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  
  - numbers that are limited to integers. Example: The number of cars passing by.
* Continuous Data  
  - numbers that are of infinite value. 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.

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

## 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:

## Median

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

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

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

## Mode

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

## 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.

## Variance

Variance is another number that indicates how spread out the values are.

In fact, if you take the square root of the variance, you get the standard deviation!

Or the other way around, if you multiply the standard deviation by itself, you get the variance!

To calculate the variance you have to do as follows:

1. Find the mean:

(32+111+138+28+59+77+97) / 7 = 77.4

2. For each value: find the difference from the mean:

 32 - 77.4 = -45.4  
111 - 77.4 =  33.6  
138 - 77.4 =  60.6  
 28 - 77.4 = -49.4  
 59 - 77.4 = -18.4  
 77 - 77.4 = - 0.4  
 97 - 77.4 =  19.6

3. For each difference: find the square value:

(-45.4)2 = 2061.16  
 (33.6)2 = 1128.96  
 (60.6)2 = 3672.36  
(-49.4)2 = 2440.36  
(-18.4)2 =  338.56  
(- 0.4)2 =    0.16  
 (19.6)2 =  384.16

4. The variance is the average number of these squared differences:

(2061.16+1128.96+3672.36+2440.36+338.56+0.16+384.16) / 7 = 1432.2

Standard Deviation is often represented by the symbol Sigma: σ

Variance is often represented by the symbol Sigma Squared: σ2

## 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 of the ages of all the people that live in 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:

### 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.

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.

## Normal Data Distribution

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.

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.

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.

## 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:

The x array represents the age of each car.

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

The y array represents the speed of each car.

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

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.

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:

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.

## 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()

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.

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.

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

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)

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?

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)

The result: 0.013 indicates a very bad relationship, and tells us that this data set is not suitable for linear 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()

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()

**Example Explained**

Import the modules you need.

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:

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))

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:

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

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?

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)))

The result: 0.00995 indicates a very bad relationship, and tells us that this data set is not suitable for polynomial regression.

## Multiple Regression

Multiple regression is like [linear regression](https://www.w3schools.com/python/python_ml_linear_regression.asp), but with more than one independent value, meaning that we try to predict a value based on ****two or more**** variables.

## How Does it Work?

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

import pandas

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](https://www.w3schools.com/python/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)

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.

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 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 1 cm3, the CO2 emission increases by 0.00780526 g.

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

We have already predicted that if a car with a 1300cm3 engine weighs 2300kg, the CO2 emission will be approximately 107g.

What if we increase the weight with 1000kg?

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)

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

## 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](https://www.w3schools.com/python/python_ml_multiple_regression.asp), but this time the ****volume**** column contains values in liters instead of cm3 (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](https://www.w3schools.com/python/trypandas.asp?filename=demo_ml_scale_mean1" \t "/home/jean/Documents\\x/_blank)) / [238.74](https://www.w3schools.com/python/trypandas.asp?filename=demo_ml_scale_std1" \t "/home/jean/Documents\\x/_blank) = -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](https://www.w3schools.com/python/trypandas.asp?filename=demo_ml_scale_mean2" \t "/home/jean/Documents\\x/_blank)) / [0.38](https://www.w3schools.com/python/trypandas.asp?filename=demo_ml_scale_std2" \t "/home/jean/Documents\\x/_blank) = -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.

Scale all values in the Weight and Volume columns:

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)

## Predict CO2 Values

The task in the [Multiple Regression chapter](https://www.w3schools.com/python/python_ml_multiple_regression.asp) 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:

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)

# 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.

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()

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:

plt.scatter(train\_x, train\_y)  
plt.show()

## Fit the Data Set

What does the data set look like? In my opinion I think the best fit would be a [polynomial regression](https://www.w3schools.com/python/python_ml_polynomial_regression.asp), 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:

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()

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.

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.

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.

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

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))

print(mymodel(5))

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

## 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.

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.

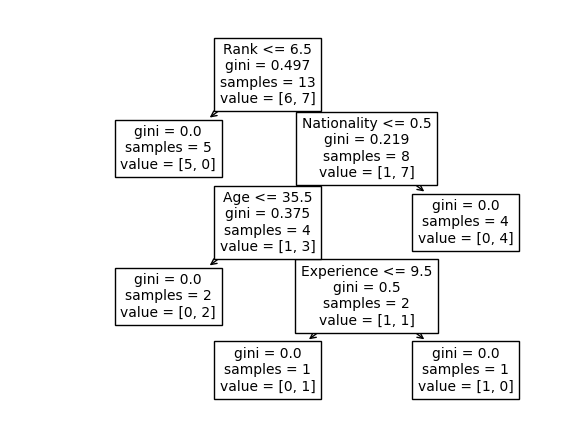
X is the feature columns, y is the target column:

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:

Create and display a Decision Tree:

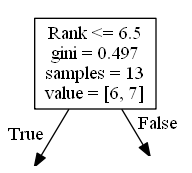
import pandas  
from sklearn import tree  
from sklearn.tree import DecisionTreeClassifier  
import matplotlib.pyplot as plt  
df = pandas.read\_csv("data.csv")  
d = {'UK': 0, 'USA': 1, 'N': 2}  
df['Nationality'] = df['Nationality'].map(d)  
d = {'YES': 1, 'NO': 0}  
df['Go'] = df['Go'].map(d)  
features = ['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

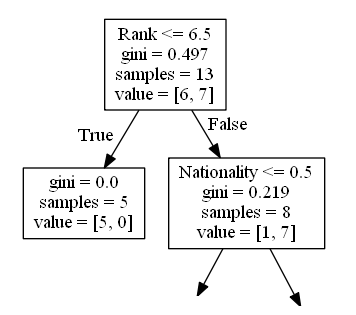
There are many ways to split the samples, we use the GINI method in this tutorial.

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:

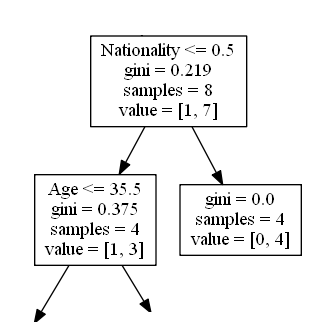
### Nationality

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).

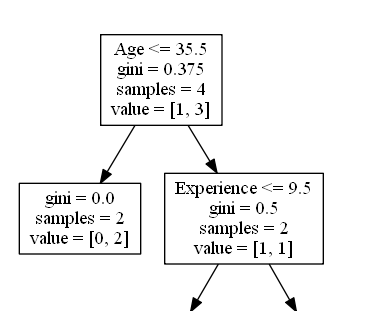
value = [1, 3] means that of these 4 comedians, 1 will get a "NO" and 3 will get a "GO".

### 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:

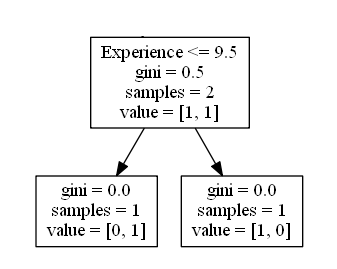
### Experience

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?

Use predict() method to predict new values:

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

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.

## 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 = [False, True])

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 = [False, True])  
cm\_display.plot()  
plt.show()