# From Scripts to Reusable Software and Reproducible Research

**Dr. André Anjos (anjos.ai)**
**Prof. Manuel Günther (www.ifi.uzh.ch)**

June, 29th 2022



idiap RESEARCH INSTITUTE

Universität Zürich[UZH]

# Outline

# Outline

# Course contents

Achieving Full Reproducibility can be challenging. It requires many components that work cooperatively to implement it. In this course, we will tackle the most important bits:

- Part I: Introduction (30')
- Part II: Workflow and code organisation (45', hands-on)
- *Coffee Break (30')*
- Part III: Command-line and configuration (75', hands-on)
- *Lunch (60')*
- Part IV: Revision control with Git (40', hands-on)
- Part V: Packaging and deployment (60', hands-on)
- *Coffee Break (30')*
- Part VI: Documentation (60', hands-on)
- Part VII: Unit testing (30')

# Preparation

1. Follow the setup instructions at
   **https://gitlab.uzh.ch/manuel.guenther/ifiss2022**
2. Change directories to `a-intro`. You will find 3 exercise directories we will refer to in a bit:

```
$ cd a-intro
```

**Notation**

- Shell commands will be examplified in boxes, with a $ prefix.
- The actual command to be typed comes after the $.
- We assume the `ifiss` environment is **always** active.

# This part

In this part of the course we will:

1. Motivate the need for reproducibility in data sciences
2. Define what reproducibility is
3. Explain to you how the different pieces of reproducibility come together
4. Motivate, via illustrative exercises, how to approach reproducibility at your own work

Materials:

1. Lecture slides
2. A (very) small lab will be used to illustrate

# Toy Project

You talked to a colleague from work, who told you:

> *It is possible to perfectly classify the Iris Flower Dataset.*

# Toy Project: Reproduction

And we decide to reproduce it.

```
$ cd ex0
$ # figure it out
```

# What is missing?

# What is missing?

- How was the system trained?
- Was there any pre-processing?
- What was the data/protocol used to verify these results?

# What is missing?

- How was the system trained?
- Was there any pre-processing?
- What was the data/protocol used to verify these results?

**In brief...**

So that you can verify the claim from your colleague, you need more information about what/when/how!

# You contact the colleague

You decide to e-mail the colleague(s) and ask for more information:

- They point you to the data, that you download
- They explain you that they *just* applied “a two-layer Neural Network" available on “Neural Network Toolbox" from Matlab

**You think...**

That is it! Now, I can try it out...

# Second Exercise

Now we reproduce it, for sure!

```
$ cd ../ex1
$ cat email.txt
$ matlab
$ # start matlab
```

# Issues?

- Well, you may have to buy a Matlab license, and have it installed on your laptop...
- Are there untold parameters?
- How was the analysis carried out? What is it?
- Would it work for a different data set?
- How can I apply those findings on my work?
- ...

**In brief...**

So that you can verify the claim from your colleague(s), you need not only the **data**, but also (preferably free) **software** and **instructions** to reproduce the claimed results.

# Reproducing a Paper



## A Scalable Formulation of Probabilistic Linear Discriminant Analysis: Applied to Face Recognition

Laurent El Shafey, Chris McCool, Roy Wallace, and Sébastien Marcel

### Appendix A
### Mathematical Derivations

The goal of the following section is to provide more detailed proofs of the formulae given in the article for both training and computing the likelihood.

The following proofs make use of a formulation of the inverse of a block matrix that uses the Schur complement. The corresponding identity can be found in [1] (Equations 1.11 and 1.10),

$$\begin{bmatrix} \boldsymbol{L} & \boldsymbol{M} \\ \boldsymbol{N} & \boldsymbol{O} \end{bmatrix}^{-1} = \begin{bmatrix} \boldsymbol{R}, & -\boldsymbol{R}\boldsymbol{M}\boldsymbol{O}^{-1} \\ -\boldsymbol{O}^{-1}\boldsymbol{N}\boldsymbol{R}, & \boldsymbol{O}^{-1}+\boldsymbol{O}^{-1}\boldsymbol{N}\boldsymbol{R}\boldsymbol{M}\boldsymbol{O}^{-1} \end{bmatrix}, \tag{51}$$

where we have substituted $\boldsymbol{R} = \left(\boldsymbol{L} - \boldsymbol{M}\boldsymbol{O}^{-1}\boldsymbol{N}\right)^{-1}$.

Another related expression is the Woodbury matrix identity (Equation $C.7$ of [2]), which states that,

$$(\boldsymbol{L}+\boldsymbol{M}\boldsymbol{O}\boldsymbol{N})^{-1} = \boldsymbol{L}^{-1} - \boldsymbol{L}^{-1}\boldsymbol{M}\left(\boldsymbol{O}^{-1}+\boldsymbol{N}\boldsymbol{L}^{-1}\boldsymbol{M}\right)^{-1}\boldsymbol{N}\boldsymbol{L}^{-1}. \tag{52}$$

#### A. Scalable training

The bottleneck of the training procedure is the expectation step (E-Step) of the Expectation-Maximization algorithm. This E-Step requires the computation of the first and second order moments of the latent variables.

*1) Estimating the first order moment of the Latent Variables:* The most computationally expensive part when estimating the latent variables is the inversion of the matrix $\mathring{\bar{\boldsymbol{\mathcal{P}}}}$ (Equation (27)). This matrix is block diagonal, the two blocks being $\boldsymbol{\mathcal{P}}_0$ (Equation (28)) and (a repetition of) $\boldsymbol{\mathcal{P}}_1$ (Equation (29)),

$$\mathring{\bar{\boldsymbol{\mathcal{P}}}} = \begin{bmatrix} \boldsymbol{\mathcal{P}}_0 & 0 & \cdots & 0 \\ 0 & \boldsymbol{\mathcal{P}}_1 & \ddots & 0 \\ 0 & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \boldsymbol{\mathcal{P}}_1 \end{bmatrix}. \tag{53}$$

The inverse of $\boldsymbol{\mathcal{P}}_1$ is equal to the matrix $\boldsymbol{\mathcal{G}}$, defined by (30). This matrix is of constant size ($D_G \times D_G$), irrespective of the number of training samples for the class. In addition, the inversion of $\boldsymbol{\mathcal{P}}_0$ can be further optimised using the block matrix inversion identity introduced at the beginning of this section, leading to

$$\boldsymbol{\mathcal{P}}_0^{-1} = \begin{bmatrix} \boldsymbol{\mathcal{F}}_{J_i} & \sqrt{J_i}\boldsymbol{\mathcal{H}}^T \\ \sqrt{J_i}\boldsymbol{\mathcal{H}} & \left(\boldsymbol{I}_{D_G} - J_i\boldsymbol{\mathcal{H}}\boldsymbol{F}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{G}\right)\boldsymbol{\mathcal{G}} \end{bmatrix}, \tag{54}$$

where $\boldsymbol{\mathcal{F}}_{J_i}$ is defined by (33) and $\boldsymbol{\mathcal{H}}$ by (37).

Then, the computation of $\mathring{\bar{\boldsymbol{\mathcal{P}}}}^{-1}\mathring{\bar{\boldsymbol{A}}}^T\bar{\boldsymbol{\Sigma}}^{-1}$ gives a block diagonal matrix, the first block being

$$\begin{bmatrix} \sqrt{J_i}\boldsymbol{\mathcal{F}}_{J_i}\boldsymbol{F}^T\boldsymbol{\mathcal{S}} \\ \boldsymbol{\mathcal{G}}\boldsymbol{G}^T\boldsymbol{\Sigma}^{-1}\left(\boldsymbol{I}_{D_x} - J_i\boldsymbol{F}\boldsymbol{\mathcal{F}}_{J_i}\boldsymbol{F}^T\boldsymbol{\mathcal{S}}\right) \end{bmatrix},$$

and the other ones being equal to $\boldsymbol{\mathcal{G}}\boldsymbol{G}^T\boldsymbol{\Sigma}^{-1}$.

As explained in section III.B.a of the article, $\boldsymbol{h}_i$ corresponds to the upper sub-vector of $\mathring{\bar{\boldsymbol{y}}}_i$ and is not affected by the change of variable, as depicted in (21). Therefore, the first order moment of $\boldsymbol{h}_i$ is directly obtained by multiplying the first block-rows of the matrix $\mathring{\bar{\boldsymbol{\mathcal{P}}}}^{-1}\mathring{\bar{\boldsymbol{A}}}^T\bar{\boldsymbol{\Sigma}}^{-1}$ with $\mathring{\bar{\boldsymbol{x}}}_i$, which gives (31).

Considering only the $\mathring{\bar{\boldsymbol{w}}}_i$ (lower) sub-vector of $\mathring{\bar{\boldsymbol{y}}}_i$, the corresponding (lower) part $\mathring{\bar{\boldsymbol{\mathcal{B}}}}$ of the matrix $\mathring{\bar{\boldsymbol{\mathcal{P}}}}^{-1}\mathring{\bar{\boldsymbol{A}}}^T\bar{\boldsymbol{\Sigma}}^{-1}$ can be decomposed into a sum of two matrices, the first one being sparse with a single non-zero block (upper left) equal to $\boldsymbol{\mathcal{B}}_0 = -J_i\boldsymbol{\mathcal{G}}\boldsymbol{G}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{F}\boldsymbol{\mathcal{F}}_{J_i}\boldsymbol{F}^T\boldsymbol{\mathcal{S}}$, and the second one being diagonal by blocks with identical blocks $\boldsymbol{\mathcal{B}}_1 = \boldsymbol{\mathcal{G}}\boldsymbol{G}^T\boldsymbol{\Sigma}^{-1}$,

$$\mathring{\bar{\boldsymbol{\mathcal{B}}}} = \begin{bmatrix} \boldsymbol{\mathcal{B}}_0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} \boldsymbol{\mathcal{B}}_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \boldsymbol{\mathcal{B}}_1 \end{bmatrix}. \tag{55}$$

Furthermore, the first order moment of the variables $\bar{\boldsymbol{w}}_i$ is given by

$$\begin{aligned} E\left[\bar{\boldsymbol{w}}_i|\bar{\boldsymbol{x}}_i,\boldsymbol{\Theta}\right] &= \left(\bar{\boldsymbol{U}}^T \otimes \boldsymbol{I}_{D_G}\right)\begin{bmatrix} \boldsymbol{\mathcal{B}}_0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathring{\bar{\boldsymbol{x}}}_i \\ &+ \left(\bar{\boldsymbol{U}}^T \otimes \boldsymbol{I}_{D_G}\right)\begin{bmatrix} \boldsymbol{\mathcal{B}}_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \boldsymbol{\mathcal{B}}_1 \end{bmatrix}\left(\bar{\boldsymbol{U}} \otimes \boldsymbol{I}_{D_x}\right)\bar{\boldsymbol{x}}_i. \end{aligned} \tag{56}$$

The previous decomposition greatly simplifies the computation, and leads to the following expression for each $\boldsymbol{w}_{i,j}$,

$$\begin{aligned} E\left[\boldsymbol{w}_{i,j}|\bar{\boldsymbol{x}}_i,\boldsymbol{\Theta}\right] &= \boldsymbol{\mathcal{G}}\boldsymbol{G}^T\boldsymbol{\Sigma}^{-1}\bar{\boldsymbol{x}}_{i,j} \\ &- \boldsymbol{\mathcal{G}}\boldsymbol{G}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{F}\boldsymbol{\mathcal{F}}_{J_i}\boldsymbol{F}^T\boldsymbol{\mathcal{S}}\sum_j \bar{\boldsymbol{x}}_{i,j} \end{aligned} \tag{57}$$

L. El Shafey is with Idiap Research Institute and Ecole Polytechnique Fédérale de Lausanne, Switzerland e-mail: laurent.el-shafey@idiap.ch

C. McCool, R. Wallace and S. Marcel are with Idiap Research Institute, Martigny, Switzerland e-mail: {christopher.mccool,roy.wallace,sebastien.marcel}@idiap.ch

The research leading to these results has received funding from the European Communitys Seventh Framework Programme (FP7) under grant agreements 238803 (BBfor2) and 257289 (TABULA RASA).



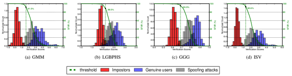



Fig. 10: Score distributions of baseline face verification systems. The full green line shows how SFAR changes with moving the threshold.

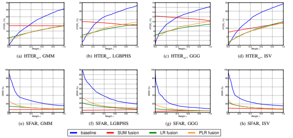



Fig. 12: EPSC for comparison of fusion techniques of baselines with LBP anti-spoofing algorithm

#### D. Performance of fused systems

In our last experiment, we compare the four face verification systems when fused with ALL counter-measures using PLR fusion scheme. Firstly, we illustrate how fusion changes the score distribution for each of them separately in Figure 14. Then, in Figure 15 we compare which of the fused systems performs the best.

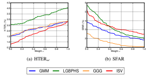
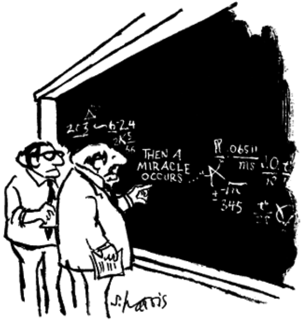



Fig. 15: EPSC curves to compare fused systems

While Figure 10 shows that the spoofing attacks of Replay-Attack are in the optimal category when fed to the baseline face verification systems, Figure 14 illustrates that their effectiveness has vastly changed after fusion. The score distribution of the spoofing attacks is now mostly overlapping with the score distribution of the zero-effort impostors, allowing for better discriminability between the positive class and the two negative classes. The results are reflecting this observation: even when the threshold is obtained using the licit scenario, SFAR has dropped to less then 6%.

The comparison between the EPSC curves given in Figure 11(a) and Figure 15(a), confirms the above observations: while HTER$_\omega$ increases rapidly with $\omega$ and reaches up to 25% for some of the baseline systems, it increases very mildly and does not exceed 4.1% for the fused systems. The major augmentation of the robustness to spoofing of the systems after

# Reproducing a Paper



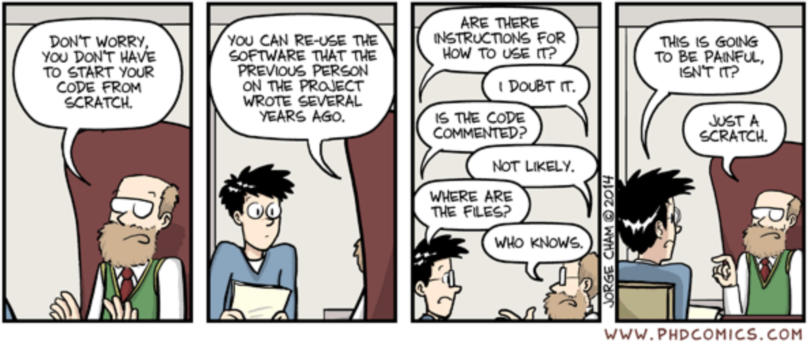

"I think you should be more explicit here in step two."

# Reproducing a Paper

*You go for it – work day and night to* ***incorporate some results*** *on your own project but:*

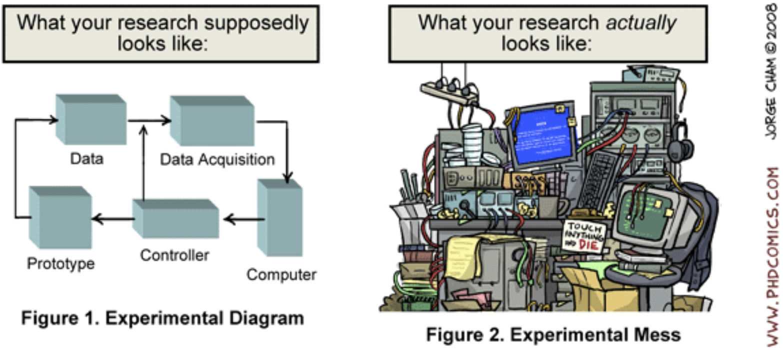
- There were **untold parameters** that needed adjustment and you couldn't get hold of them
- You realized the proposed solution **worked only on the specific data** shown at the original paper
- You realized that something did **not quite add up** in the end

# Reproducing a Paper




*Had **to take over** the work from another colleague that left and had to start from scratch - months into programming to make things work again*

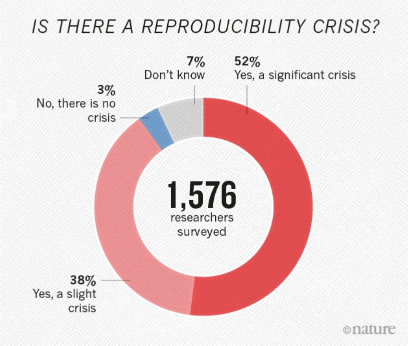
# Reproducing a Paper




*Would have liked to* ***replay to someone about your work****, but you couldn't really remember all details when you first made it work? Or you* ***could not make it work*** *at all?*

# Is there a crisis?[1]

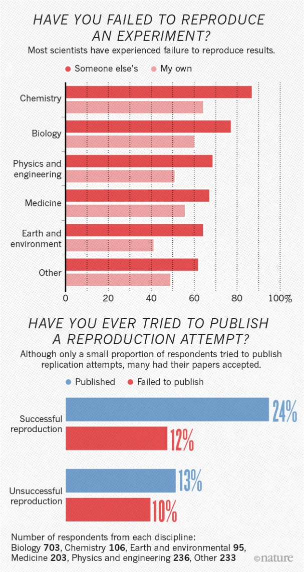





A survey in Nature revealed that irreproducible experiments are a problem across all domains of science.

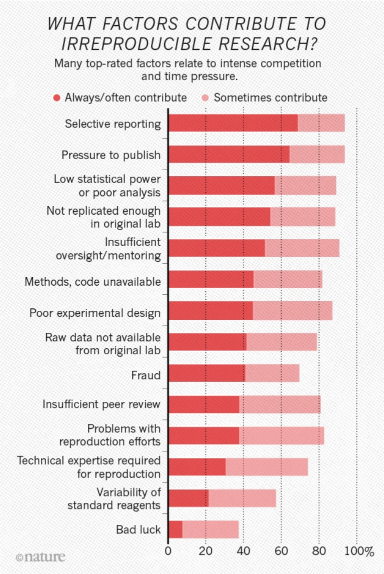
[1] 1,500 scientists lift the lid on reproducibility, Monya Baker, Nature, 2016

# Who else is affected?




Engineering is among the most affected research fields. For example, a study in Nature found that 47 out of 53 medical research papers focused on cancer research were irreproducible.

# Reasons


WHAT FACTORS CONTRIBUTE TO IRREPRODUCIBLE RESEARCH?
Many top-rated factors relate to intense competition and time pressure.
Always/often contribute
Sometimes contribute
Selective reporting
Pressure to publish
Low statistical power or poor analysis
Not replicated enough in original lab
Insufficient oversight/mentoring
Methods, code unavailable
Poor experimental design
Raw data not available from original lab
Fraud
Insufficient peer review
Problems with reproduction efforts
Technical expertise required for reproduction
Variability of standard reagents
Bad luck
0
20
40
60
80
100%
©nature

# Enter “Reproducible Research" (RR)[2]

One term that aggregates work comprising of:

- a **report**, that describe your work in all relevant details
- **code** to reproduce all results
- **data** required to reproduce the results
- **instructions**, on how to apply the *code* on the *data* to repeat the results on the *report*.

[2] http://reproducibleresearch.net

# Make the difference[3]

| | | Data | |
|---|---|---|---|
| | | **Same** | **Different** |
| **Code** | **Same** | **Reproducible** | Replicable |
| | **Different** | Robust | Generalisable |

[3] *Identifying and Overcoming Threats to Reproducibility, Replicability, Robustness, and Generalizability in Microbiome Research*, Patrick D. Schloss, 2018

# Levels of Reproducibility[4]

With respect to an independent researcher (reader):

0. Irreproducible
1. Cannot seem to reproduce
2. Reproducible, with extreme effort ($>$ 1 month)
3. Reproducible, with considerable effort ($>$ 1 week)
4. Easily reproducible ($\sim$ 15 min.), but requires proprietary software (e.g. Matlab)
5. **Easily reproducible ($\sim$ 15 min.), only free software**

[4] *Reproducible Research in Signal Processing: What, why and how*, Vandewalle, Kovacevic and Vetterli, 2012

# Fields of Application

## Requirements

- Data that serves as input must be copiable
- Procedure must be easily copied:
  - Computer-based routines
  - Statistical or Deterministic methods

## Counter-Examples

- Theoretical Physics (or disciplines of any sort)
- Biological Experimentation (see “replicability")
- Humanities
- ...

# Why? (1/2)

***You** are the winning party!*


- **You**, first!
  - Improved project structure and organization
  - Easy to replay analysis and generate results *after* changing mistakes
  - Easy to *extend* study to different tools and data
- **Collaborators**:
  - Closer interaction between collaborators
  - Scientific reports practically “write themselves"
  - Easy to pass-on work to colleagues
- **Others**:
  - Increased visibility (researchers)

# Why? (2/2) - Research Oriented

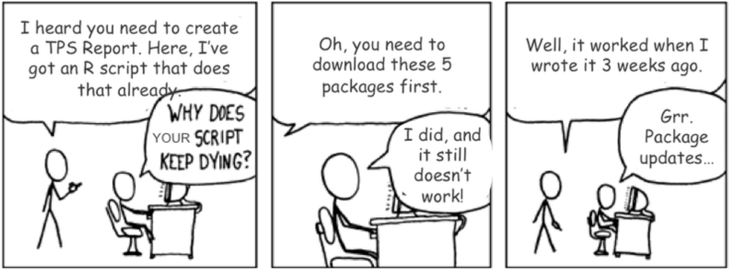
Boost your research **impact (visibility)**:

- **Lower entrance barrier** to your publications
- The current number of reproducible papers is **rather small** - you have a clear chance to stand out today:
  - Only **10% of TIP** papers provide source code[5].
- Statistically, your work is **more valuable** if it is RR:
  - **13 out of the top 15 most cited** articles in TPAMI or TIP provide (at least) source code
  - The average number of citations for papers that provide source-code in TIP is **7 fold** that of papers that do not.

[5] *Code Sharing is Associated with Research Impact in Image Processing*, Patrick Vandewalle, 2012

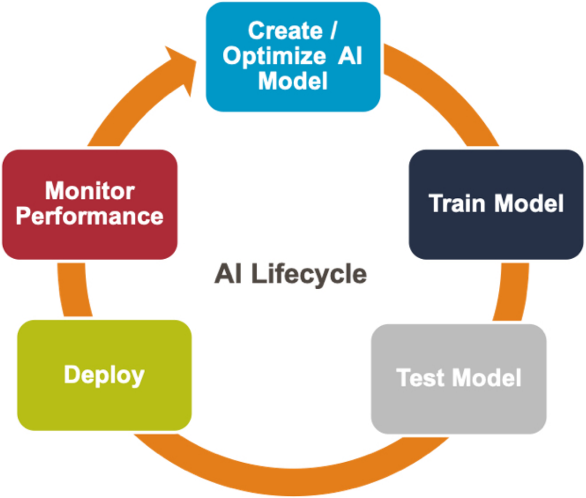
# Continual Reproducibility

It is rarely the case your system is reproduced on the exact day you make it available.



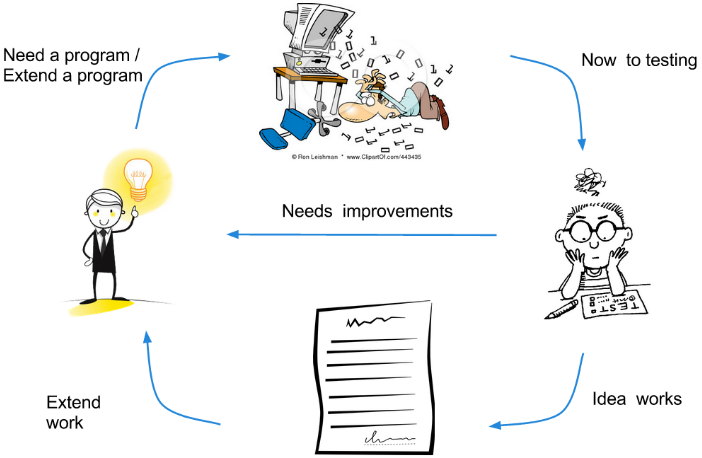
# Lifecycle of AI projects

More commonly, the lifecycle of AI projects is a continuous optimization loop



# Continual or Full Reproducibility

It does **not** make sense to make a single time effort for reproducibility in AI. It is more sensible to think of it as an iterative process in search for better solutions. This is typically called **Continual** or **Full** Reproducibility.



# Key Elements

These are important elements in Continual Reproducibility[6][7]:

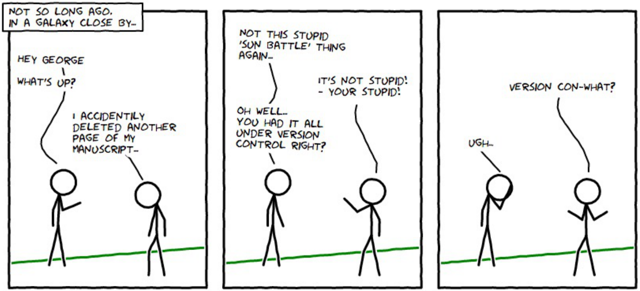
1. Revision Control (& Code Sharing)
2. Data Management & Framework Organization
3. Documentation
4. Packaging and Deployment
5. Unit testing (& Continuous Integration)

[6] Lack of these may affect long-term reproducibility, but do **not** deny it.

[7] *Continuously Reproducing Toolchains in Pattern Recognition and Machine Learning Experiments*, A. Anjos and others, 2017

# Revision Control

**Revision control** everything!




- As a consequence, prefer text files to other types of input
  - Programs are good candidates
  - Document your code using simple mark-up
  - Use LaTeX or mark-up for your reports
  - Microsoft `<name>` is not a good candidate...

# Code Sharing

**Keep** track of issues, annotate recipes

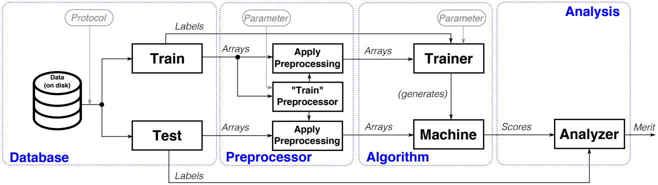
- There are several alternatives in the free-world: GitHub, GitLab, Bitbucket, etc. Just pick one!
- If maintaining an in-house solution, make it backup regularly
- Interact with your code-sharing repository often to avoid local, unbacked-up, changes
- Use it as a portal during reports to your management and to keep track of decisions

# Data Management

- **Keep** raw data.
  - Meaningful data is precious
  - Ensure you can always go back to the source
  - Keep backups
- **Make** data machine readable.
  - Use meaningful names for your variables
  - Avoid proprietary formats (to improve long term storage)
- **Record** all the steps used to produce and process data
  - Do not improvise, script everything
  - Write down documentation for your data that is meaningful to you and collaborators
- **Distribute** the data (if open-access)
  - Through DOI-capable portal (e.g. Zenodo)
  - Disclaimer: More “research" oriented

# Framework Organization

**Encapsulate** components you would like to test




- Data and evaluation protocols must be recorded and provided like simple iterators
- Preprocessing steps
- Your Machine Learning solution
- How to analyze the data
- Encapsulate components for easily replacing each upon need

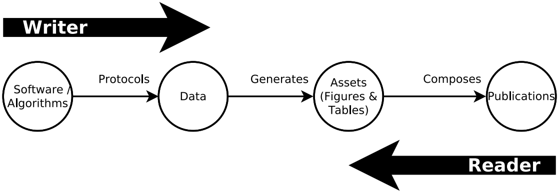
# Unit testing and Continous Integration

**Test** every piece of code you can

- Do **NOT** think you are not subject to errors
  - At first, the code is probably going to be OK
  - As the project develops, you will add code and skip checking basics - that is when errors tend to appear
  - Having a thorough check you can run under a minute helps **a lot**
- This is probably the hallmark of **reliable** code
- There is no *right* amount of testing. It should be thorough and pertinent
- Use your code sharing solution to run tests for you at every push, so you can **trust** your own changes
- If you collaborate (internally or externally), this also ensures your colleagues' additions do not break your analysis!
- Underlying libraries and code may also contain unexpected behaviour!

# Documentation

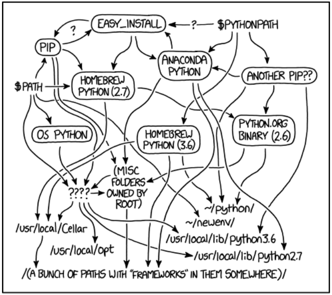
Documentation is there to help you remember, others to figure out.




- Data (what data is, how it was acquired, how to access it)
- Code (what your code does, how to install it)
  - Remember: Code is never self-documenting!
  - You can unit-test your documentation (where relevant)
- Instructions (how to take code and apply to the data)
  - Code usage examples may work as documentation
- Achievements (plots, tables, conclusions)

# Packaging and Deployment

Quickly deploying and switching environments is a time saver.




MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

- Key points:
  - Tracking own build/run/test-time dependencies
  - Fast to create new environments for quick tests or deployments
- There are various standards for packaging code. It is very easy to get lost!

# Takeaway Message

*Organize yourself so you are* ***always*** *doing Reproducible Research*

# Takeaway Message

*Organize yourself so you are **always** doing Reproducible Research*

For existing projects, do what you can, in this order:

1. Revision Control (& Code Sharing)
2. Data Management & Framework Organization
3. Documentation
4. Packaging and Deployment
5. Unit testing (& Continuous Integration)

# Hands-on!

You insisted with your colleague the project and told that you did not have access to Matlab - maybe they had another implementation flying around?

*They replied positively: "Here is some Python code a student wrote which does what our Matlab code does. I hope it is useful to reproduce our findings."*

# Hands-on!

You insisted with your colleague the project and told that you did not have access to Matlab - maybe they had another implementation flying around?

*They replied positively: "Here is some Python code a student wrote which does what our Matlab code does. I hope it is useful to reproduce our findings."*

```
(ifiss) $ cd ../ex2
(ifiss) $ python doit.py
...
Error: 0.00000  #great job!
```

# Hidden Gem(s)

At this point, you may think you have it!

# Hidden Gem(s)

At this point, you may think you have it!

However, note the authors made a mistake on the code. Can you guess where it is?

The code is also very messy and difficult to understand.

# Hidden Gem(s)

At this point, you may think you have it!

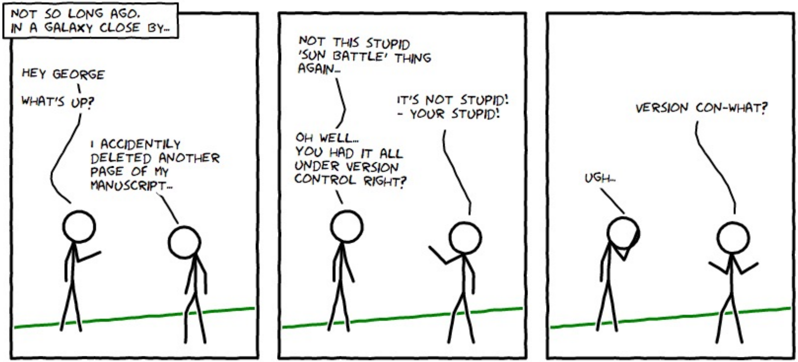
However, note the authors made a mistake on the code. Can you guess where it is?

The code is also very messy and difficult to understand.

Let's now go through the steps on making this code **Fully Reproducible**.

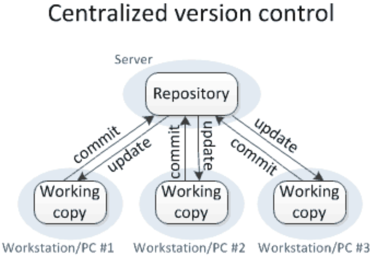
# Outline

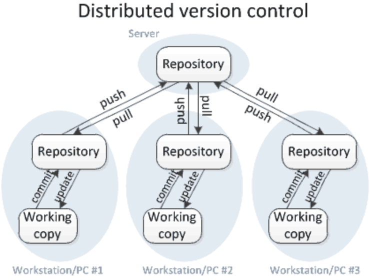
# Revision/Version Control



# What is Revision Control?

- Management of changes to documents/code or any sorts of collections of information
- It is normally done by specialized software packages such as `git`
- There are two types:
  - Centralized: Revision history is kept on a remote server
  - Distributed: History is copied with the repository



# Why is it necessary?

Imagine a world w/o version control:

- You released version 1.0 of your software. It has a bug. Which other versions are affected?
- When was the last time I touched this file? Which changes did I do?
- You introduced a bug on the software: Where is that *fracking* backup?

**It is possible!**

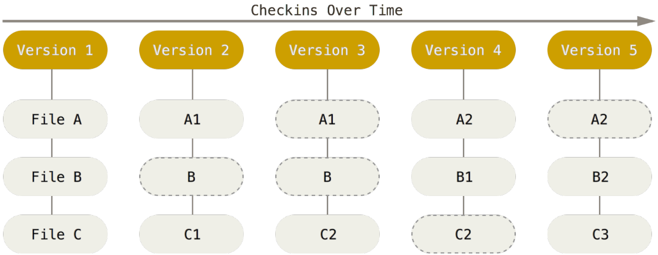
Actually, the Linux project stayed 11 years w/o version control!

This was possible thanks to an “extremely" organized procedure for diff/patching changes that gave birth to what is “Git" today!

# Git

What is Git?

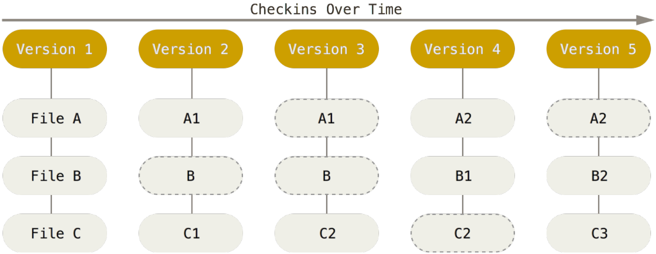
Git is a distributed revision control system. It keeps snapshots of **the entirety** of your versioned directory through time using patches.



# Git

What is Git?

Git is a distributed revision control system. It keeps snapshots of **the entirety** of your versioned directory through time using patches.




## Old tools, new usage

In order to create a snapshot, git uses *diffs*, *patches* and (SHA-1) *hashes*

# Diff/Patch

A *diff* is a set of textual differences between files.

```
file1.txt:

I need to go to the store.
I need to buy some apples.
When I get home, I'll wash the dog.
```

```
file2.txt:

I need to go to the store.
I need to buy some apples.
I also need to buy grated cheese.
When I get home, I'll wash the dog.
```

To create a *patch*, use the `diff` command:

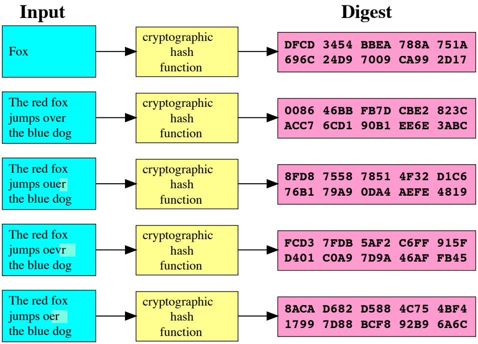
```
$ diff file1.txt file2.txt > patch.txt
$ cat patch.txt
2a3
> I also need to buy grated cheese.
```

**Translating**

After line 2 in the first file, a line needs to be added: line 3 from the second file.

# Hash

A (crypto) hash function is a function that can be used to map digital data of arbitrary size to a fixed length string, that is practically impossible to invert.




*Notice that small changes on the input make the hash change a lot.*

# Hash (collision)

**Nearly impossible to clash**

It is nearly **impossible** that two natural sequences collide on the **same** repository.

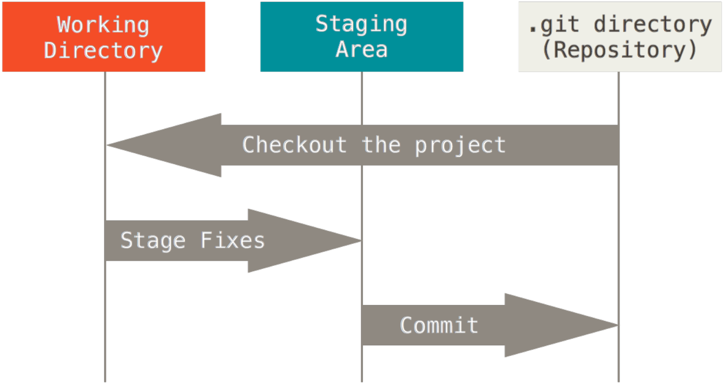
If all world population would be developers and every one of them would commit to the **same** repository every second, the probability of 50% collision would be reached in[8]:

$$6.6 \times 10^6 \text{years}$$

[8] http://diego.assencio.com/?index=eacd6eedf46c9dd596a5f12221ad15b8

# Git states

Git contains 3 states for your project.



# Git workflow

Easy

- You modify files in your working directory.
- You stage the files, adding snapshots of them to your staging area.
- You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

# Hands on: Configuring `git` (first time only)

To tell git it should log every commit using your name and e-mail, you need to configure it once:

```
$ git config --global user.name "First Last"
$ git config --global user.email first.last@example.com
# to list all configuration set for you
$ git config --list
```

**Tip: Finetune your Git configuration**

You can configure more in Git including, e.g., which editor to launch for doing commits or viewing changes. Read more here: **https://git-scm.com/book/en/v2/Customizing-Git-Git-Configuration**

# Hands on: Initializing a new repository

Let's create a new Git repository for the code we saw earlier. We will use `git init` to do so:

```
$ cp -r x-to-package ifiss  # project name
$ cd ifiss
$ git init
Initialized empty Git repository in ...
```

# Hands on: What is staged?

The `status` command gives an overview of the staging area.

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add ...
```

# Hands on: Let's do the first commit

The `commit` command instructs git to register the snapshot (patch) to its `.git` directory.

```
$ git add *.py *.yaml #adds all files to staging area
$ git commit -m "Initial commit"
$ git status
On branch master
nothing to commit, working directory clean
```

**Tip: Configuring the default editor**

```
$ git config --global core.editor /usr/bin/nano #default
$ git config --global core.editor /usr/bin/gedit
$ git config --global core.editor /usr/bin/vim
$ git config --global core.editor /usr/bin/gvim
```

# Making changes

The power of version control is shown when you make changes. Let's add a newline on the file `evaluation.py` and see what happens.

```
$ edit evaluation.py #change something
$ git diff
diff --git a/evaluation.py b/evaluation.py
index 14c1f22..2af2428 100644
--- a/evaluation.py
+++ b/evaluation.py
@@ -1,4 +1,5 @@
 import numpy
+
 def classification_error(prediction, dataset):
```

Note: The commit hashes might differ

# Commiting changes (faster)

You can stage and commit changes with one command.

```
$ git commit -m
  "evaluation: Added a space between imports and code" -a
```

# Unstage no content deletion

You can unstage a file from the staging area without deleting content.

```
$ git reset <file>
```

# Logs and Diffs

At all times, you have access to history and can revert back.

```
git log --oneline
97fd521 (HEAD -> master) evaluation: Added a space between imports and
e6d22a2 Initial commit
$ #figured out I did not like my previous change, so will revert
$ git revert 97fd521
# edit the comment, and save (<ESC>:wq)
$ git log --oneline
0925559 (HEAD -> master) Revert "evaluation: Added a space between imp
97fd521 evaluation: Added a space between imports and code
e6d22a2 Initial commit
$ git diff e6d22a2..0925559
# OK!
```

# Deletion/Renaming a file

These operations are tracked by the version history!

```
$ #delete a file
$ git rm <file>
$ #rename a file
$ git mv <file-before> <file-after>
```

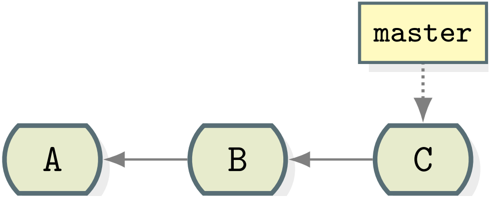
# Tags

Git allows you to set labels to refer to repository versions (instead of hash initials). You should use the `tag` command to do so.

```
$ git tag final #makes final == a43f4c6..
$ git tag buggy df7bc06 #makes buggy == df7bc06...
$ git diff buggy..final
...
$ git tag
buggy
final
```

# Branches

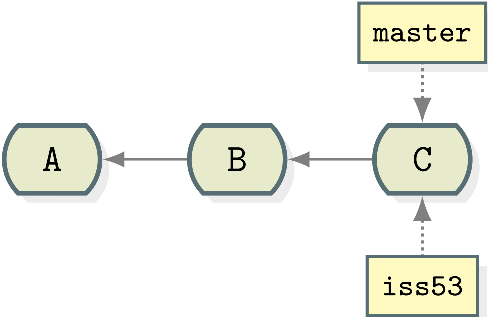
A branch is marker that identifies where the current changes will be patched on. By default, there is only one branch (called `master` or `main`) on a git repository.



# Branches (2)

You may create a new branch, to develop something new while keeping the master stable.

```
$ git branch iss53
```



# Branches (3)

To tell Git to consider a new branch as the default for commiting, use the `checkout` command:

```
$ git checkout iss53
Switched to a branch "iss53"
```

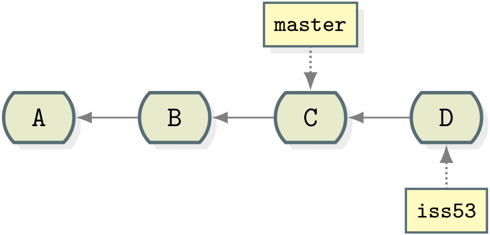
**Tip: One liner command**

```
# create a new branch and switch to it
$ git checkout -b iss53
```

# Branches (4)

If you commit a change, only the marker `iss53` will be modified:

```
$ gedit ...
$ git commit -m "My change to the special branch" -a
$ git log --oneline #has the new commit
...
$ git log --oneline master #as before
...
$ git checkout master #To go back to the master
$ git merge iss53 #To merge all changes back
$ git branch -d iss53 #To remove the branch
```



# How to Properly Use Tag/Branches

As rule of thumb:

- Everytime you make a **new release**, you **tag** your repository so you know what was distributed
- You use branches to:
  - Test new features w/o disturbing the stability of `master`
  - Fix problems with old versions of the software:

```
$ git branch old-release tag-1.2.4
$ git checkout old-release #state of version 1.2.4
# edit the changes
$ git commit -m "..."
# release version 1.2.5 from that point
```

# Git: Explore further!

Here is a list of resources if you're interested to know more about this powerful tool:

- Reference website: **https://git-scm.com/documentation**
  - Detailed tutorials
  - Videos
  - Reference documentation

  videos)
- ProGit Book (2nd Edition): **http://git-scm.com/book/en/v2**
- MOOC: **https://www.codeschool.com/courses/try-git**
- YouTube clips from GitHub: **https://www.youtube.com/user/GitHubGuides**

# GitLab

GitLab is a website that integrates a powerful interface to git repositories:

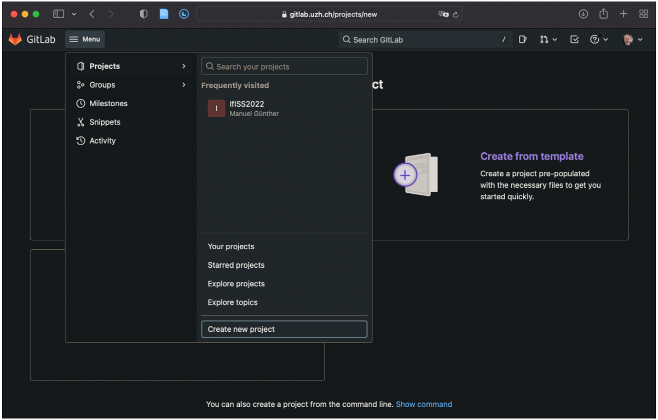
- Allows easy code sharing and free web-hosting for your projects
- Integrates wiki so you can setup various sorts of guides
- Integrates an issue tracker so people can report bugs and you can keep track of development

**Interact**

Open a web-browser and go to **https://gitlab.uzh.ch**.

# Hands On: New project

Let's move your project into GitLab. Create a new repository by clicking on the Menu (top-left), then Create new project (bottom right of the menu). Choose a "Python-valid" name (start with letter or underscore, continue with letter, digit or underscore) for your project.



# Hands on: Push the code

You can now "push" your local code to the remote repository.

```
$ git remote add origin git@gitlab.uzh.ch:anjos/abc.git
$ git push -u origin master
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 10 threads
Compressing objects: 100% (13/13), done.
Writing objects: 100% (13/13), 3.79 KiB | 3.79 MiB/s, done.
Total 13 (delta 4), reused 0 (delta 0), pack-reused 0
To gitlab.idiap.ch:andre.anjos/abc.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

**Reload GitLab**

Reload the GitLab repository page and see what happens!

# Hands on: "Social" fixes

If you are working on a new feature or fix, follow the branch-push-review-merge workflow:

1. Fix the code locally:

```
$ git checkout -b myfix
# work on the fix
$ git push -u origin myfix
```

2. Go to the web interface and create a new merge-request
3. On the merge-request page, review the changes (or allow another person to review them)
4. Merge the change and delete the work branch
5. Done!

# GitLab: Explore further!

Here is a list of resources you should explore:

- Issue Tracker: **https://docs.gitlab.com/ee/user/project/issues/**
- Merge Requests: **https://docs.gitlab.com/ee/user/project/merge_requests/**
- Push branch and create a merge request in a single command: **https://docs.gitlab.com/ee/user/project/push_options.html**
- Programmatic access: **https://python-gitlab.readthedocs.io/en/stable/**
- Various other tutorials: **https://docs.gitlab.com/ee/tutorials/**

# Revision Control best practices

- **Always** commit related changes together. Two bug fixes = Two different commits
- **Commit** often
- **Don't** commit half-done work to the main/master branch
- **Test** your code before you commit
- **Write** good commit messages
- **Use** branches
- **Follow** formatting rules:
    - short clear summary on the first line
    - **Always leave the second line blank**
    - Longer description after the second line if required
- **Tag** to keep track of important moments (paper status for example)
- Use **Branches** to apply fixes to distributed software
- Use **GitLab**/GitHub/BitBucket/etc to: centralize, share, keep issues and instructions accessible to all parties.

# Outline

# Where are we?

We just uploaded our project into a Git server, so it is now easier to share it, and inspect changes. Social code hosting is an important step into full reproducibility!

It is now time we work on the installation of the code. Let's start from our knowledge of the environment required to run it, and build a fully installable package.

# Packaging - Motivation

If we grep all imports in the code, we can distinguish: Python, numpy, scikit-learn, yaml, tabulate and matplotlib.

# Packaging - Motivation

If we grep all imports in the code, we can distinguish: Python, numpy, scikit-learn, yaml, tabulate and matplotlib.

**What happens if?**

This is a (very) simple project with a small number of dependencies. What if:

- You have a project with several dependencies?
- These packages depend on other packages?
- How do you manage this?

# Messy

You guessed it right!

# Our Objectives

Before we start doing things, what should be our objectives?

# Our Objectives

Before we start doing things, what should be our objectives?

- Create an archive of our software

# Our Objectives

Before we start doing things, what should be our objectives?

- Create an archive of our software
- Encapsulate metadata related to the software:
  - Dependence versioning
  - LICENSE
  - Further metadata (name, version, description, etc.)
  - (Documentation)

# Our Objectives

Before we start doing things, what should be our objectives?

- Create an archive of our software
- Encapsulate metadata related to the software:
  - Dependence versioning
  - LICENSE
  - Further metadata (name, version, description, etc.)
  - (Documentation)
- Make the software easily accessible (via URL?)

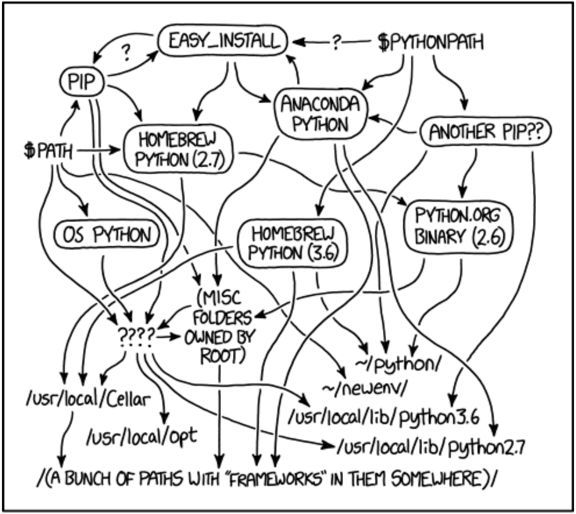
# Packaging is not New

The idea of bundling software is not new.

- In the past, mostly used to ship (base) Operating Systems
- Next in the pipe was providing software/updates to users:
  - Linux/Debian (dpkg/apt), Linux/RedHat (rpm/yum), Free-BSD/port, Snap (**https://snapcraft.io**)
  - MacOS/MacPorts or Homebrew: **https://brew.sh**
  - Windows/Chocolatey: **https://chocolatey.org**
- As complementary part of a programming language itself:
  - Python: **https://packaging.python.org**
  - NPM/Javascript: **https://www.npmjs.com**
  - Ruby Gems: **https://rubygems.org**
- More recently, we saw the apperance of OS- and language-independent package managers, such as Conda **https://docs.conda.io/en/latest/**

# State of Python Packaging

This inevitably created a gigantic mess!




MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

# Python Packaging: Disclaimer

Whereas Python provides its own packaging standards:

- Python packaging does not include non-Python packages
- It requires Python is pre-installed (chicken-and-egg problem)

# Python Packaging: Disclaimer

Whereas Python provides its own packaging standards:

- Python packaging does not include non-Python packages
- It requires Python is pre-installed (chicken-and-egg problem)

In our hands-on example today, we will:

- Wrap our package into a Python package
- Test our installation to make sure it works in
  - The `ifiss` environment
  - Another fresh environment with Python-only (if time allows)

# Python Packaging

Python has a standard for tracking the inter-dependencies between packages and more!

- Dependence tracking: **https://pypi.org/project/setuptools/**
- Version numbering: **https://www.python.org/dev/peps/pep-0386/**
- Package Repository: **https://pypi.org**

Let's explore each of these concepts and package our code!

# Anatomy of a Python Package

A Python Package is a zip or a tar archive containing an organized **directory** structure.

```
.
+-- <package>/      # your code in a directory
|   +-- __init__.py # package marker
|   +-- ...         # other files
+-- setup.py        # installation + requirements
+-- MANIFEST.in     # extras to be installed, besides the Python files
+-- README.md       # basic information
+-- LICENSE         # licensing information (optional)
```

# Hands on! Let's package our prototype

**Step 1**: You must re-structure the code, so all Python files and everything you want to ship to your users, is within the Python-package directory.

See next!

# Hands on! Let's package our prototype (2)

Hint: here is our organization [5 minutes]

```
ifiss
+-- __init__.py
+-- algorithm.py
+-- config
|   +-- __init__.py
|   +-- network-ablation.yaml
|   +-- network.yaml
|   +-- svm-ablation.yaml
|   +-- svm.yaml
+-- data
|   +-- __init__.py
|   +-- iris.csv  #renamed from data.csv
+-- dataset.py
+-- evaluation.py
+-- loader.py
+-- scripts
    +-- __init__.py
    +-- ablation_study.py
    +-- execute.py  #renamed from script.py
```

# Hands on! Let's package our prototype

**Step 2**: Write the `setup.py` file, place it in the root of the package. This file tells `setuptools` what is inside the package, how to install it (5 minutes).

https://gitlab.uzh.ch/manuel.guenther/ifiss2022/-/tree/master/b-packaging

Inspect the file `setup.py` and check what is inside. For more information on the syntax, check:
**https://packaging.python.org/tutorials/packaging-projects/**

# Note on Pinning Dependencies

Inside the `setup.py` file, we must indicate which other packages this package depends on:

```
...
install_requires=[
    "setuptools",
    "numpy==1.23.0",
    "scikit-learn==1.1.1",
    "pyyaml==6.0",
    "matplotlib==3.5.2",
    "tabulate==0.8.10",
],
...
```

By selecting a specific version of a dependence, we say to have "pinned" that dependence. This is good practice to ensure the version you tested with is the version that is going to be deployed.

If you would like to have a more flexible deployment environment, than you need to test your package with more variants.

# Note on version numbers

Version numbers are easy in Python[9]:

```
0.9.6
1.0.4
3.1.2a3
0.0.1b1
10.10.12c1
```

## In short

- 3 numbers separated by dots
- (optional) `[abc]N`, indicating *alpha*, *beta* or *candidate* versions
- Apply Semantic Versioning (**https://semver.org**). Given a version number `MAJOR.MINOR.PATCH`, increment the:
  - MAJOR version when you make incompatible API changes,
  - MINOR version when you add functionality in a backwards compatible manner, and
  - PATCH version when you make backwards compatible bug fixes.

[9] https://www.python.org/dev/peps/pep-0386/

# Note on `MANIFEST.in`

Setuptools will by default install all files ending in `.py`, from within the package directory. The file `MANIFEST.in` contains a list of all **other** elements that also need to be shipped.

```
include LICENSE
recursive-include ifiss *.yaml *.csv
```

We are saying: *Also ship the `LICENSE` file with my package. Find and also ship any file inside the directory `ifiss` that ends in `.yaml` or `.csv`.* Note the file `README.md` is included automatically.

More information: **https://packaging.python.org/en/latest/guides/using-manifest-in/**

# Hands on! Let's package our prototype

**Step 3**: Change package imports to *relative* imports. Internal module imports within the packages must be marked as so, to avoid undesired external module loading (5 minutes).

```python
# old
from dataset import Data

# new
from ..dataset import Data
```

Reference: **https://docs.python.org/3/reference/import.html#package-relative-imports**

# Note: Finding non-Python files

When you install the package, things like `import ifiss` will work out of the box. But how to find CSV and YAML files we need to load to execute our scripts? Where is the package installed? How?

# Note: Finding non-Python files

When you install the package, things like `import ifiss` will work out of the box. But how to find CSV and YAML files we need to load to execute our scripts? Where is the package installed? How?

We will use Python's importlib.resources to load package resources:

```
# old
for line in open(filename, "rt"):
    ...

# new
import importlib.resources
# package is, e.g., "ifiss.config"
# filename is, e.g., "network-ablation.yaml"
for line in importlib.resources.open_text(package, filename):
    ...
```

# Hands on! Let's package our prototype

**Step 4**: Change the method for loading text files shipped with the distribution, like below (5 minutes).

We will create a new module called `loader.py` which contains a special version of the `open()` function:

```
import importlib.resources

def open_textfile_from_package_or_filesystem(name, extension):
    if name.endswith(extension):
        name = name[:-len(extension)]
    if name.startswith("self"):
        name = name.replace("self", __name__.split(".", 1)[0])
    try:
        package, filename = name.rsplit(".", 1)
        return importlib.resources.open_text(package, filename + extension)
    except Exception:
        return open(name + extension, "rt")
```

Find the instances in which a text file needs to be loaded, and replace the calls with this function. In doubt, check our implementation of packaging in the shared repository.

# Hands on! Test package installation

**Step 5**: We should be ready to test the installation of the code. To do so, just run (3 minutes):

```
# Install the package alongside other system packages
(ifiss) $ pip install .
# Install a link to the current directory,
# allowing local edits - very nice for development!
(ifiss) $ pip install -e .
```

Play with your installation:

- Uninstall with `pip uninstall ifiss`
- Install using one of the two methods above, change working directories, confirm you can still use the packages, `python -c 'import ifiss; print(ifiss)'`
- Write a "mistake" on `loader.py`, see what happens when you call the script `execute` in each case

# Hands on! Install from away

You don't have to be within the directory containing `setup.py` to install your package. Try these things (5 minutes):

1. Uninstall the package, go to the parent directory and now install from there: `pip install -e <directory-with-setup.py>`
2. Commit the code to your GitLab repository (remember to follow the branch/push/review/merge workflow!) and install from it directly:

```
# installs the default branch
(ifiss) $ pip install -e
git+ssh://git@gitlab.uzh.ch/<user>/<project>.git#egg=ifiss
```

More information on syntax, installing tags or different branches: **https://packaging.python.org/en/latest/tutorials/installing-packages/**

# Where do (Python) packages come from?

**E.g.: `pip install numpy`**

They come from a central Python Package Index, or simply said `PyPI`.

- PyPI is a standard Python package distribution mechanism
- The web address is: **https://pypi.org**
- You may distribute your packages using it instead of GitLab:
  - Can hold large packages (>100MB OK)
  - Separates development from ready-to-use states
  - Simplifies package installation (no need to know the Git repository address)

Know more:

- Test server (for checking things work, wiped out every now and then): **https://test.pypi.org**
- Official server: **https://pypi.org**
- Upload your packages with twine

# Pip-based Workflow

Recommended workflow:

- Keep your software project at Git server
- Create instructions to package your software
- Everytime you need a release, tag the package on the server
- Upload package to PyPI
- Deploy from PyPI using pip

# Python Virtual Environments

Virtual environments allow one to achieve executable/library installation "insulation" for multiple installations at the same time.

For example, in these conditions, virtual environments are useful:

- Test software you are developing in various combination of dependencies before you ship it
- Test third-party software, without breaking your machine's Python installation
- Work on various projects in parallel with different dependencies (e.g. project A depends on numpy 1.14, whereas project B depends on numpy 1.9)

# How do they work?

In Python, virtual environments are created by:

- Copying (or linking) all python binaries and installed packages to a new directory
- Modifying the loading path of that copy to locate and install packages on the new directory (and not on the original location anymore)
- Modifying your environment so that you find binaries and libraries first on that new location

Pip-installing on the new directory, only affects the new directory installation of Python.

By deleting the new directory, the virtual environment is erased. The old environment remains *unaffected*.

# Python Implementations

To create virtual environments, use one of the following packages:

- python venv: **https://docs.python.org/3/library/venv.html**
- virtualenv: **https://pypi.org/project/virtualenv/**

```
$ python -m venv ~/venv  #a new env with **no packages!**
$ source ~/venv/bin/activate
(venv) $ pip list
...
(venv) $ pip install
git+ssh://git@gitlab.uzh.ch/<user>/<project>.git#egg=ifiss
(venv) $ ablation-study
...
```

# Better Packaging with Conda

Motivations for (yet) another Package Manager:

- Pick were Python/Pip/PyPI/virtualenv left-off
- Merge Python/Pip/PyPI/virtualenv into a single tool that provides a complete service
- Allow handling of non-Python packages (e.g. MKL, HDF5, LLVM, etc.)
- Virtual environments with different Python versions possible! (not possible with venv or virtualenv)
- Architecture, OS **and** Language Agnostic - works well for Python, R, C, C++, Javascript, or any other language.
- Better package resolution and does not require base environment (except for libc) to install other packages.

Read:
**https://www.anaconda.com/understanding-conda-and-pip/**

# Conda enviroments

You can manage conda environments[10] pretty much like you do for virtualenv. You can check which environments are available with:

```
$ conda env list
...
```

You can switch between available environments by *first* deactivating the current environment and then activating the environment you want.

```
# activate an environment:
# update search paths for libraries and executables
$ conda activate <envname>
# deactivate it:
# remove environment's library and executables from search path
$ conda deactivate
```

[10] https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html

# Creating environments

You can create new conda environments easily. You call `conda create`, pass a name, and a package list, with requirements.

```
$ conda create --name newenv python=3 pip numpy
...
$ conda activate newenv
(newenv) $ which python
(newenv) $ python
...
>>> import numpy  #works because it is installed
```

Environments are completely isolated, and you can create as many as you may need. Try adding more packages to this environment, for example.

# Freezing conda environments

You can save the complete specification of an environment, and give it to somebodyelse so they can reproduce your setup.

To do so, you can use either commands:

```
$ conda env export --name newenv > environment.yml
$ #or
$ conda list --export --name newenv > environment.txt
$ #for ultimate reproducibility:
$ conda list --explicit --name newenv > explicit-environment.txt
```

# Repeating an environment

Re-creating an environment from a list of packages can be done like the following:

```
$ conda create --name copyenv --file environment.txt
# or
$ conda create --name copyenv --file explicit-environment.txt
# or
$ conda env create -f environment.yml
```

Choose the `YAML` file format for a more complete specification, including download URLs for the packages.

# Hands on! New conda environment and testing

Let's create a new conda environment, and test our package installation in the case nothing is pre-installed.

```
(ifiss) $ conda deactivate
$ conda create --name ifiss2 python=3 pip
$ conda activate ifiss2
(ifiss2) $ which pip
...
(ifiss2) $ pip install
git+ssh://git@gitlab.uzh.ch/<user>/<project>.git#egg=ifiss
```

# Outline

# Unit testing

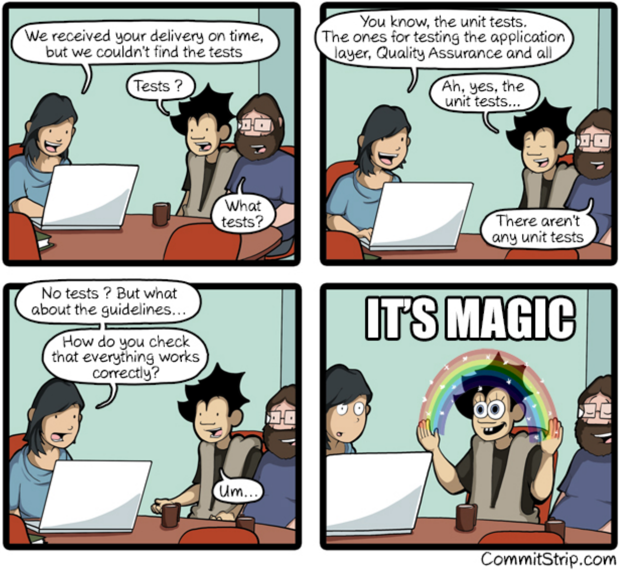



Image courtesy: **http://www.commitstrip.com/en/?**

# There is more than unit testing

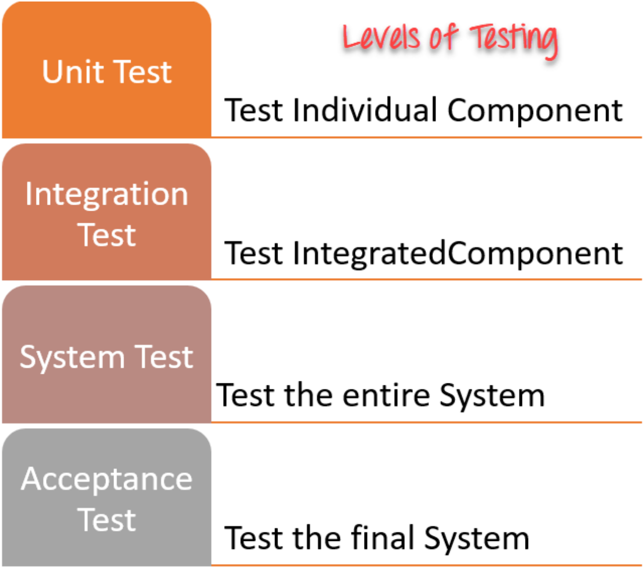



Image courtesy: https://www.guru99.com/levels-of-testing.html

# What is unit testing?

- A *unit test* is a piece of code that checks another piece of code for correctness
- A set of unit tests can evaluate different parts of the software
- There are specialized tools that help writing and executing unit tests
- **Testing improves the overall reproducibility of your code across different setups (architectures, software stacks, etc.)**

# Why do we need unit testing?

- Software is **hard (expensive) to create** and bugs are a major issue
- Software is **complex** and changes in one place can add bugs to another
- Software is **always changing** and unit testing helps performing sanity checks

# Hands-on! Setup

Let's start by looking the existing packaged solution at `7-docs`.

Pip-install (`-e`) this solution on your current environment, replacing your own.

```
(ifiss) $ cd 7-docs
(ifiss) $ pip install -e .
```

# Go back to `execute`

Re-run the program `execute`

```
(ifiss) $ execute
Loading internal resource at 'ifiss.config.network.yaml'
Loading internal resource at 'ifiss.data.iris.csv'
Loading internal resource at 'ifiss.data.iris.csv'
Loading internal resource at 'ifiss.data.iris.csv'
Error: 0.000%
```

Have you noticed something weird?

# Go back to `execute`

Re-run the program `execute`

```
(ifiss) $ execute
Loading internal resource at 'ifiss.config.network.yaml'
Loading internal resource at 'ifiss.data.iris.csv'
Loading internal resource at 'ifiss.data.iris.csv'
Loading internal resource at 'ifiss.data.iris.csv'
Error: 0.000%
```

Have you noticed something weird?

**Well...**

... the classification error is **now** zero!

# There was an error!

Your colleague, unwillingly, has introduced a **programming bug** in their analysis code.

Can you guess where it is?

**Tip**

In Python, a integer division gives an integer output!

# There was an error!

Your colleague, unwillingly, has introduced a **programming bug** in their analysis code.

Can you guess where it is?

**Tip**

In Python, a integer division gives an integer output!

**Error**

Open the file `ifiss/evaluation.py` and search for:

```
return wrong // count
```

# To err is human

Unfortunately, programming is a tricky business. You're bound to make mistakes:

- Buggy, untested cases which become important later
- Naive "improvements" which cause errors somewhere else
- Bug fixes may also introduce more bugs
- ...

But how to prevent these errors?

# To err is human

Unfortunately, programming is a tricky business. You're bound to make mistakes:

- Buggy, untested cases which become important later
- Naive "improvements" which cause errors somewhere else
- Bug fixes may also introduce more bugs
- ...

But how to prevent these errors?

**For starters...**

(from experience) Be humble! Assume from the start, you will make mistakes.

# Defensive programming

That basically means: “Prevent problems before they appear”. For example:

```
def division(a, b):
    return a/b
```

Could be best written like this:

```
def division(a, b):
    assert b != 0
    return a/b
```

# Unit testing

In unit testing we test each functional unit of our code with a tuple of **expected inputs and outputs**. The behavior of the unit must respect the expected input and output or the test **fails**.

Each unit test is encoded in a single function. For example:

```
import ifiss

def test_CER_50_50():
    assert ifiss.evaluation.classification_error_rate(5, 10) == 0.5
```

# Tools

We can use a couple of tools for this:

- A program to run your tests (a.k.a. "test runner")
- A set of utility functions that help you to check for specific conditions (and errors)

# Unit Testing in Python

In Python, there are few packages for this:

- The native `unittest` module: **https://docs.python.org/3/library/unittest.html** (utility functions)
- `pytest`: **https://docs.pytest.org/en/latest/**
- `nose`: **https://nose.readthedocs.org/en/latest/** (halted development, refrain from using in production)

In this course, we will examplify using `pytest`, but you are free to adopt any tool you prefer.

N.B.: You will find similar tools for other programming languages.

# Introduction to Pytest

The package `pytest` extends the native `unittest` with specific tools and other useful functionality, also providing a simpler way to write tests.

The `pytest` runner searches for objects on a given directory structure that match a search expression (basically, anything starting with `test` on their name). It considers those objects to be test units and executes them.

`pytest` provides a set of utilities to help encoding test units in a readable manner.

# Basic skeleton of a test program

```python
import pytest

def test_one():  #no parameters!
    assert True  #use the stock Python assertion

def test_two():
    assert False  #this test will fail, obviously

def test_equal():
    a = 5
    b = 5
    assert a == b

@pytest.mark.skip(reason="no way of currently testing this")
def test_the_unknown():
    assert False  #this will never be executed, nor will it fail
```

# Python Testing Tips

- The test code for function `foo()` should be named like `test_foo_variant_1()`
- Start with easy, obvious cases where the function would work
- Call every function a few times differently in different test
- Test boundary inputs of the function
- Try to have tests that ‘cover’ all functionality (more on "coverage" later)

# Patching the issue

How to patch an issue?

1. Implement test units that trigger the observed problem
2. Verify the problem is reproducible
3. Patch the issue and re-run the test suite (check the problem is gone)
4. Branch/Push/Review/Merge as per usual

Let's see how we fixed the code so that the tests run correctly.

```
(ifiss) $ cd c-tested
(ifiss) $ pip install -e .
# have a look at ifiss/evaluation.py, run test units to check
(project) $ pytests test.py
....
================================ 4 passed in 0.90s ===
```

# Exercise

Open the file `test.py`. Try to understand it. Do you?

```
# edit test.py
```

# Solution

Here is the solution to the problem, in `ifiss/evaluation.py`:

```
return wrong / count
```

Running the tests should now get you:

```
(project) # using the "-v" flag on the runner may help you
(project) $ pytest -v test.py
...
test.py::test_CER_0 PASSED                    [ 25%]
test.py::test_CER_50_50 PASSED                [ 50%]
test.py::test_CER_20_80 PASSED                [ 75%]
test.py::test_CER_1 PASSED                    [100%]

====================== 4 passed in 0.57s =====...
```

# Other useful testing tools

**https://docs.pytest.org/en/stable/example/index.html**

- Fixtures: useful to setup and cleanup one or more test units (e.g. setup connection to server for multiple tests, delete generated files)
- Attributes: tag test functions for a quick selection (e.g. "run only fast tests")
- Logging support: capture logging messages to ensure they are happening in the right order
- Doctest integration: test code snippets (examples) in your documentation
- ...and many more!

# How much should I test?

This is a common question when you are confronted for the first time with unit testing.

There is no *right* answer to this, but:

- Avoid complicated functions, but if you can't, test them thoroughly
- Use test units to *pin* behavior you'd like to be tested when changes occur
- Test as much as you can (don't believe you aren't prone to errors!)

**Tip**

Test units checks any changes did not negatively impacted your code, **quickly**.

# Coverage

Coverage gives you *an idea* on how much of the code functionality the unit tests evaluate or *cover*.

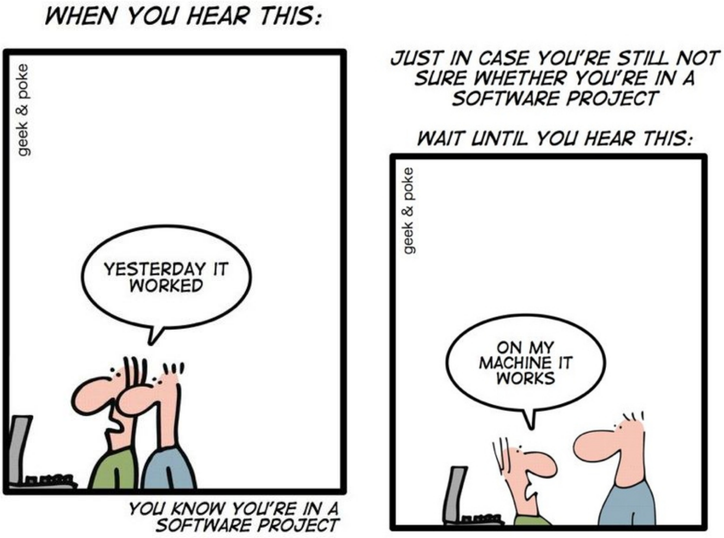
```
(ifiss) $ pytest --cov=ifiss test.py
...
Name                               Stmts   Miss  Cover
-----------------------------------------------------
ifiss/__init__.py                      6      0   100%
ifiss/algorithm.py                    51     32    37%
ifiss/config/__init__.py               0      0   100%
ifiss/data/__init__.py                 0      0   100%
ifiss/dataset.py                      24     16    33%
ifiss/evaluation.py                   24     14    42%
ifiss/loader.py                       15     13    13%
ifiss/scripts/__init__.py              0      0   100%
ifiss/scripts/ablation_study.py       36     36     0%
ifiss/scripts/execute.py              37     37     0%
-----------------------------------------------------
TOTAL                                193    148    23%
```

# Coverage

The Python package `coverage.py` provides more functionality for the test coverage **https://coverage.readthedocs.io**. For example, it can generate a full HTML report.

```
(ifiss) $ coverage report -m
(ifiss) $ coverage html
```

# Continuous Integration (CI)




Images courtesy: **http://geek-and-poke.com/**

# What is Continuous Integration (CI)?

- CI helps you running your tests after pushing changes to your repository, *automatically*
- It is typically implemented using programmable *hooks* - every time you push code to your repository, the (GitLab) server sends a message (triggers hook) to a service that runs a script on particular machine, on your behalf.
- We define and use this *script* to run our test units. Errors are reported back to the user (typically via an e-mail).
- A CI service can be used for more than just running (unit) tests. E.g.: they can be used to automatically calculate and publish software coverage, update the package documentation, or publish new versions of your package to PyPI.
- Gitlab CI **https://docs.gitlab.com/ee/ci/**, if you have a GitLab installation, that is the way to go.
- Follow the tutorials on **https://docs.github.com/en/actions/learn-github-actions** for further details on a GitHub-based CI.

# Thank you for your attention!

Dr. André Anjos (anjos.ai)
Prof. Manuel Günther (www.ifi.uzh.ch)

June, 29th 2022


idiap RESEARCH INSTITUTE Universität Zürich[UZH]

# From Scripts to Reusable Software and Reproducible Research

## Part II: Code Organization

Instructor: Manuel Günther
Email: guenther@ifi.uzh.ch
Office: AND 2.54

Wednesday, June 29, 2022

# Outline

1. Current Situation
2. Toward Re-Usable Code
3. Toward Algorithm Comparison

# Outline

1. Current Situation

# Current Situation

## One script that does it all

- Hard-coded parameters
- Implicit data structures
- Unreadable variable names
- Repeated code
- Commented-out code
  - → No other comments
- Star-imports

⇒ Write-only code

## Example

`1-single-script/OneForAll.py`

## Issues

- Data not accessible
  - → Hard-coded paths
- Bugs in code
  - → This code did not produce the published results!
- Algorithm unclear
- When run several times
  - → Very different results

# Current Situation

## Why is This Code bad?

- Hard-coded paths need to be changed before being able to run
- Knowledge about data files (class names) are hard-coded
  - → Not applicable to other data files
- Hard-coded parameters with various commented-out options
  - → Not clear which version produced which results
- Everything in one file, no functions
  - → Impossible to replace parts of the code by something else
  - → Running on different dataset requires to copy and adapt code

## Bottomline

This code runs exactly this experiment, but nothing else.

# Outline

2. Toward Re-Usable Code
   - Collect Parameters
   - Code Implementation
   - Split Your Code to Files

# Toward Re-Usable Code

## Structure your Code

- Software design
  - → Provide code interface
- Define classes and functions
  - → Include appropriate parameters
  - → Use meaningful names
- Encapsulate internal data
  - → Provide access functions
- Foresee different use cases
  - → Provide flexible implementation
- Rely on standard library

## The Golden Rule: DRY

- **Don't Repeat Yourself**
  - → Write function with parameters
  - → Call function wherever needed

## Step 1: Identify Code Blocks

# Toward Re-Usable Code

## Structure your Code

- Software design
  - → Provide code interface
- Define classes and functions
  - → Include appropriate parameters
  - → Use meaningful names
- Encapsulate internal data
  - → Provide access functions
- Foresee different use cases
  - → Provide flexible implementation
- Rely on standard library

## The Golden Rule: DRY

- **Don't Repeat Yourself**
  - → Write function with parameters
  - → Call function wherever needed

## Step 1: Identify Code Blocks

- Data handling / loading
- Algorithm training
- Algorithm testing
- Evaluation

# Toward Re-Usable Code

## Encapsulated Code

`1-single-scripts/Encapsulated.py`

# Toward Re-Usable Code

## Step 2: Collect Parameters (a)

- Data loading:
- Target value handling:

## Step 2: Collect Parameters (b)

- Algorithm implementation:
- Internal parameters:

# Toward Re-Usable Code

## Step 2: Collect Parameters (a)

- Data loading:
  - → Data file name
  - → Indexes of data
  - → Index of label
  - → Header rows
  - → Delimiter; how is data stored
- Target value handling:

## Step 2: Collect Parameters (b)

- Algorithm implementation:
- Internal parameters:

# Toward Re-Usable Code

## Step 2: Collect Parameters (a)

- Data loading:
  - → Data file name
  - → Indexes of data
  - → Index of label
  - → Header rows
  - → Delimiter; how is data stored
- Target value handling:
  - → Class names (from file)
  - → Class indexes (evaluation)
  - → One-hot representation (training)

## Step 2: Collect Parameters (b)

- Algorithm implementation:
- Internal parameters:

# Toward Re-Usable Code

## Step 2: Collect Parameters (a)

- Data loading:
  - → Data file name
  - → Indexes of data
  - → Index of label
  - → Header rows
  - → Delimiter; how is data stored
- Target value handling:
  - → Class names (from file)
  - → Class indexes (evaluation)
  - → One-hot representation (training)

## Step 2: Collect Parameters (b)

- Algorithm implementation:
  - → Input dimension (given by data)
  - → Number of outputs (given by data)
  - → Hidden layer size (to be specified)
  - → Learning rate (to be specified)
  - → Training epochs (to be specified)
- Internal parameters:
  - → Network topology
  - → Activation function

# Toward Re-Usable Code

## Step 3: Code Implementation

- Encapsulate your code
  - → Implement classes and functions
  - → With internal data, use classes
- Take any possible parameter as a parameter
  - → Can have default values
  - → Parameter handling will follow

## Writing Classes

- Instantiate everything in constructor `__init__()`
  - → Load data from files
  - → Instantiate matrices of algorithm
  - → Lazy initialization possible
- Store data internally
  - → Indicate private data with `_`
  - → Access to data through functions
- Define interface functions
  - → Requirements of inputs and outputs

# Toward Re-Usable Code

## Step 3 (a): Dataset

- What kind of information needs to be provided to the outside?
- What functionality do we need to provide?

# Toward Re-Usable Code

## Step 3 (a): Dataset

- What kind of information needs to be provided to the outside?
  - → Input data
  - → Target data in different formats
  - → ~~Class names~~
- What functionality do we need to provide?

# Toward Re-Usable Code

## Step 3 (a): Dataset

- What kind of information needs to be provided to the outside?
  - → Input data
  - → Target data in different formats
  - → ~~Class names~~
- What functionality do we need to provide?
  - → Load data from file → `__init__(dataset_params)`
  - → Provide data → `data()`
  - → Provide labels → `labels(how)`

# Toward Re-Usable Code

## Step 3 (a): Dataset

- What kind of information needs to be provided to the outside?
  - → Input data
  - → Target data in different formats
  - → ~~Class names~~
- What functionality do we need to provide?
  - → Load data from file → `__init__(dataset_params)`
  - → Provide data → `data()`
  - → Provide labels → `labels(how)`
- What data format do we provide, how do we arrange data?
  - → Provide clear documentation, see later today

# Toward Re-Usable Code

## Step 3(b): Algorithm

- What kind of information needs to be provided to the outside?
- What functionality do we need to provide?

# Toward Re-Usable Code

## Step 3(b): Algorithm

- What kind of information needs to be provided to the outside?
  $\longrightarrow$ None!
- What functionality do we need to provide?

# Toward Re-Usable Code

## Step 3(b): Algorithm

- What kind of information needs to be provided to the outside?
  - → None!
- What functionality do we need to provide?
  - → Initialize our model → `__init__(network_params)`
  - → Train our model → `train(data, training_params)`
  - → Test our model → `test(data)`

# Toward Re-Usable Code

## Step 3(b): Algorithm

- What kind of information needs to be provided to the outside?
  - → None!
- What functionality do we need to provide?
  - → Initialize our model → `__init__(network_params)`
  - → Train our model → `train(data, training_params)`
  - → Test our model → `test(data)`
- What is the data format that we expect?
  - → Rely on interface from data set!

# Toward Re-Usable Code

## Step 3(c): Evaluation

- Do we need to store information?

## Step 3(d): Code Execution

- Instantiate everything we need
  → Select parameters, follow-up on this comes soon
- Train your algorithm
- Test your algorithm
- Evaluate your result

# Toward Re-Usable Code

## Step 3(c): Evaluation

- Do we need to store information?
  → No! Implement as function

## Step 3(d): Code Execution

- Instantiate everything we need
  → Select parameters, follow-up on this comes soon
- Train your algorithm
- Test your algorithm
- Evaluate your result

# Toward Re-Usable Code

## Step 3(c): Evaluation

- Do we need to store information?
  $\rightarrow$ No! Implement as function

## Step 3(d): Code Execution

- Instantiate everything we need
  $\rightarrow$ Select parameters, follow-up on this comes soon
- Train your algorithm
- Test your algorithm
- Evaluate your result

# Toward Re-Usable Code

## Step 4: Split your Code to Files

- Separate implementations from code execution
- Collect semantically similar code into files
  → Provide several implementations of related classes/structures
- Store files with implementations in same directory (for now)
- Import functionality from code files into script
- Use `if __name__ == "__main__"` guard for script
  → Maybe do the same in other files to include test code

## The Code split up into Four Files

`2-split-code`: `dataset.py`, `algorithm.py`, `evaluation.py`, `script.py`

# Outline

3 Toward Algorithm Comparison
- Comparing Algorithms
- Extending your Code

# Toward Algorithm Comparison

## We Still Have a Single Script for Everything

- Sometimes training takes a long time
  - → You do not want to call perform training each time that you want to test
  - → You want to be able to distribute pre-trained models
- Sometimes testing also takes a long time
  - → When changing plotting parameters, you do not want to re-test
  - → You need to evaluate your pre-trained model on different datasets
- You want to plot results of several algorithms into one plot?
  - → You might also want to change the evaluation procedure?

# Toward Algorithm Comparison

## Splitting Training, Testing and Evaluation Scripts

- One script/function to perform training
  - → Trained models need to be readable and writable
  - → Implement I/O functionality into algorithms
  - → You can use pythons `pickle` or other native methods
- One script/function to perform testing
  - → Load trained model from file
  - → Write resulting score files (`pickle`, `numpy.tofile`, CSV)
- One script/function to run the evaluation
  - → Load several score files from previous experiments
  - → Evaluate and plot results together in one plot

# From Scripts to Reusable Software and Reproducible Research

## Part III: Command Line Interface

Instructor: Manuel Günther
Email: guenther@ifi.uzh.ch
Office: AND 2.54

Wednesday, June 29, 2022

# Outline

# Outline

1. Let's Talk about Parameters
   - Basic Interface: `sys.argv`

# Let's Talk about Parameters

## Parameters in Scripts

- We have identified all parameters
  - → All classes and functions have all their parameters parametrizable
- Currently, we assign values in the script
- We run an experiment with these parameters and evaluate it
- We select different parameters and run again
  - → How do we know which results were generated with which parameters?

## Sad but True

For many experiments published in papers, even the authors do not know which set of parameters led to their published results!

# Let's Talk about Parameters

## Ways of Providing Parameters

- Command line interface on the console
- One way: Provide all parameters on command line
  → `$ python script.py param1 param2 param3`
- Maybe better: Provide named parameters
  → `$ python script.py --option1=param1 --option2=param2 --option3=param3`
- Include default values, only overwrite non-defaults:
  → `$ python script.py --option2=param2`
- Combine all parameters in a configuration file
  → `$ python script.py config.yaml`

# Basic Interface: `sys.argv`

## Python's Basic Interface

- Python provides command line parameters via `sys.argv`
  → List of strings, where `sys.argv[0]` is the current executable
- We can have many parameters, e.g.:
  → `dataset_file = sys.argv[1]`
- We can include default values
  → `dataset_file = sys.argv[1] if len(sys.argv) > 1 else "data.csv"`

## Disadvantages of Basic Interface

- You need to remember the order of parameters
- When you want to change only the third parameter, you need to remember the default values for all previous parameters

# Outline

2. Pythons Argument Parser `argparse`
   - Defining Parameters in `argparse`
   - Using Parameters in `argparse`
   - Hands-on: Implement Parser for our Script

# Python's Argument Parser `argparse`

## Command Line Interfaces

- There are several command line interfaces in python
  → `argparse`, `getopt`, `optparse`, `click`, and more
- Today we will focus on `argparse`, pythons standard parser
- There are several tutorials online, e.g.:
  https://docs.python.org/3/howto/argparse.html

# Python's Argument Parser `argparse`

## Basic use cases of Argument Parsers in Code

1. Create a parser object: `parser=argparse.ArgumentParser()`
2. Add all parameters: `parser.add_argument("param1", ...)`
   → Each parser has one default option: `--help / -h`
3. Collect arguments: `args = parser.parse_args()`
4. Use arguments: `args.param1`

## Obtaining the Help on how to Use the Script

- On command line: `$ python script.py --help`
  → Shows all parameters and usage instructions

# Python's Argument Parser `argparse`

## Parameters of Argument Parsers

- `description` provides a description of the script
  - → Should mention what the script is doing
- The `epilog` is written after listing the parameters
  - → Often shows usage examples
- `formatter_class` selects ways of formatting the help output
  - → I usually prefer `argparse.ArgumentDefaultsHelpFormatter`

## Example Parser Creation

```
parser = argparse.ArgumentParser(
    description = "Trains a machine learning model",
    formatter_class = argparse.ArgumentDefaultsHelpFormatter
)
```

# Defining Parameters in `argparse`

## Two Types of Parameters

- Required parameters, so-called arguments
  - → Often used as positional arguments, names are regular variable names
- Optional parameters, so-called options
  - → Names start with `--`; abbreviation with `-` exist

## Defining Arguments

```
parser.add_argument("required")
parser.add_argument("--optional", "-o")
```

## Command Line Usage

```
$ python script.py required_param -o optional-param
```

# Defining Parameters in `argparse`

## Options of `add_argument`

- `type`: the data type to accept
  - → Typically used with `int` or `float`, default: `str`
- `choices`: list of possible values, typically strings
  - → Example: `choices=["A", "B", "C"]`
- `default`: Provides a default when not provided on command line
  - → Shall be of the correct `type`, from the `choices`; not checked
  - → If `default` is not provided for an option, the default is `None`
- `nargs`: allows providing more than one parameter
  - → Can be a fixed number, e.g.: `nargs=4`
  - → Can be `"?"` (zero or one), `"+"` (one or more), or `"*"` (zero or more)
  - → The resulting variable will be a list; default is still `None`

# Defining Parameters in `argparse`

## Options of `add_argument` (continued)

- `required=True` option needs to be specified, contradicts `default`
  - → Turns options into arguments, but keeps the `--` notation
- `action` has several possibilities:
  - → `action="store_true"` turns options into flags, no parameters required
  - → `action="count"` counts how often this flag was used
- `dest` provides the destination variable in the final `args`
  - → By default, the name itself is used: `--param` will turn into `args.param`
  - → Dashes are replaced by underscore: `--my-param` → `args.my_param`
  - → Only for options, arguments require variable name
- `help` **the most important flag** shown in `$ python script.py -h`
  - → Tell users what this parameter/option/falg should be used for

# Defining Parameters in argparse

## Example 1: Input File

```
parser.add_argument(
  "input_file",
  type = pathlib.Path,
  help = "The file to be used"
)   {required}
```

## Example 2: List of Integers

```
parser.add_argument(
  "--gpu-indexes", "-g",
  type = int,
  nargs = "+",
  help = "Specify  GPU indexes"
)   {optional, default: None}
```

## Example 3: Choice of Functions

```
functions = {
  "max": numpy.max,
  "mean": numpy.mean,
  "sum": numpy.sum
}

parser.add_argument(
  "--numpy-function", "-f",
  choices = functions.keys(),
  default = "max",
  help = "Define function"
)   {optional, given values only}
```

# Defining Parameters in `argparse`

## Options vs. Arguments, Required vs. Optional

- Options start with `-` are optional
  - → Options always have a default (`None`)
  - → Default can be overwritten by `default = ...`
  - → You can make options required by `required = True`
- Arguments start without `-` and are required
  - → You can also have arguments with default values
  - → When defining, add `nargs = "?", default = ...`

# Using Parameters in `argparse`

## Parsing Arguments

- Simplest form: `args = parser.parse_args()`
  - → This is equivalent to `parser.parse_args(sys.argv[1:])`
- Can pass list of arguments, for example:
  - → `parser.parse_args(["filename", "-g", "0", "1"])`
  - → Particularly useful for test code or sub-procedure calls
  - → Often seen: `parser.parse_args("filename -g 0 1".split())`
- Result `args` is a `Namespace` object
  - → Stores values in variable with the provided names
  - → Can be converted into a `dict` via `vars(args)` or `args.__dict__`
  - → Can be written to file to store parameters together with results

# Hands-on: Implement Parser for our Script

## Hands-On: Command Line Parser (20 Minutes)

- Basis for the code: `3-command-line/script.py`
- Implement an argument parser with the following:
  - → A task `"train"`, `"test"`, `"eval"` to choose from
  - → A data file to read from, with a default value
  - → All parameters of the network training
  - → Filenames for intermediate files, with default values
- Replace hard-coded parameters with arguments
  - → You will need to pass `args` to all functions
- Call training, testing and evaluation from command line
- Write arguments of parser into text file

# Outline

3. Configuration Files
   - When Configuration Files are Better
   - Types of Configuration Files
   - YAML Ain't Markup Language
   - Using YAML Configuration Files

# When Configuration Files are Better

## Adding one More Algorithm

- We want to compare the result to another algorithm
- Here, we choose a Support Vector Machine
  → We rely on the `sklearn` implementation
- We need a small wrapper with the API of our `TwoLayerNetwork`
- We support two parameters: `C` and `kernel`

## Additional SVM Algorithm

`4-config-file/algorithm.py`

## Algorithm Selection

- How can we select the algorithm from the command line?
  → How can we specify all its parameters without changing the code?

# When Configuration Files are Better

## Advantages of Command Line Parsers

- You can run different experiments directly from command line
  - → No source code or other file needs to be changed
- You can adapt execution-specific parameters quickly
  - → E.g., use a specific GPU ID, limit number of processes, increase verbosity

## Disadvantages of Command Line Parsers

- If you have too many parameters, you can lose track
  - → You need to specify all parameters of all your algorithms
- Writing on console is error-prone and unpleasant

## Better Alternative

- Work with one or more configuration files

# Types of Configuration Files

## Simple Text Files

- Write parameter names and values line by line:

```
param1 = value1
param2 = "value2"
param3 = elem1,elem2,elem3
flag1 = True
```

- Disadvantages:
  - → No grouping of parameters
  - → Need to write own parser to deal with different data types

# Types of Configuration Files

## Structured Text Files

- Several standard formats available
  → Standard library implements parsers
- Well-known libraries:
  → Extensible Markup Language (XML)
  → JavaScript Object Notation (JSON)
  → YAML Ain't Markup Language (YAML), superset of JSON

## Python Scripts (done in Bob)

- Can assign variables in Python
- Can implement functionality in configuration file
  → Disadvantage: provides incentives to fall-back into bad coding habits

# YAML Ain't Markup Language

## Structural Elements in YAML

- Basic element: dictionary
  - → Using colon separator `:`
  - → Using curly braces `{}`
- Nested dictionaries: indentation
- Lists of scalars
  - → Using square brackets `[]`
  - → Using itemization `-`
- Various data types supported:
  - → `int`, `float`, `str`, `bool`
- More complex operations

## YAML Structure Examples

```yaml
# Dictionaries
str_param: string value
int_param: 42

# Nested dictionaries
nested_dict: {key1: str1, key2: 2}
other_nested_dict:
  key3: 1e-4
  key4: true

# Lists
int_list: [0,4,8,22]
param_list:
- elem1
- elem2: {key6: v6, key7: v7}
```

# YAML Ain't Markup Language

## Parsing YAML Files

- No native support (yet), requires library
  - → Recommended: PyYAML, but others also exist
  - → Easy installation via `$ pip install pyyaml`
- Loading YAML (or JSON) files via `yaml.safe_load()`
  - → Loads a **nested dictionary** with `str` keys
- Writing YAML files also possible via `yaml.dump())`

## Reading YAML File

```
import yaml
with open("my_config.yaml", "r") as f:
  config = yaml.safe_load(f)
```

# Using YAML Configuration Files

## Writing Configuration Files

- Define a dictionary with all parameters
- Create nested dictionaries for each code block
- Selecting the algorithm that you want to test
  → In the `algorithm` YAML block, provide `type` and `options`

## Using Configuation Files

- Pass the sub-dictionary to each code block
  → As a dictionary: `Dataset(config["dataset"])`
- Select the algorithm based on your `config["algorithm"]["type"]`
- Instantiate with loaded parameters **for this algorithm**:
  → `SupportVectorMachine(**config["algorithm"]["options"])`

# Using YAML Configuration Files

## Mixing YAML Configuration and `argparse`

- Combination of configuration and command line options
- Configuration for algorithm parameters
- Command line options for execution parameters
  - → Includes the configuration file

## Exemplary Code

- Script implementation: `4-config-file/script.py`
- Network configuration: `4-config-file/network.yaml`
- SVM configuration: `4-config-file/svm.yaml`
- Run on console

# Outline

4. Use Case: Ablation Study

# Use Case: Ablation Study

## Test Various Parameter Settings

- We have our algorithm, but which parameters work best?
- You want to test several combinations
  - → 4 different learning rates
  - → 5 different hidden neuron counts
- You want to get the results in a table to include into your report
- You also want a visual display of your results

## Naïve Approach

- Produce 20 configuration files
- Run the 20 experiments and collect results in an Excel sheet
  - → This is a root cause of trouble! Copy-paste errors are inevitable!

# Use Case: Ablation Study

## How to Run an Ablation Study

- Collect all the results in one script
- Run training, testing and evaluation for parameter combinations
  - → Can use IO capabilities if algorithms run longer
- Use the script to write results in a table
- Possibly make use of `python-tabulate` package
  - → Can write different formats, including LaTeX
- I often write LaTeX-`\input`-able files by hand
  - → You can automatically compute values to highlight
- Use the script to plot results into a `.pdf` figure
  - → Use `matplotlib` library to produce plots
  - → Do not write `.jpg` or `.png` when writing publications in LaTeX!

# Use Case: Ablation Study

## Modify your Configuration File

- Add a list `options-test` of tested parameters: `hidden_size`
  → Remove this from the default `options` dictionary
- Add a list `training-options-test` of tested parameters: `learning_rate`
  → Remove this from the default `training-options` dictionary

## Implement your Ablation Study Script

- Iterate over `options-test` and `training-options-test` lists
  → Add corresponding options to `options` and `training-options`
- Run training, test and evaluation for all parameter combinations

## Example Files for Ablation Study

`5-ablation-study`: `network-ablation.yaml`, `ablation_script.py`

# Use Case: Ablation Study

## Hands On: Ablation Study for SVM Parameters (10 Minutes)

- Write a configuration file to run ablation study for SVM
  - → Four values for `C`: `10, 1, 0.1, 0.01`
  - → Four different `kernels`: `"linear", "rbf", "poly", "sigmoid"`
  - → Plot the results into `svm.pdf`
  - → Write a LaTeX table `svm.tex`
- Modify the `ablation_script.py` to test two parameters
  - → We have two different `options-test`, and no `training-options-test`

# Final Remarks before Lunch

## Parameters of Functions

- For our `TwoLayerNetwork` we had two sets of parameters
  → Construction parameters, training parameters
- Difficult to handle different sets of parameters
  → We needed to modify our `ablation_study.py` script for SVM
- Solution: provide **all** parameters to constructor
  → Default behavior in `sklearn`

## Different Versions of Code

- We have now five different versions of the code
- How to organize it better than copying and modifying?
  → Follows after lunch.

# From Scripts to Reusable Software and Reproducible Research

## Part VI: Documentation and Usage Instructions

Instructor: Manuel Günther
Email: guenther@ifi.uzh.ch
Office: AND 2.54

Wednesday, June 29, 2022

# Outline

# Outline

1. Package Documentation
   - The README File
   - Package Usage Instructions
   - Jupyter Notebooks
   - Hands-On: Write Jupyter Notebook Example

# Why Code Documentation

## You Want People to Use your Code

- If people don't know why your project exists, they won't use it.
- If people can't figure out how to install your code, they won't use it.
- If people can't figure out how to use your code, they won't use it.

https://www.writethedocs.org/guide/writing/beginners-guide-to-docs/0

## You want People to Cite your Work

- If people don't use your code, you won't get cited.

$\longrightarrow$ At least not that often

# The README File

## The Entry Point to your Code

- README files are shown on front page of Git, PyPI
  - → Written in ReStructuredText README.rst or Markdown README.md
- Provide all relevant information of your package
  - → What is the purpose of the package
  - → What is the software license that applies
  - → Which paper should be cited when using this code
- Provide installation instructions
  - → Write how to install it using pip, if applicable
  - → Provide requirements of other non-standard packages, if applicable
- Add links to resources: datasets, pre-trained models or alike

# The README File

## What is Contained in the Package

- Talk about all important files and directories
  - → Where to find the implementations of functionality
  - → Which of the files contain runnable scripts
  - → Where to find configuration files and for which experiments
- List all relevant scripts
  - → Provide detailed information about scripts
  - → Talk about dependencies: in which order the scripts must be executed?
  - → Provide command line (sequences) to reproduce your results
- Make sure that the scripts reproduce your results
  - → **Optimally you have used these scripts to produce your results!**

# Package Usage Instructions

## Provide Use Cases for your Package

- Go step by step and explain functionality
- Illustrate complicated structure with graphics or math
- Explain in simple words how to use your code
- Provide a simple example how to **extend** your code

## Building a Documentation

- Write documentation in MarkDown or ReStructuredText
- Use sphinx to turn into HTML documentation

  $\rightarrow$ Can also include auto-generated API documentation
- Not included into our small course today

# Package Usage Instructions

## Excursion: The `__init__.py` File

- Defines how your package/module can be used
- An (empty) `__init__.py` file defines a module
  - → Allows to import from module, e.g., `from ifiss import dataset`
  - → When importing the module: `import ifiss` it is **empty**
- You can add contents by local import inside `__init__.py`
  - → Add code files: `from . import dataset, algorithm, evaluation`
  - → Add submodules: `from . import scripts, data, config`
    Submodules also need to contain an (empty) `__init__.py` file
- Afterward, you can use `import ifiss` and access `ifiss.dataset`
- Sometimes this is required for `importlib` to find the module

# Jupyter Notebooks

## One Famous Tool for Showcasing your Code

- Jupyter Notebooks (formerly known as IPython Notebooks)
- Incorporates source code and explanation
- Supported by most Git servers

## Writing Jupyter Notebooks

- Start notebook server in any directory
  - → Code is found since it is installed locally via pip
- Two types of blocks
  - → Text block in MarkDown syntax to explain your code
  - → Code block for running your Python code

# Jupyter Notebooks

## Starting a Jupyter Notebook

- The most simple way: go to console and write `jupyter notebook`
  → This opens a browser window and shows current directory
- If you have an existing notebook, click on it
  → Or directly open it on command line `jupyter notebook Network.ipynb`
- You can also create new notebook by selecting `New` (top-right)

## Other Jupyter Notebook Editors

- JupyterLab: https://jupyterlab.readthedocs.io/en/stable
- Integrate into Visual Studio Code:
  https://code.visualstudio.com/docs/datascience/jupyter-notebooks

# Jupyter Notebooks

## Creating Code Blocks

- Click on the + icon (second from left)
- Import the modules that you require
- Write any regular Python code
- Add new block via +
- Continue code from above (base indent only)

## Running Cells

- Click on the icon next to the cell
- Output of the **last** operation written

→ Use `print` otherwise

# Jupyter Notebooks

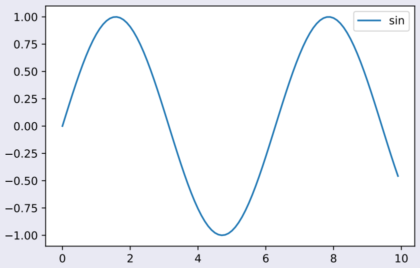
## Plotting in Jupyter Notebooks

- Plots via `matplotlib` are displayed in notebook
  → Can also be saved into files (current directory)

## Plotting Code

```
from matplotlib import pyplot
import numpy
x = numpy.arange(0,10,0.1)
y = numpy.sin(x)
pyplot.plot(x,y, label="sin")
pyplot.legend()
pyplot.savefig("sin.pdf")
```

## Jupyter Notebook Output



# Jupyter Notebooks

## Creating Documentation Blocks

- Click on the + icon (second from left)
- Change the type to `Markdown` (second from right)

## Writing Documentation in Markdown

- Headers start with `#` (first level), `##` (second level), `###` and so on
- Separate paragraphs of text via empty lines
- Include links: `[Link Text](https://url "Optional Title")`
- Write LaTeX-style equations: `$ $` (inline) or `$$ $$` (display style)
- Code highlighting via `` `code example` ``
- **Run Markdown Cell to apply and display correctly**

# Jupyter Notebooks

## Typical Contents of a Machine Learning Notebook

- Load the data and show some statistics
  - → This makes the code easier to be understood
  - → For example, use the `tabulate` package
- Instantiate your models and explain all its parameters
  - → Mention why you chose which parameter
- Train the models on your training data
- Apply the model on your validation/test data
- Provide some (graphical) interpretation of the results
  - → Plot confusion matrices, ROC plots or whatever applies

# Hands-On: Write Jupyter Notebook Example

## Example Notebook for the Network

`6-jupyter/notebooks/Network.ipynb`

## Showcase how to use a Different Dataset

- Use the Wine dataset: https://archive.ics.uci.edu/ml/datasets/wine
  → Download data file and adapt parameters for our `Data` class
  → OR implement a wrapper class for the `sklearn` implementation:

  https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_wine.html
- Show some statistics of the dataset
- Train a support vector machine and predict the classes
- Plot the confusion matrix between the classes

# Outline

2 Code Documentation
- API Documentation
- Hands-on: Document the `TwoLayerNetwork` class
- In-Code Documentation

# API Documentation

## Document Classes and Functions

- Document all **public** classes and functions
- Write some general usage instructions
- For classes, document constructor parameters and members
- For (member) functions, provide information for all parameters
  → Talk about expected data types, data ranges, and other expectations
- Also, describe in detail the returned data
  → Include data types, alignment properties and alike

## The Python Docstring

- Multiline comment starting and ending with `"""`
- Is placed below declaration of class or function

# API Documentation

## Exemplary Function Documentation

```
def apply_and_yield(callable, iterable):
  """This function applies the given ‘callable‘ onto the provided ‘iterable‘,
  and yields the results one by one.

  Parameters
  ----------
  callable : callable such as function, or class with overloaded ‘__call__‘ operator
    A function that takes one parameter and returns one value

  iterable : iterable such as ‘tuple‘, ‘list‘, or generator function
    The iterable should iterate over all of the data that should be transformed

  Yields
  ------
  element : any
    The result of the ‘callable‘ function on an element of the ‘iterable‘
  """
  for it in iterable:
    yield callable(it)
```

# Hands-on: Document the `TwoLayerNetwork` class

## Provide Documentation for `TwoLayerNetwork` (15 Minutes)

- Load the `7-docs/ifiss/algorithm.py` file
- Write a docstring for the `TwoLayerNetwork` class
  → Document the purpose of the class
  → Document the parameters and usage
- Write a docstring for all **public** functions
  → Provide information about the functionality
  → Document the parameters of all functions
  → Document the return values, if existent

# In-Code Documentation

## Code Comments are like Traffic Signs

- If none available, it is chaotic
- If there is too much, you get overwhelmed with information

## Can you Find your Way?

# In-Code Documentation

## Well-Written Code Documents Itself

- Use long and explanatory variable and function names
  → Write `input_dimension = 10` instead of `N = 10 # input dimension`
- Use clear and concise data structures
- Avoid complicated and nested list- or dictionary-comprehensions
  → Can you guess what this does (found in this Jupyter notebook):
  `cat(list(dict(list(zip(*result))[1]).values()))`
- Use fully-quoted functions and variables to increase legibility
  → Prefer: `import module; module.function()`
  over: `from module import function; function()`
  → Try to avoid: `from module import function as method; method()`
  → **Avoid at all costs**: `from module import *; function()`

# In-Code Documentation

## How to Write Good Documentation

- Write documentation such that **you** can understand it in 6 months
- Comment the **why**, not the *how* and not the *what*
- Do not document the obvious, e.g. avoid:

```
for k in values:  ...  # iterates over values using k
```

- Comments should be self-contained, short and concise
- Document your code while writing it

$\rightarrow$ If you can guess what 6-month-old code does, there is need to document it

$\rightarrow$ If you cannot guess, it is too late; consider rewriting the code plus comments!

## Documentation of the Data Class

`7-docs/ifiss/data.py`

# In-Code Documentation

## Hands-On: Commit your Changes

- Go through the `git` flow and commit your changes
- Push your commits to the `git` repository

# Extending your Code

## Comparison to the State of the Art

- Split your data into training and test set
  → Add a parameter `train=True` to your `Dataset` class
  → Or: Split your data into two files
  → Not done in this small example today, but **easy** now
- Implement wrappers around other algorithms
  → Assure to have same interface as your function
- Example: a wrapper around `sklearn.svm.SVC`
  → Can take various different parameters
- Adapt your script(s) to work with new algorithm
  → How to do that in a clever way: after the break