Lesson 4 - Streamlit

Raoul Grouls & Jan-Willem Lankhaar

Table of contents

Introduction

Enabling a business user to interact with the data and the outcomes of a project is often very important. With `streamlit` you can easily create web apps that allow your client to interact with the data without having to deal with the details and complexities of the analysis.

Communication is key

Communication is a key part of data mining and exploration. In the business understanding phase, communication is essential to get a profound understanding of the client's problem and to define the objective and data mining goals of the project.

In later phases, it is essential to translate the outcome of the project to the client and the organization.

Communication is more than mere visualization

Good communication of the outcomes of an analysis often requires well thought out and balanced data visualization. However, how effective and attractive they may be, showing only charts and visuals in a slide deck is not always sufficient.

Often, clients needs to be able to interact with the data (e.g. slice, filter, zoom etc.) in a dashboard.

With packages like `streamlit` it is relatively easy to build a web app or dashboard that allow a user to interact with the data.

In many projects, this ability can make or break the adoption of the project outcomes.

What you will learn

- The basic principles of a web app
- How to configure your development environment to work with `streamlit`
- How to create a simple `streamlit` app
- The most important `streamlit` methods

Why `streamlit`?

There are many options for web app frameworks in Python. Rather simple (e.g. Dash for dashboards or Flet for more general purpose web apps) and more sophisticated/flexible frameworks (e.g. Flask or Django).

We have chosen `streamlit` because of its popularity and ease of use. It allows you to create a good looking web app with a few lines of code.

Installation

You can install `streamlit` with `poetry`:

```
$ poetry add streamlit
```

Important note!

`streamlit` is not compatible with Python 3.9.7. Even, if you don't have that exact version installed, the installation may fail because your `pyproject.toml` is too permissive for the Python version.

To prevent this, modify your `pyproject.toml` as follows:

```
[tool.poetry.dependencies]
python = ">=3.9,<3.9.7 || >3.9.7,<3.10"
```

This line allows all Python versions between 3.9 and 3.10 except 3.9.7. Alternatively, you could fix the Python version that `poetry` uses to a specific version (other than 3.9.7).

How a web app works

A web app works in a client server architecture, which is a bit different from a normal command line app.

The basic concept is as follows:

1. It starts with running a web server, which is an app that continuously listens to a port on your computer (or a server) and waits for a request from a web browser.
2. If you enter a URL in your webbrowser, the browser sends a request to the web server.
3. On receipt of a request, the web server processes the request and returns an HTML page in response.
4. The web browser receives the response, renders an HTML page and shows the output in your web browser window.

In full-fledged web apps (e.g. Django or Flask apps), steps 3 is the most difficult part. The web server hands over the request to your app and you have to make sure that your app processes it correctly and returns the proper response in all cases. In `streamlit` most of these complexities are abstracted away. So, you don't have to worry about all kinds of technical details.

You can start the `streamlit` web server by running the following command from the command line:

```
$ streamlit run yourapp.py
```

Note that instead of running your app directly with `python yourapp.py`, you run it via `streamlit`.

A minor complication when working remote

When you work remote (as we do on virtual machines), there is a minor complication. The web server that runs on your virtual machine cannot be reached directly from the outside world. We have to manually set up a connection from the ports of our local pc to the ports of the virtual machine.

Fortunately, Visual Studio Code provides functionality for this so-called port mapping (i.e. connecting a remote to a local port) and it only requires one simple additional step.

In the Ports panel, we have to configure a port forwarding. Once this has been done, we can submit requests in our local browser to the webserver on the (remote) virtual machine and we can receive responses.

Follow these steps to configure port forwarding in Visual Studio Code:

1. Select the **PORTS** tab in the lower panel of the screen

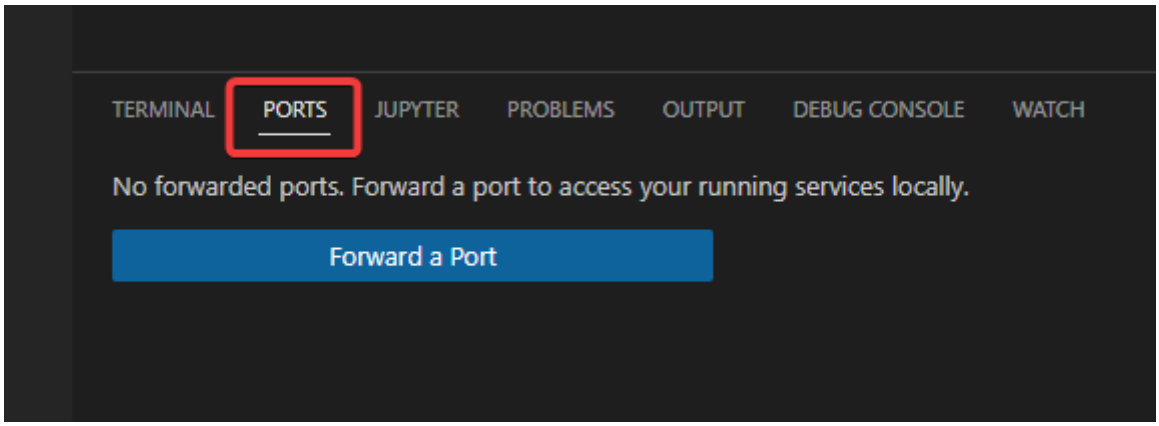

Figure 1: The **PORTS** tab in Visual Studio Code can be used to configure a port forwarding.

2. Click on **Forward a Port**
3. Enter the port number and press `<Enter>`. The port number is typically `8501` for `streamlit`, but it may be different in your case. You can find the port number the URL (after the `:`) that is shown when you run `streamlit`.

Starting with `streamlit`

How it works

If you want to turn your Python code into a `streamlit` app, all you have to do is add

```
import streamlit as st
```

to your code, call the proper `streamlit` "output" function in your code and run your code with `streamlit`.

Note that we abbreviate the `streamlit` with `st`, so that we can call the `streamlit` functions with the conveniently short prefix `st`.

The most basic output function is `st.write`. We will use `st.write` for our first 'hello world' `streamlit` app.

`streamlit` and Jupyter Notebooks don't go together

As you can imagine, Jupyter Notebooks and `streamlit` don't go together very well because of the client-server architecture. You have to write code in modules (separate `.py` files) and run them with `streamlit`.

Hello world in `streamlit`

We will start with a simple but not very usefull app that shows a static text.

```
import streamlit as st

st.write(
    r"""
    # Hello

    My name is **[put your name her]**.

    This is my first `streamlit` app for the Data Mining & Exploration course of the HU Un

    As you can see, the `st.write` function supports (GitHub flavored) Markdown.

    # Header 1

    ## Header 2

    ### Header 3

    My personal goals for this course are:
    1. Learn robust data analysis in Python
    2. Conduct my own analysis in a project from my own practice
    3. Have some fun!

    $ x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} $

    I'm a list:
    - Item 1
    - Item 2 with some _emphasis_
    - Goto [Hello](#hello)
    ---
    """
```

```
)
```

Save this to a separate module `src/hello.py` and run the code from the command line in your shell:

```
$ streamlit run src/hello.py
```

You will see that the webserver has started and that it shows the address you can visit.

If you open the link in your browser (after having correctly configured port forwardig as shown above), you will see the output of your module.

As you can see, the `st.write` function is simple in use, but because it supports Markdown you can already create a good looking page with formatting, headers, lists, hyperlinks and mathematical equations. Visit this page if you want to learn more about Markdown.

Modifying your code

If you want to change your code, you don't have to stop the web server. All you have to do is save your module. The webserver will detect the changes and will ask you (in the right upper corner of your web page) whether you want to rerun the code.

If you click **Always rerun** it will automatically refresh your page each time you save your file.

Note that if not otherwise specified, `streamlit` will always run the entire module when you save it. This may involve long-during analyses. In these cases, it is wise to have the page not refreshed automatically.

Adding plots

We will now move on to a more realistic example with a plot. Create a new module `dashboard.py` for the code below and import the following:

```
from datetime import datetime

import matplotlib as mpl
import palmerpenguins
import seaborn as sns
import streamlit as st
```

Load the penguins data into a `pandas` dataframe.

```
penguins = palmerpenguins.load_penguins()
```

It is important to inform the user of a dashboard about how current the information on the dashboard is. Therefore, use `st.write` and the appropriate `datetime` function to add the folllowing text with the current date in it:

```
Last updated: <dd MMM yyyy>
```

We want to enable the user to investigate the correlation between the bill depth and bill length of each penguin species. So, let's add a plot that shows exactly that.

Add the following to your code:

```
# Create a (scatter) plot object.
# f, ax = mpl.subplots(1,1)
g = sns.relplot(
    ax=ax,
    data=penguins,
    x='bill_length_mm',
    y='bill_depth_mm',
    hue='species'
)
g.set_axis_labels('Bill length (mm)', 'Bill depth (mm)')
g.set(title='Bill length and depth correlation across species')

# Add the object to the page.
st.pyplot(g)
```

Note that style and details matter on a dashboard. We add a title, use legible axis labels and add units to quantities.

Adding interactivity

We want to add interactivity to our dashboar. As a first step, we will add check boxes that allow us to select/unselect a species shown in the plot.

In order to be able to do so, we have to prepare the data. We need the following:

- A list `species` with the names of the species.
- A list of booleans `is_checked` (the same length as `species`) that indicates which species are selected.
- A dataframe `penguins_selection` that includes only data for the selected species.

Complete the following code with the appropriate `pandas` functions:

```
species = penguins['species']...
is_selected = []        # Will be initialized below.

# Put your checkbox code (see below) here.

penguins_selection = penguins[ ... ]
        # Hint: combine .isin() with species and is_selected.

# Add your plot to the page here.
```

In `streamlit`, checkboxes are created with `st.checkbox`.

Since the number of species is limited, we will position the checkboxes horizontally. Each checkbox will get its own column. Columns are created with `st.columns`. It returns a context manager that you can use to position objects:

```
with column_object:
    ...
    # create objects that are placed in this column
    ...
```

We create a checkbox and a column for each species and combine them in pairs with `zip()`:

```
columns = st.columns(spec=len(species), gap='small')
for col, spec in zip(columns, species):
    with col:
        is_selected.append(st.checkbox(spec, key=f'{spec}_checkbox', value=True))
```

Make sure to place the dataframe filtering and the checkbox code above your `st.pyplot(...)`. The `key` argument comes in handy later.

If all went well, you are now able to select and unselect the species shown in the plot and the plot will change accordingly.

We also want to show the exact data, stored in our data frame. `streamlit` provides an easy way to add an interactive table to the page with the `st.dataframe()` function:

```
st.dataframe(penguins_selection)
```

We want to hide details in the table and have neat and legible column titles. Use `pandas` to rename the columns properly and modify your code to have it only show the columns `species`, `bill_length_mm` and `bill_depth_mm`.

Rearranging the plot and the table

Not every business user is interested in the table and not every user is interested in the plot. To rearrange them, we will put the plot and the data table on their own tab. However, we want the check boxes to appear above both, because they apply to both.

```
tabs = st.tabs(['Plot', 'Table'])
```

This will create a list of tab context managers that we can use to position our objects in. Alternatively, we could use tuple unpacking and write:

```
plot_tab, table_tab = st.tabs(['Plot', 'Table'])
```

```
with tab[0]:    # Alternatively: with plot_tab
    ...
    # Place objects on tab0.
    ...

with tab[1]:    # or: table_tab
    ...
    # Place objects on tab1.
```

Modify your code so that it:

- Always shows the checkboxes
- Shows the scatter plot on the `Plot` tab
- Shows the data frame on the `Table` tab
- Shows a table caption above the table on the `Table` tab (use the `st.caption` function)

Buttons

Besides checkboxes and tabs, `streamlit` has a lot of other controls (so-called *widgets*) for user interaction. We will add a reset button that resets our species selection to the default selection (i.e. all species selected).

We will give the button the label **Reset selection** and when it is clicked by the user, the selection should be reset. We can achieve this by defining a callback function and attach it

to the button. Callback functions should be defined *before* they can be used, so we will first define the function and then add the button:

```
def reset_selection():
    """Reset the species selection checkboxes to their default value (i.e. True)."""
    for spec in species:
        key = f'{spec}_checkbox'        # Corresponds to the `key` argument used above.
        st.session_state[key] = True
```

Once, we have this function defined, we can add a button to the page (with `st.button`) and attach the callback function to it.

A few comments on this function. The function uses the `session_state`, a dictionary that holds the state of the variables of your app on the current page. You can use the `session_state` to look up variable values and modify them.

Also note that this function accesses a variable outside of its own scope (`species`), which is often not a desirable feature of a function but in this case it is defendable.

```
st.button('Reset selection', on_click=reset_selection)
```

This concludes this part of the lesson. You have now a simple dashboard. You may want extend it with other plots and controls.

A more interactive map app

In many projects, geographical data plays an important role (e.g. sales per regions). In this second part of the lesson, we walk through an app that shows some data on the map. On the fly, you will learn how to use some additional `streamlit` widgets.

Note that it is more a demonstration than an actual lesson.

Working with geographical data is an art in itself. In this course, we won't dive into that. Fortunately, there are many packages and tools that make working with geographical data easy. `streamlit` also provides some basic mapping functionality out of the box.

What the app does

The app loads hourly temperature observation history from the KNMI weather stations for a given number of days. The location of the weather stations is shown on a map.

If the user selects one or more weather stations (in a multiselect box), the temperature of the selected stations is plotted in a chart.

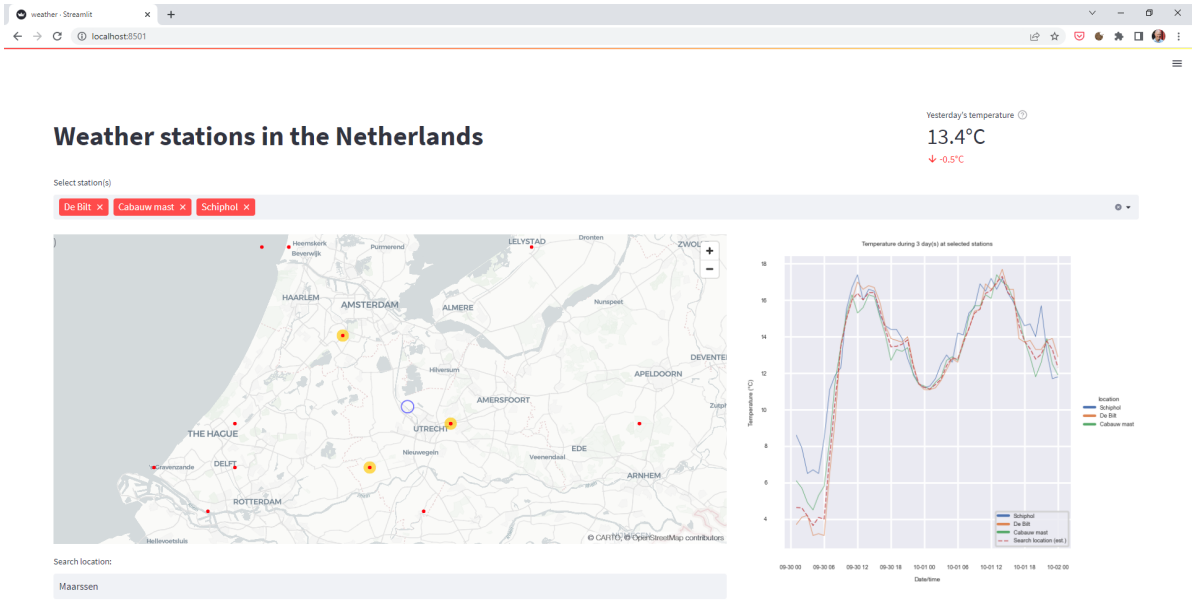

Figure 2: The weather app.

Below the map, the user may enter a search query for a location. Using a dedicated search engine (`Open Streetmap`'s Nominatim), the location is translated to coordinates and is plotted on the map.

In addition, the weighted average temperature (progression over time) of the weather stations nearest to the searched location is shown in the chart.

In the upper right corner of the dashboard, a `metric` widget shows yesterday's average temperature (averaged across all weather stations) and its change with respect to the day before.

Structure of the app

The app is defined in the following modules:

- `weather.py` The main module, that should be run with `streamlit`.
- `knmi.py` Module that defines some functions to load and average the KNMI temperature data.
- `settings.py` Settings for the app.