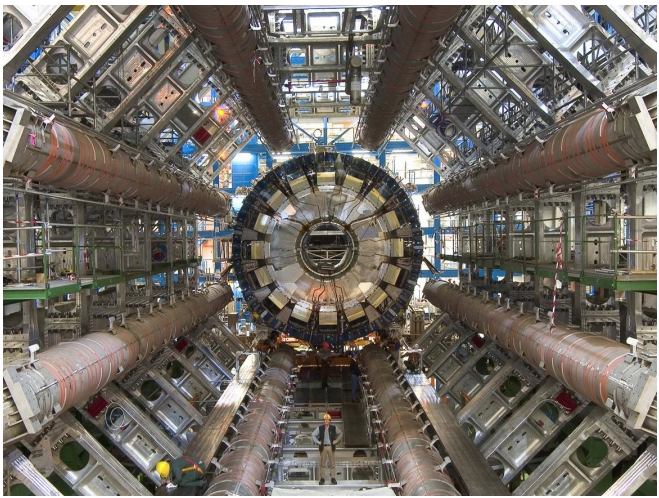
# 7. Machine Learning: Part 1

# Application of the week: theoretical physics



DataKirk

# Session overview

1. What is machine learning?
2. Typical steps of an ML task
3. Introduction to K-nearest neighbours
4. A worked example

# What is machine learning?

- Machine learning is a broad term. In general, it can be thought of as a collection of algorithms and techniques for learning from and extracting insight from data.

- The line between statistics, machine learning and artificial intelligence is often quite blurred.

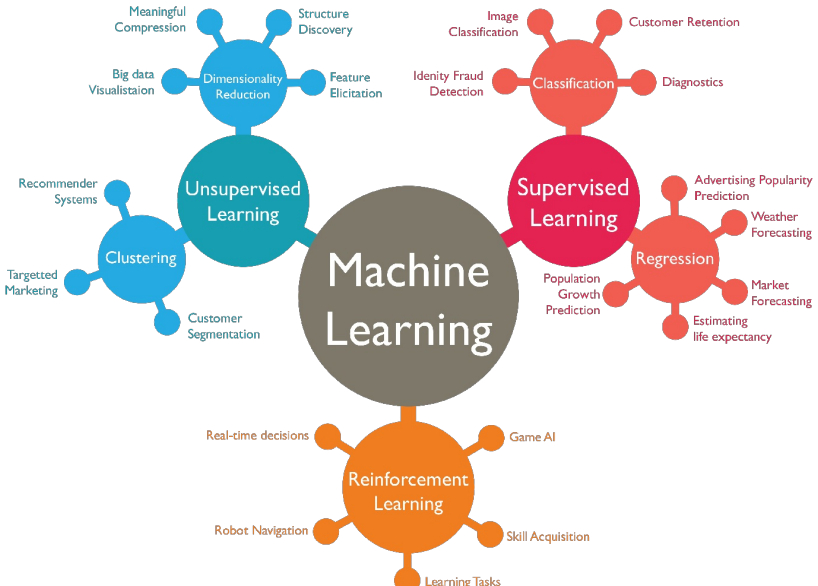
- As the amount of data being generated by everyone every day has exploded, so too has the usage of machine learning algorithms. We all interact with machine learning algorithms every day, every time we make a google search or open Facebook.

- Machine learning offers the possibility of automated decision making and is capable of making sometimes *scarily* accurate predictions.

# What is machine learning?

- On one end of machine learning there are relatively simple algorithms, such as standard regression and K-nearest neighbours.
- Somewhere in the middle are more sophisticated algorithms such as XG-Boost and support vector machines.
- On the far end is the area of deep neural networks. This is the state-of-the-art for making complex predictions, with thousands of academic papers being published on this topic every year.
- Good news: Python is the number one language for designing machine learning algorithms.

# What is machine learning?



# Typical steps of an ML task

- Most machine learning tasks often follow a typical pattern:

    1. Collect the raw data
    2. Preprocess this data into a usable format
    3. Apply your chosen ML model to the data
    4. Analyse the accuracy/effectiveness
    5. Adjust any model hyperparameters
    6. Report the final accuracy/effectiveness

# Typical steps of an ML task

- Steps 1 and 2 are often overlooked! There is a lot of hype and attention given to ML algorithms, but the reality is that, often, the majority of a data scientist's time is spent collecting and preprocessing data.

## 1. Data collection

- Sometimes data will be generated automatically and stored on a database.
- Other times you will either have to look for data sources on the internet. This may involve writing a *web-scraper*.

## 2. Data preprocessing

- Once you have the data it is unlikely that it will be in usable right away! There will often be missing values or unexpected errors, and is unlikely to be in a useful format.

# Typical steps of an ML task

## 3. Apply the ML model

- This is the most fun stage! Here you get to apply your chosen model to the data you have collected. This is sometimes called "training".
- Depending on the complexity of the model, the amount of computation resources available and the amount of data used, this may take anywhere between a fraction of a second and many weeks.

## 4. Analyse the model

- At this stage it is important to know how well the model is performing. Often data is split into "training data" and "test data". Accuracy is measured on the test data.
- Measuring accuracy on training data can give false optimism! This is known as "overfitting"

# Typical steps of an ML task

## 5. Adjust the hyperparameters

- Often a given ML model will have a list of possible settings known as "hyperparameters". One job of the data scientist is to optimally choose these settings.
- This can be done by testing the accuracy of the model for a range of different settings, and then choosing the best. This is known as "tuning" the hyperparameters.

## 6. Report the final accuracy

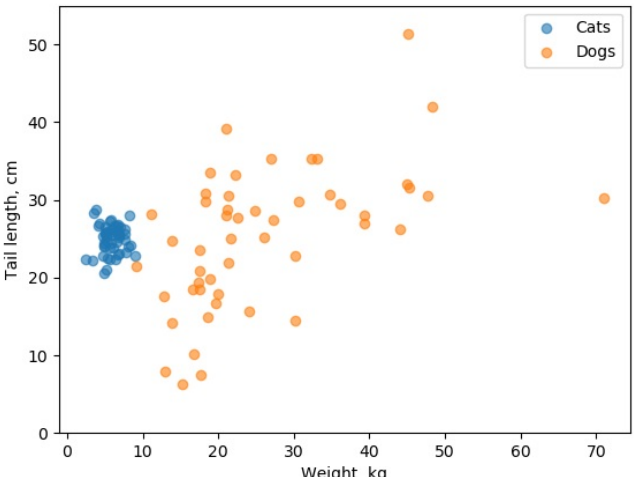
- Once the hyperparameters have been chosen, the only thing left to do is to report the final results of the model.
- Ideally this should be done on an extra test set, which hasn't been used to tune the hyperparameters.

# K-nearest neighbours

- The KNN or K-nearest neighbour algorithm is a supervised learning algorithm. It can be used both for classification and regression; however, it is more widely used for classification purposes.
- Consider a dataset where we have measured the weight in kg and the tail length in cm for a number of dogs and cats.

| Cat or dog? | Weight (kg) | Tail length (cm) |
|---|---|---|
| cat | 8.03 | 24.02 |
| dog | 16.75 | 10.1 |
| dog | 24.85 | 28.53 |
| cat | 5.66 | 27.23 |
| ⋮ | ⋮ | ⋮ |

# K-nearest neighbours



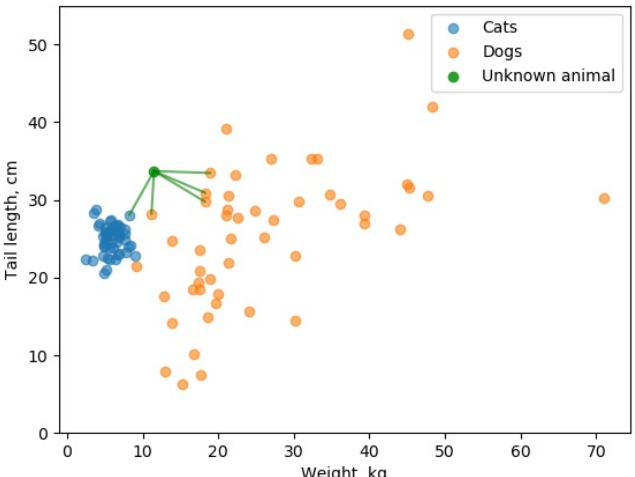
# K-nearest neighbours

- For a new unknown animal, for which we have both measurements, the KNN algorithm measures the "distance" between this point and all the labelled points.

- The closest K are selected (K may be 5, for example).

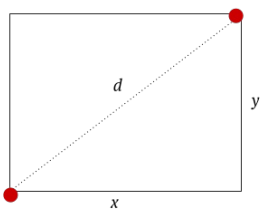
- The new data point is then assigned a class label based on the labels of its neighbours. If the majority of these close points are dogs, the new point is predicted to be a dog. If more are cats, the new point is predicted to be a cat.

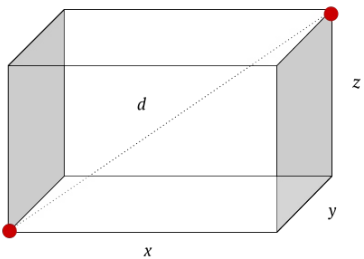
# K-nearest neighbours



# Measuring distance

- In two dimensions, measuring distance is intuitive. However, what if there were more measurements, say, paw width?
- "Distance" can actually be extended to any number of dimensions.




...

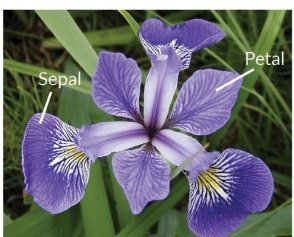
$$d^2 = x^2 + y^2$$

$$d^2 = x^2 + y^2 + z^2$$

$$d^2 = x^2 + y^2 + z^2 +$$

# A worked example: the Iris dataset

- The Iris dataset is a classic starter dataset in machine learning.
- It consists of 150 samples having three classes, namely `Iris-Setosa`, `Iris-Versicolor`, and `Iris-Virginica`. Four measurements contribute to uniquely identifying each plant: `sepal-length`, `sepal-width`, `petal-length` and `petal-width`.




**Iris Versicolor**

**Iris Setosa**

**Iris Virginica**

# The Iris dataset

- A widely used and very powerful Python library for machine learning tasks is `scikit-learn`. This can be installed on your own computer by going to the command line and typing

```
pip install scikit-learn
```

- Once installed, scikit learn can be used to load the Iris dataset directly.

```
from sklearn.datasets import load_iris

data = load_iris(as_frame=True).frame
```

# The Iris dataset

| sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) | target |
|---|---|---|---|---|
| 5.1 | 3.5 | 1.4 | 0.2 | 0 |
| 4.9 | 3 | 1.4 | 0.2 | 0 |
| 4.7 | 3.2 | 1.3 | 0.2 | 0 |
| 4.6 | 3.1 | 1.5 | 0.2 | 0 |
| 5 | 3.6 | 1.4 | 0.2 | 0 |
| ... | ... | ... | ... | ... |

# The Iris dataset

- The target column indicates the plant type. The first 50 rows have a target of 0, meaning Iris-Setosa. The second 50 have a value of 1 meaning Iris-versicolor, and the final 50 have a value of 2, meaning Iris-virginica.

- A good first step is to make some plots to get a feel for the data. We can use matplotlib to scatter two of the four variables against each other.

- For this example, let's look at petal length vs petal width. Although we could do this for any two columns.

# The Iris dataset

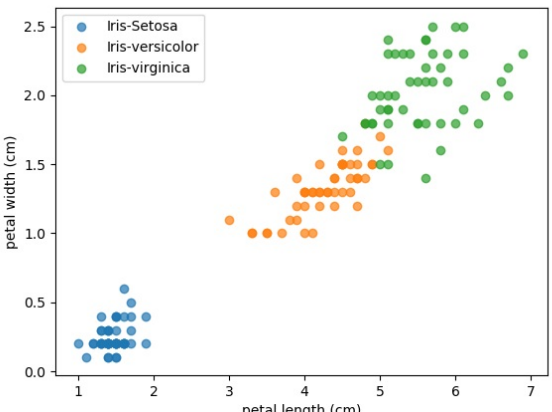
```
# create a new figure
plt.figure()

# these two columns will be our x and y values
x = data['petal length (cm)']
y = data['petal width (cm)']

# scatter in 50s...
plt.scatter(x.iloc[:50], y.iloc[:50], label='Iris-Setosa')
plt.scatter(x.iloc[50:100], y.iloc[50:100], label='Iris-Versicolor')
plt.scatter(x.iloc[100:], y.iloc[100:], label='Iris-Virginica')

plt.xlabel('petal length (cm)')
plt.ylabel('petal width (cm)')
plt.legend()
```

# The Iris dataset



# Splitting into train and test data

- As mentioned, a key step in any machine learning task is to split the data into a training set and a test set.
- This means when we test our accuracy, the test points will not be used to make the prediction, which could lead to overfittting.
- Scikit learn has a special function called `train_test_split()` which will randomly split the data into two groups.

```
from sklearn.model_selection import train_test_split

train_data, test_data = train_test_split(data, test_size=0.2,
random_state=37)
```

- Providing the value `random_state` will make sure the "random" split is reproducible.

# Splitting into train and test data

`train_data`:

| sepal length | sepal width | petal length | petal width | target |
|---|---|---|---|---|
| 4.6 | 3.1 | 1.5 | 0.2 | 0 |
| 6.3 | 2.5 | 5 | 1.9 | 2 |
| ... | ... | ... | ... | ... |

`test_data`:

| sepal length | sepal width | petal length | petal width | target |
|---|---|---|---|---|
| 6.4 | 2.9 | 4.3 | 1.3 | 1 |
| 5.2 | 3.5 | 1.5 | 0.2 | 0 |
| ... | ... | ... | ... | .. |

# Training the model

- The next step is to apply the model to the training data. Again, we will use scikit learn's pre-built model.

```
from sklearn.neighbors import KNeighborsClassifier

# generate a new classifier: KNN with k=5
my_classifier = KNeighborsClassifier(n_neighbors=5)

# these are the columns that we will use to predict
input_cols = ['sepal length (cm)', 'sepal width (cm)', 'petal length
(cm)', 'petal width (cm)']

# this is the column we are trying to predict
target_col = 'target'

# fit the classifier using the input data and targets
my_classifier.fit(train_data[input_cols], train_data[target_col])
```

# Testing the model

- Now that we have trained the model on the input data, we can use it to make predictions on the test data.

```
# use the .predict() function on the test data
prediction = my_classifier.predict(test_data[input_cols])

print(prediction)
```

```
array([1, 0, 2, 2, 0, 1, 0, 2, 2, 2, 0, 2, 2, 0, 0, 2, 1, 2, 2, 2, 1, 0,
2, 2, 0, 1, 2, 1, 0, 1])
```

- This gives a prediction for each row of the test data.

# Testing the model

- Is this prediction any good?

```
# get the true labels from the test data
true_label = test_data['target'].values

print(true_label)
```

```
array([1, 0, 2, 2, 0, 1, 0, 2, 2, 2, 0, 2, 2, 0, 0, 2, 1, 2, 2, 2, 1, 0,
2, 1, 0, 1, 2, 1, 0, 1])
```

- It looks pretty good but how can we tell for sure?

# Testing the model

- Let's compare the predicted values against the true values

```
print(prediction == true_label)
```

```
array([True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True, True, True, True, True,
False, True, True, True, True, True, True])
```

- In Python `True` is the same as 1, and `False` is the same as 0.

```
print(sum(prediction == true_label))
```

```
29
```

# Testing the model

- Since there were 30 items in the test data, we can see the accuracy is 29/30 = 96.7%.

```
print(sum(prediction == true_label) / 30)
```

```
0.9666666666666667
```

- Not bad!

# Tuning hyperparameters

- The next stage in the ML pipeline is to adjust hyperparameters.
- In this algorithm there was only one hyperparameter: the number of neighbours to look at, k.
- In order to decide which value of k to use, we should train a classifier for a range of values of k, and find out which one has the best accuracy.

# Tuning hyperparameters

```
# perform a for-loop over different k-values
for k in [1, 2, 3, 4, 5, 6, 7, 8, 9]:

    # create a new classifier with k neighbours
    my_classifier = KNeighborsClassifier(n_neighbors=k)

    # fit this classifier on the training data
    my_classifier.fit(train_data[input_cols], train_data[target_col])

    # make a prediction on the test data
    prediction = my_classifier.predict(test_data[input_cols])

    # analyse the accuracy
    print(k, sum(prediction == true_label))
```

# Tuning hyperparameters

```
1 29
2 28
3 29
4 28
5 29
6 28
7 28
8 28
9 28
```

- This isn't too helpful! Can we find a better way to estimate accuracy?

# Tuning hyperparameters

```
def estimate_accuracy(k):

    # create a fresh test/train split with no random_state
    train_data, test_data = train_test_split(data, test_size=0.2)

    # train a new classifier
    my_classifier = KNeighborsClassifier(n_neighbors=k)
    my_classifier.fit(train_data[input_cols], train_data[target_col])

    # make a prediction on the test data
    prediction = my_classifier.predict(test_data[input_cols])

    # return the number of correct guesses
    return  sum(prediction == test_data['target'].values) / 30
```

# Tuning hyperparameters

```
# perform a for-loop over different k-values
for k in [1, 2, 3, 4, 5, 6, 7, 8, 9]:

    # make an empty list to hold accuracy estimates
    accuracies = []

    # perform 100 loops
    for i in range(100):

        # estimate the accuracy 100 times
        accuracy = estimate_accuracy(k)
        accuracies.append(accuracy)

    # print the average accuracy
    print(k, sum(accuracies) / 100)
```

# Tuning hyperparameters

```
1 0.9589333333333291
2 0.9461999999999965
3 0.961399999999995
4 0.9571333333333285
5 0.962866666666662
6 0.9611999999trainig999947
7 0.962933333333329
8 0.9647333333333286
9 0.9662666666666618
```

- From this we can see that the optimal number of neighbours could possibly be 9 with an accuracy of ~96.6%.

# Thanks!