Module 7 : Machine Learning

Session 7C : Machine Learning – Who Would Survive the Titanic?

Dr Daniel Chalk

"My lecture will go on"

NIHR | Applied Research Collaboration South West Peninsula

@peninsula_ARC #HSMA4

HSMA 4

HSMA 4 Everyone

With Thanks

Big thanks to my colleague, Mike Allen, for developing the notebooks upon which much of today’s content is based.

You should check out his full materials as further reading available here : https://bit.ly/titanic_machine_learning

Simpler Machine Learning Algorithms

In the first ML session, you learned about some key concepts in Machine Learning (as well as my taste in movies).

In this session we’ll start looking at some “simpler” (non-Neural Network) Machine Learning algorithms, such as Logistic Regression.

We’ll apply these algorithms to see if we can make predictions about who would have survived the sinking of the Titanic.

The Titanic ML Problem

Kaggle (https://www.kaggle.com/) is an online Data Science community that hosts resources such as data sets, web-hosted data science environments, and *competitions* in which companies post problems, and participants compete to develop Machine Learning solutions, often with cash prizes.

“Titanic – Machine Learning from Disaster” (https://www.kaggle.com/c/titanic) is one of the classic competitions that is a really useful one with which to engage when learning about Machine Learning.

The challenge is to use ML to predict which passengers survived the sinking of the Titanic, based on data about the passengers. People compete with their approaches in terms of the accuracy with which their ML algorithm can determine which passengers survived and which passengers did not survive.

ML Key Concept Refresher

We’ll come back to the Titanic data in a bit. But let’s start with a quick reminder of some of the key concepts we covered in the first session of this module.

Features

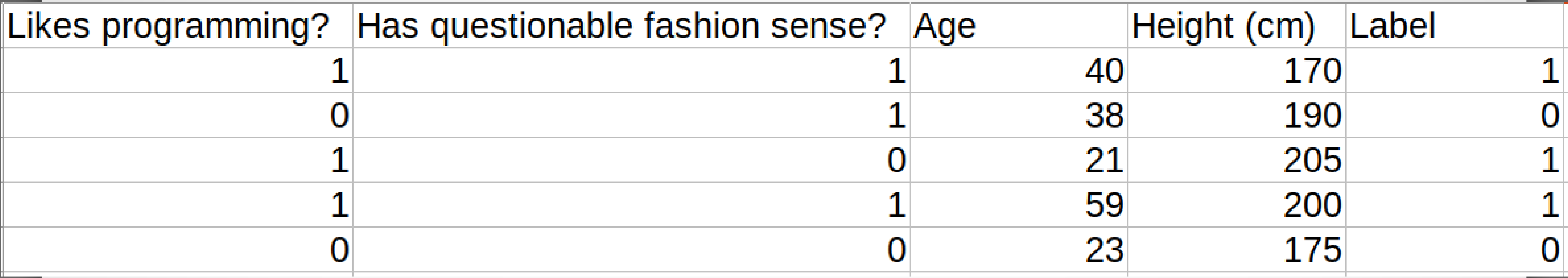
When we say that we want to teach a machine to make a prediction or classification, we mean that we want it to predict an *output* (or *label*) given a certain set of *inputs*. In ML terminology, inputs are referred to as *features.*

Examples :

- Predicting whether a patient has a certain condition or can receive a treatment based on aspects of their health, demographics, time since onset etc
- Predicting the likelihood of someone reoffending based on features such as the nature of their original crime, whether they served a custodial sentence etc
- Classifying whether the image presented is an image of a flower
- Classifying whether the text presented is positive or negative in tone.

Feature Weighting

Classifying whether someone is a PenCHORDian

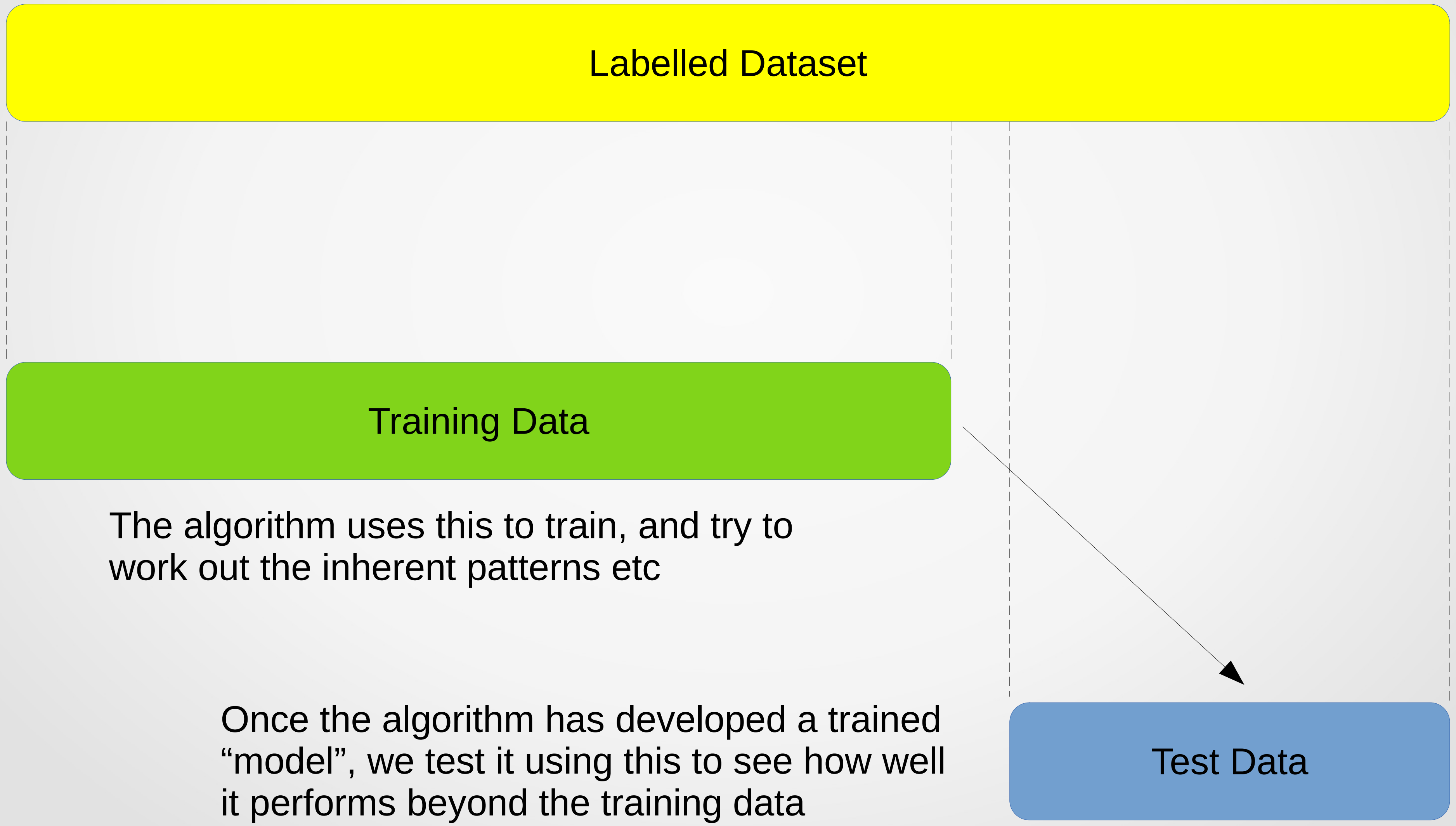


Likes programming?	Has questionable fashion sense?	Age	Height (cm)	Label
1	1	40	170	1
0	1	38	190	0
1	0	21	205	1
1	1	59	200	1
0	0	23	175	0

In Machine Learning, features become *weighted* by the algorithm being used, so that their relative contribution to making a prediction of the label can be determined. This allows us to determine which features are more or less important for making a prediction.

Looking at the above, which features do you think are more important for predicting if someone is a PenCHORDian?

Train / Test Splits

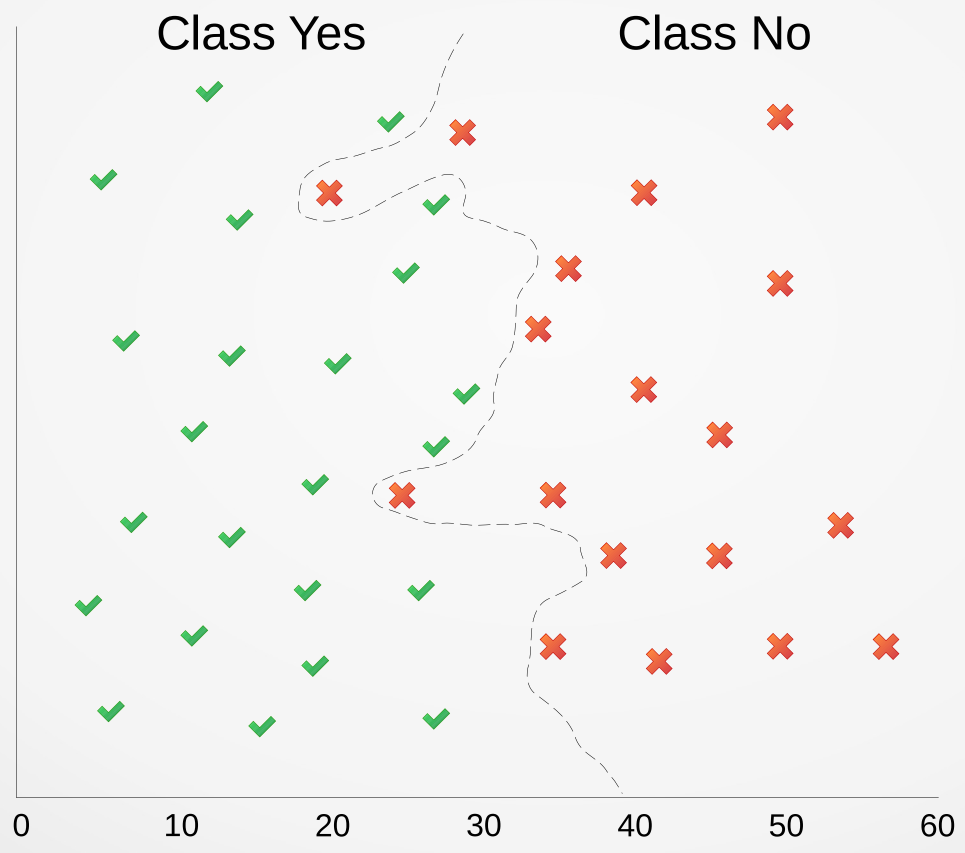


The algorithm uses this to train, and try to work out the inherent patterns etc

Once the algorithm has developed a trained “model”, we test it using this to see how well it performs beyond the training data

Overfitting

A model that has been overfitted.



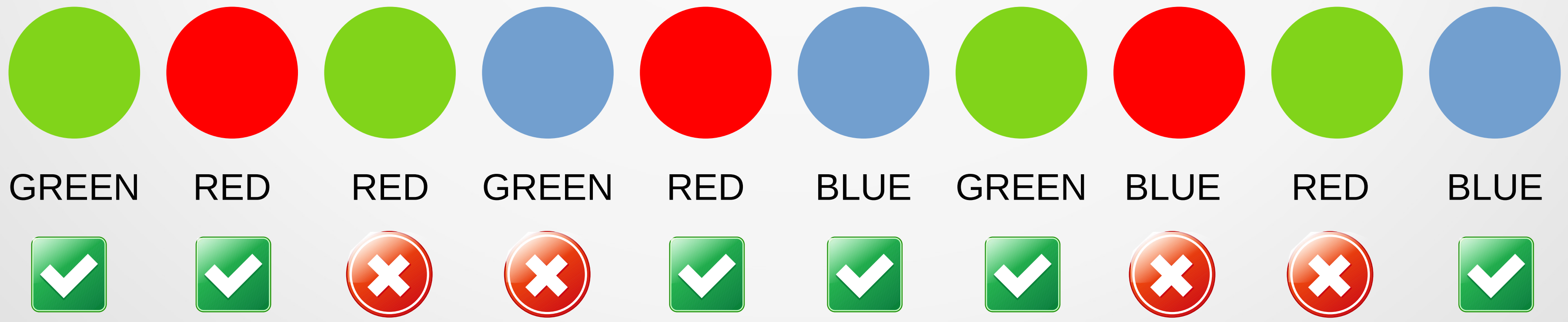
It’s got everything correctly classified in the training set, but let’s now add some other points from the wider data...

Accuracy

When we’re building a Classifier (a ML model that makes a classification prediction), the three most commonly used metrics to assess performance are Accuracy, Precision and Recall.

Accuracy gives the percentage of predictions that the model makes correctly in the Test Set. In other words :

accuracy = total correct / number of predictions



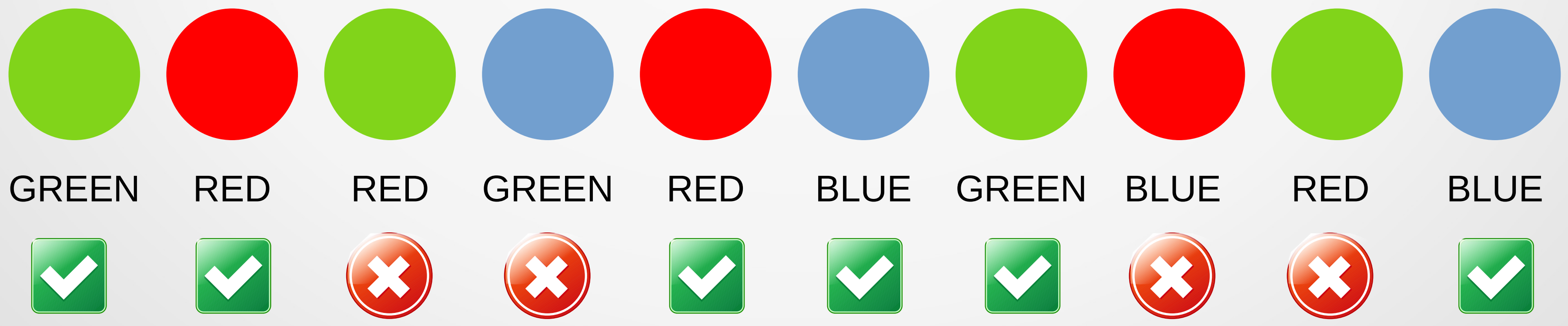
What is the accuracy of these results?

Precision

Precision gives the percentage of correct predictions for the classification of interest, out of the total number of times this classification was predicted (both correctly and incorrectly). So :

$\text{precision}_c = \text{true positives}_c \,/\, (\text{true positives}_c + \text{false positives}_c)$

Precision is a useful metric when classes aren't evenly represented.



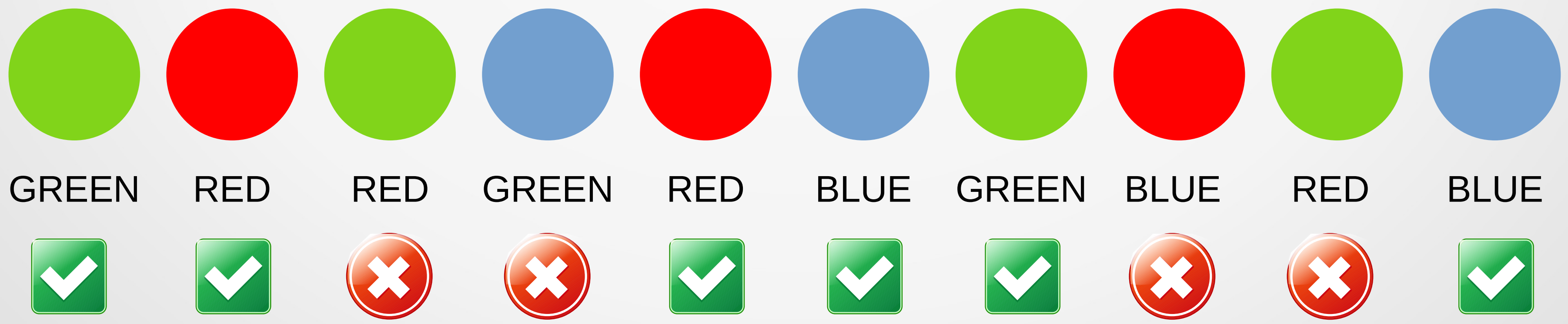
What is the precision for classifying blue circles in the example above?

Recall

Recall gives the percentage of correct predictions for the classification of interest, out of the total number of examples that truly have that classification. So :

$$\textbf{recall}_c = \textbf{true positives}_c \,/\, (\textbf{true positives}_c + \textbf{false negatives}_c)$$

Recall is a useful metric when classes aren't evenly represented.



What is the recall for classifying green circles in the example above?

Regression

Regression refers to a set of approaches which try to predict an output value given one or more input values.

The output value is referred to as the *dependent variable* (also often referred to as the *outcome / response variable*). This is the value we're trying to *predict*.

The input values are referred to as the *independent variables* (often referred to as *explanatory variables / features*). These are the data values from which we are trying to predict the dependent variable.

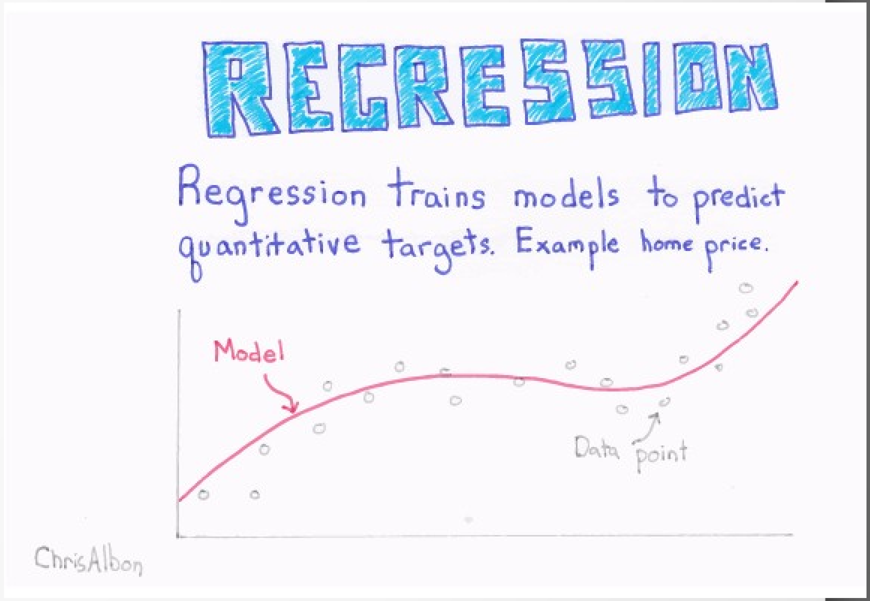

REGRESSION
Regression trains models to predict quantitative targets. Example home price.
Model
Data point
ChrisAlbon

Image from www.machinelearningflashcards.com

Examples :
- Predicting the level of revenue from advertising spending
- Predicting house price from size of house

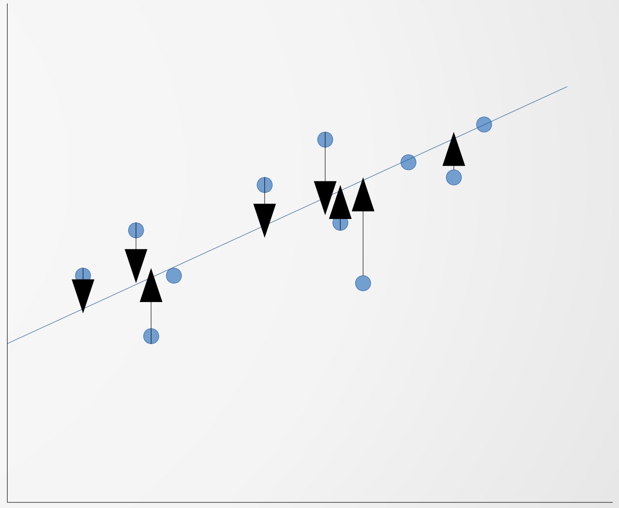
Linear Regression

Linear Regression is a type of regression where we are trying to fit a straight line that best fits the data points.

The line is described using the equation **y = a + bx**, where :

- **y** is the dependent variable (the one we're trying to predict)
- **a** is the intercept (where the line cuts through the y-axis at x=0)
- **b** is the slope / gradient of the line
- **x** is the independent variable / feature

A common method for fitting a Linear Regression model is using *Least Squares Regression*, in which the line of best fit is the one which minimises the sum of the squares of vertical deviations between data points and the line.



Where there is more than one independent variable, the data (and the line) will be in greater than 2 dimensions

Logistic Regression

With *Logistic Regression*, we are trying to predict the *probability* that a given set of inputs belongs to a given *class*.

For example, we might use Logistic Regression to predict :
- the probability of passing an exam against mock exam grades
- the probability that a passenger on the Titanic with a given set of features would survive the sinking…

Logistic Regression is therefore used in Machine Learning for *Classification* problems.

The simplest form of Logistic Regression considers only binary classes (x or y, True or False) but there are more complex forms that allow for more than two classifications. However, we won’t worry about those here.

Logistic Regression

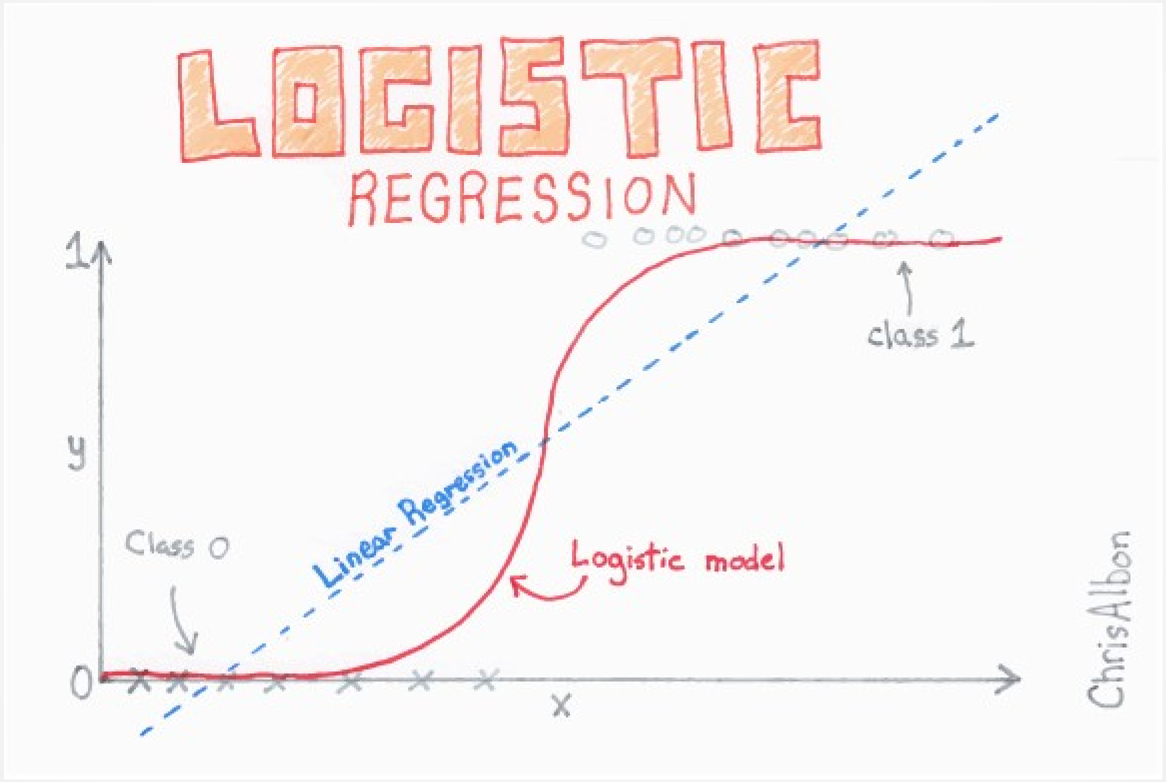


Image from www.machinelearningflashcards.com

The Logistic Function

Logistic Regression is named as such because of the function that defines the regression line – the *Logistic Function*, which is an example of a *Sigmoid Function* – one which has an S-shaped (sigmoid) curve.

We can set up the Logistic Function to take any value and map it to a value between (but never reaching) 0 and 1. This makes it very useful for generating outputs that represent probabilities.

$$f(x) = \frac{L}{1 + e^{-k(x - x_0)}}$$

f(x) : output, given input x
L : maximum value of the curve (we can use 1 here to map between 0 and 1)
e : the natural number (Euler's number)
k : steepness of the curve
x : value to be mapped (input)
x_0 : midpoint of the sigmoid curve (the point of intersection with the y-axis)

Logistic Regression Equation

Logistic Regression equations combine the *Logistic Function* with a *Linear Function* that has *weights* (or *coefficients)* associated with each input value.

In simple terms, the equation fits a linear model with weights for each input, and then converts that using a logistic model to output a probability between 0 and 1.

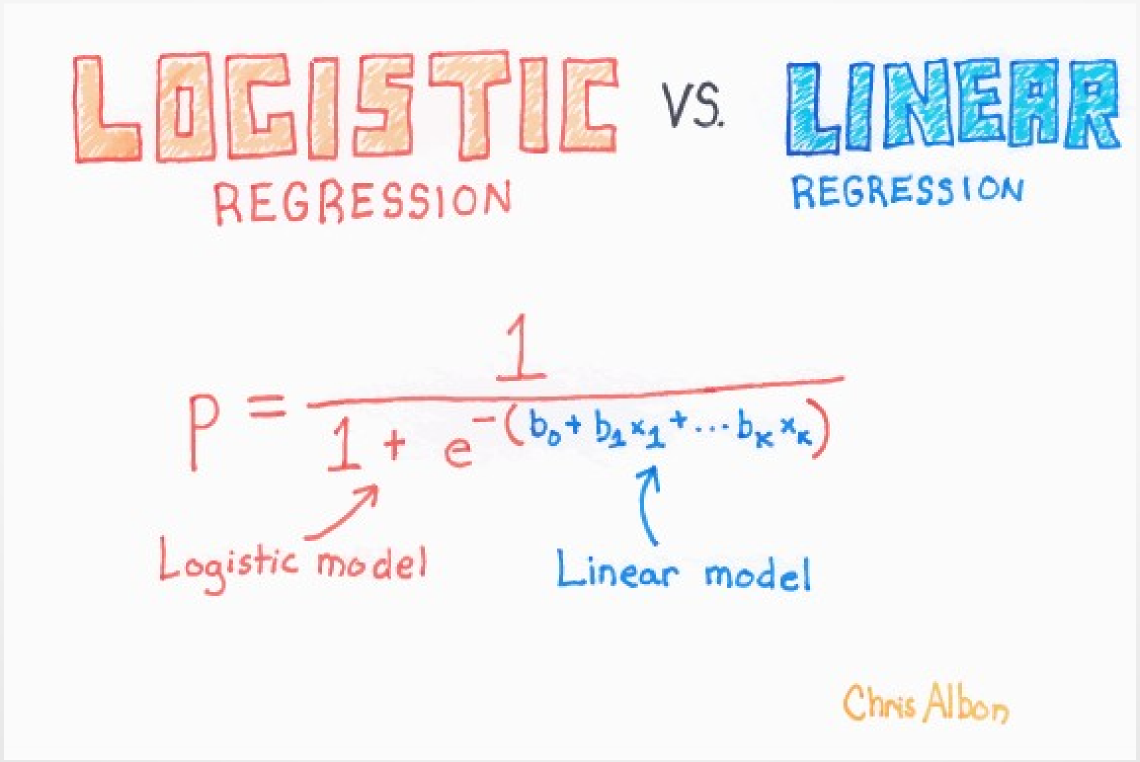


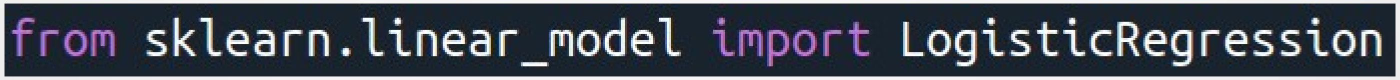
Image from www.machinelearningflashcards.com

Logistic Regression in Python

It’s very easy to fit a Logistic Regression model in Python. We can import a library called *sklearn* (sci-kit learn), which is a free ML library that contains lots of ML algorithms, as well as handy functions for splitting data into training and test data, preprocessing data for ML algorithms and more.

The Anaconda distribution comes with sci-kit learn already installed.

We can import the Logistic Regression module as follows :



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

Preparation

Assuming our data has been preprocessed to be in the format we need (we’ll come back to that), there are three things we need to do with our data before we give it to the Logistic Regression algorithm :

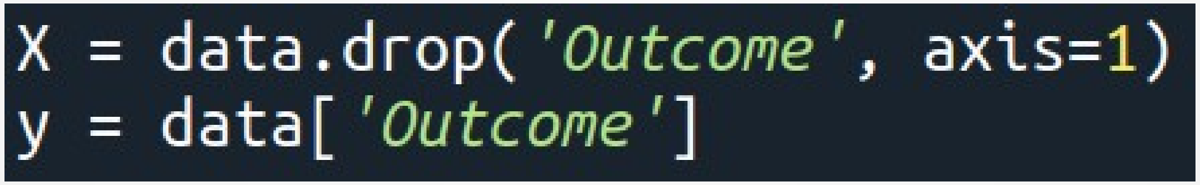
1. Divide the data into *features* (inputs) and *labels* (outputs)
2. Divide the data into *training* and *test* sets
3. Apply feature scaling, so that the feature data values are all on a similar scale

Let’s look at each of these steps in turn.

X and y

We need to divide our data into features (which we denote with X – the capital letter signifying there are multiple features for each example), and labels (which we denote with y – the lower case indicating there is a single output for each example).

If our data is stored in a Pandas DataFrame called *data*, and our label is stored in a column called *‘Outcome’*, we can store our features and labels as follows :



```
X = data.drop('Outcome', axis=1)
y = data['Outcome']
```

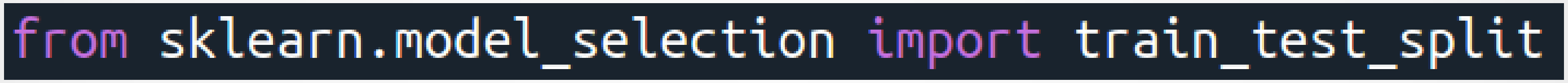
The first line above tells Python to assign to X (our features) all the columns in the data frame *except* for the ‘Outcome’ column (axis=1 tells the drop function you’re dropping one of the columns, rather than a row (which would be axis=0)).

The second line above tells Python to assign the data in the ‘Outcome’ column to y.

Train / Test Splits

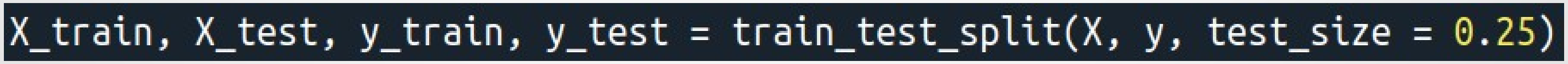
We need to split our data into training and testing sets (and do this for both the features (X) and labels (y)).

Sci-kit Learn has a module that allows us to split our data up randomly really easily :



```
from sklearn.model_selection import train_test_split
```

Then we can just use the following line of code to automatically split our data into training and test sets (here we specify the size of the test set as being 25% of our data – leaving 75% of our data for training) :



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

(The train_test_split function returns the training and test data for X and the training and test data for y, in that order)

Feature Scaling

The features that we include in a Machine Learning algorithm may well be diverse, and have very different kinds of values.

For example, consider the following set of features :
- Age
- Number of admissions to hospital
- Mean daily calorie intake

Now let’s consider the kinds of ranges of values we might expect to see for each feature :
- Age (up to around 100, may well be peaks around 60-85 depending on area we’re looking at)
- Likely less than 10 for most
- Likely 2000 – 3000

The values from these features are on very different scales.

Feature Scaling

It would be better for Machine Learning if our values were on similar scales. Vastly different kinds of scales for values leads to :

- Poorer learning performance (because the algorithm may tend to to give more weight to features with values of higher magnitude)
- Difficulty comparing weights between features, making it difficult to ascertain what are the more / less important features

The two most common ways to scale feature values are :

- **Normalisation** – this is where we rescale values so all values in our data fall between two values (most commonly 0 and 1). This is also known as *Min-Max Normalisation*.
- **Standardisation** – this is where we centre all our values around a mean of 0 with a standard deviation of 1. This means that we don't have a bounded range, and can accommodate outliers more naturally – but most of our values will fall within a more manageable distribution.

MinMax Normalisation

MinMax Normalisation typically involves scaling all values in our data between 0 and 1. We can do this using the equation :

$$z = \frac{x - min(x)}{max(x) - min(x)}$$

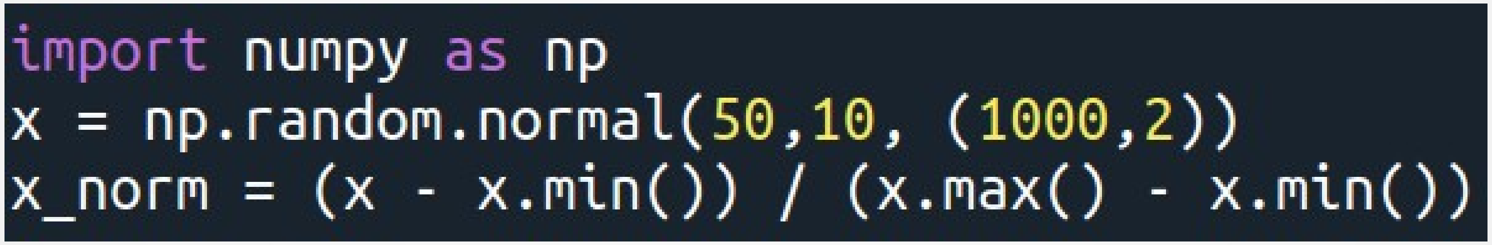
The above equation basically says that to get a normalised value for a data point, you subtract the minimum value in the data from your data point, and divide it by the range of values (the maximum value minus the minimum value).

So, if our data contained the values 2, 5, 7, 9, 100, then the value 5 would be normalised as : (5 – 2) / (100-2) = 3 / 98 = 0.03. The highest value (100) would be normalised as : (100 – 2) / (100 – 2) = 98 / 98 = 1. The lowest value (2) would be normalised as : (2 – 2) / (100 – 2) = 0 / 98 = 0.

MinMax Normalisation (Manual Method)

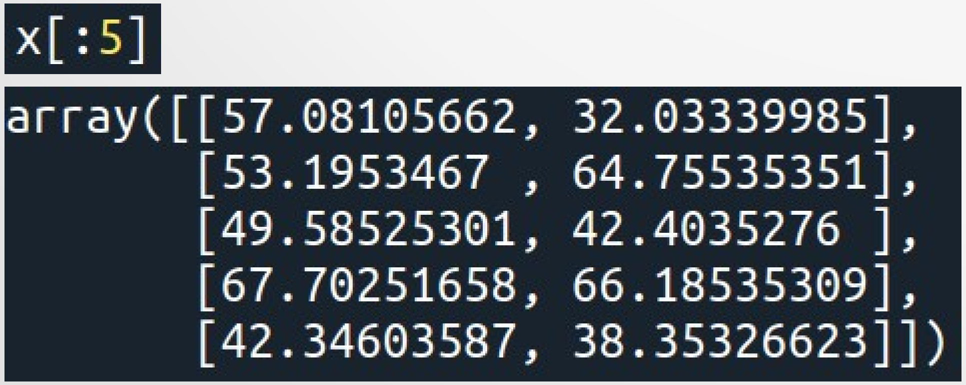
To MinMax Normalise a NumPy array of data values, we can either do it manually or use the MinMaxScaler module from Sci-Kit Learn.

To do it manually, for a 2D NumPy array called x of shape (1000, 2) (1000 samples, 2 columns (features)) which we randomly generate from a normal distribution with mean = 50 and standard deviation = 10 here :



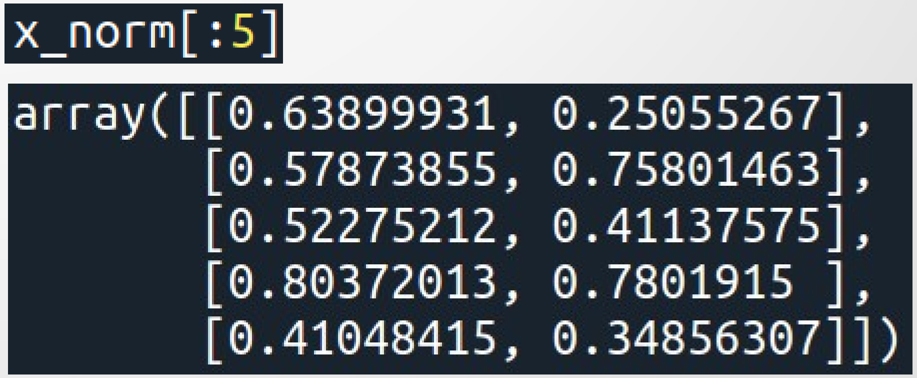
```python
import numpy as np
x = np.random.normal(50,10, (1000,2))
x_norm = (x - x.min()) / (x.max() - x.min())
```

(remember – NumPy allows us to easily apply the same operation to all values in a NumPy array without having to specify a loop)



```
x[:5]
```

```
array([[57.08105662, 32.03339985],
       [53.1953467 , 64.75535351],
       [49.58525301, 42.4035276 ],
       [67.70251658, 66.18535309],
       [42.34603587, 38.35326623]])
```

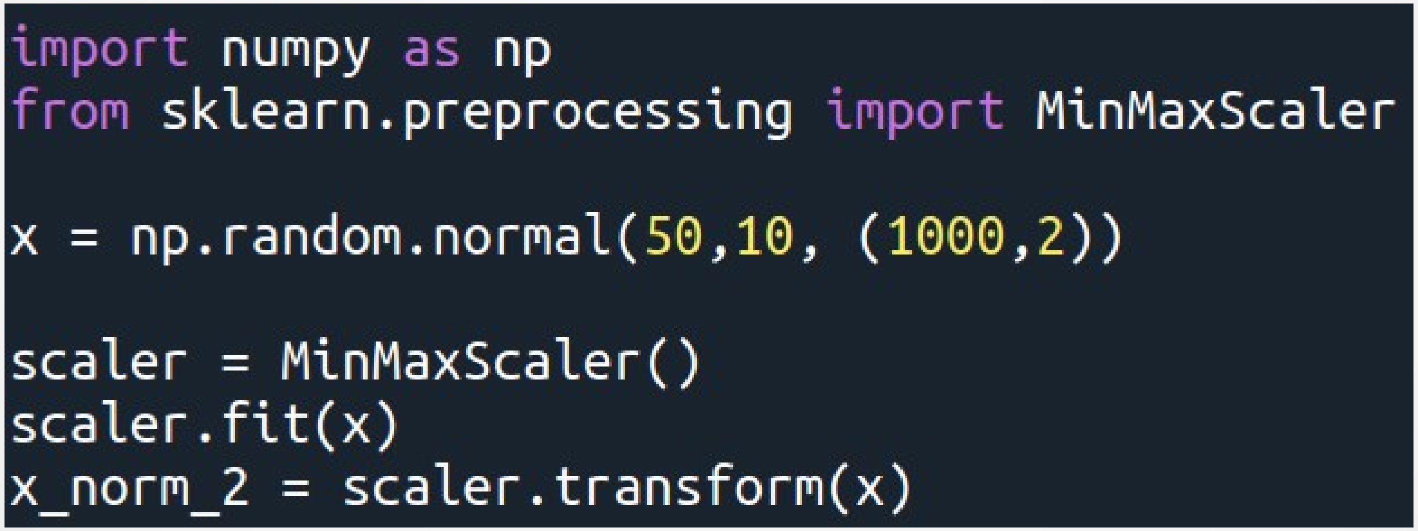


```
x_norm[:5]
```

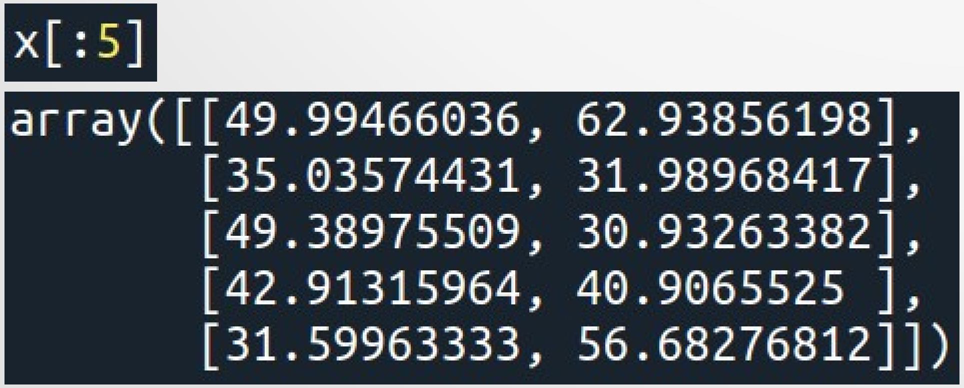
```
array([[0.63899931, 0.25055267],
       [0.57873855, 0.75801463],
       [0.52275212, 0.41137575],
       [0.80372013, 0.7801915 ],
       [0.41048415, 0.34856307]])
```

MinMax Normalisation (MinMaxScaler)

To use the MinMaxScaler library for the same data, we’d do the following :

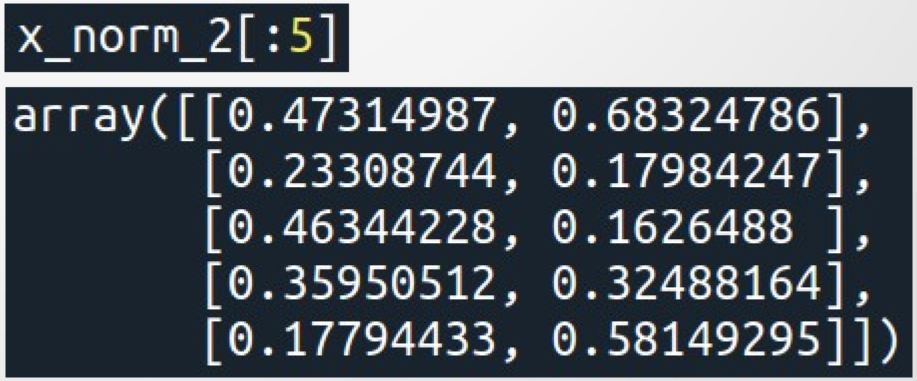


```
import numpy as np
from sklearn.preprocessing import MinMaxScaler

x = np.random.normal(50,10, (1000,2))

scaler = MinMaxScaler()
scaler.fit(x)
x_norm_2 = scaler.transform(x)
```



```
x[:5]
```

```
array([[49.99466036, 62.93856198],
       [35.03574431, 31.98968417],
       [49.38975509, 30.93263382],
       [42.91315964, 40.9065525 ],
       [31.59963333, 56.68276812]])
```



```
x_norm_2[:5]
```

```
array([[0.47314987, 0.68324786],
       [0.23308744, 0.17984247],
       [0.46344228, 0.1626488 ],
       [0.35950512, 0.32488164],
       [0.17794433, 0.58149295]])
```

Standardisation

To standardise, we scale such that all features have a mean of 0 and a standard deviation of 1. We can do this using the following equation :

$$z = \frac{x - \mu}{\sigma}$$

This equation says that to standardise our data we take each value and subtract the mean of the data, then divide it by the standard deviation of the data.

Consider an example to see how this works. Let's again take our values 2, 5, 7, 9, 100. The mean of these values is 24.6, and the standard deviation is 42.23.

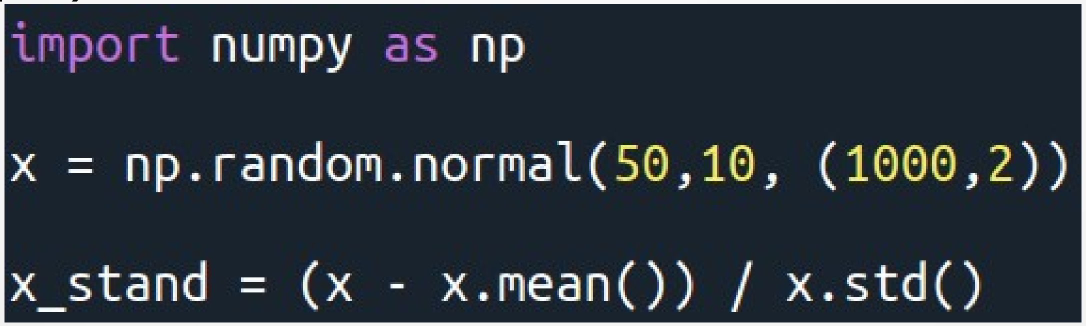
2 → (2 – 24.6) / 42.23 = -0.54
5 → (5 – 24.6) / 42.23 = -0.46
7 → (7 – 24.6) / 42.23 = -0.42
9 → (9 – 24.6) / 42.23 = -0.37
100 → (100 – 24.6) / 42.23 = 1.79

Mean of standardised values = 0, Standard Deviation of standardised values = 1

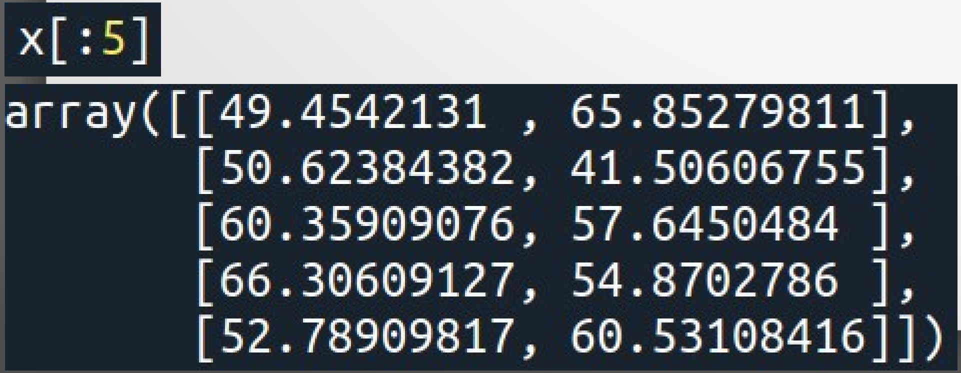
Standardisation (Manual Method)

As with normalisation, there are different ways in which we can standardise data in Python. We could :
- do it manually
- use the preprocessing.scale() function in Sci-Kit Learn
- use the StandardScaler() function in Sci-Kit Learn

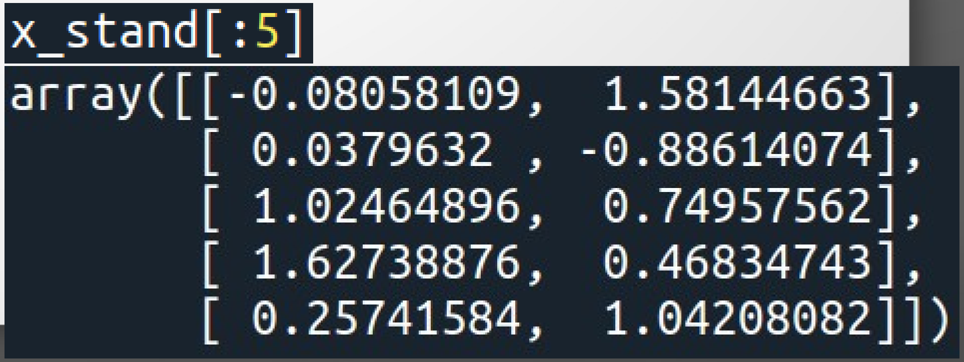
To manually do it (using the same randomly generated data in the previous example) :



```
import numpy as np

x = np.random.normal(50,10, (1000,2))

x_stand = (x - x.mean()) / x.std()
```



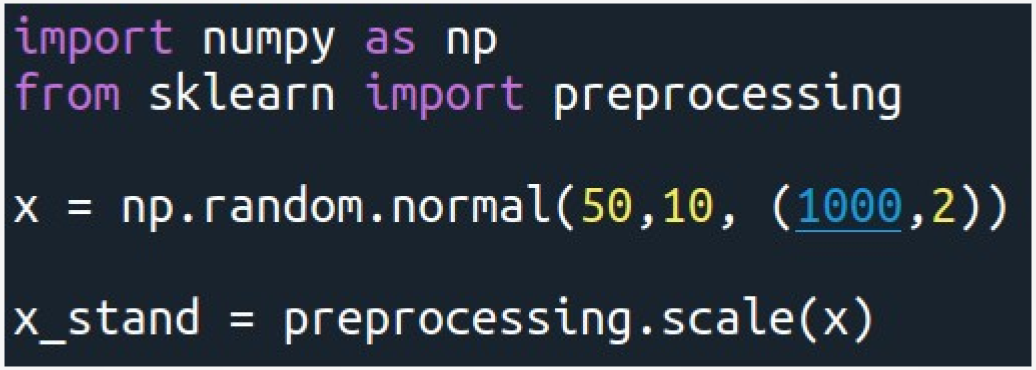
```
x[:5]
array([[49.4542131 , 65.85279811],
       [50.62384382, 41.50606755],
       [60.35909076, 57.6450484 ],
       [66.30609127, 54.8702786 ],
       [52.78909817, 60.53108416]])
```



```
x_stand[:5]
array([[-0.08058109,  1.58144663],
       [ 0.0379632 , -0.88614074],
       [ 1.02464896,  0.74957562],
       [ 1.62738876,  0.46834743],
       [ 0.25741584,  1.04208082]])
```

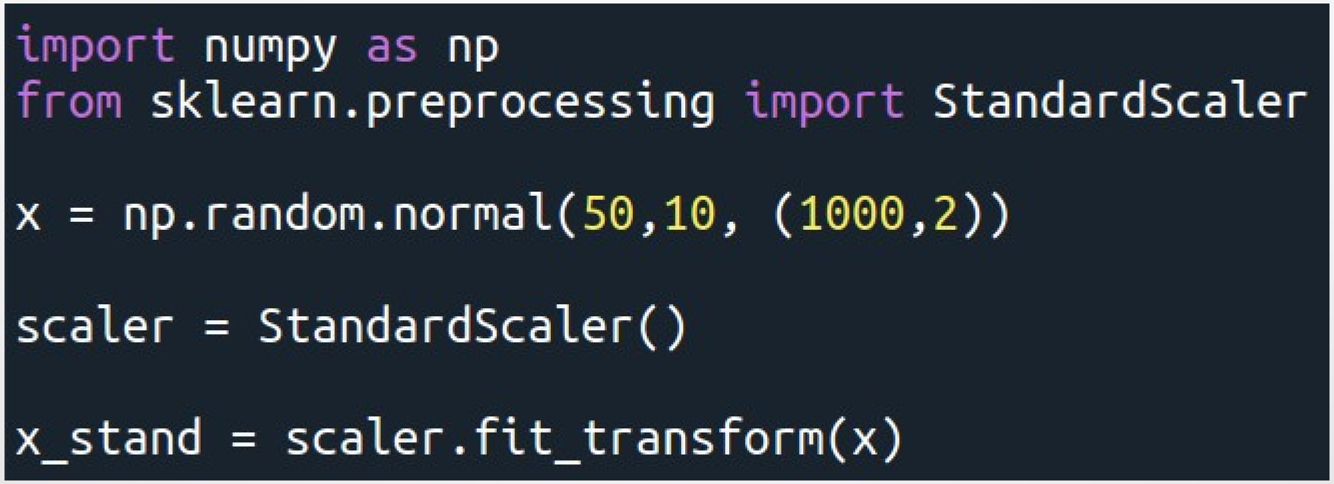
Standardisation (Sci-Kit Learn)

To use the scale() function of the preprocessing module of Sci-Kit Learn :



```python
import numpy as np
from sklearn import preprocessing

x = np.random.normal(50,10, (1000,2))

x_stand = preprocessing.scale(x)
```

Or to use the StandardScaler() module :



```python
import numpy as np
from sklearn.preprocessing import StandardScaler

x = np.random.normal(50,10, (1000,2))

scaler = StandardScaler()

x_stand = scaler.fit_transform(x)
```

A couple notes

1. So, when should we use normalisation, and when should we use standardisation?

There aren’t any hard and fast rules here. The best option is to try both and see what performs best for your data and problem.

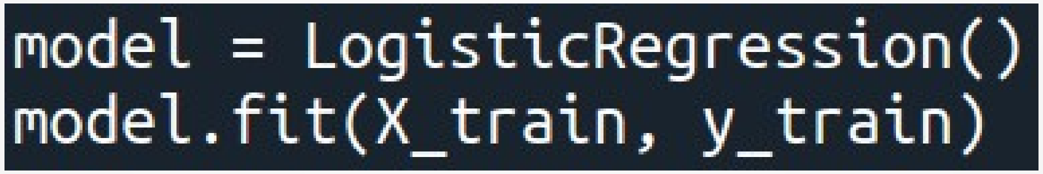
But you will generally find that Linear Regression works best with standardised data, and Neural Networks work best with MinMax Normalised data.

2. When standardising, we standardise the training set data, and use the training data’s mean and standard deviation to standardise the test data (rather than the test data’s mean and standard deviation). This is so that the test data is on the same scale as the predictions generated by the model.

Fitting the Model

Once we’ve done our prep work (splitting into features and labels, splitting into training and test data, and scaling our features), we’re ready to fit the model (ie – do the actual Logistic Regression).

Which, you may imagine, involves lots of complex code. But because we’re using the Sci-Kit Learn library, it’s actually really easy :



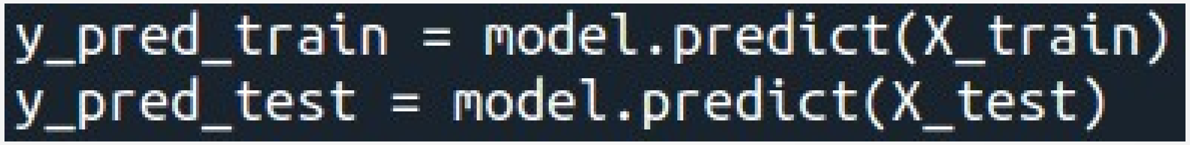
```
model = LogisticRegression()
model.fit(X_train, y_train)
```

(Note that X_train above should be the standardised data, which you may have named differently (e.g. X_train_stand) to distinguish it)

Predicting

Once the model's been trained ("fitted"), then we can use it to generate predictions. We can then test the accuracy (and other metrics) of the predictions on both the training set and the testing set.

To generate predictions from a fitted model using the Sci-Kit Learn library, we simply use the predict() method of the model :



```
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)
```

The above code calls the predict method on the feature data of the training data and the test data in turn. We can then see how well the fitted model predicts the y values (labels) in the data it's seen (training data) and hasn't seen (test data) separately.

Exercise 1

We'll now take a 10 minute comfort break. After the break, I want you to :
1) work through the two Jupyter Notebooks HSMA_tutorial_1.ipynb and HSMA_tutorial_2.ipynb. Please ensure that you have created the "titanic" Conda environment (as per the prep instructions) and make sure you switch into this environment before launching the notebooks.
2) have a go at HSMA_exercise.ipynb, where you're asked to write code that trains and uses a Logistic Regression model to predict whether stroke patients will receive clot-busting treatment.

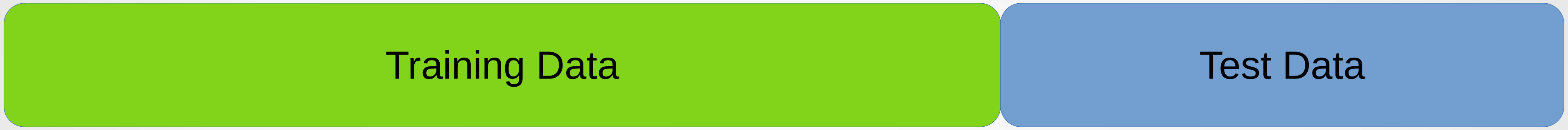
For 1) above, you should read through the workbooks, and run each code cell as you work your way through. Make sure you understand how all the code is working by working through it line by line.

I'd recommend that you work through the workbooks as a group, but all follow along in your own notebooks. Don't go too fast – make sure everyone is following and understands it before you move on. I'd also recommend you tackle the second part (writing the stroke ML algorithm) as a group.

You have **1 hour**.

Stratified k-fold Validation

So far, when we've split our data into training and test data, we've made one split. We randomly select x% to be training data, and y% to be test data.



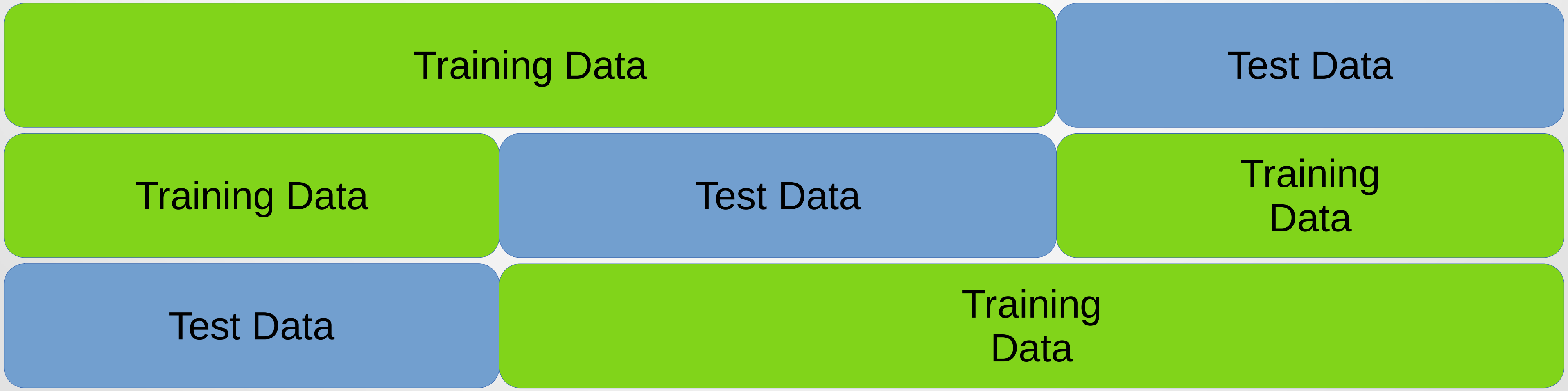
The problem is that this split may not be a good split for training, and so may give a misleading estimate of the accuracy of our model. Much as in a Discrete Event Simulation – we wouldn't just run the simulation once, because the results we get may not be representative.

A better approach is to use something called *Stratified k-fold Validation*.

Stratified k-fold Validation

In *k-fold Validation,* we repeat the model training and testing *k* times, such that all data is used once (but only once) as part of the test set. In other words, all our data gets an opportunity to be the thing against which we *check*, rather than against which we *learn*.

In ***Stratified*** *k-fold Validation*, we also ensure that the balance between different outcomes (e.g. survived vs died in the Titanic data) across all the data is maintained in each train-test split. (e.g. if it's a 60/40 split in the data as a whole, that's maintained in each train-test combination used).



ROC Curves

Receiver Operator Characteristic (ROC) Curves are really useful in assessing the performance of a Machine Learning algorithm. They plot the *True Positive Rate* (proportion of cases correctly identified as positive; *sensitivity*) against the *False Positive Rate* (proportion of cases incorrectly identified as positive; *1 – specificity*).

We plot these rates for varying prediction *thresholds*. The prediction threshold is the value above which we classify something as *positive* (and below which, something is classified as *negative*). By default, this is usually 0.5, but that may not always be the best threshold, particularly for imbalanced datasets.

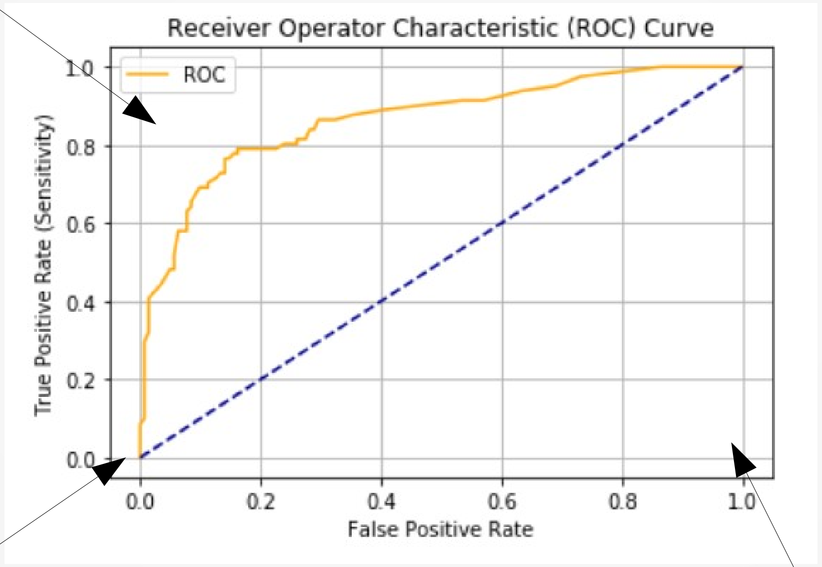
	Real World Positive	Real World Negative
Predicted Positive	TRUE POSITIVE	FALSE POSITIVE
Predicted Negative	FALSE NEGATIVE	TRUE NEGATIVE

Sensitivity = TP / (TP + FP)

Specificity = TN / (FN + TN)

ROC Curves

Thresholds that result in points near the top-left corner are nearer optimal (Very good at saying something is positive when it is, and rarely saying something is positive when it isn't)



Thresholds that result in points near the bottom right says most things are positive when they're not, and rarely says something is positive when it is. **This will likely indicate an error in your algorithm (ie you've got things the wrong way around)**

The blue dotted line represents a "Worthless Test" – if points fall along this line, then the algorithm is no better at predicting a positive correctly than incorrectly. This is a useful *Reference Line* – this is the poorest outcome for our algorithm. If we're above this line, we're doing better than random.

AUROC

A visual inspection of a ROC curve can give us a good indication of how well our model is performing. But we can more precisely assess this by calculating the *AUROC* – the Area Under the Receiver Operator Characteristic curve.

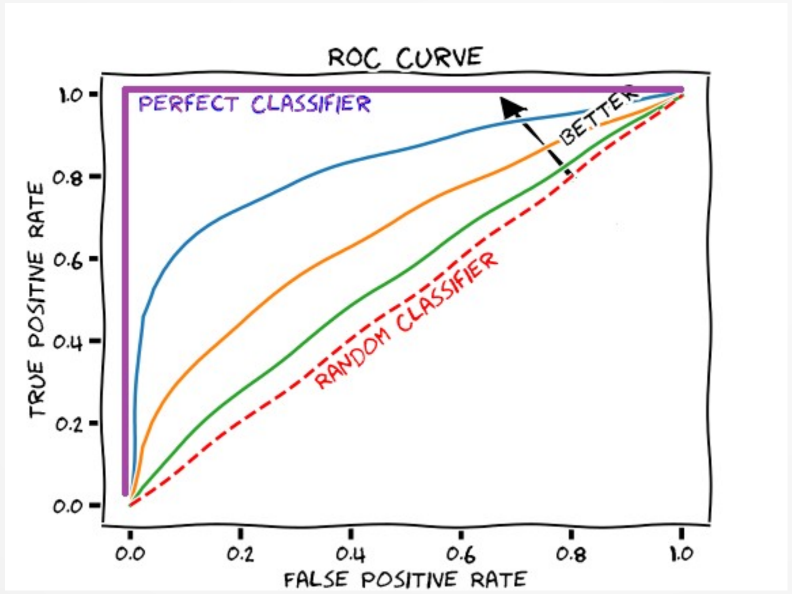
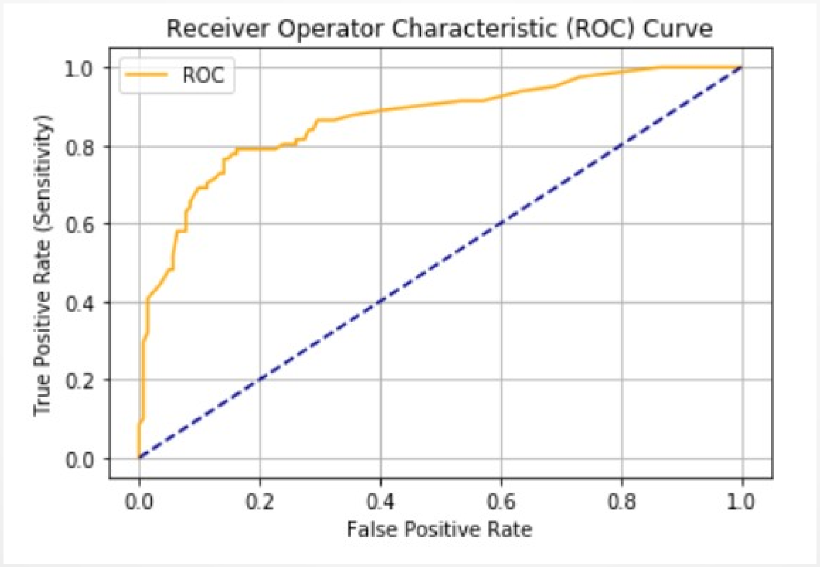


Image from https://glassboxmedicine.com/2019/02/23/measuring-performance-auc-auroc/

Any area greater than 0.5 (half the area) is better than random. An area of 1.0 represents a "Perfect" model – one which has a perfect sensitivity for all specificity values.

Threshold Selection

We can use ROC curve data to determine what threshold we will use for our classifier. We determine the “trade-off” that we’re willing to accept – ie the % of false positives that we’re willing to accept to get a higher rate of true positives.



Exercise 2

In your groups, you should spend the remainder of this session working through the notebooks **03_k_fold.ipynb** and **06_roc_sensitivity_specificity.ipynb**. As before, you should work through them as a group and make sure that everyone understands how the code works.

If you have extra time, or for further reading after the session, I would also recommend that you check out **17_random_forest.ipynb**.