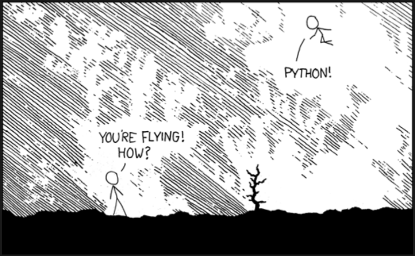
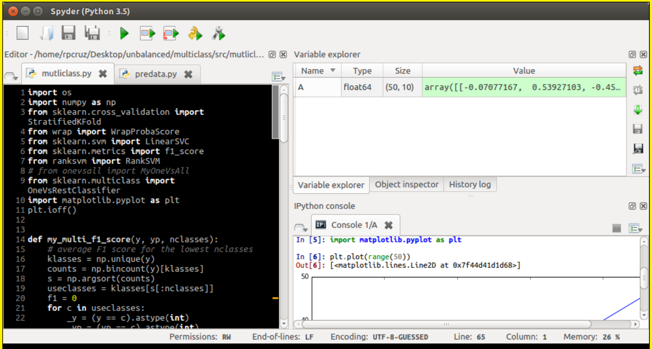
# Machine Learning

## Lecture Zero – October 1st, 2019

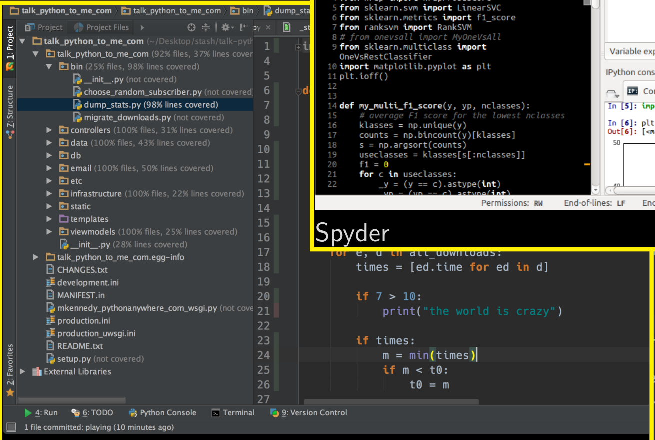

Eduardo Castro emcastro@inesctec.pt
Wilson Silva wilson.j.silva@inesctec.pt
Tiago Gonçalves tiago.f.goncalves@inesctec.pt

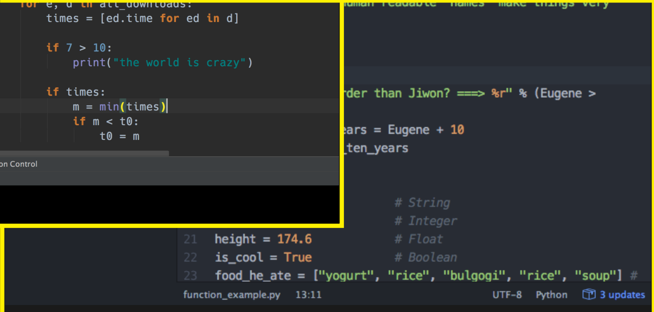

# Editors




Spyder



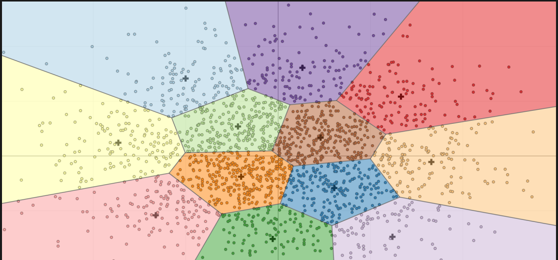

PyCharm




Atom

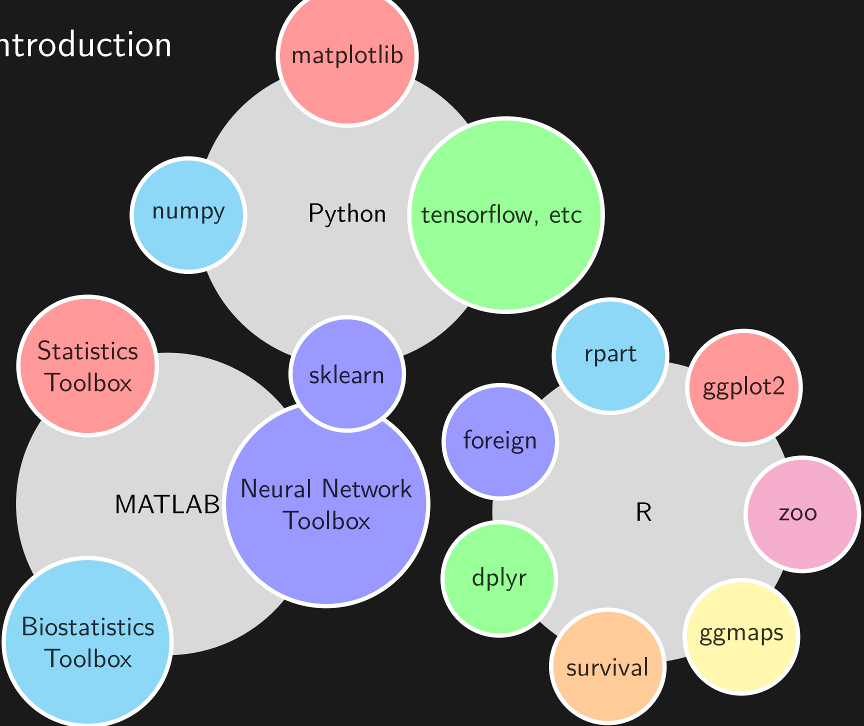
# Agenda

| **Theory** | **Practice** |
|---|---|
| ~Introduction~ | |
| Numpy | Implement Euclidean distance |
| Cycles/functions | K-means algorithm |
| File handling | Use a real dataset |
| Matplotlib | Plots |
| Classes/modules | — |
| Scikit-learn | Use its k-means |
| Keras+TF | Demo |
| ~Finishing touches~ | |



# Introduction

# Introduction



# Introduction

- Furthermore, Python is a **general** purpose language
  - It is used in everything from user interfaces to web servers
  - Linear algebra is **not** a primary citizen

# Introduction: Syntax

## Python

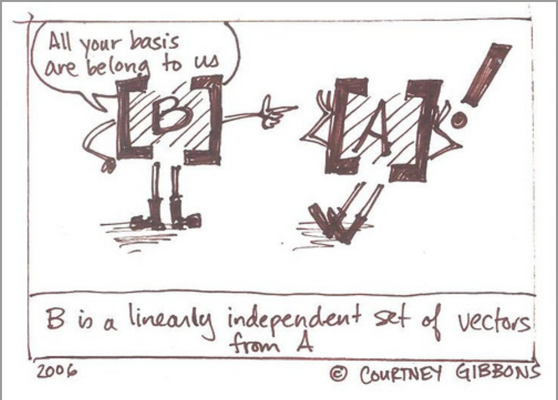
```python
def is_prime(n):
  for m in range(2, n):
    if n % m == 0:
      return False
  return True
```

## MATLAB

```matlab
function ret = is_prime(n)
  ret = true;
  for m=1:(n-1)
    if mod(n, m) == 0
      ret = false;
      break
    end
  end
end
```

Warning: In Python you **must** properly indent your code.

# NumPy: linear algebra package



# NumPy

The *de-facto* package for linear algebra is `numpy`.

Usually, it is shortened to `np`:

```
import numpy as np
```

NumPy features two main structures:

- `np.ndarray` — array of multiple dimensions
- `np.matrix` — matrix (two dimensions)

Why ever use `np.matrix` when `np.ndarray` supports two dimensions?

# NumPy vs MATLAB

Summary of **NumPy for MATLAB users** [1] [2]

| MATLAB | NumPy |
|---|---|
| `size(a)` | `a.shape` |
| `a(1:5,:)` | `a[0:5,:]` or `a[0:5]` or `a[:5]` |
| `a(end-4:end,:)` | `a[-5:]` |
| `a.'` | `a.transpose()` or `a.T` |
| `a * b` | `a.dot(b)` |
| `a .* b` | `a * b` |

The main conceptual difference is that Numpy supports **arithmetic broadcasting.** That is, you can do the following element-wise multiplication: (6,3) * (6,1). It automatically assumes you want to multiply by column. In MATLAB, you would have to use `bsxfun(@times,r,A)` or first use `repmat()`.

[1] https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html
[2] http://mathesaurus.sourceforge.net/matlab-numpy.html

# NumPy Example

Create $B_{ij} = \begin{cases} 5, & \text{if } A_{ij} > 1 \\ 0, & \text{otherwise.} \end{cases}$

Suggestions:

```
B = A.copy()
B[A > 1] = 5
B[A <= 1] = 0
```

---

```
B = np.zeros(A.shape)
B[A > 1] = 5
```

---

```
B = (A > 1) * 5
```

---

```
B = np.asarray(
  [[5 if aij > 1 else 0 for aij in ai] for
ai in A])
```

It is probably a good idea to finish this off with:

```
B = B.astype(np.int8)
```

to save memory!

# API

## Python Lists

1. Create a list – `l = [1, 5, 2]`
2. Access item – `l[1]`
3. Modify item – `l[1] = 7`
4. Add item – `l.append(8)`, `l += [9, 10]`

## NumPy

1. Create ndarray from list – `np.array()`
2. Create ndarray of zeros/ones – `np.zeros()`, `np.ones()`
3. Sample Uniform(0,1) – `np.rand`, `np.random.random()`
4. Sample Normal(0,1) – `np.randn`, `np.random.normal()`
5. Horizontal concatenation – `np.vstack()`, `np.c_[]`
6. Vertical concatenation – `np.hstack()`, `np.r_[]`
7. Any concatenation – `np.concatenate(list, axis)`

# Project: Implement Euclidean distance

Get the project files on **https://github.com/jtrpinto/ml2018**.

1. Implement a function called **euclidean_distance(a, b)**
   - Given two vectors **a** and **b**, computes the square-root of the squared difference between the two vectors
   - $d = \|x_1 - x_2\|_2$
   - TIP: you might want to use vectorisation!
2. Use it for the two example vectors given in the script
   v1 = [1.1, 2.5, 4.4, 0.1, 2.3, 3.4]
   v2 = [2.0, 2.2, 1.0, 1.0, 2.5, 3.4]

Possibly useful functions:

`np.random.randn`, `np.mean`, `np.std`, `np.sqrt`, `np.square`, `np.linalg.norm`

# Project: Implement Euclidean distance (solution)

```python
# Implement the euclidean_distance function here:
def euclidean_distance(a, b):
    # Computes and returns the Euclidean distance between vectors 'a' and 'b'
    distance = np.sqrt(np.sum(np.square(a - b)))
    return distance

# Check if it is working! Here are two vectors:
v1 = np.array([1.1, 2.5, 4.4, 0.1, 2.3, 3.4])
v2 = np.array([2.0, 2.2, 1.0, 1.0, 2.5, 3.4])

# Call the euclidean_distance function below to compute the distance between them:
dist = euclidean_distance(v1, v2)

print('Distance between v1 and v2: ', dist)  # (For these vectors, it should be something like 3.6482...)
```

# Cycles & Functions

# Loops

Python has some interesting functions that can help you in loops.

```
for n in range(10, 30, 2):
  print(n)
# 10 12 14 16 18 20 22 24 26 28

for i, n in enumerate(range(10, 20, 2)):
  print((i, n))
# (0, 10) (1, 12) (2, 14) (3, 16) (4, 18)

for a, b in zip(range(10, 20), range(40, 50)):
  print((a, b))
# (10, 40) (11, 41) (12, 42) (13, 43) (14, 44) (15, 45) (16, 46)
  (17, 47) (18, 48) (19, 49)
```

You will probably use these functions a lot in your loops: `range`, `enumerate` and `zip`.

# List Comprehension and Functools

## List comprehension

```python
[i**2 for i in range(5)]
```

```python
[i for i in range(5) if i % 2 == 0]
```

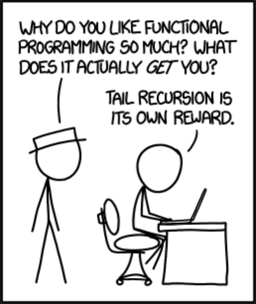
## Functools

```python
map(lambda x: x**2, range(5))
```

```python
filter(lambda x: x % 2 == 0, range(5))
```

```python
functools.reduce(lambda a,b: a+b, range(5))  # sum numbers (10)
```

- List comprehension is considered the most Pythonic approach.
- But...



# Project: Implement the k-means algorithm

Given data $X_{ij}$, the user first defines $K$ clusters.

The algorithm:

1. Random cluster centroids are selected randomly using the MacQueen method;
2. Each observation is assigned the closest cluster:
   - $c_i(t+1) = \arg\min_k \|x_i - \tilde{c}_k\|_2$
3. Compute the new centroid of each cluster (using the Euclidean distance you implemented before):
   - $\forall k, j\colon \tilde{c}_{kj} = \frac{1}{\sum_{i=1}^{N} \mathbb{1}_{c_i=k}} \sum_{i=1}^{N} \mathbb{1}_{c_i=k} X_{ij}$
4. Repeat steps 2-3 for a set number of iterations.

Please place this inside a `kmeans`$\colon \mathbb{Z}, \mathbb{R}^{N\times M} \to \mathbb{Z}^N$ function, which maps $K$ and $X$ to its respective clusters.

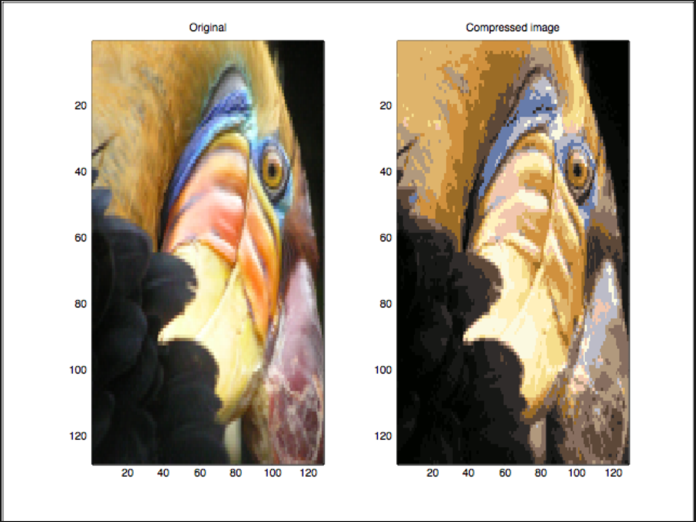
Note: you can also implement a convergence condition that stops the loop as soon as the clusters stabilise.

# Project: Implement the k-means algorithm (solution)

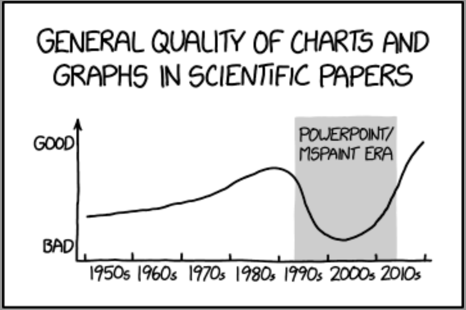
```
def k_means(X, k=3, n_iterations=10):
    # (...)
    # The k_means algorithm is iterative. Start a loop here for 'n_iterations':
    for ii in range(n_iterations):
        # In each loop, for each data point:
        for index in range(X.shape[0]):
            # Compute the Euclidean distance between the data point and each centroid:
            distance = [euclidean_distance(X[index], cc) for cc in centroids]

            # Then, assign each data point to the cluster with the nearest centroid:
            membership[index] = np.argmin(distance)

        # Now, recompute the centroids of each cluster, computing the mean of the cluster data points:
        for cc in range(k):
            centroids[cc] = np.mean(X[membership == cc], axis=0)
    # (...)
```

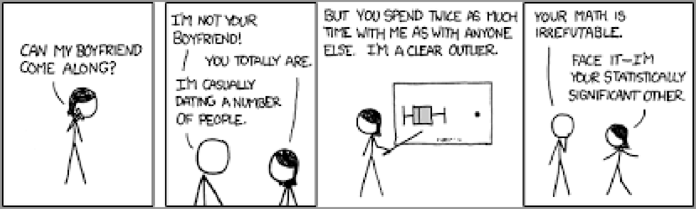
# Kmeans: usage example

K-means can be used for e.g. image compression...
24 ($8 \times 3$) bits $\rightarrow$ k=64 (6 bits)



# Matplotlib


GENERAL QUALITY OF CHARTS AND GRAPHS IN SCIENTIFIC PAPERS
GOOD
BAD
POWERPOINT/ MSPAINT ERA
1950s 1960s 1970s 1980s 1990s 2000s 2010s


CAN MY BOYFRIEND COME ALONG?
I'M NOT YOUR BOYFRIEND!
YOU TOTALLY ARE.
I'M CASUALLY DATING A NUMBER OF PEOPLE.
BUT YOU SPEND TWICE AS MUCH TIME WITH ME AS WITH ANYONE ELSE. I'M A CLEAR OUTLIER.
YOUR MATH IS IRREFUTABLE.
FACE IT—I'M YOUR STATISTICALLY SIGNIFICANT OTHER.

# Matplotlib

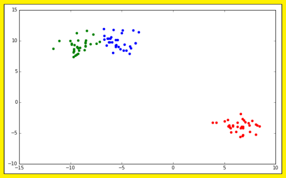
The *de-facto* package for plotting graphics in python is matplotlib.

Matplotlib contains an API[3] called `pyplot` that is inspired in MATLAB.

See: `http://matplotlib.org/1.4.3/api/pyplot_api.html`

[3] API = Application Programmer's Interface

# Matplotlib




Example using synthetically created data:

```
from matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
X, y = make_blobs(centers=3)
```

```
plt.scatter(X[y == 0, 0], X[y == 0, 1], color='red')
plt.scatter(X[y == 1, 0], X[y == 1, 1], color='green')
plt.scatter(X[y == 2, 0], X[y == 2, 1], color='blue')
plt.show()
```

OR

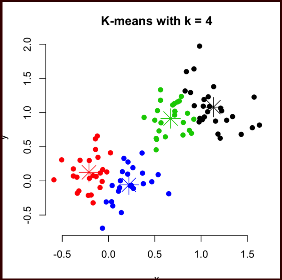
```
colors = ['red', 'green', 'blue']
plt.scatter(X[:, 0], X[:, 1], color=[colors[_y] for _y in y])
plt.show()
```

OR

```
colors = plt.cm.rainbow(np.linspace(0, 1, 3))
plt.scatter(X[:, 0], X[:, 1], color=colors[y])
plt.show()
```

# Project: k-Means in a simulated dataset

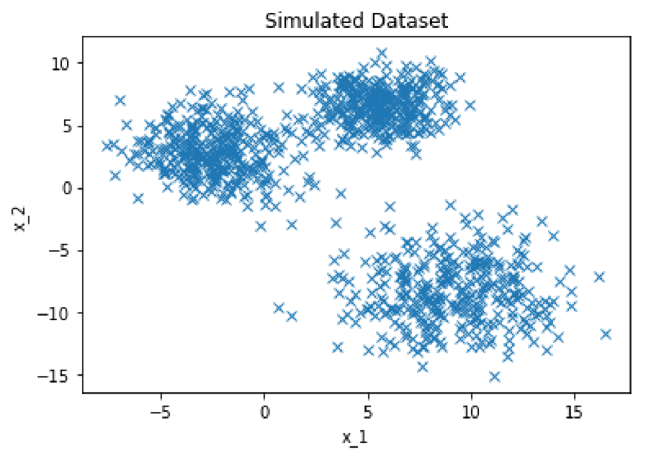
1. Use matplotlib to **plot the simulated dataset**
2. Use the implemented k-Means function to **cluster the dataset** (note: use $k = 3$)
3. Use matplotlib again, to **plot the k-Means clustering result** (use a different color for the points of each cluster)
4. At last, try to **plot the movement of the centroids** over the iterations



# Project: k-Means in a simulated dataset (solution)

```
# Now, to test it, we shall use a simulated dataset:
X, _ = additional_stuff.simulated_dataset()

# We can use matplotlib to visualise the simulated data points:
# First, we plot the data points:
pl.plot(X[:,0], X[:,1], 'x')
# Then, some additional stuff:
pl.title('Simulated Dataset')
pl.xlabel('x_1')
pl.ylabel('x_2')
pl.show()

# Now, we can apply our implementation of 'k-means' to this
    dataset:
membership, centroids, centroids_history = k_means(X, k=3)
```

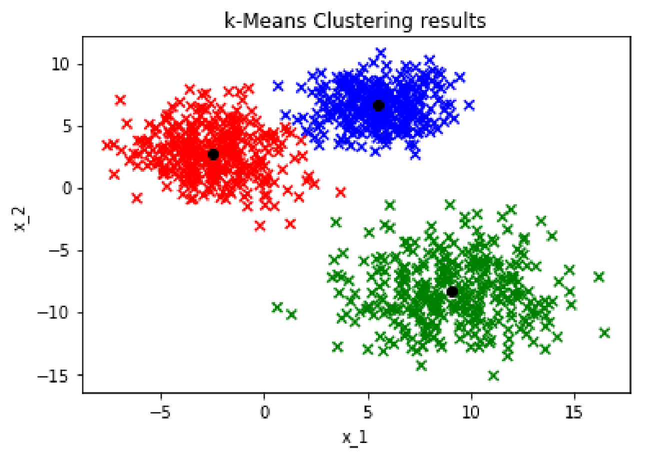
# Project: k-Means in a simulated dataset (solution)



# Project: k-Means in a simulated dataset (solution)

```
# Using matplotlib, we can plot the result:
# First, plot the data points:
colors = ['red', 'green', 'blue']
pl.scatter(X[:, 0], X[:, 1], marker='x', color=[colors[int(_y)]
    for _y in membership])
# Then, plot the final cluster centroids:
pl.plot(centroids[:,0], centroids[:,1], 'o', color='black')
# And some additional stuff:
pl.title('k-Means Clustering results')
pl.xlabel('x_1')
pl.ylabel('x_2')
pl.show()
```

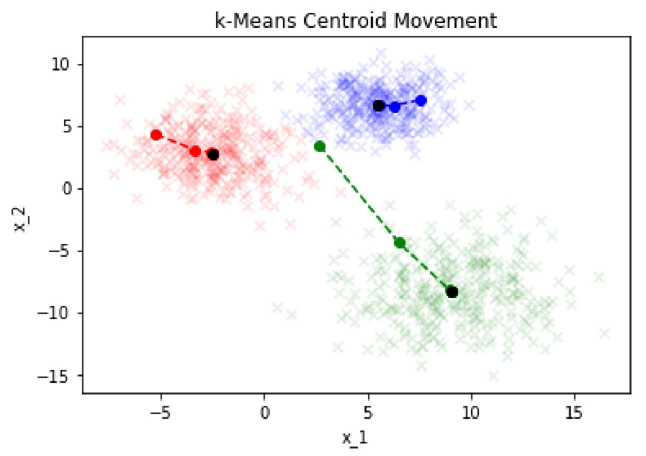
# Project: k-Means in a simulated dataset (solution)



# Project: k-Means in a simulated dataset (solution)

```
# We can also see the movement of each centroid during the iterations:
# Plot the data points again (use alpha to make them transparent):
pl.scatter(X[:, 0], X[:, 1], marker='x', alpha=0.1, color=[colors[int(_y)] for _y in membership])
# And plot each centroid's movement:
pl.plot(centroids_history[:,0,0], centroids_history[:,0,1], '--', marker='o', color='red')
pl.plot(centroids_history[:,1,0], centroids_history[:,1,1], '--', marker='o', color='green')
pl.plot(centroids_history[:,2,0], centroids_history[:,2,1], '--', marker='o', color='blue')
# Then, plot the final cluster centroids:
pl.plot(centroids[:,0], centroids[:,1], 'o', color='black')
# And some additional stuff:
pl.title('k-Means Centroid Movement')
pl.xlabel('x_1')
pl.ylabel('x_2')
pl.show()
```

# Project: k-Means in a simulated dataset (solution)



# File Handling

# Open files using Numpy

Open a CSV file and add some random noise.

```
import numpy as np
d = np.loadtxt('file.csv', [np.float64, np.float64, np.int8],
                 delimiter=',', skiprows=1)
X = d[:, :-1]
y = d[:, -1]
```

# Open files using Numpy

Open a CSV file and add some random noise.

```
import numpy as np
d = np.loadtxt('file.csv', [np.float64, np.float64, np.int8],
                delimiter=',', skiprows=1)
X = d[:, :-1]
y = d[:, -1]
```

By default numpy always works with `np.float64`. But you may want to change `dtype` when opening a CSV file or creating a vector in order to use less memory. The following types are available:

| | |
|---|---|
| `np.int` | 8, 16, 32 and 64 bits |
| `np.uint` | 8, 16, 32 and 64 bits |
| `np.float` | 16, 32 and 64 bits |
| `np.complex` | 64 and 128 bits |

Complex numbers are represented using two 32-bits or 64-bits floats.

By default numpy always works with `np.float64`. If you work with GPUs, it's a good idea to convert everything to 32-bits.

# Open images using Scikit-image

```
from skimage.io import imread
img = imread('filename.png')

print(img.shape)
# (512, 512, 3)  # if color (RGB)
```

# Pandas

If you are used to R `data.frame` then you are going to miss things like accessing columns by column name.

There is a widely used data-frame package for Python called `pandas`.

(This package is particularly useful when dealing with time series because it supports a lot of timeseries functionality we are not going to cover.)

Opening is faster than numpy...

```
import pandas as pd
df = pd.read_csv('titanic.csv')
df = pd.read_excel('a.xlsx')

df.groupby('gender').mean()
#                  age        height
# gender
# female   53.870850   158.201459
# male     53.491803   158.971695

df.to_csv('blabla.csv')
df.to_excel('blabla.xlsx')
```

Indexing

```
df['nome-coluna']

df.loc['nome-linha']
df.ix[5]   # nro linha
df.ix[:, 5]   # linha,coluna
```

# Save Session

There are two ways to save your session/variables to a file:

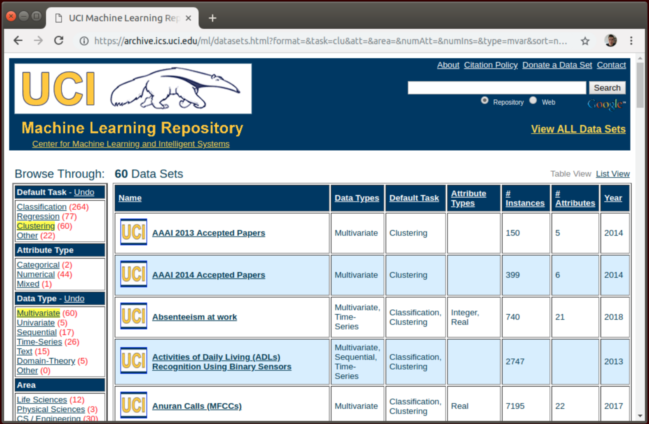
- `np.save`: numpy binary format
- `pickle`: binary file (python version-specific)
- `json`: text file (universal)

Pickle is the native Python approach:

```
import pickle
pickle.dump(d, open('save.pickle', 'wb'))
# and then you can just load it again:
d = pickle.load(open('save.pickle', 'rb'))
```

# Project: Use a real dataset

We are going to use the Iris dataset from “UCI Machine Learning”




It includes 150 samples of flowers of three classes (iris-setosa, iris-versicolor, iris-virginica) and includes measurements of sepal width and length and petal width and length.

# Project: Use a real dataset

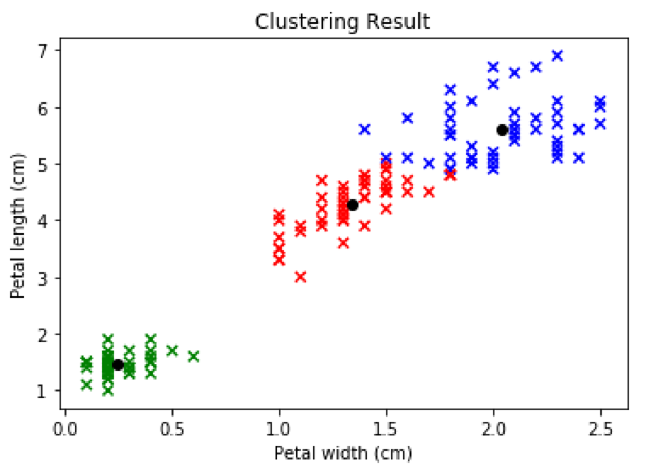
Tasks:

1. **Open 'iris.txt'** using pandas read_csv() function;
2. **Check the dataset** using Spyder's Variable Explorer;
3. Make a **numpy array with just two features** of the dataset;
4. Use k-means to **cluster the dataset** (use $k = 3$);
5. Encode the labels and **print the mutual information score**;
6. Use matplotlib to **plot the results**.

# Project: Use a real dataset (solution)

```python
# We can also apply our k-means algorithm to a real dataset.
# To do that, we can use pandas to import the dataset:
data = pd.read_csv('iris.txt')

# To use k-means, let's select just two columns of the dataset:
X = np.array(data[['petal width', 'petal length']])

# And perform k-Means clustering with k = 3 (no. of classes)
membership, centroids, _ = k_means(X, k=3)

# Getting and encoding the labels:
# (...)

# And plot the results:
colors = ['red', 'green', 'blue']
pl.scatter(X[:, 0], X[:, 1], marker='x', color=[colors[int(_y)]
    for _y in membership])
pl.plot(centroids[:,0], centroids[:,1], 'o', color='black')
pl.title('Clustering Result')
pl.xlabel('Petal width (cm)')
pl.ylabel('Petal length (cm)')
pl.show()
```

# Project: Use a real dataset (solution)



# Project: Use an image

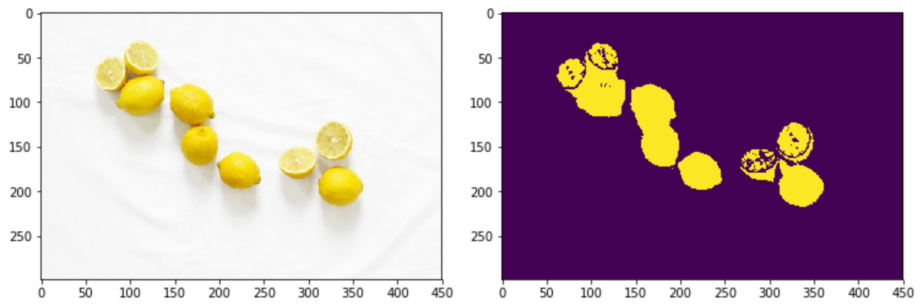
Tasks:

1. **Open the image** 'lemons.jpg' using scikit-image `io.imread()` function;
2. **Visualise the image** using matplotlib `imshow()` function;
3. Change the shape from $(w, h, c) \rightarrow (w \times h, c)$ with numpy's `reshape()` function;
4. Use **k-means on the reshaped image**;
5. Reshape the k-means result back: $(w \times h, c) \rightarrow (w, h, c)$;
6. **Visualise the result** using the `imshow()` function;
7. **Save the result** using scikit-image: `io.imsave(filename, image)`.

# Project: Use an image (solution)

```
# First, we use scikit-image to import the image:
image = io.imread('lemons.jpg')

# Visualise the image using matplotlib:
pl.imshow(image)
pl.show()

# Each pixel should be considered a single sample (in this case, with three
# features - RGB). We need to restructure the image accordingly:
image_reshaped=image.reshape((image.shape[0]*image.shape[1], 3))

# Now, we finally apply k-means to it:
membership, _, _ = k_means(image_reshaped, k=2)

# Reshaping and visualising the result:
result = membership.reshape((image.shape[0], image.shape[1]))
pl.imshow(result)
pl.show()

# Save the result as an image:
io.imsave('result.jpg', result)
```

# Project: Use an image (solution)



# Classes & Modules

# Classes

```
class Animal:
  def talk(self):
    raise NotImplementedError('Implement me')

class Cat(Animal):
  def talk(self):
    print("Miau")

class Human(Animal):
  def talk(self):
    print("Bla bla")

Cat().talk()  # what is the output?
```

Two notes:

- Python is very dynamic: there are no formal contracts like abstract methods
- in Python, the reference to the object itself is passed explicitly

# Modules

Different ways to access a module:

- ```python
  import fib
  fib.whatever()
  ```
- ```python
  from fib import whatever
  whatever()
  ```
- ```python
  from fib import *
  whatever()
  ```

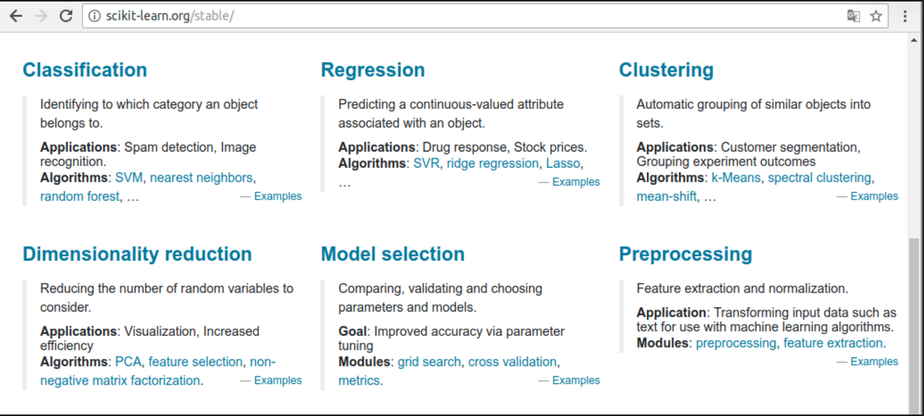
  (import everything into your namespace — usually not recommended)

Inside each directory, you ~~must~~ may have a `__init__.py` file where you can put initialization code.
  ```

# Scikit-learn

# Scikit-learn

Scikit-learn has literally **everything!**


scikit-learn.org/stable/

## Classification

Identifying to which category an object belongs to.

**Applications**: Spam detection, Image recognition.
**Algorithms**: SVM, nearest neighbors, random forest, ... — Examples

## Regression

Predicting a continuous-valued attribute associated with an object.

**Applications**: Drug response, Stock prices.
**Algorithms**: SVR, ridge regression, Lasso, ... — Examples

## Clustering

Automatic grouping of similar objects into sets.

**Applications**: Customer segmentation, Grouping experiment outcomes
**Algorithms**: k-Means, spectral clustering, mean-shift, ... — Examples

## Dimensionality reduction

Reducing the number of random variables to consider.

**Applications**: Visualization, Increased efficiency
**Algorithms**: PCA, feature selection, non-negative matrix factorization. — Examples

## Model selection

Comparing, validating and choosing parameters and models.

**Goal**: Improved accuracy via parameter tuning
**Modules**: grid search, cross validation, metrics. — Examples

## Preprocessing

Feature extraction and normalization.

**Application**: Transforming input data such as text for use with machine learning algorithms.
**Modules**: preprocessing, feature extraction. — Examples

# Scikit-learn interface

| | |
|---|---|
| **Estimator:** | Implements the fit method to learn from data.<br>For Supervised Learning:<br>`estimator = estimatorObj.fit(data, labels)`<br>and Unsupervised Learning:<br>`estimator = estimatorObj.fit(data)` |
| **Predictor:** | `labels = predictorObj.predict(data)`<br>May implement `predict_proba` to return the degree of certainty. |
| **Transformer:** | Filters or modifies the data:<br>`new_data = transformerObj.transform(data)` |
| **Model:** | Measures goodness of fit:<br>`score = modelObj.score(data, labels)` |

# Project: Apply K-means with sklearn

Tasks:

1. Find the sklearn documentation for the K-means algorithms;
2. Go to the code you have previously written for the Iris dataset and substitute your k-means function by the KMeans object of sklearn. You should:
   - **2.1** Create the object.
   - **2.2** Estimate the parameters with the data (fit(.)).
   - **2.3** For each sample predict the assigned cluster (predict(.)).
3. Plot and check you have obtained an equivalent result.

# Project: Apply K-means with sklearn

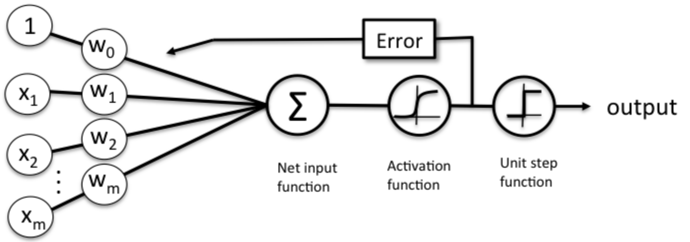
Tasks:

1. Find the sklearn documentation for the K-means algorithms;
2. Go to the code you have previously written for the Iris dataset and substitute your k-means function by the KMeans object of sklearn. You should:
   2.1 Create the KMeans object.
   2.2 Estimate the parameters with the data (fit(.)).
   2.3 For each sample predict the assigned cluster (predict(.)).
3. Plot and check you have obtained an equivalent result.

Solution:

```
from sklearn.cluster import KMeans as kmeans
model = kmeans(n_clusters=3)
model.fit(X)
membership = model.predict(X)
centroids = model.cluster_centers_
```

# Classification with Logistic Regression




**Schematic of a logistic regression classifier.**

# Project: Apply Logistic Regression for Classification

We now focus on the problem of categorizing data.

1. Find the sklearn documentation for the K-means algorithms;
2. Split the data into training and testing.
3. Create the LogisticRegression object and estimate the parameters with the training data.
4. Obtain the accuracy of the model in the training and testing data (score(.)).

Further exercises:

5. Run the same code again. Did you obtain the same result?
6. Try the same exercise with another classifier.

Useful functions: `train_test_split`, `LogisticRegression`, fit, score.

# Project: Apply Logistic Regression for Classification (solution)

Solution:

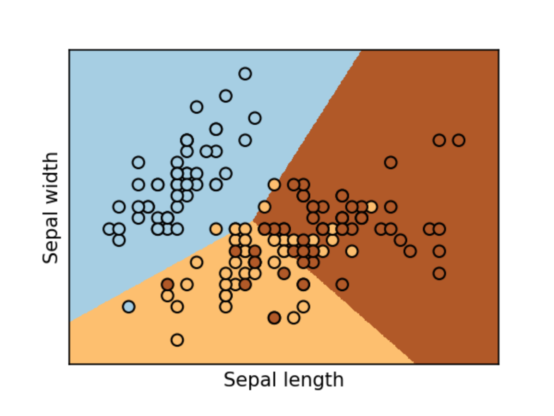
```
1        from sklearn.linear_model import LogisticRegression
2        X_train, X_test, y_train, y_test = train_test_split(X, y)
3        logreg = LogisticRegression()
4        logreg.fit(X_train, y_train)
5
6        print("Train set: ", logreg.score(X_train, y_train))
7        print("Test set: ", logreg.score(X_test, y_test))
```

Note: If you want to obtain the same exact results you need to remove the randomness in the process. Some common sources of variation are the train and test split and the initial parameters of the logistic regression.

Note: Because different classifiers have the same interface, we can simply substitute the 3rd line by another model. In this way we can quickly estimate the performance of multiple models for any machine learning problem we want to solve.

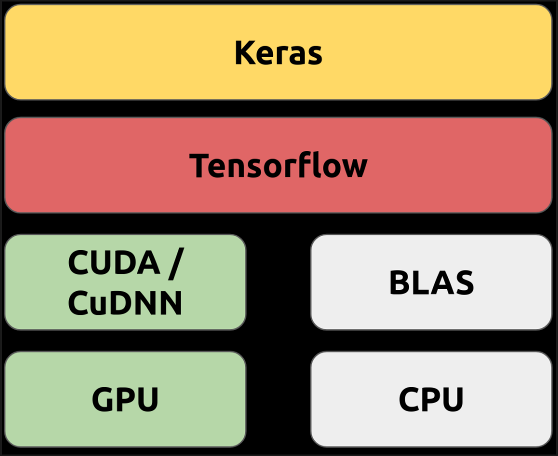
# Plotting the Feature Space

Using the provided function (plot_fs()) plot the feature space. This function might be useful for other projects!



# Keras & TensorFlow

# Keras & TensorFlow



# Keras

Keras is a high-level neural networks API.

Models in Keras (and Tensorflow) are defined as a graph of operations which transform the data from the input to the output. The simplest possible model in Keras is a `Sequential` which represents a linear stack of operations. We construct the model by adding layers:

```
from keras.layers import Dense
from keras.models import Sequential
model = keras.Sequential()
model.add(Dense(64, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
```

After creating the model we need to compile it before we start the training process.

```
model.compile(loss='categorical_crossentropy', optimizer=
    'sgd', metrics=['accuracy'])
```

# Keras

After a Keras model is compiled its interface is similar to models in sklearn:

| Sklearn | Keras |
| --- | --- |

Fit the training data:

**Sklearn**

```
model.fit
```

**Keras**

```
model.fit
```

Predict the output for input data:

**Sklearn**

```
model.predict
```

**Keras**

```
model.predict
```

Evaluate the performance for input data:

**Sklearn**

```
model.score
```

**Keras**

```
model.evaluate
```

# Logistic Regression in Keras

A Logistic Regression classifier can easily be implemented in Keras as a one layer neural network. For :

```
model = Sequential()
model.add(Dense(n_categories, input_dim=n_inputs, activation=
    'softmax'))
# The model is optimized by stochastic gradient descent with
    momentum
sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(optimizer=sgd, loss='categorical_crossentropy',
     metrics=['accuracy'])
```

Note that although we are doing logistic regression the structure and optimization of this model is different from the sklearn's logistic regression. More on this will be explained on the lecture about logistic regression.

# Project: Implement Logistic Regression in Keras

We now focus on the problem of categorizing data.

1. Find the keras documentation for the sequential model;
2. Go to the code you have previously written for classification of the Iris dataset and substitute the sklearn implementation of the logistic regression by your own keras implementation.
3. Fit the training data and obtain the accuracy for the train and test data.
4. Obtain a plot of the feature space (use plot_fs())

# When to use sklearn vs Keras

Keras is used to work with neural networks. It allows faster training due to using GPU accelaration. Generally, if you are working with `Deep Learning` models you will be using Keras or a similar module. For traditional machine learning algorithms you use sklearn.

In this course sklearn will be more useful.

# MNIST in Keras Demo

An example where modules like keras are used often: Image Recognition with Convolutional Neural Networks.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
    activation='relu',input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adadelta(), metrics=['accuracy'])
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          verbose=1,
          validation_data=(x_test, y_test))
score = model.evaluate(x_test, y_test, verbose=0)
```

Logistic Regression: 81%
CNN: 99%

# Finishing Touches

**Pointers**

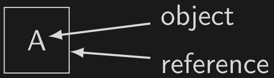
# References

In programming languages when calling a function, there are two possible behaviors:

- pass by value (or copy)
- pass by reference

Python uses what some people like to call 'pass-by-object-reference'. Everything is an object and the reference to the object is copied (not the object itself).

```python
def fn(b):
  print(id(b))
  print(a is b)
a = []
print(id(a))
fn(a)
```




Output:

```
139663236727240
139663236727240
True
```

# References

```
a = [1,2,3]
b = a
b[1] = 7
print(a)
```

What is the value of a?

```
a = [1,2,3]
b = a
b = [1,7,3]
print(a)
```

What is the value of a?

```
a = 2
b += 5
print(a)
```

What is the value of a?

# Python2 vs Python3

Python2 will be discontinued in 2020. The major differences are that:

- In Python3, functions like `range()` are generators/iterators, while in Python2 they are lists. In most use cases, this will make no difference to you.
- In Python3, strings are codified in Unicode by default. This makes it nicer to work with when using, for instance, Portuguese characters like ç or á.
- Probably the when you'll notice the most is that in Python2 `print 'Hello World'` was valid. In Python3, you need to use parenthesis, `print('Hello World')`
- There are many other small ones. **Most importantly, not all packages are yet compatible with Python3.** (Though 99% are.)

# Python Implementations

There are several implementations of the Python language:

- CPython
  - $\boxed{\text{source code}} \xrightarrow{\text{is compiled to}} \boxed{\text{bytecode}} \rightarrow \text{execute}$
  - CPython is the most popular implementation of Python, and is developed by the Python Foundation
  - This is the implementation people *mean* when they say Python
- PyPy
  - $\boxed{\text{source code}} \rightarrow \boxed{\text{bytecode}} \rightarrow \text{execute} \leftrightarrow \boxed{\text{machine code}}$
  - This is also a very popular and highly supported implementation
  - Faster; this is how Java and MATLAB work
- Jython
  - $\boxed{\text{source code}} \rightarrow \boxed{\text{Java bytecode}}$
  - The Java interpretor: $\boxed{\text{bytecode}} \rightarrow \text{execute} \leftrightarrow \boxed{\text{machine code}}$
  - It outsources work to Java

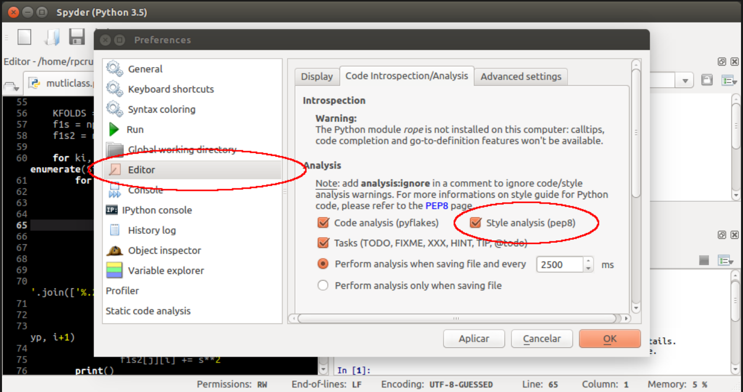
# PEP8

Using a pretty style makes reading your code much easier.

In Python, there is a coding style that is recommended called PEP8.

**Exercise:** Enable code style checking in Spyder and fix the errors.

Under the Preferences window:




Eduardo Castro emcastro@inesctec.pt
Wilson Silva wilson.j.silva@inesctec.pt
Tiago Gonçalves tiago.f.goncalves@inesctec.pt