Introduction to Python

Carl Henrik Ek

Week 1

Abstract

In this worksheet we will have a look at some operations essential to machine learning and at the same time become comfortable with `Python` as a programming language and how to use the packages `numpy`, `scipy` and `matplotlib`. There is a lot of tutorials on how to use `Python` out there[1] and this is by no means a complete one. We have tried to focus on the elements that you will need to work on the labs in the unit. Importantly, many of you are very likely much better programmers than me, in this case trust your own judgement and don't list too much to what I have to say about programming.

1 Environment

The first thing we are going to do is to install an environment and all the necessary packages. To not make a mess of your previous installs we are going to create a new environment. Follow the

1. Download 64-bit Linux Miniconda installation script for Python 3.X from here.
2. From a terminal execute

   Code
   ```
   sh <path-to-folder>/Miniconda3-latest-Linux-x86_64.sh
   ```

3. Scroll through the license (press the space bar to move through quickly), type 'yes' to approve the terms, and then accept all the installation defaults.
4. Close the Terminal.
5. Open the Terminal again and execute the following commands,

   Code
   ```
   conda update conda
   conda create -n myenv python
   conda install -n myenv jupyter scipy numpy matplotlib
   ```

6. Now we have installed the necessary packages and created a miniconda environment called `myenv` which can be started by,

   Code
   ```
   source activate myenv
   ```

[1] an excellent *cheat-sheet* can be found here.

1.0.1 Development

Now there are many ways that you can write the code and perform the experiments. If you have a sadistic nature you could even run them `Matlab` but that would be wrong on so many levels.

Option 1 stand-alone python, pick your editor of choice[2] simply execute the code in python. Say that you write some code called `coms30007.py` you simply open a terminal and run,

Code

```
python coms30007.py
```

Option 2 Using the `iPython` interactive shell. This approach allows you to open a shell where the variables that is generated by your code are resident. This means that you can interactively play around with your workspace. This is my preferred method when I work as the machine learning process is often very interactive. First open the `iPython` shell from a terminal,

Code

```
ipython
```

You can now run code and interact with the results. Lets say that we have some code called `coms30007.py` that creates a variable `x`.

Code

```
run coms30007
print(x)
```

Option 3 Using `Jupyter Notebook`. This is a browser based way to interact with the `iPython` shell. Start the `Jupyter` server by executing the following in a terminal and follow the instructions. Personally I find it dreadful to write code in a browser but for `emacs` users there is obviously a better alternative.

Code

```
jupyter notebook
```

Do play around with the different methods and find the one that is most suitable for you. The labs are here not for you to simply finish a bit of code and be done but rather to play around with code to get a better understanding of the underlying theory.

2 Python

Programming languages comes and goes but currently Python is the king of the hill when it comes to machine learning. The main benefit of Python is that it is very easy to interact with other languages, parsing data is super simple and there now exists good nummerical libraries and tools for plotting. If you are using a interactive Python shell you can write `foo.bar?` to print out the help of `foo.bar`, if you write `foo.bar??` you instead print the actual implementation.

[2] emacs obviously

2.1 Numpy

The main library that we use to build machine learning models and methods will be `Numpy`. Numpy is very well implemented and contains a lot of useful functions to work on vectors and matrices. The normal convention is to import numpy as `np`. To create vectors and matrices you use `np.array`.

Code

```
import numpy as np
x = np.array([1, 2, 3]) # create 3 vector
y = np.array([[2], [3], [4]]) # create a 3x1 vector
A = np.array([[1, 2, 4],[2, 6, 8],[3, 3, 3]]) # create a 3x3 matrix

# print dimensionality
print('x:', x.shape, 'y:', y.shape, 'A:', A.shape)
```

There are several prebuilt constructions that returns commonly used matrices,

Code

```
np.zeros((2,2))      # create a 2x2 zero matrix
np.ones((2,3))       # create a 2x3 matrix of ones
np.eye(5)            # create a 5x5 identity matrix
np.empty((3,4))      # create a 3x4 placeholder matrix
np.arange(1,9,2)     # create a vector with values from 1 to 9 with increment of 2
np.linspace(0,1,100) # create a vector of 100 linearly spaced values between 0 and 1
```

Often we want to inspect a matrix/vector and get its dimensions/shape this can be done as follows,

Code

```
x = np.eye(5)
x.shape
```

2.1.1 Arithmetic

The normal operations for addition and subtraction have been overloaded to `numpy` objects, so we can simply use `+,-,*,/` as we would normally do. Importantly, the operations are element-wise so the do not implement matrix arithmetics. Lets first generate some data,

Code

```
import numpy as np

x = np.arange(3).reshape(3,1) # 3x1 vector
y = np.arange(3).reshape(1,3) # 3x1 vector
A = np.arange(9).reshape(3,3)  # 3x3 matrix
```

As long as we are working with objects with the same dimensionality everything makes but you can also do a few more involved operations where the results are a bit more surprising. Try the following operations out and make sure you understand what they do.

Code

```
print(x-y)
print(y-x)
print(A-x)
print(A-y)
print(x*y)
print(A*x)
print(x*A)
print(A*y)
```

To work with matrix operations we use the following notation,

Code

```
A.dot(x)        # scalar product
A.dot(y.T)      # scalar product with transpose of y
np.outer(x,y.T) # outer product
```

Now its time to introduce one of the annoying things with `numpy` its very relaxed Broadcasting rules. Test the following,

Code

```
A = np.arange(9).reshape(3,3)
x = np.arange(3)
y = np.arange(3).reshape(3,1)
A.dot(x)
A.dot(x.T)
A.dot(y)
A.dot(y.T)
```

The last dot-product above should fail, which makes sense as you are trying take the scalar product between a `3x3` matrix and a `1x3` vector. However both the first and the second dot-product computes, isn't this weird? Yes it is, but what is happening is that in numpy you can create arrays which have 1 dimension. If you check `x.shape` you will se that it is `(3,)`. This is very annoying as effectively it picks the shape of the vector to be whatever is needed to get the operation to work. One way to avoid this is to always reshape vectors so that they have a two dimensional shape. You can easily do this by,

Code

```
x = x.reshape(-1,1)
```

where I have used the `-1` notation which means if you take the index of a vector which is `-1` you get the last index. So for the vector above replacing `-1` with `3` would have the same effect. However many functions that you will call in numpy assumes that a vector have the shape `(dim,)` so when calling them you have to revert back to the one-dimensional numpy array, see below for an example that you will probably use a lot,

Code

```
mu = np.ones(2).reshape(-1,1)
x = np.random.multivariate_normal(mu.flatten(), np.eye(2))
```

Here is another strange thing,

Code

```
x = np.ones((32,1))
y = np.ones(32)
z = x+y
print(z.sum())
```

at least I was a bit surprised by the result of this computation. So my advice, make sure that each vector is of shape `(dim, 1)` and use `.flatten()` when required.

To keep things consistent I try to always follow this convention for the shape of my data, vectors are column vectors and when I concatenate `N` `d` dimensional vectors into a matrix I do it so that each row of the matrix corresponds to a data-point thereby generating an `N` by `d` matrix.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1d} \\ x_{21} & x_{22} & \cdots & x_{2d} \\ \vdots & & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{Nd} \end{bmatrix} \tag{1}$$

This notation might feel rather unatural, and in many ways it really is but its a legacy from computer science. In statistics one would concatenate $\mathbf{X}$ such that each column is a vector as this makes more mathematical sense. As an example, say that I have a linear mapping from a d_2 dimensional to a d_1 dimensional space parametrised as a matrix $\mathbf{A}$. Now I can evaluate the mapping of a point $\mathbf{x}_i$ as,

$$\underbrace{\mathbf{y}_i}_{d_1 \times 1} = \underbrace{\mathbf{A}}_{d_1 \times d_2} \cdot \underbrace{\mathbf{x}_i}_{d_2 \times 1},$$

just as we are used to. The difference comes if we want to evaluate `N` data-points at the same time. By concatenating the data as a statistician its follows naturally,

$$\underbrace{\mathbf{Y}}_{d_1 \times N} = \underbrace{\mathbf{A}}_{d_1 \times d2} \cdot \underbrace{\mathbf{X}}_{d_2 \times N}$$

while the concatenation used in machine learning will be the same in the individual point case it will be rather different in the matrix case as,

$$\underbrace{\mathbf{Y}}_{N \times d_1} = \underbrace{\left(\underbrace{\underbrace{\mathbf{A}}_{d_1 \times d_2} \cdot \underbrace{\mathbf{X}^{\mathrm{T}}}_{d_2 \times N}}_{d_1 \times N} \right)^{\mathrm{T}}}_{N \times d_1} = \left(\mathbf{X}^{\mathrm{T}}\right)^{\mathrm{T}} \mathbf{A}^{\mathrm{T}} = \mathbf{X}\mathbf{A}^{\mathrm{T}}.$$

So in the statistics case everything works out naturally and the order of the operation stays the same while in the computer science notation it doesn, why would anyone want to use the latter. Personally I believe this is a legacy from thinking of implementation first. If I have a matrix in memory I most often want to access each dimension in turn, this means that if I have a pointer at the start of the matrix I can do this by incrementing the pointer one step at the time. In the statistics notation I will have to increment the pointer with the number of columns in the matrix. I don't know if this is true but its the best explanation I can come up with. Sadly this notation have gotten stuck in machine learning so in the literature you will see the row-vector concatenation, to that end we will use this notation throughout the unit.

2.1.2 Linear Algebra

To compute the inverse of a matrix we want to solve the following system of linear equations for an unknown $\mathbf{A}$ and a known $\mathbf{M}$,

$$\mathbf{A}\mathbf{M} = \mathbf{I}.$$

The most straightforward way to do this is to call the matrix inverse function in `np.linalg`,

Code

```
# generate a random matrix
M = np.random.randn(100).reshape(10,10)

# compute the inverse of M
A = np.linalg.inv(M)

# print the diagonal elements
print(np.diag(A.dot(M)))

# pring the sum of the absolute values of the off-diagonal elements
print(np.abs(A.dot(M)).sum() - np.abs(A.dot(M))[np.diag_indices(M.shape[0])].sum())
```

In the code above we print first the values of the diagonal elements using the function `np.diag()` which extracts them from the matrix. Then we print the sum of the off-diagonal elements using another useful function `np.diag_indices()` this function returns the indices of the diagonal elements. As you can see the diagonal elements are all `1.0` while the off-diagonal elements are all very close to `0`.

1. **EXTRA** Cholesky Decomposition Now computing the inverse of a matrix is something that we will do a lot and its a very expensive operation with cubic complexity, not something that you want to have in your inner-loop. Furthermore `np.inv` is a completely general function and often we have matrices with specific structures or types. This is something that we should take into account in our computation[3]. The Cholesky decomposition is valid for any positive definite matrix as,

   $$\mathbf{M} = \mathbf{L}\mathbf{L}^{\mathrm{T}},$$

   where $\mathbf{L}$ is an lower-triangular matrix. As the decomposition is unique to the matrix it is first much cheaper to store as it has fewer values. So lets that we want to compute the inverse in order to solve the following system of equations,

   $$\mathbf{A}\mathbf{x} = \mathbf{b},$$

   where $\mathbf{A}$ and $\mathbf{x}$ is known and $\mathbf{b}$ is unknown. Furthermore we know that the matrix $\mathbf{A}$ is positive-definite[4]. We know that we can compute the inverse of $\mathbf{A}$ and get the solution by,

   $$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b},$$

   but this is not exploiting the fact that we know that $\mathbf{A}$ is positive definite. In this case we can exploit the Cholesky decomposition of a matrix to good effect making the computation both faster and more stable.

[3] Good blog post on this
[4] You will see this a lot later in the unit

Code

```
# generate a "hopefully" PSD matrix
A = np.random.randn(100).reshape(10,10)
A = A.dot(A.T)+np.eye(A.shape[0])

# Ax=b
b = np.ones(A.shape[0]).reshape(-1,1)

# compute cholesky factor
L = scipy.linalg.cholesky(A, lower=True)
# solve system of equations using cholesky
x = scipy.linalg.cho_solve((L, True), b)

# print the "difference" between the two
print(np.abs(b-A.dot(x)).sum())
```

Most of the problems that we do in this unit you will involve small and reasonably well behaved matrices so directly computing the inverse will be fine but when you get to large problems understanding what is stable and efficient is key to writing well behaved and efficient code.

2. Eigenvalues We often want to do eigen-decompositions of matrices as this representation is much more interpretable. The eigen-decomposition takes a square matrix `M` and decomposes it into,

$$\mathbf{M} = \mathbf{VUV}^{\mathrm{T}},$$

where $\mathbf{V}$ are the eigenvectors which forms an orthonormal basis and the diagonal matrix $\mathbf{U}$ which have the eigenvalues on the diagonal. This can easily be done using,

Code

```
# create a hopefully PSD matrix
M = np.random.randn(100).reshape(10,10)
M = M.dot(M.T)+np.eye(M.shape[0])

# compute eigen decomposition
U, V = np.linalg.eig(M)

# print element-wise error
print(np.abs(V.dot(np.diag(U).dot(V.T))-M).sum())
```

as you can see above when calling `np.diag` with a vector as an argument it creates a matrix with the vector on the diagonal.

2.2 Indexing

One of the real benefits of `numpy` and `Python` in general is how easy it is to *slice'n'dice* objects and apply operations only on a subset of the matrix. A matrix is indexed as `[row, col]` with the first index being `0`. There are several different indexing modes. For `numpy` arrays its important to know that there is no copying of data involved, you are simply providing a different view of an array by indexing. A good guide to the indexing modes in `numpy` is available here

2.2.1 Direct

Matrices in `Python` are indexed by `[row, col]`, in order to access all elements you can use a `:` as in the examples below,

Code

```
# create a 10x10 matrix
A = np.arange(100).reshape(10,10)

# differnet direct index modes
print(A[0,0]) # first row and column
print(A[:,0]) # all rows of first column
print(A[4,:]) # all columns of fourth row
print(A[:])   # all elements
```

You can also use the `:` notation to access parts of the matrix. The notation `[i:j,3]` implies column `3` and row `i` up till, but not including `j`. If you do not explicitly write identify the `start` it is assumed to be `0`

Code

```
print(A[1:3, 2])  # access elements at row 1,2 and column 2
print(A[:3, 2])   # access element at row 0,1,2 and column 2
print(A[:2, 4:6]) # access element at row 0,1 and column 4,5
```

As we have seen before index `-1` implies the last index,

Code

```
print(A[-1,-1])  # access the element at the last row and column
print(A[:-1,0])  # access all the rows except for the last of the first column
print(A[0:-1,0]) # same as the above
print(A[:10, 0]) # access all the row of the first column
```

2.2.2 Operations along axes

Many operations have an `axis` argument, what this argument indicates is the `axis` of the matrix where the operation is applied to. So as an example

Code

```
print(A.mean())        # compute the mean value of the matrix
print(A.mean(axis=0))  # compute the mean value of each column of the matrix
print(A.mean(axis=1))  # compute the mean value of each row of the matrix
```

Many commands take `axis` arguments so its worth looking at the documentation by `?` as it can often lead to cleaner code. A very useful example is `np.sort` where you can decided to sort column or row-wise.

2.2.3 Boolean

Many operations that are boolean in `Python` are vectorised. For example, if you write,

Code

```
A = np.random.randn(10,10)
B = A>0
```

`B` will be a boolean matrix with `True` at locations where the element is larger than 0 and `False` elsewhere. The nice thing about this is that we can use the matrix `B` as a means of indexing a matrix. So say that we would want to get all the values of the matrix `A` which are positive we could simply do,

Code

```
A = np.random.randn(10,10)
A[A>0]
```

This can be used to write very compact and interpretable code and I find it especially useful for drawing conditional plots when I am just interested in plotting points that fore fill a certain criteria.

2.3 Matplotlib

A friend of mine explained machine learning not as programming but as debugging[5]. His analogy of the machine learning task is that you walk down the street and you find a hard-drive lying on the street, you know that there is something important on this but that's all you know. You have no idea what file-system it is, what the important information is etc. So what do you do, you start poking it with all the tools that you have to figure out what the important information is. A very important tool for this process is to visualise your data, plot different things to give you a visual understanding of the data. However horrible `Python` is as a language the biggest benefit it has is the amazing visualisation packages that exists. We will use `matplotlib` a lot but there are other more specific packages for plotting things like distributions[6]. We will follow the convention of importing `matplotlib.pyplot` as `plt`.

Disclaimer

In the following examples there will be a set of commands towards the end of the code that saves the image and a return statement, please ignore these its just so that the image gets saved and included in the LaTeX document. You can replace this by `plt.show()` as this will display the figure you have generated.

There are mainly two objects that you need to know about, `figures` and `axes`. The former corresponds to a "physical" figure while the latter corresponds to sections of the latter. You first create a `figure` and then you add `axes` to this figure. In the example below we create a single `figure` called `fig` and then we add `2*3=6` subplots to this figure and call the call the first one of these `ax`.

Code

```
import matplotlib.pyplot as plt #import packages

fig = plt.figure(figsize=(10,5)) #create figure handle
ax = fig.add_subplot(231)        #create axis handle
```

Once we have create a figure we can do some simple plotting, in the example below we create a figure with one row and two columns and create a simple line plot and a scatter plot. These are really

[5] `http://inverseprobability.com/2017/03/14/data-science-as-debugging`
[6] A very useful package you might want to look at is seaborn

Code

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

fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

# create data
x = np.linspace(-np.pi, np.pi, 100)
y = np.sin(x)

# line plot
ax1.plot(x, y, linewidth=2.0, color='b')

# add noise
y += 0.3*np.random.randn(y.shape[0])

# scatter plot
ax2.scatter(x, y, s=20, color='r')

# IGNORE THESE
plt.tight_layout()
plt.savefig(path, transparent=True)
return path
```

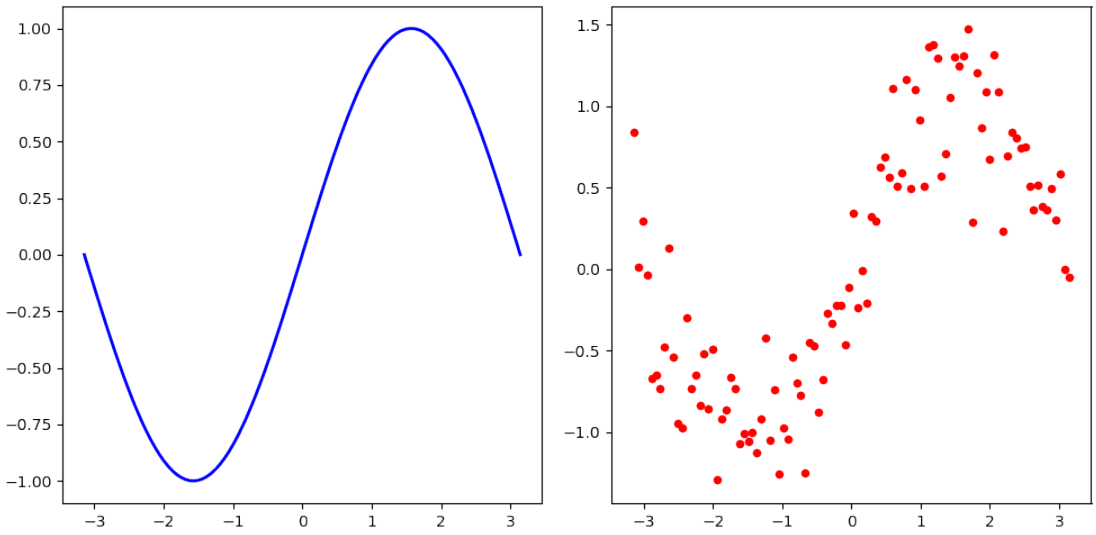

Figure 1: The left most pane is associated with `ax1` and the rightmost with `ax2`

Code

```
ax.set_xlim()
ax.set_xticks()
```

We will often work with rather large matrices in machine learning and getting an idea of how they are structured is often very telling. If the matrix consists of real values we can simply consider it to be an image and visualise it in this way.

Code

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

M = np.sort(np.random.randn(256,256))
fig = plt.figure(figsize=(10,5))
ax1 = fig.add_subplot(121)
im1 = ax1.imshow(M)
ax2 = fig.add_subplot(122)
im2 = ax2.imshow(20*M)
ax2.set_yticks([])
plt.colorbar(im1, ax=(ax1))
plt.colorbar(im2, ax=(ax2))

plt.savefig(path, transparent=True)
return path
```

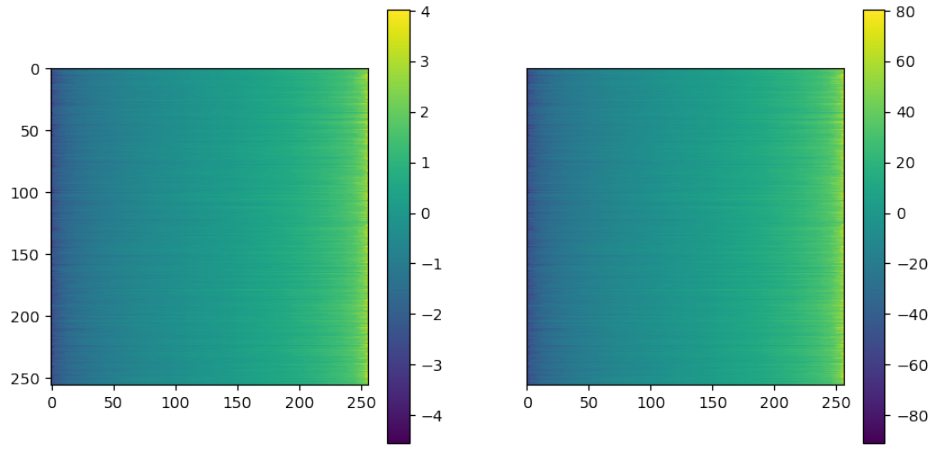

Figure 2: Figure showing how we can visualise matrices as images. Note that `imshow()` rescales the colours so to the range of the values in the matrix. The colourbars to the left shows the different maps used.

There is lots more to `matplotlib` than this and its really an excellent package, if you are interested in learning more look at the documentation here.

3 Distributions

`Numpy` has a class called `random` which contains a lot of different distributions. The distributions in `np.random` are used for generating samples from distributions. To see all the functions/members of the object `object` you can execute.

Code

```
{ k:v for k,v in vars(object).items() if not k.startswith('_') }
```

Hopefully you will recognise quite a few of the names of distributions under here. Most of the distributions have the same structure. The distribution that we will use the most in the unit is the `Gaussian` or `Normal` distribution so let us have a

Code

```
loc = np.array([0,0])
scale = np.array([[1, 0],[0, 1]])
f = np.random.normal(loc, scale)
```

In many cases we are instead interested in the actual probability functions rather than samples from it. We can access them through the package `scipy.stats` and directly create random variables. In next weeks lab we will use the `beta` distribution so lets start with this. We first import the distribution from `scipy.stats` and then we create an instance of this which we now will refer to as a *random variable*. This is a concept that will be central to anything that we do in machine learning. In most cases we are used to *deterministic variables* these are variables that takes a specific value, for example in C syntax `int a = 3` is a variable `a` that takes the value `3`. In contrast a random variable is a variable that takes values following a specific distribution as an example,

$$\epsilon \sim \mathcal{N}(0, 1),$$

is a variable ϵ that takes values that follows a normal distribution with mean `0` and variance `1`. Don't worry if this is not clear right now, you will see a lot of this over the next few weeks. When we have created the random variable `rv` in the code below we have access to lots of different functions that we can interact with. Below we plot the *density* the *cummulative density* and also sample from the variable.

Code

```
from scipy.stats import beta
import numpy as np
import matplotlib.pyplot as plt

# create random variable
rv = beta(10.0, 10.0)

# create an index set for the distribution
index = np.linspace(0+1.0e-8, 1, 100)

# create plot
fig = plt.figure(figsize=(10,5))
ax = fig.add_subplot(111)

# plot density function
ax.plot(index, rv.pdf(index), color='g')
ax.fill_between(index, rv.pdf(index), color='g', alpha=0.3)

# plot cumulative density function
ax.plot(index, rv.cdf(index), color='r')
ax.fill_between(index, rv.cdf(index), color='r', alpha=0.3)

# sample from random variable
Y = rv.rvs(1000)

# plot histogram
hist, bins = np.histogram(Y)
ax.bar(bins[:-1], hist/hist.sum(), alpha=0.4, color='b',width=0.05)

plt.tight_layout()
plt.savefig(path, transparent=True)
return path
```

When you have gotten this code to work now re-write this to work with the normal distribution instead. First try to repeat the same thing in one dimension and then change to do the same with a two dimensional normal distribution instead. The two distributions that you want to use are,

Code

```
from scipy.stats import norm
from scipy.stats import multivariate_normal
```

make sure that you check the documentation by writing `multivariate_normal?` to see how it needs to be initialised. For a two dimensional distribution the index-set will be different a simple way to create a two dimensional grid is this,

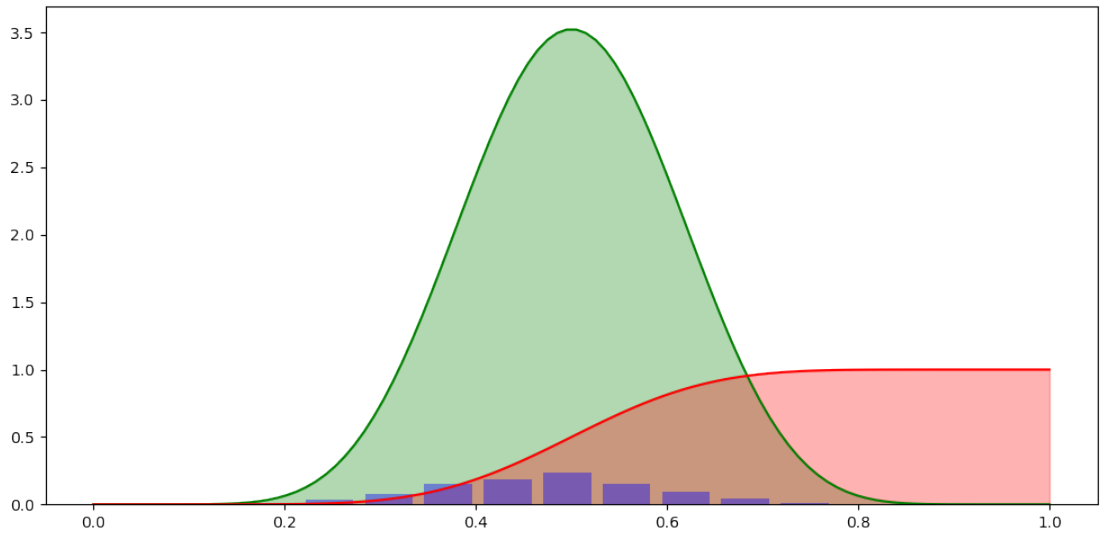

Figure 3: The green curve shows the density function, the red the cummulative density function and the blue a histogram of samples from the random variable.

Code

```
# create a 1d index
x = np.linspace(-1.5,1.5,100)
# create all combinations of locations from the 1d index
x1p, x2p = np.meshgrid(x,x)
# stack these positions into a single vector
pos = np.vstack((x1p.flatten(), x2p.flatten())).T
```

4 Summary

Hope that you managed to get through this worksheet and found a sensible way to working with Python. As said in the beginning you are most likely much better programmers than me and I'm sure I've said more than one wrong thing in this document. The important thing is that this unit is **not** a programming unit, we will only write code in oder to experiment to give ourselves an intuition of the underlying math. Most exercises will not contain much more code than what we did today. Next week will look at distributions and do a first easy example of how we can learn from data. Hope that you are looking forward to this.