



# Introduction to Scientific Computation

## Lecture 8

### Fall 2021

Packaging, Unsupervised learning: t-SNE

## Package types:

A package can be essentially in two forms:

- source
- binary

Focusing now on the Python world:

As Python is a dynamic language, this distinction can get a bit blurred:

There is little difference between a source and binary package *for a pure python package*

But if there is any compiled code in there, building from source can be a challenge:

- Binary packages are very helpful

## Source Packages

A source package is all the source code required to build the package.

Package managers (like pip) can automatically build your package from source.

**But:**

- Your system needs the correct tools installed, compilers, build tools, etc
- You need to have the dependencies available
- Sometimes it takes time, sometimes a LONG time

## Binary Packages

A collection of code all ready to run.

- Everything is already compiled and ready to go – makes it easy.

**But:**

- It's likely to be platform dependent
- May require dependencies to be installed

## Python Packaging

There are two package managers widely used for Python.

**pip:** The “official” solution.

- Pulls packages from PyPy
- Handles both source and binary packages (wheels)
- Python only

**conda:** Widely used in the scipy community.

- Pulls packages from anaconda.org
- Binary only (does not compile code when installing)
- Supports other languages / libraries: C, Fortran, R, Perl, Java (anything, really)
- Manages Python itself!

# WHY ARE WE DOING THIS?

One of the most powerful things about coding for the sciences is that it costs nothing to re-use code we've written in the past, allowing us to build on past work rather than starting over every project or paper.

However, one practice we see again and again is copying and pasting code from one project into another. Sometimes it will just be a function, other times (coughMATLABcough) it's files.

Assembling code in packages makes it really easy to re-use old code: all the scripts and functions end up in a central location and can be called and imported from anywhere on the computer - just like the famous packages [numpy](#) or [matplotlib](#).

# SETTING UP

## 1. THE BASICS

The most basic directory structure for a Python package looks like this:

```
project
|
|__ setup.py
|
|__ myPackage
    |
    |__ somePython.py
    |__ __init__.py
```

But at the moment, we've just got some flat files.

```
project
|
|__ norms.py
|__ metrics.py
```

So, the first step is to move files around. First comes the hardest part: choosing a package name. I'll call mine **measure**. Create a directory with that name, and move the python files in there.

```
project
|
|__ measure
    |__ norms.py
    |__ metrics.py
```

There is one more crucial file: `__init__.py` lets the Python interpreter know that there are importable modules in this directory. This is the script that gets run when you execute `import measure`. For more about what you can do with modules, you can see the (Python docs)[<https://docs.python.org/3/tutorial/modules.html>]. After adding `__init__.py`, the project directory should be

```
project
|
|__ measure
    |__ __init__.py
    |__ norms.py
    |__ metrics.py
```

## 2. SETUP.PY

At this point, the library can be imported if we're in the same directory, but it isn't a package. To let `setuptools` and `pip` know how to handle it, we need to add the `setup.py` file.

A very basic version of `setup.py` is

```
from setuptools import setup

setup(
    # Whatever arguments you need/want
)
```

If you were to run that file, you'd get a whole bunch of warnings, and nothing would actually get packaged. As a bare minimum, here is our `setup.py` for it to work.

```
from setuptools import setup

setup(
    # Needed to silence warnings (and to be a worthwhile package)
    name='Measurements',
    url='https://github.com/jladan/package_demo',
    author='John Ladan',
    author_email='jladan@uwaterloo.ca',
    # Needed to actually package something
    packages=['measure'],
    # Needed for dependencies
    install_requires=['numpy'],
    # *strongly* suggested for sharing
    version='0.1',
    # The license can be anything you like
    license='MIT',
    description='An example of a python package from pre-existing code',
    # We will also need a readme eventually (there will be a warning)
    # long_description=open('README.txt').read(),
)
```

```
project
|
|__ setup.py
|__ MANIFEST.in
|
|__ measure
|   |__ __init__.py
|   |__ norms.py
|   |__ metrics.py
|
|__ tests
|   |__ test_suite.py
|
|__ doc
|   |__ docs.rst
|
|__ scripts
|   |__ hello.py
|
|__ README.txt
|__ CHANGES.txt
|__ LICENSE.txt
```

## Basic definitions

We have a set of objects:  $X$

And the set of possible answers:  $Y$

We define the target function:  $y^* : X \rightarrow Y$



Supervised



Unsupervised  $Y \in \emptyset$

## Unsupervised

We have a set of objects:  $X$

And the set of possible answers:  $Y$

## Unsupervised

We have a set of objects:

$X$

And the set of possible answers:  $Y$



This one we don't have

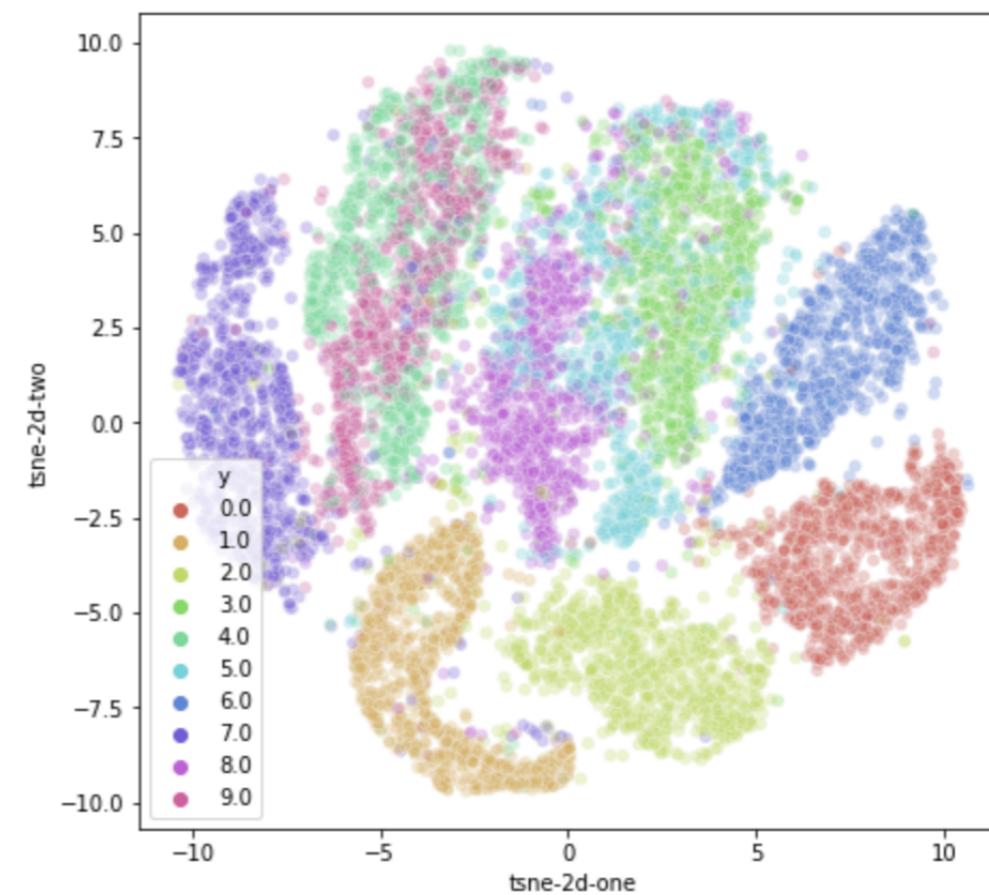
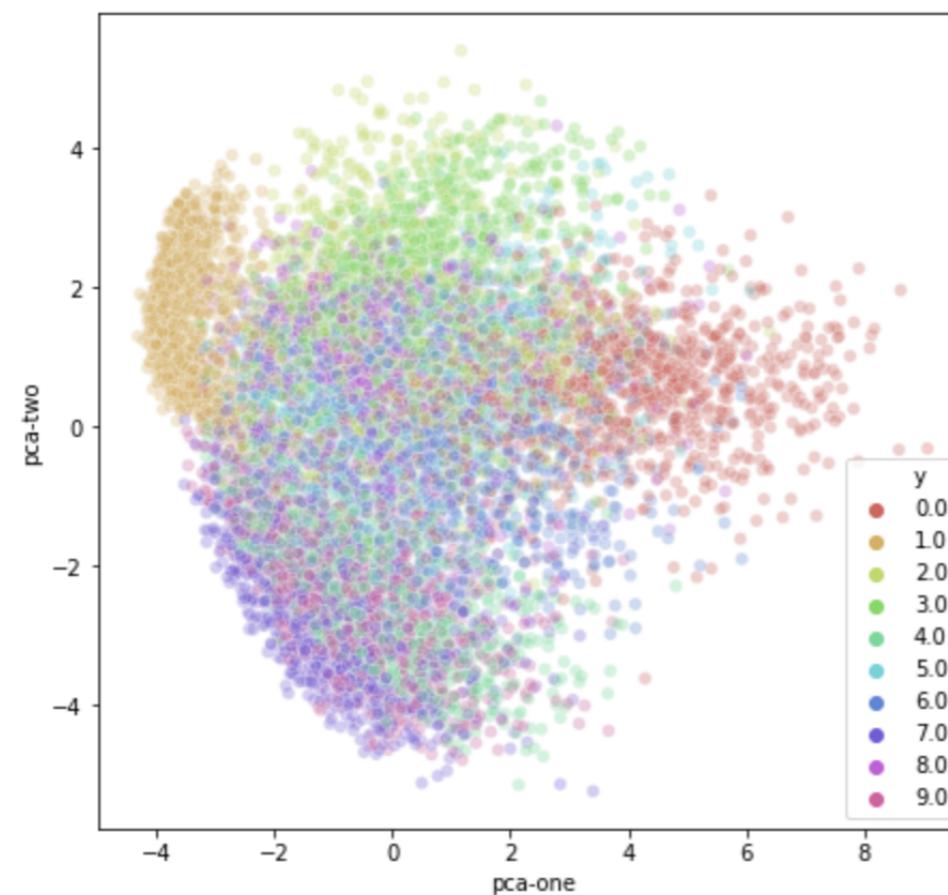
## Unsupervised

We have a set of objects:  $X$

And the set of possible answers:  $Y$



This one we don't have



## SNE

- “Encode” high-dimensional information into distribution
- Random walk between data points
  - Higher probability to jump to a closer point
- Find low dimensional points with similar neighbourhood distribution
- Not easy to embed new points

## Neighbourhood distributions

- Consider the neighbourhood around an input data point  $x_i \in \mathbb{R}^d$
- Imagine Gaussian distribution centred around  $x_i$
- The probability that  $x_i$  chooses some other datapoint  $x_j$  as neighbour is proportional to the area under gaussian curve.
- The point closer to  $x_i$  is more likely than a further one

- The  $i \rightarrow j$  probability to choose point  $j$  from  $i$   
$$P_{j|i} = \frac{\exp(-||x_i - x_j||^2/\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||^2/\sigma_i^2)}$$
- $P_{i|i} = 0$
- $\sigma_i$  sets the size of neighbourhood:
  - Low - always choose nearest point
  - High - uniform weights

- $\sigma_i$  is chosen differently for each data point
- The distribution is symmetric:  $P_{ij} = \frac{1}{2N}(P_{i|j} + P_{j|i})$ 
  - Pick  $i$  or  $j$  uniformly and jump according to distribution

## Perplexity

- For each  $P_{j|i}$  (depends on  $\sigma_i$ ) perplexity:
  - $\text{perp}(P_{j|i}) = 2^{H(P_{j|i})}, H(P_i) = - \sum_i P_i \log(P_i)$
- If  $P$  is uniform over  $k$  elements, perplexity is  $k$ 
  - Smooth version of  $k$  in kNN
  - Low perplexity - small  $\sigma_i$
  - Large perplexity - large  $\sigma_i$
- $\sigma_i$  sets the size of neighbourhood:
  - Low - always choose nearest point
  - High - uniform weights

## SNE objective

- Given  $x^{(1)}, \dots, x^{(N)} \in \mathbb{R}^d$  we define distribution  $P_{ij}$
- Goal: find good embedding  $y^{(1)}, \dots, y^{(N)} \in \mathbb{R}^{\hat{d}}$ , where  $\hat{d} < d$  (normally 2 or 3)
- For points  $y^{(1)}, \dots, y^{(N)} \in \mathbb{R}^{\hat{d}}$  we can define distribution  $Q$  (no  $\sigma_i$  and not symmetric):

$$Q = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_k \sum_{k \neq i} \exp(-\|y_i - y_k\|^2)}$$

- Optimize  $Q$  to be close to  $P$ 
  - Use  $KL$ -divergence

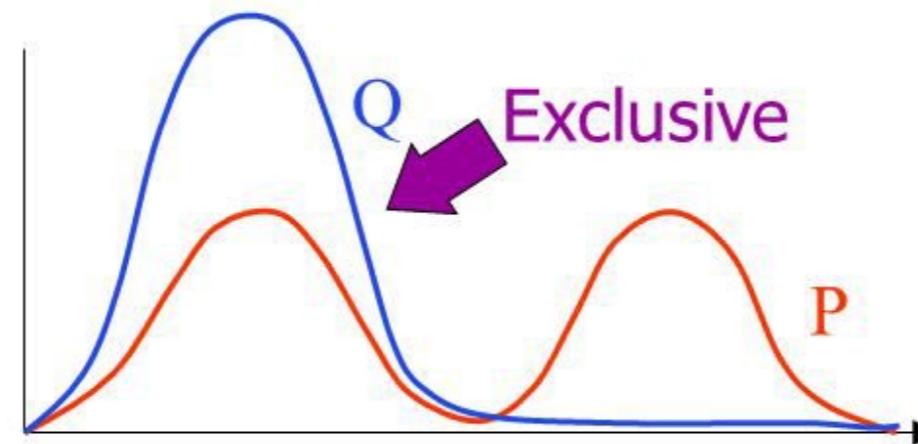
**KL**

- $$KL(P || Q) = \sum_{ij} Q_{ij} \log \frac{Q_{ij}}{P_{ij}}$$
- Not a metric function - not symmetric
- $$KL(Q || P) = - \sum_{ij} P_{ij} \log Q_{ij} + \text{const}$$
- In lower dimensions it is harder to pack neighbours than in high dimension: crowding problem
- t-SNE: change gaussian in  $Q$  to a heavy tailed distribution

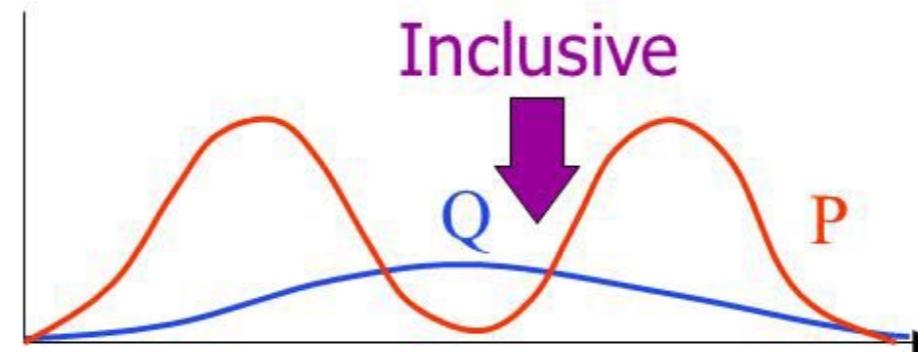
## KL

- $KL(Q \parallel P) \geq 0$  and 0 if  $Q = P$
- $KL(Q \parallel P)$  is convex
- If  $P_{ii} = 0$  but  $Q_{ii} > 0$  then  $KL(Q \parallel P) = \infty$

Minimising  
 $KL(Q \parallel P)$   
 $= \sum_H Q(H) \ln \frac{Q(H)}{P(H|V)}$



Minimising  
 $KL(P \parallel Q)$   
 $= \sum_H P(H|V) \ln \frac{P(H|V)}{Q(H)}$



## Algorithm

- We optimise  $KL(P || Q)$  by choosing  $y^{(1)}, \dots, y^{(N)} \in \mathbb{R}^{\hat{d}}$
- $$\frac{\partial KL(P || Q)}{\partial y^{(i)}} = \sum_j (P_{ij} - Q_{ij})(y^{(i)} - y^{(j)})$$
- Not a convex problem! No guarantees, we can use multiple restarts
- Main issue: crowding problem
- In lower dimensions it is harder to pack neighbours than in high dimension: crowding problem
- t-SNE: change gaussian in  $Q$  to a heavy tailed distribution

**t-SNE**

- Student t-probability density  $p(x) \propto (1 + \frac{x^2}{\nu})^{-(\nu+1)/2}$
- Probability goes to zero much slower than for Gaussian (that's heavy tailed)
- $$Q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_k \sum i \neq k (1 + ||y_i - y_k||^2)^{-1}}$$
- $$\frac{\partial KL(P || Q)}{\partial y^{(i)}} = \sum_j (P_{ij} - Q_{ij})(y^{(i)} - y^{(j)}) (1 + ||y_i - y_j||^2)^{-1}$$

## t-SNE algorithm

- We learn  $y^{(1)}, \dots, y^{(N)} \in \mathbb{R}^{\hat{d}}$
- Compute  $P_{ij}$  with perplexity
- Make  $P_{ij}$  symmetric
- Sample initial solution  $y^{(1)}, \dots, y^{(N)} \in \mathbb{R}^{\hat{d}}$
- For T iterations:
  - Compute  $Q_{ii}$
  - Compute gradient of the loss
  - Update  $y^{(1)}, \dots, y^{(N)} \in \mathbb{R}^{\hat{d}}$