GESIS Fall Seminar in Computational Social Science 2024
Introduction to Machine Learning for Text Analysis with Python

Setup Guide

Welcome! In this guide, we will walk you through setting up a programming environment on your computer. The following steps will help you to configure the same environment that we will be working with during the course. However, you are free to deviate from this, if for example you already have a setup that works for you, or you simply prefer to experiment. Just make sure that, before the course begins, you can run python code, *inside of a jupyter notebook*. Reach out to us if you encounter issues!

Important: if you are using a work-managed laptop, you might run into permissions issues when performing this setup (as you, as the user, may not have the necessary administrative rights). Therefore, make sure that you attempt this setup with enough time to contact your ICT administrator should their assistance be needed.

For the following, you will need to access your system terminal to enter commands:
```
MacOS: press ⌘ + space and enter: terminal
Windows: press ctrl + r and enter: cmd
```

Step 1: Install Python.

We recommend installing **miniconda**: https://docs.anaconda.com/miniconda/#quick-command-line-install

Miniconda is a free, lightweight tool for managing python environments and packages. It allows you to create isolated environments for different projects, each with its own set of packages and dependencies. This helps avoid conflicts between projects and keeps your system organized.

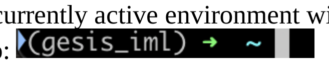
For **MacOS** (arm64), copy/paste the following commands into your terminal:
```
mkdir -p ~/miniconda3
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-arm64.sh
-o ~/miniconda3/miniconda.sh
bash ~/miniconda3/miniconda.sh -b -u -p ~/miniconda3
rm ~/miniconda3/miniconda.sh
```
Note: if you have an older (intel) mac, substitute the 'arm64' in the second line for 'x86_64'.

For Windows (x86) copy/paste the following commands into your terminal:
```
curl https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-
x86_64.exe -o miniconda.exe
start /wait "" .\miniconda.exe /S
del miniconda.exe
```

With miniconda installed, let's try creating an environment. In your terminal, run:
```
conda create -n gesis_iml python=3.10 ipykernel
```

With this, we're asking conda to create a new environment called 'gesis_iml' (or whatever you want to call it) and we're setting the python version to 3.10. By default, this will also install a list of commonly used packages. In addition, we also ask it to install the ipykernel package, which we need for working with jupyter notebooks.

Now that we have created an environment, we activate it by running:
```
conda activate gesis_iml
```

That's it! Remember that you must check that your desired environment is currently active **before** running python code or installing packages. The name of the currently active environment will always be shown in parentheses next to the current cursor position, like so: 
```
(gesis_iml) → ~
```

You can find more about creating and managing your environments here:
https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html

Step 2: Install GitHub Desktop and clone the course repository:

We recommend installing **GitHub desktop**:
https://docs.github.com/en/desktop/installing-and-authenticating-to-github-desktop/installing-github-desktop

This is the easiest way (for now) to access *git-versioned repositories*. Think of these as project folders that keep track of changes to files over time, including who made each change and when.

In this course, we only need to know how to 'clone' (take a copy of) a repository.

From within the app, select Add >> Clone Repository >> URL and clone the repository at 'https://github.com/annekroon/gesis-machine-learning.git' to a local folder of your choosing.

That's it! This folder will contain your copy of the course materials – lecture slides and exercises.

Step 3: Install an Integrated Development Environment (IDE).

Your IDE will be where you spend many long and hopefully *enjoyable* hours working on your code… so it is important to pick a good one! We recommend installing **VSCode**: https://code.visualstudio.com/download

Note: if you are prompted to choose between a 'user' and a 'system' level setup, choose user. Downstream tasks (like updates) will require less permission elevations, so there is less chance of encountering issues.

We recommend VSCode primarily as it is quite easy to get running ('out of the box'), but at the same time, provides plenty of room to grow. Many of its advanced features (which we will not need) can be hidden from view. You can also expand its functionality by installing 'extensions' as you need them. As we will now do:

Click on the extensions tab on the left-hand side of the interface, search for, and install:
Python – this extension allows VSCode to work seamlessly with your python environment(s).
Jupyter – this allows VSCode to run Jupyter notebooks (where we will perform exercises).

With these extensions installed, let's check that we can run python code inside of a jupyter notebook:

Create a new Jupyter Notebook via the *command palette* (a very useful menu for most tasks):
```
MacOS: press ⌘ + Shift + P and enter: Create: New Jupyter Notebook
Windows: press ctrl + shift + P and enter: Create: New Jupyter Notebook
```

In the empty cell, enter the following, and then run the cell (shift + return):
```
print('hello world!')
```

At this point, the command palette will ask you which python kernel you would like to use to run the code. You should see the python environment you created earlier with miniconda. Select it. The cell should execute, and underneath it, you should see the output (hello world!). Note that for each new file (like this notebook), you only need to select the kernel once – it will remember and use it in the future.

Optional: Co-pilot (AI assisted coding)

While not required for this course, we recommend learning to work with AI-assisted programming, as it will save you a good deal of time and frustration fixing errors with your code and can be an excellent learning tool.

For this, we recommend **Microsoft Copilot**, since it integrates into VSCode.

You will need a GitHub Pro account. This can be discounted (academic staff), or free (PhD).
https://education.github.com/discount_requests/application (approval can take a couple of days).

Once you have upgraded your account, follow these instructions to set up Copilot in VSCode:
https://code.visualstudio.com/docs/copilot/setup

There are two main ways to use Copilot: (1) **Chat** or (2) **Code completion**.
https://code.visualstudio.com/docs/copilot/getting-started

Chat can be accessed by clicking the button in the primary sidebar. You can ask the assistant questions without context, *or* with context by highlighting the part of your code that you are asking about, before asking your question. The latter is extremely useful: for example, you can highlight code from this course and ask it to explain the code to you step-by-step. Or, to troubleshoot code that isn't behaving as expected.

Code completion is switched on by default and works in two ways -

The first is **coding by comments**.

This involves describing what you want in natural language and allowing the assistant to predict the required code for you. For example, you could type:

```
#print the string: 'hello world!'
```

…and in the next line, the assistant will suggest something like this, which you can accept by pressing **tab**:

```
print('hello world!')
```

This useful for relatively simple operations, but the assistant can become overwhelmed quite easily and produce code that – whilst might run – may not behave the way that you expect or may be needlessly long or complex. Therefore, instead of asking it to perform a more complex operation all at once, try to break the operation into simpler steps, using comments to 'guide' the assistant through each in order. This will ensure the most success.

The second is **inline completion**.

This involves writing the beginning of your code and allowing the assistant to predict the rest of your code for you. For example, if you type:

```
print(‘hell
```

... the assistant will try to predict the remainder of your code, which you can accept by pressing **tab**:

```
print('hello world!')
```

Like coding by comments, the more wiggle-room the assistant has, the more likely it will get this prediction wrong, and produce code that does something that you don't want it to do. Therefore, it is best to leave a trail of 'breadcrumbs' by completing as much of the code you can yourself, until the suggestion is what you were after.

Importantly: Copilot is aware of your existing code, so it will often try to predict what you want to write based on previous lines (even before you start typing). This can be useful, but it can also be quite distracting at times. Therefore, you can **toggle on/off code completions** using the ***command palette***:

```
GitHub Copilot: Enable/Disable Copilot completions
```

Disclaimer: as with all commercial AI offerings, presume that your interactions with Copilot (and the contents of your code) are retained by Microsoft. This may influence your decision to use these tools.