandas module allows us to read csv files and manipulate DataFrame objects:

```
cars = pandas.read_csv("data.csv")
```

It also allows us to create the dummy variables:

```
ohe_cars = pandas.get_dummies(cars[['Car']])
```

Then we must select the independent variables (X) and add the dummy variables columnwise.

Also store the dependent variable in y.

```
X = pandas.concat([cars[['Volume', 'Weight']], ohe_cars],
axis=1)
y = cars['CO2']
```

We also need to import a method from sklearn to create a linear model

Learn about linear regression.

```
from sklearn import linear_model
```

Now we can fit the data to a linear regression:

```
regr = linear_model.LinearRegression()
regr.fit(X,y)
```

Finally we can predict the CO2 emissions based on the car's weight, volume, and manufacturer.

```
##predict the CO2 emission of a VW where the weight is 2300kg,
and the volume is 1300cm3:
predictedCO2 =
regr.predict([[2300, 1300,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]])
```

Example

```
import pandas
from sklearn import linear_model

cars = pandas.read_csv("data.csv")
ohe_cars = pandas.get_dummies(cars[['Car']])

X = pandas.concat([cars[['Volume', 'Weight']], ohe_cars], axis=1)
y = cars['CO2']

regr = linear_model.LinearRegression()
regr.fit(X,y)

##predict the CO2 emission of a VW where the weight is 2300kg, and the
```

```
volume is 1300cm3:
predictedCO2 =
regr.predict([[2300, 1300,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0]])

print(predictedCO2)
```

Result

```
 [122.45153299]
```

We now have a coefficient for the volume, the weight, and each car brand in the data set

---

Dummifying

It is not necessary to create one column for each group in your category. The information can be retained using 1 column less than the number of groups you have.

For example, you have a column representing colors and in that column, you have two colors, red and blue.

Example

```
import pandas as pd

colors = pd.DataFrame({'color': ['blue', 'red']})

print(colors)
```

Result

```
    color
  0  blue
  1   red
```

You can create 1 column called red where 1 represents red and 0 represents not red, which means it is blue.

To do this, we can use the same function that we used for one hot encoding, get_dummies, and then drop one of the columns. There is an argument, drop_first, which allows us to exclude the first column from the resulting table.

Example

```
import pandas as pd

colors = pd.DataFrame({'color': ['blue', 'red']})
dummies = pd.get_dummies(colors, drop_first=True)

print(dummies)
```

Result

```
   color_red
0          0
1          1
```

What if you have more than 2 groups? How can the multiple groups be represented by 1 less column?

Let's say we have three colors this time, red, blue and green. When we get_dummies while dropping the first column, we get the following table.

Example

```
import pandas as pd

colors = pd.DataFrame({'color': ['blue', 'red', 'green']})
dummies = pd.get_dummies(colors, drop_first=True)
dummies['color'] = colors['color']

print(dummies)
```

Result

```
   color_green  color_red  color
0            0          0   blue
1            0          1    red
2            1          0  green
```

Machine Learning - K-means

K-means

K-means is an unsupervised learning method for clustering data points. The algorithm iteratively divides data points into K clusters by minimizing the variance in each cluster.

Here, we will show you how to estimate the best value for K using the elbow method, then use K-means clustering to group the data points into clusters.

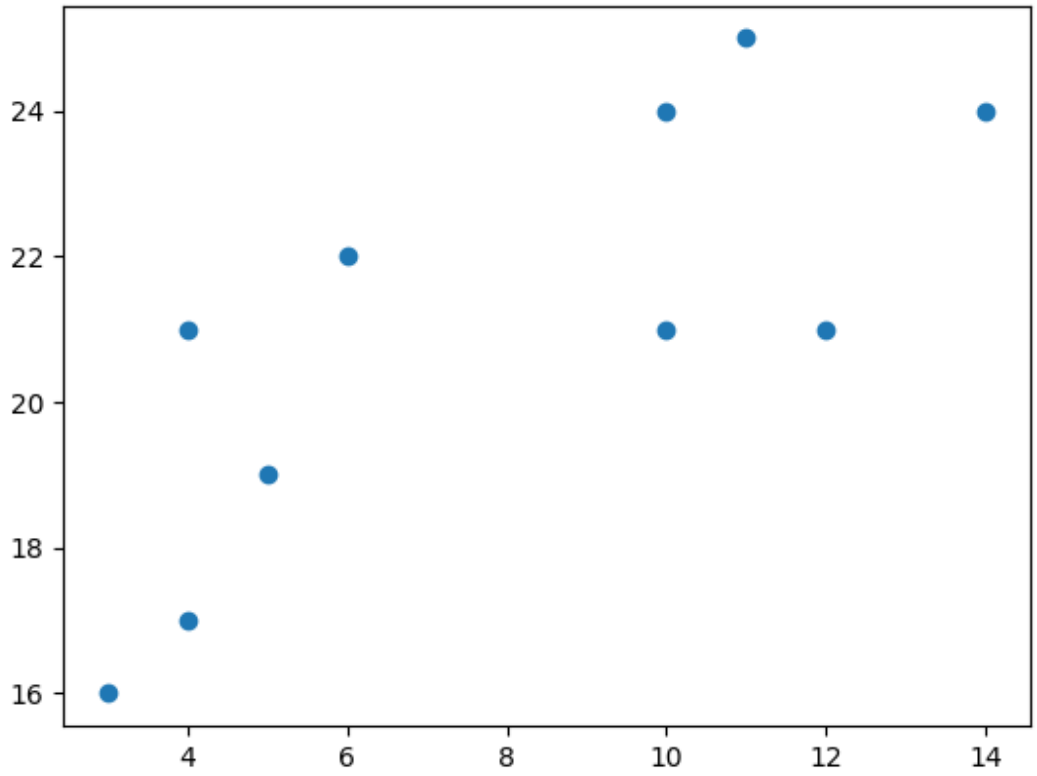
How does it work?

First, each data point is randomly assigned to one of the K clusters. Then, we compute the centroid (functionally the center) of each cluster, and reassign each data point to the cluster with the closest centroid. We repeat this process until the cluster assignments for each data point are no longer changing.

K-means clustering requires us to select K, the number of clusters we want to group the data into. The elbow method lets us graph the inertia (a distance-based metric) and visualize the point at which it starts decreasing linearly. This point is referred to as the "elbow" and is a good estimate for the best value for K based on our data.

Start by visualizing some data points:

```
import matplotlib.pyplot as plt

x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]

plt.scatter(x, y)
plt.show()
```

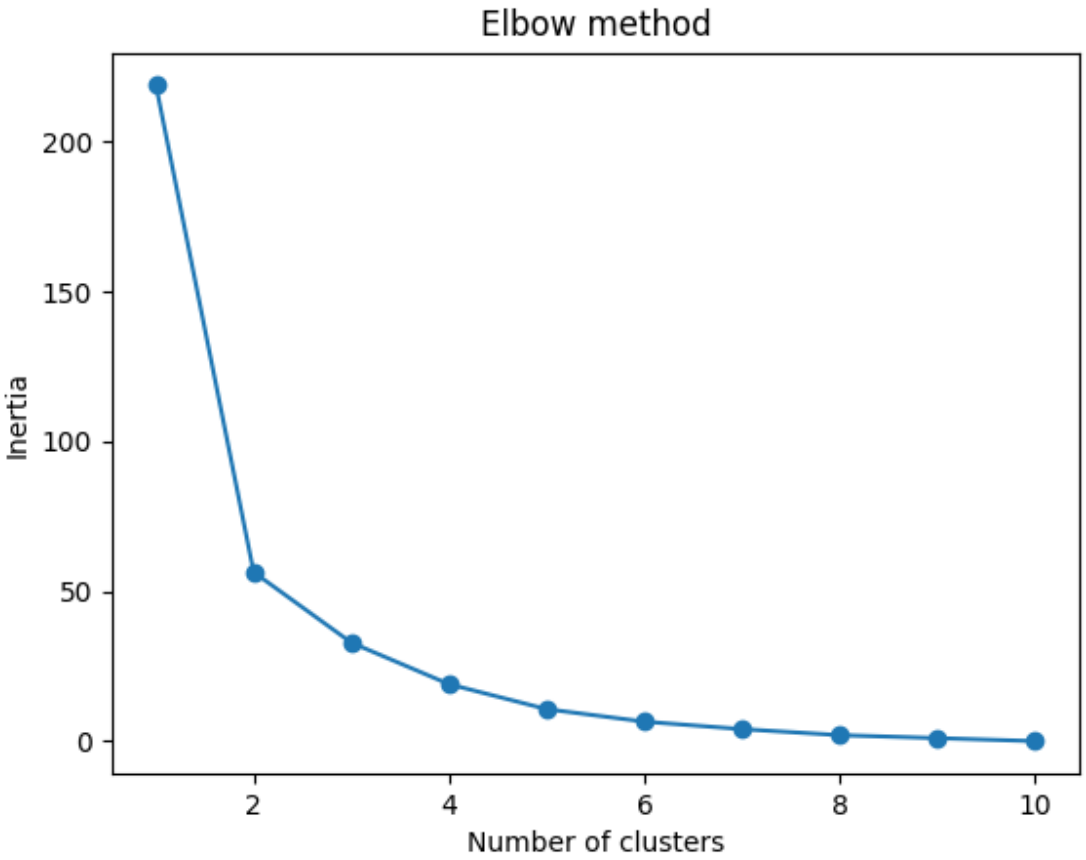


Now we utilize the elbow method to visualize the intertia for different values of K:

Example

```
from sklearn.cluster import KMeans

data = list(zip(x, y))
inertias = []

for i in range(1,11):
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(data)
    inertias.append(kmeans.inertia_)

plt.plot(range(1,11), inertias, marker='o')
plt.title('Elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```

Result

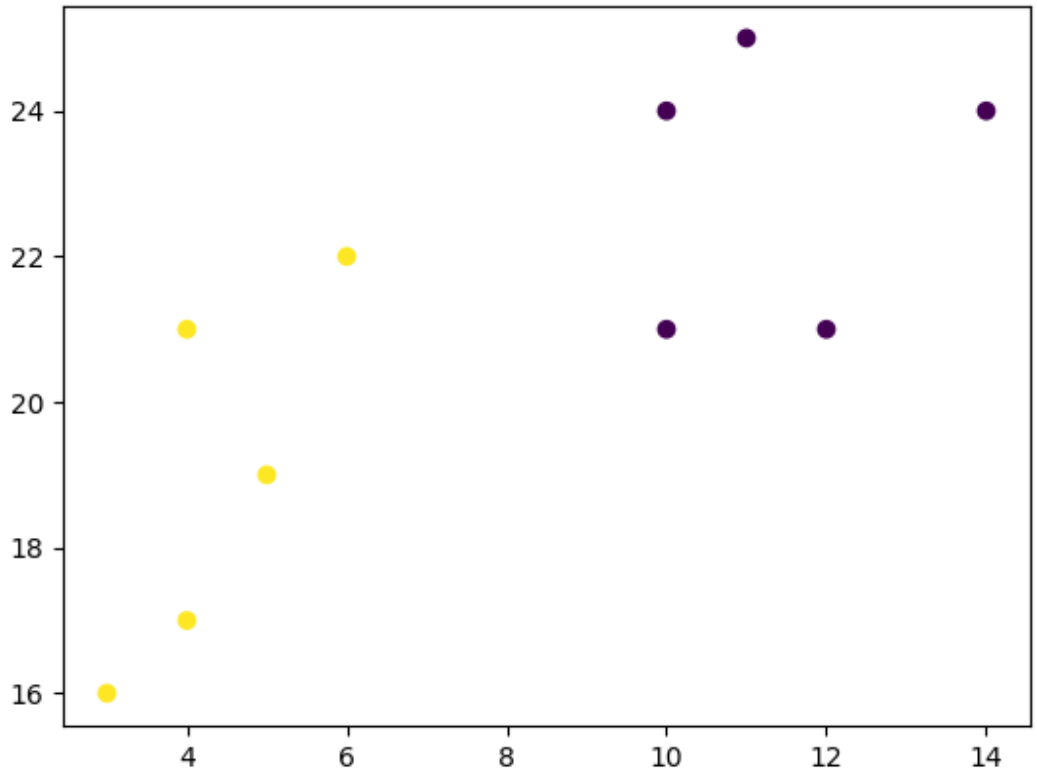


The elbow method shows that 2 is a good value for K, so we retrain and visualize the result:

Example

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(data)

plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```

Result



Example Explained

Import the modules you need.

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
```

You can learn about the Matplotlib module in our "Matplotlib Tutorial.

scikit-learn is a popular library for machine learning.

Create arrays that resemble two variables in a dataset. Note that while we only use two variables here, this method will work with any number of variables:

```
x = [4, 5, 10, 4, 3, 11, 14 , 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
```

Turn the data into a set of points:

```
data = list(zip(x, y))
print(data)
```

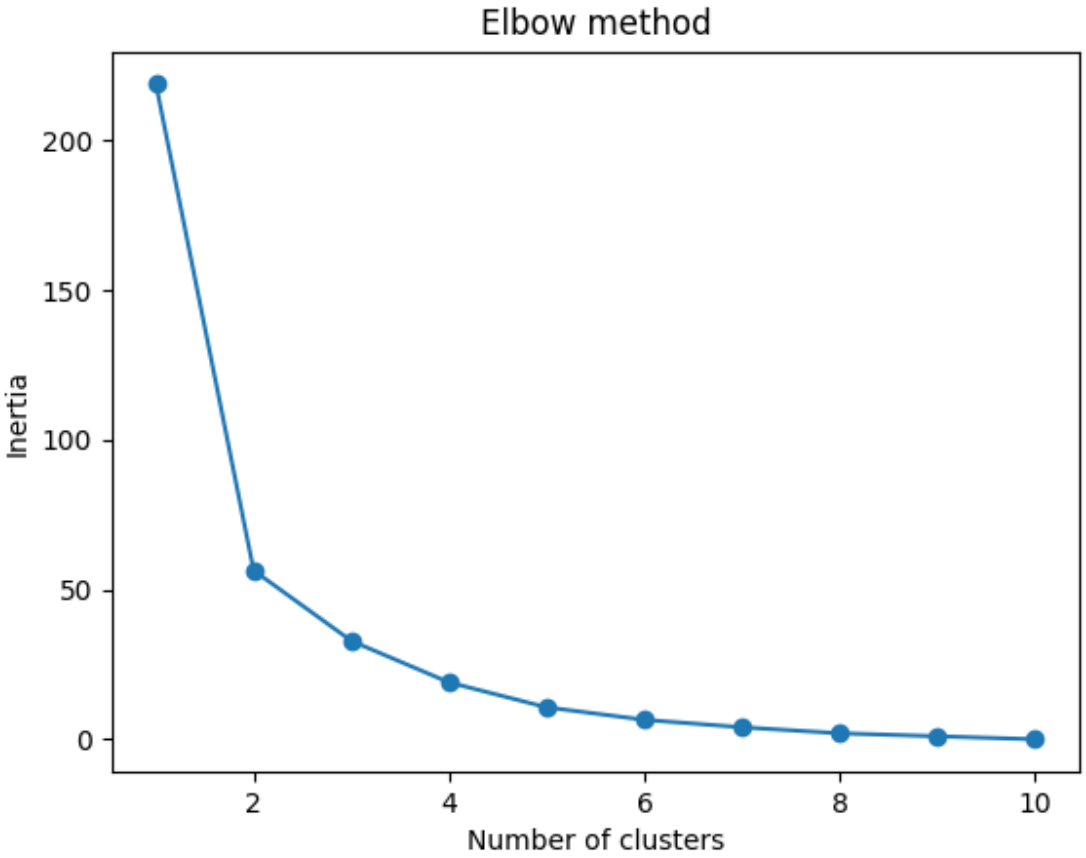
Result:

```
[(4, 21), (5, 19), (10, 24), (4, 17), (3, 16), (11, 25),
(14, 24), (6, 22), (10, 21), (12, 21)]
```

In order to find the best value for K, we need to run K-means across our data for a range of possible values. We only have 10 data points, so the maximum number of clusters is 10. So for each value K in range(1,11), we train a K-means model and plot the intertia at that number of clusters:

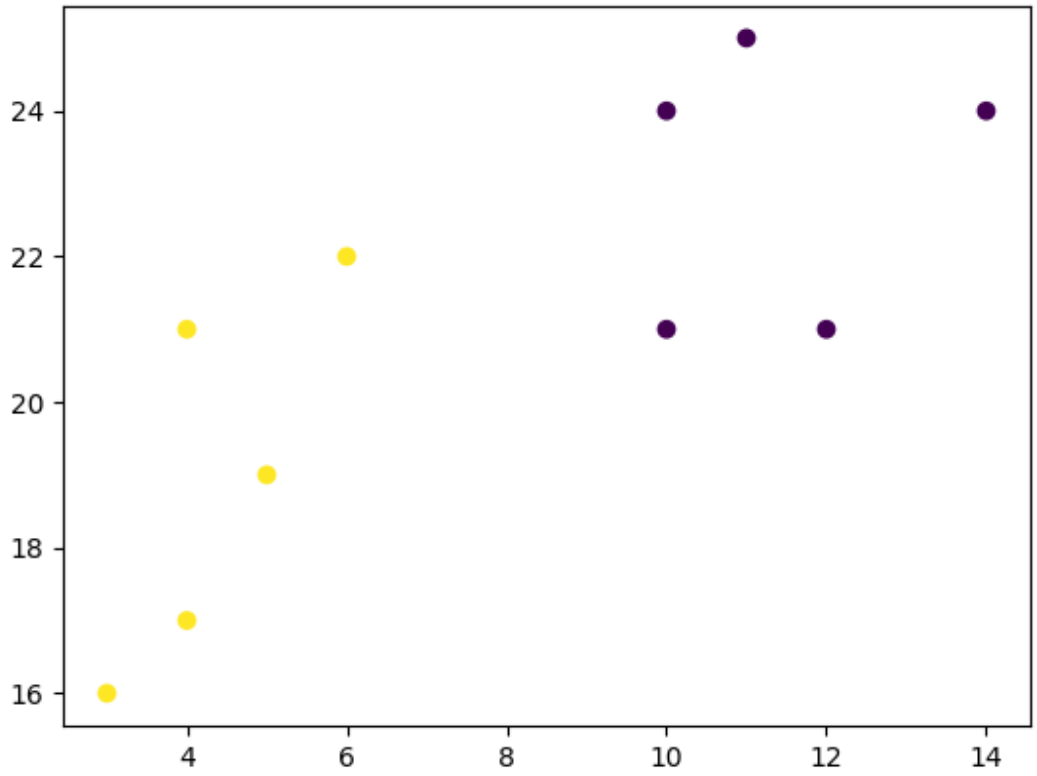
```
inertias = []

for i in range(1,11):
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(data)
    inertias.append(kmeans.inertia_)

plt.plot(range(1,11), inertias, marker='o')
plt.title('Elbow method')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```

Result:



We can see that the "elbow" on the graph above (where the interia becomes more linear) is at K=2. We can then fit our K-means algorithm one more time and plot the different clusters assigned to the data:

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(data)

plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```



Machine Learning - Bootstrap Aggregation (Bagging)

Bagging

Methods such as Decision Trees, can be prone to overfitting on the training set which can lead to wrong predictions on new data.

Bootstrap Aggregation (bagging) is a ensembling method that attempts to resolve overfitting for classification or regression problems. Bagging aims to improve the accuracy and performance of machine learning algorithms. It does this by taking random subsets of an original dataset, with replacement, and fits either a classifier (for classification) or regressor (for regression) to each subset. The predictions for each subset are then aggregated through majority vote for classification or averaging for regression, increasing prediction accuracy.

Evaluating a Base Classifier

To see how bagging can improve model performance, we must start by evaluating how the base classifier performs on the dataset. If you do not know what decision trees are review the lesson on decision trees before moving forward, as bagging is a continuation of the concept.

We will be looking to identify different classes of wines found in Sklearn's wine dataset.

Let's start by importing the necessary modules.

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
```

Next we need to load in the data and store it into X (input features) and y (target). The parameter as_frame is set equal to True so we do not lose the feature names when loading the data. (`sklearn` version older than 0.23 must skip the `as_frame` argument as it is not supported)

```
data = datasets.load_wine(as_frame = True)

X = data.data
y = data.target
```

In order to properly evaluate our model on unseen data, we need to split X and y into train and test sets. For information on splitting data, see the Train/Test lesson.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size = 0.25, random_state = 22)
```

With our data prepared, we can now instantiate a base classifier and fit it to the training data.

```
dtree = DecisionTreeClassifier(random_state = 22)
dtree.fit(X_train,y_train)
```

Result:

```
DecisionTreeClassifier(random_state=22)
```

We can now predict the class of wine the unseen test set and evaluate the model performance.

```
y_pred = dtree.predict(X_test)

print("Train data accuracy:",accuracy_score(y_true = y_train,
y_pred = dtree.predict(X_train)))
```

```
print("Test data accuracy:",accuracy_score(y_true = y_test,
y_pred = y_pred))
```

Result:

```
Train data accuracy: 1.0
Test data accuracy: 0.8222222222222222
```

Example Get your own Python Server

Import the necessary data and evaluate base classifier performance.

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier

data = datasets.load_wine(as_frame = True)

X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.25, random_state = 22)

dtree = DecisionTreeClassifier(random_state = 22)
dtree.fit(X_train,y_train)

y_pred = dtree.predict(X_test)

print("Train data accuracy:",accuracy_score(y_true = y_train, y_pred =
dtree.predict(X_train)))
print("Test data accuracy:",accuracy_score(y_true = y_test, y_pred =
y_pred))
```

The base classifier performs reasonably well on the dataset achieving 82% accuracy on the test dataset with the current parameters (Different results may occur if you do not have the `random_state` parameter set).

Now that we have a baseline accuracy for the test dataset, we can see how the Bagging Classifier out performs a single Decision Tree Classifier.

Creating a Bagging Classifier

For bagging we need to set the parameter n_estimators, this is the number of base classifiers that our model is going to aggregate together.

For this sample dataset the number of estimators is relatively low, it is often the case that much larger ranges are explored. Hyperparameter tuning is usually done with a grid search, but for now we will use a select set of values for the number of estimators.

We start by importing the necessary model.

```
from sklearn.ensemble import BaggingClassifier
```

Now lets create a range of values that represent the number of estimators we want to use in each ensemble.

```
estimator_range = [2,4,6,8,10,12,14,16]
```

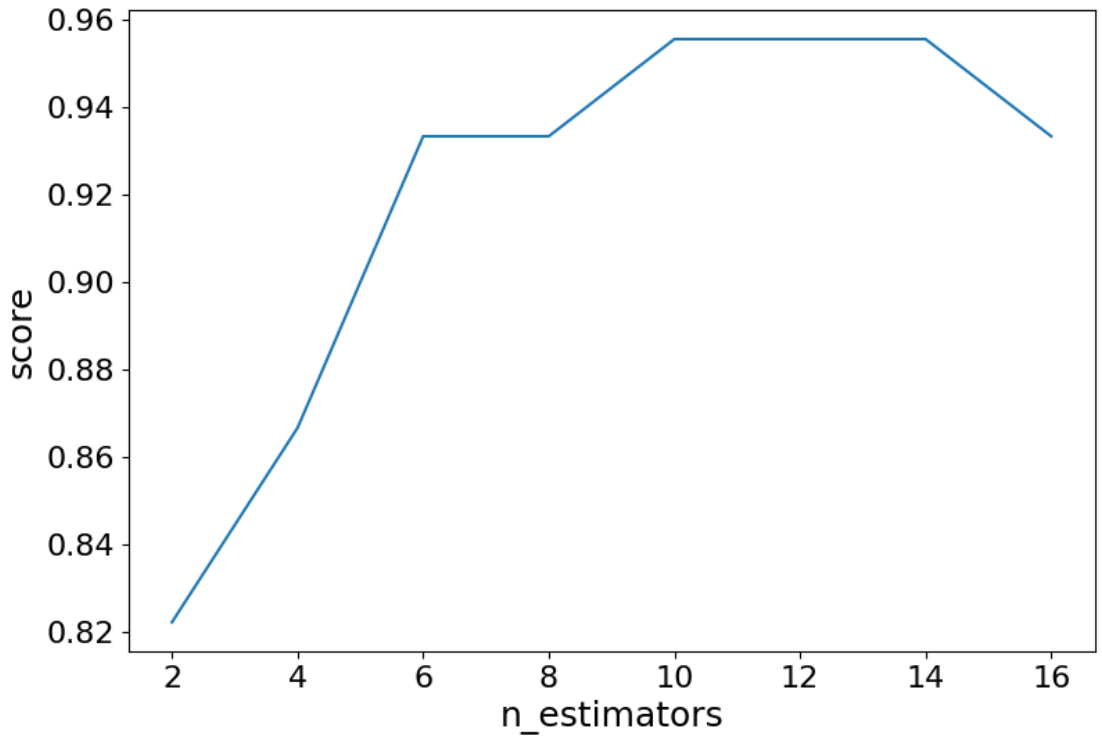
To see how the Bagging Classifier performs with differing values of n_estimators we need a way to iterate over the range of values and store the results from each ensemble. To do this we will create a for loop, storing the models and scores in separate lists for later visualizations.

Note: The default parameter for the base classifier in `BaggingClassifier` is the `DecisionTreeClassifier` therefore we do not need to set it when instantiating the bagging model.

```
models = []
scores = []

for n_estimators in estimator_range:

    # Create bagging classifier
    clf = BaggingClassifier(n_estimators = n_estimators,
random_state = 22)

    # Fit the model
    clf.fit(X_train, y_train)

    # Append the model and score to their respective list
    models.append(clf)
    scores.append(accuracy_score(y_true = y_test, y_pred =
clf.predict(X_test)))
```

With the models and scores stored, we can now visualize the improvement in model performance.

```
import matplotlib.pyplot as plt

# Generate the plot of scores against number of estimators
plt.figure(figsize=(9,6))
plt.plot(estimator_range, scores)

# Adjust labels and font (to make visable)
plt.xlabel("n_estimators", fontsize = 18)
plt.ylabel("score", fontsize = 18)
plt.tick_params(labelsize = 16)

# Visualize plot
plt.show()
```



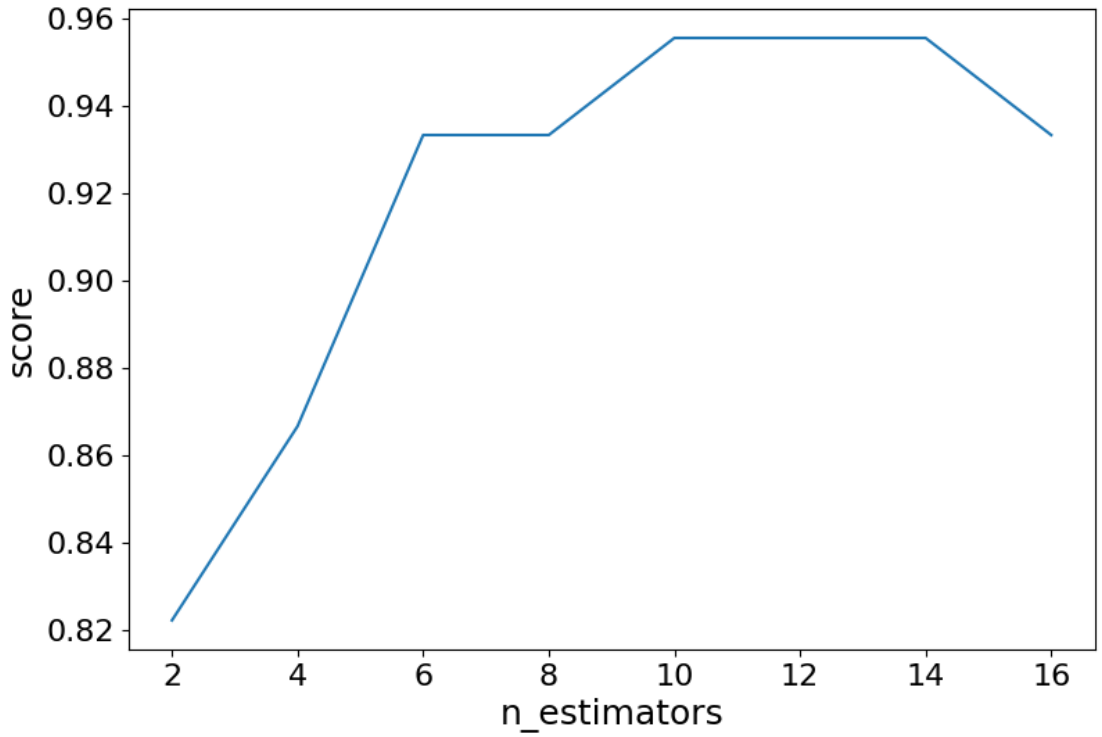
Example

Import the necessary data and evaluate the `BaggingClassifier` performance.

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

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.ensemble import BaggingClassifier

data = datasets.load_wine(as_frame = True)

X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.25, random_state = 22)

estimator_range = [2,4,6,8,10,12,14,16]

models = []
scores = []

for n_estimators in estimator_range:

    # Create bagging classifier
    clf = BaggingClassifier(n_estimators = n_estimators, random_state
= 22)

    # Fit the model
    clf.fit(X_train, y_train)

    # Append the model and score to their respective list
    models.append(clf)
    scores.append(accuracy_score(y_true = y_test, y_pred =
clf.predict(X_test)))

# Generate the plot of scores against number of estimators
plt.figure(figsize=(9,6))
plt.plot(estimator_range, scores)

# Adjust labels and font (to make visable)
plt.xlabel("n_estimators", fontsize = 18)
plt.ylabel("score", fontsize = 18)
plt.tick_params(labelsize = 16)

# Visualize plot
plt.show()
```

Result



Results Explained

By iterating through different values for the number of estimators we can see an increase in model performance from 82.2% to 95.5%. After 14 estimators the accuracy begins to drop, again if you set a different `random_state` the values you see will vary. That is why it is best practice to use cross validation to ensure stable results.

In this case, we see a 13.3% increase in accuracy when it comes to identifying the type of the wine.

Another Form of Evaluation

As bootstrapping chooses random subsets of observations to create classifiers, there are observations that are left out in the selection process. These "out-of-bag" observations can then be used to evaluate the model, similarly to that of a test set. Keep in mind, that

out-of-bag estimation can overestimate error in binary classification problems and should only be used as a compliment to other metrics.

We saw in the last exercise that 12 estimators yielded the highest accuracy, so we will use that to create our model. This time setting the parameter `oob_score` to true to evaluate the model with out-of-bag score.

Example

Create a model with out-of-bag metric.

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.ensemble import BaggingClassifier

data = datasets.load_wine(as_frame = True)

X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.25, random_state = 22)

oob_model = BaggingClassifier(n_estimators = 12, oob_score
= True,random_state = 22)

oob_model.fit(X_train, y_train)

print(oob_model.oob_score_)
```

Since the samples used in OOB and the test set are different, and the dataset is relatively small, there is a difference in the accuracy. It is rare that they would be exactly the same, again OOB should be used quick means for estimating error, but is not the only evaluation metric.

Generating Decision Trees from Bagging Classifier

As was seen in the Decision Tree lesson, it is possible to graph the decision tree the model created. It is also possible to see the individual decision trees that went into the

aggregated classifier. This helps us to gain a more intuitive understanding on how the bagging model arrives at its predictions.

Note: This is only functional with smaller datasets, where the trees are relatively shallow and narrow making them easy to visualize.

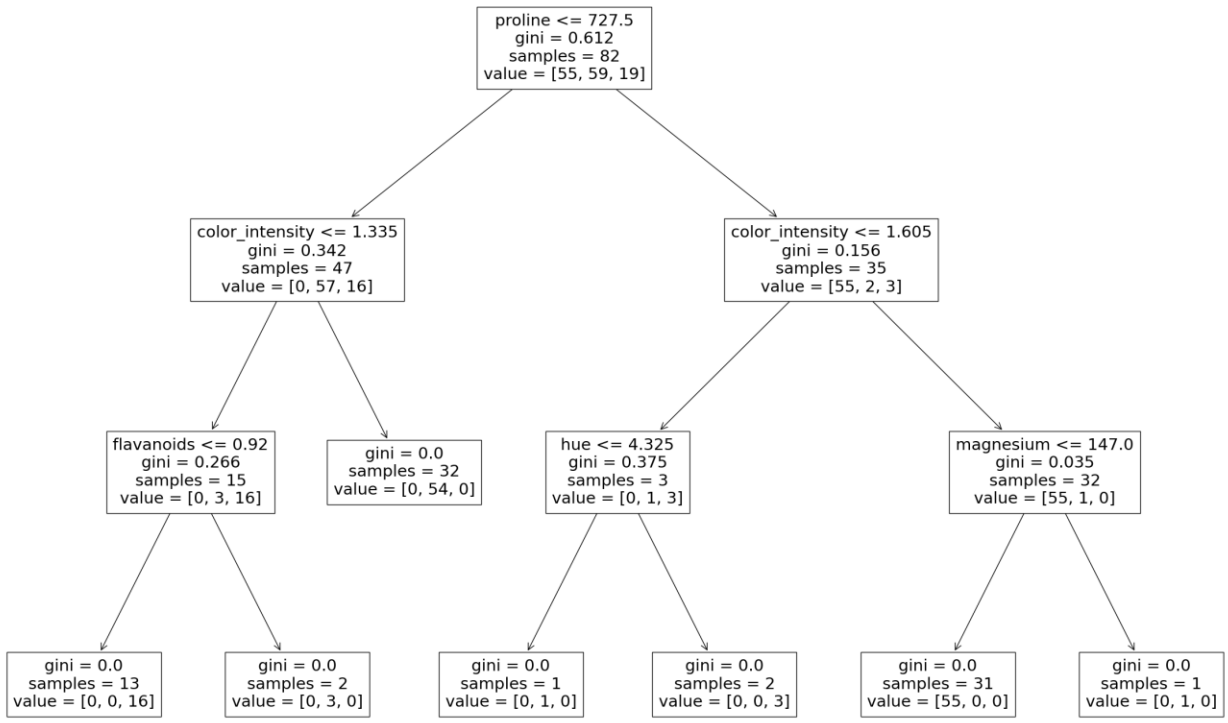
We will need to import `plot_tree` function from `sklearn.tree`. The different trees can be graphed by changing the estimator you wish to visualize.

Example

Generate Decision Trees from Bagging Classifier

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import plot_tree

X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
= 0.25, random_state = 22)

clf = BaggingClassifier(n_estimators = 12, oob_score = True,random_state
= 22)

clf.fit(X_train, y_train)

plt.figure(figsize=(30, 20))

plot_tree(clf.estimators_[0], feature_names = X.columns)
```

Result



Machine Learning - Cross Validation

Cross Validation

When adjusting models we are aiming to increase overall model performance on unseen data. Hyperparameter tuning can lead to much better performance on test sets. However, optimizing parameters to the test set can lead information leakage causing the model to preform worse on unseen data. To correct for this we can perform cross validation.

To better understand CV, we will be performing different methods on the iris dataset. Let us first load in and separate the data.

```
from sklearn import datasets

X, y = datasets.load_iris(return_X_y=True)
```

There are many methods to cross validation, we will start by looking at k-fold cross validation.

K-Fold

The training data used in the model is split, into k number of smaller sets, to be used to validate the model. The model is then trained on k-1 folds of training set. The remaining fold is then used as a validation set to evaluate the model.

As we will be trying to classify different species of iris flowers we will need to import a classifier model, for this exercise we will be using a `DecisionTreeClassifier`. We will also need to import CV modules from `sklearn`.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import KFold, cross_val_score
```

With the data loaded we can now create and fit a model for evaluation.

```
clf = DecisionTreeClassifier(random_state=42)
```

Now let's evaluate our model and see how it performs on each *k*-fold.

```
k_folds = KFold(n_splits = 5)

scores = cross_val_score(clf, X, y, cv = k_folds)
```

It is also good pratice to see how CV performed overall by averaging the scores for all folds.

Example Get your own Python Server

Run k-fold CV:

```
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import KFold, cross_val_score

X, y = datasets.load_iris(return_X_y=True)

clf = DecisionTreeClassifier(random_state=42)

k_folds = KFold(n_splits = 5)

scores = cross_val_score(clf, X, y, cv = k_folds)
```

```
print("Cross Validation Scores: ", scores)
print("Average CV Score: ", scores.mean())
print("Number of CV Scores used in Average: ", len(scores))
```

Stratified K-Fold

In cases where classes are imbalanced we need a way to account for the imbalance in both the train and validation sets. To do so we can stratify the target classes, meaning that both sets will have an equal proportion of all classes.

Example

```
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import StratifiedKFold, cross_val_score

X, y = datasets.load_iris(return_X_y=True)

clf = DecisionTreeClassifier(random_state=42)

sk_folds = StratifiedKFold(n_splits = 5)

scores = cross_val_score(clf, X, y, cv = sk_folds)

print("Cross Validation Scores: ", scores)
print("Average CV Score: ", scores.mean())
print("Number of CV Scores used in Average: ", len(scores))
```

While the number of folds is the same, the average CV increases from the basic k-fold when making sure there is stratified classes.

Leave-One-Out (LOO)

Instead of selecting the number of splits in the training data set like k-fold LeaveOneOut, utilize 1 observation to validate and n-1 observations to train. This method is an exaustive technique.

Example

Run LOO CV:

```
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import LeaveOneOut, cross_val_score

X, y = datasets.load_iris(return_X_y=True)

clf = DecisionTreeClassifier(random_state=42)

loo = LeaveOneOut()

scores = cross_val_score(clf, X, y, cv = loo)

print("Cross Validation Scores: ", scores)
print("Average CV Score: ", scores.mean())
print("Number of CV Scores used in Average: ", len(scores))
```

We can observe that the number of cross validation scores performed is equal to the number of observations in the dataset. In this case there are 150 observations in the iris dataset.

The average CV score is 94%.

Leave-P-Out (LPO)

Leave-P-Out is simply a nuanced diffence to the Leave-One-Out idea, in that we can select the number of p to use in our validation set.

Example

Run LPO CV:

```
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import LeavePOut, cross_val_score

X, y = datasets.load_iris(return_X_y=True)

clf = DecisionTreeClassifier(random_state=42)

lpo = LeavePOut(p=2)

scores = cross_val_score(clf, X, y, cv = lpo)

print("Cross Validation Scores: ", scores)
print("Average CV Score: ", scores.mean())
print("Number of CV Scores used in Average: ", len(scores))
```

As we can see this is an exhaustive method we many more scores being calculated than Leave-One-Out, even with a p = 2, yet it achieves roughly the same average CV score.

Shuffle Split

Unlike `KFold`, `ShuffleSplit` leaves out a percentage of the data, not to be used in the train or validation sets. To do so we must decide what the train and test sizes are, as well as the number of splits.

Example

Run Shuffle Split CV:

```
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import ShuffleSplit, cross_val_score

X, y = datasets.load_iris(return_X_y=True)

clf = DecisionTreeClassifier(random_state=42)

ss = ShuffleSplit(train_size=0.6, test_size=0.3, n_splits = 5)

scores = cross_val_score(clf, X, y, cv = ss)

print("Cross Validation Scores: ", scores)
```

```
print("Average CV Score: ", scores.mean())
print("Number of CV Scores used in Average: ", len(scores))
```

Ending Notes

These are just a few of the CV methods that can be applied to models. There are many more cross validation classes, with most models having their own class. Check out sklearns cross validation for more CV options.

Machine Learning - AUC - ROC Curve

AUC - ROC Curve

In classification, there are many different evaluation metrics. The most popular is **accuracy**, which measures how often the model is correct. This is a great metric because it is easy to understand and getting the most correct guesses is often desired. There are some cases where you might consider using another evaluation metric.

Another common metric is **AUC**, area under the receiver operating characteristic (**ROC**) curve. The Reciever operating characteristic curve plots the true positive (**TP**) rate versus the false positive (**FP**) rate at different classification thresholds. The thresholds are different probability cutoffs that separate the two classes in binary classification. It uses probability to tell us how well a model separates the classes.

Imbalanced Data

Suppose we have an imbalanced data set where the majority of our data is of one value. We can obtain high accuracy for the model by predicting the majority class.

Example

```
import numpy as np
from sklearn.metrics import accuracy_score, confusion_matrix,
roc_auc_score, roc_curve

n = 10000
```

```
ratio = .95
n_0 = int((1-ratio) * n)
n_1 = int(ratio * n)

y = np.array([0] * n_0 + [1] * n_1)
# below are the probabilities obtained from a hypothetical model that
always predicts the majority class
# probability of predicting class 1 is going to be 100%
y_proba = np.array([1]*n)
y_pred = y_proba > .5

print(f'accuracy score: {accuracy_score(y, y_pred)}')
cf_mat = confusion_matrix(y, y_pred)
print('Confusion matrix')
print(cf_mat)
print(f'class 0 accuracy: {cf_mat[0][0]/n_0}')
print(f'class 1 accuracy: {cf_mat[1][1]/n_1}')
```

Although we obtain a very high accuracy, the model provided no information about the data so it's not useful. We accurately predict class 1 100% of the time while inaccurately predict class 0 0% of the time. At the expense of accuracy, it might be better to have a model that can somewhat separate the two classes.

Example

```
# below are the probabilities obtained from a hypothetical model that
doesn't always predict the mode
y_proba_2 = np.array(
    np.random.uniform(0, .7, n_0).tolist() +
    np.random.uniform(.3, 1, n_1).tolist()
)
y_pred_2 = y_proba_2 > .5

print(f'accuracy score: {accuracy_score(y, y_pred_2)}')
cf_mat = confusion_matrix(y, y_pred_2)
print('Confusion matrix')
print(cf_mat)
print(f'class 0 accuracy: {cf_mat[0][0]/n_0}')
print(f'class 1 accuracy: {cf_mat[1][1]/n_1}')
```

For the second set of predictions, we do not have as high of an accuracy score as the first but the accuracy for each class is more balanced. Using accuracy as an evaluation

metric we would rate the first model higher than the second even though it doesn't tell us anything about the data.

In cases like this, using another evaluation metric like AUC would be preferred.

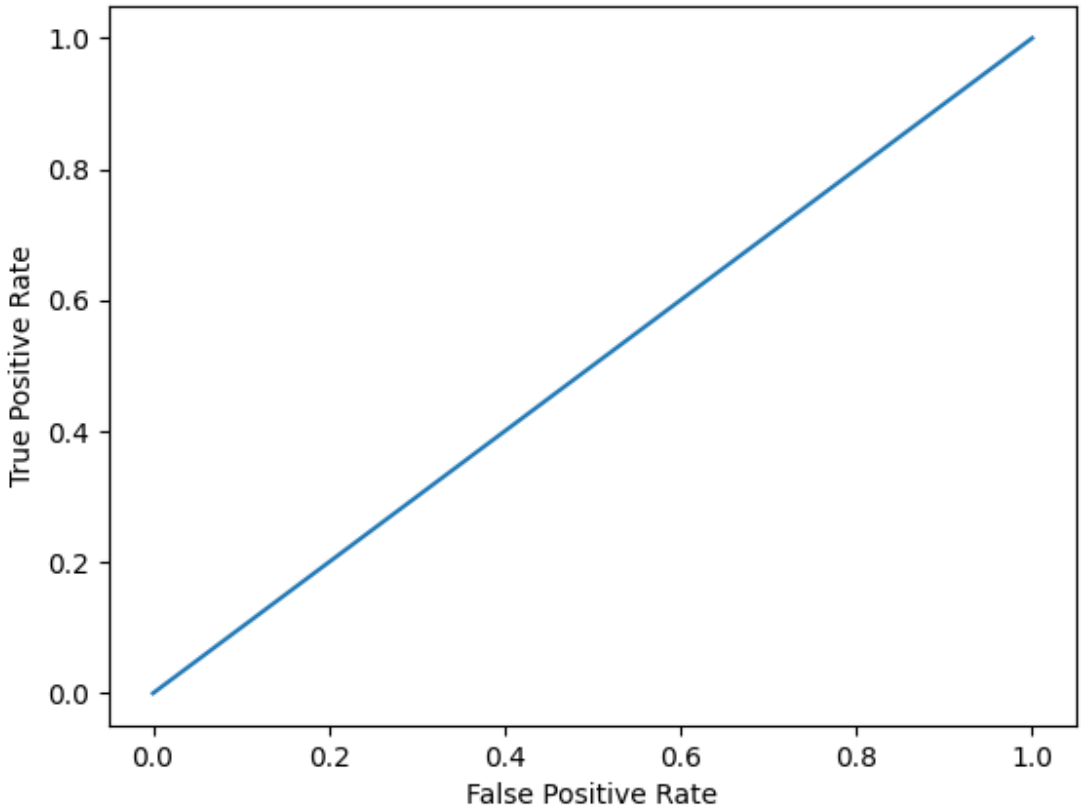
```
import matplotlib.pyplot as plt

def plot_roc_curve(true_y, y_prob):
    """
    plots the roc curve based of the probabilities
    """

    fpr, tpr, thresholds = roc_curve(true_y, y_prob)
    plt.plot(fpr, tpr)
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
```

Example

Model 1:

```
plot_roc_curve(y, y_proba)
print(f'model 1 AUC score: {roc_auc_score(y, y_proba)}')
```

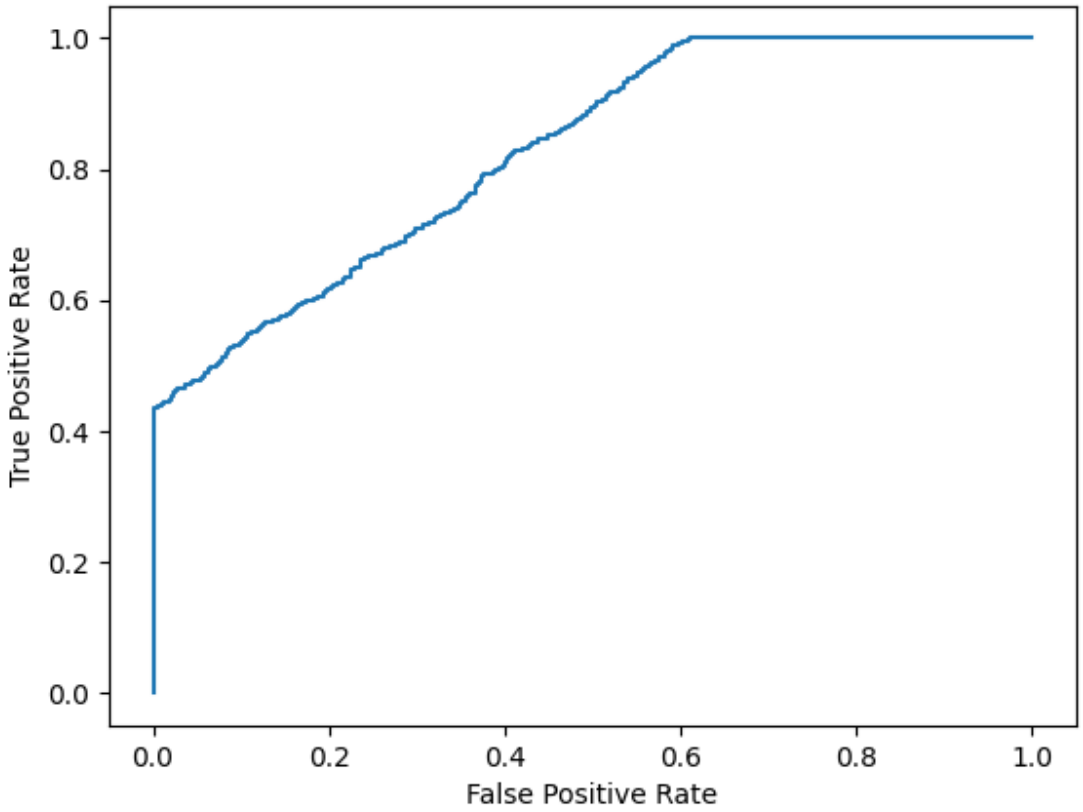
Result



xample

Model 2:

```
plot_roc_curve(y, y_proba_2)
print(f'model 2 AUC score: {roc_auc_score(y, y_proba_2)}')
```



An AUC score of around .5 would mean that the model is unable to make a distinction between the two classes and the curve would look like a line with a slope of 1. An AUC score closer to 1 means that the model has the ability to separate the two classes and the curve would come closer to the top left corner of the graph.

Probabilities

Because AUC is a metric that utilizes probabilities of the class predictions, we can be more confident in a model that has a higher AUC score than one with a lower score even if they have similar accuracies.

In the data below, we have two sets of probabilites from hypothetical models. The first has probabilities that are not as "confident" when predicting the two classes (the probabilities are

close to .5). The second has probabilities that are more "confident" when predicting the two classes (the probabilities are close to the extremes of 0 or 1).

Example

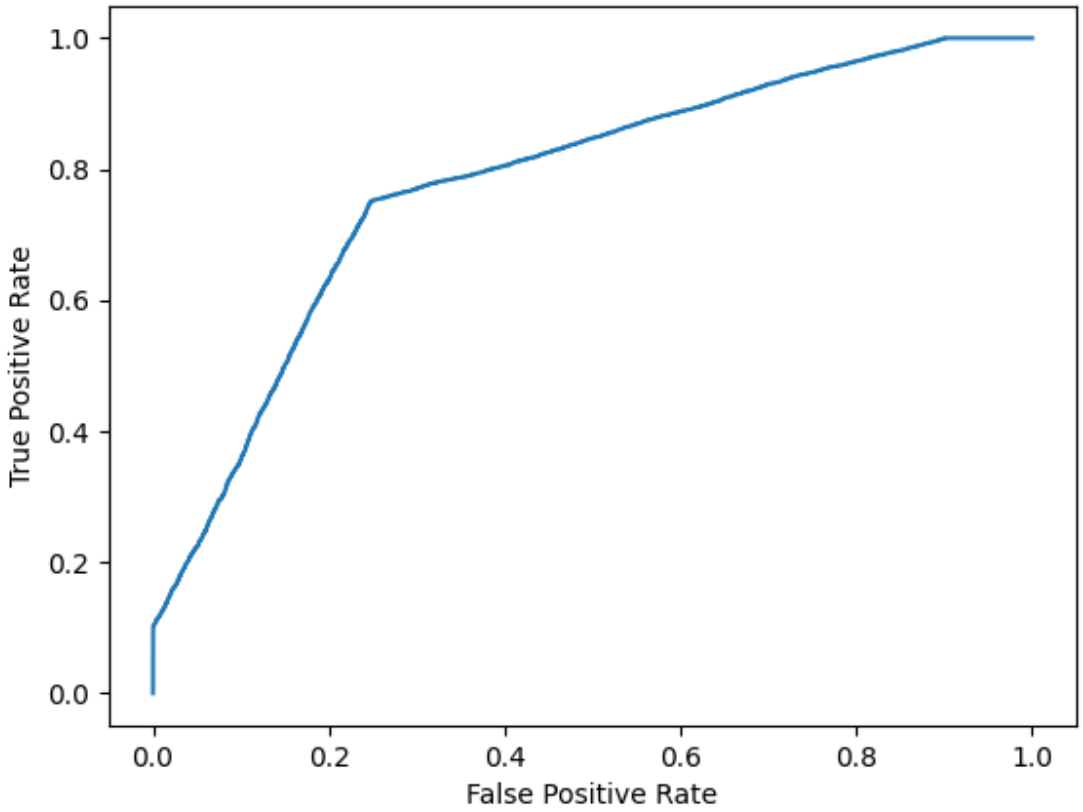
```
import numpy as np

n = 10000
y = np.array([0] * n + [1] * n)
#
y_prob_1 = np.array(
    np.random.uniform(.25, .5, n//2).tolist() +
    np.random.uniform(.3, .7, n).tolist() +
    np.random.uniform(.5, .75, n//2).tolist()
)
y_prob_2 = np.array(
    np.random.uniform(0, .4, n//2).tolist() +
    np.random.uniform(.3, .7, n).tolist() +
    np.random.uniform(.6, 1, n//2).tolist()
)

print(f'model 1 accuracy score: {accuracy_score(y, y_prob_1>.5)}')
print(f'model 2 accuracy score: {accuracy_score(y, y_prob_2>.5)}')

print(f'model 1 AUC score: {roc_auc_score(y, y_prob_1)}')
print(f'model 2 AUC score: {roc_auc_score(y, y_prob_2)}')
```

Example

Plot model 1:

```
plot_roc_curve(y, y_prob_1)
```

Result



Example

Plot model 2:

```
fpr, tpr, thresholds = roc_curve(y, y_prob_2)
plt.plot(fpr, tpr)
```

Result

Machine Learning - K-nearest neighbors (KNN)

KNN

KNN is a simple, supervised machine learning (ML) algorithm that can be used for classification or regression tasks - and is also frequently used in missing value imputation. It is based on the idea that the observations closest to a given data point are the most "similar" observations in a data set, and we can therefore classify unforeseen points based on the values of the closest existing points. By choosing *K*, the user can select the number of nearby observations to use in the algorithm.

Here, we will show you how to implement the KNN algorithm for classification, and show how different values of *K* affect the results.

How does it work?

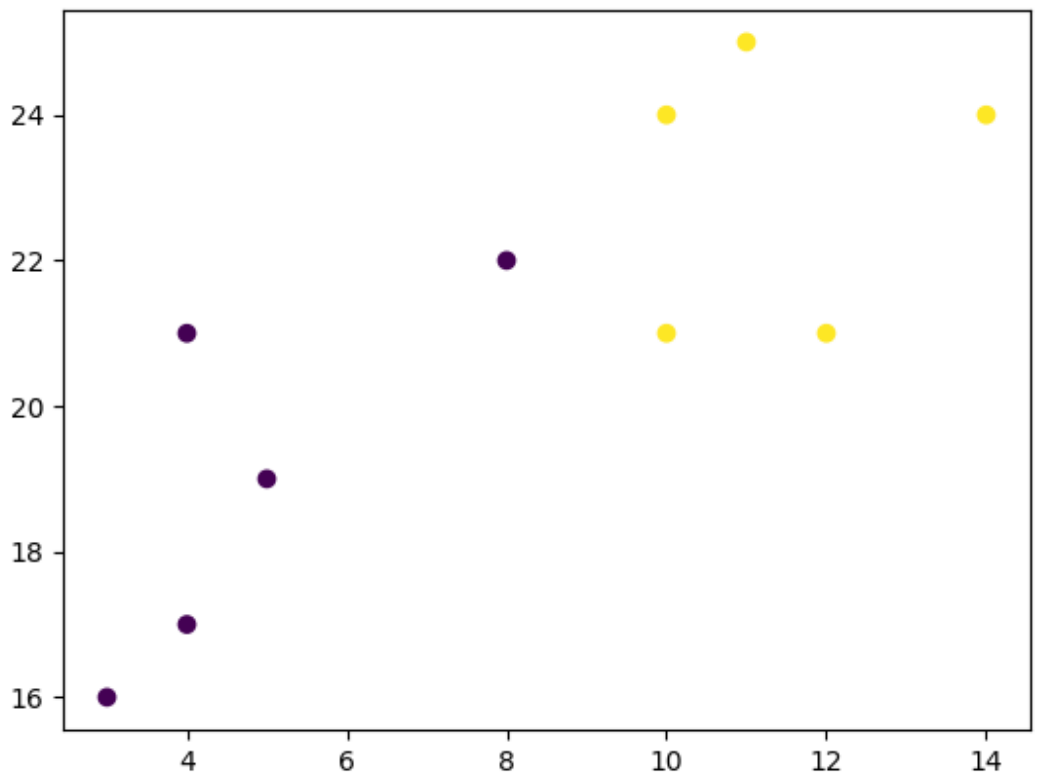
K is the number of nearest neighbors to use. For classification, a majority vote is used to determined which class a new observation should fall into. Larger values of *K* are often more robust to outliers and produce more stable decision boundaries than very small values (*K=3* would be better than *K=1*, which might produce undesirable results.

Example

Start by visualizing some data points:

```
import matplotlib.pyplot as plt

x = [4, 5, 10, 4, 3, 11, 14 , 8, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
classes = [0, 0, 1, 0, 0, 1, 1, 0, 1, 1]

plt.scatter(x, y, c=classes)
plt.show()
```

Result



Now we fit the KNN algorithm with K=1:

```
from sklearn.neighbors import KNeighborsClassifier

data = list(zip(x, y))
knn = KNeighborsClassifier(n_neighbors=1)

knn.fit(data, classes)
```

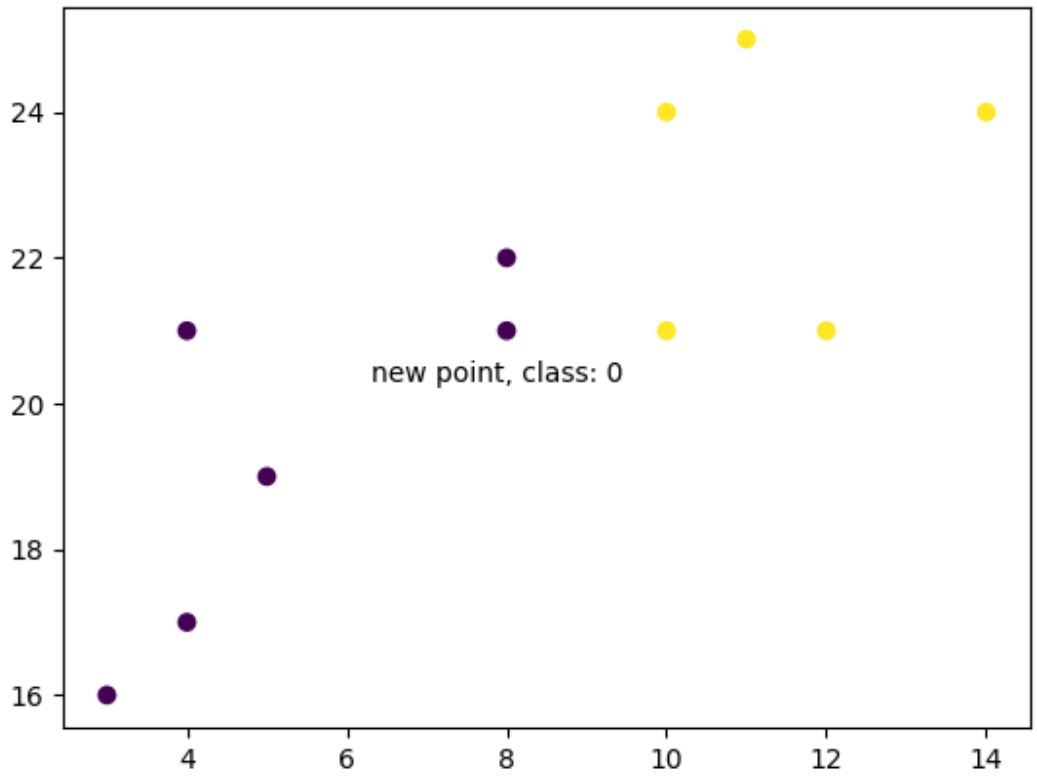
And use it to classify a new data point:

Example

```
new_x = 8
new_y = 21
new_point = [(new_x, new_y)]

prediction = knn.predict(new_point)

plt.scatter(x + [new_x], y + [new_y], c=classes + [prediction[0]])
```

```
plt.text(x=new_x-1.7, y=new_y-0.7, s=f"new point, class: {prediction[0]}")
plt.show()
```

Result

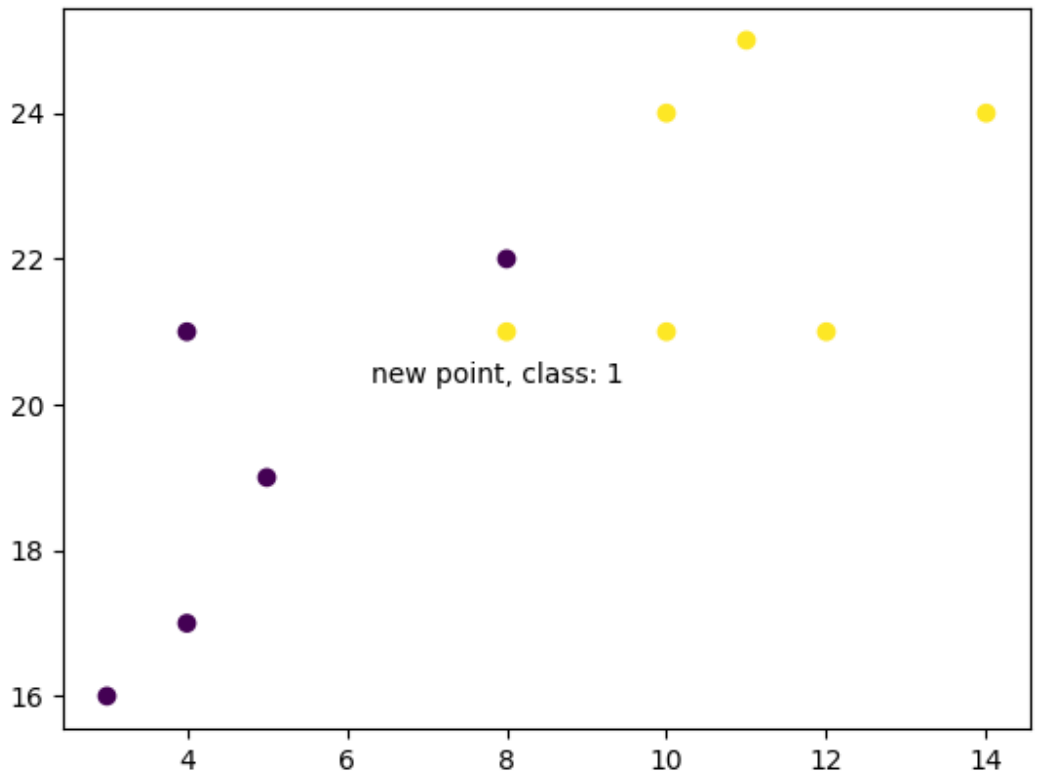


Now we do the same thing, but with a higher K value which changes the prediction:

Example

```
knn = KNeighborsClassifier(n_neighbors=5)

knn.fit(data, classes)

prediction = knn.predict(new_point)

plt.scatter(x + [new_x], y + [new_y], c=classes + [prediction[0]])
plt.text(x=new_x-1.7, y=new_y-0.7, s=f"new point, class: {prediction[0]}")
plt.show()
```

Result



Example Explained

Import the modules you need.

You can learn about the Matplotlib module in our "Matplotlib Tutorial.

scikit-learn is a popular library for machine learning in Python.

```
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
```

Create arrays that resemble variables in a dataset. We have two input features (`x` and `y`) and then a target class (`class`). The input features that are pre-labeled with our target

class will be used to predict the class of new data. Note that while we only use two input features here, this method will work with any number of variables:

```
x = [4, 5, 10, 4, 3, 11, 14 , 8, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
classes = [0, 0, 1, 0, 0, 1, 1, 0, 1, 1]
```

Turn the input features into a set of points:

```
data = list(zip(x, y))
print(data)
```

Result:

```
[(4, 21), (5, 19), (10, 24), (4, 17), (3, 16), (11, 25),
(14, 24), (8, 22), (10, 21), (12, 21)]
```

Using the input features and target class, we fit a KNN model on the model using 1 nearest neighbor:

```
knn = KNeighborsClassifier(n_neighbors=1)
knn.fit(data, classes)
```

Then, we can use the same KNN object to predict the class of new, unforeseen data points. First we create new x and y features, and then call `knn.predict()` on the new data point to get a class of 0 or 1:

```
new_x = 8
new_y = 21
new_point = [(new_x, new_y)]
prediction = knn.predict(new_point)
print(prediction)
```

Result:

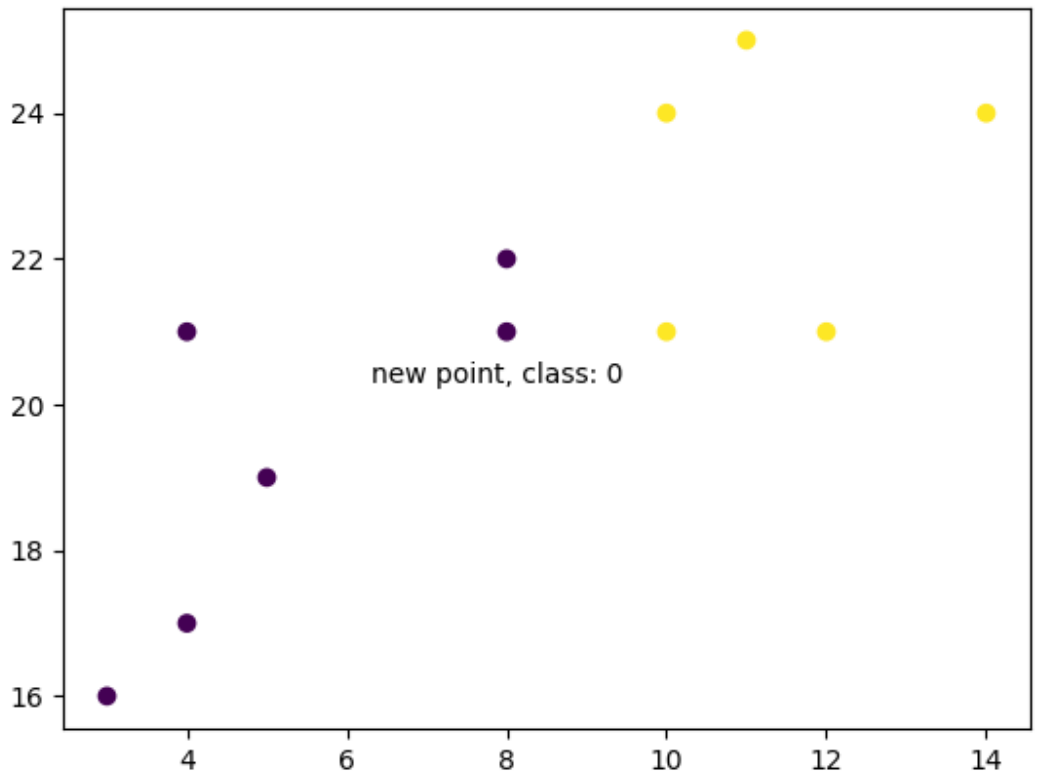
```
[0]
```

When we plot all the data along with the new point and class, we can see it's been labeled blue with the `1` class. The text annotation is just to highlight the location of the new point:

```
plt.scatter(x + [new_x], y + [new_y], c=classes +
[prediction[0]])
plt.text(x=new_x-1.7, y=new_y-0.7, s=f"new point, class:
```

```
{prediction[0]}")
plt.show()
```

Result:



However, when we changes the number of neighbors to 5, the number of points used to classify our new point changes. As a result, so does the classification of the new point:

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(data, classes)
prediction = knn.predict(new_point)
print(prediction)
```

Result:

```
[1]
```

When we plot the class of the new point along with the older points, we note that the color has changed based on the associated class label:

```
plt.scatter(x + [new_x], y + [new_y], c=classes +
[prediction[0]])
plt.text(x=new_x-1.7, y=new_y-0.7, s=f"new point, class:
{prediction[0]}")
plt.show()
```

Result: