

STREAMLIT TIPS AND TRICKS

10 Features Your Streamlit ML App Can't Do Without — Implemented

Add Jupyter lab, session management, multi-page, files explorer, conda envs, parallel processing, and deployment to Streamlit app



Dor Meir

Published in Towards Data Science

13 min read · Dec 29, 2021



*

M

uch has been written about [Streamlit killer data apps](#), and it is no surprise to see Streamlit is the fastest growing platform in this field:

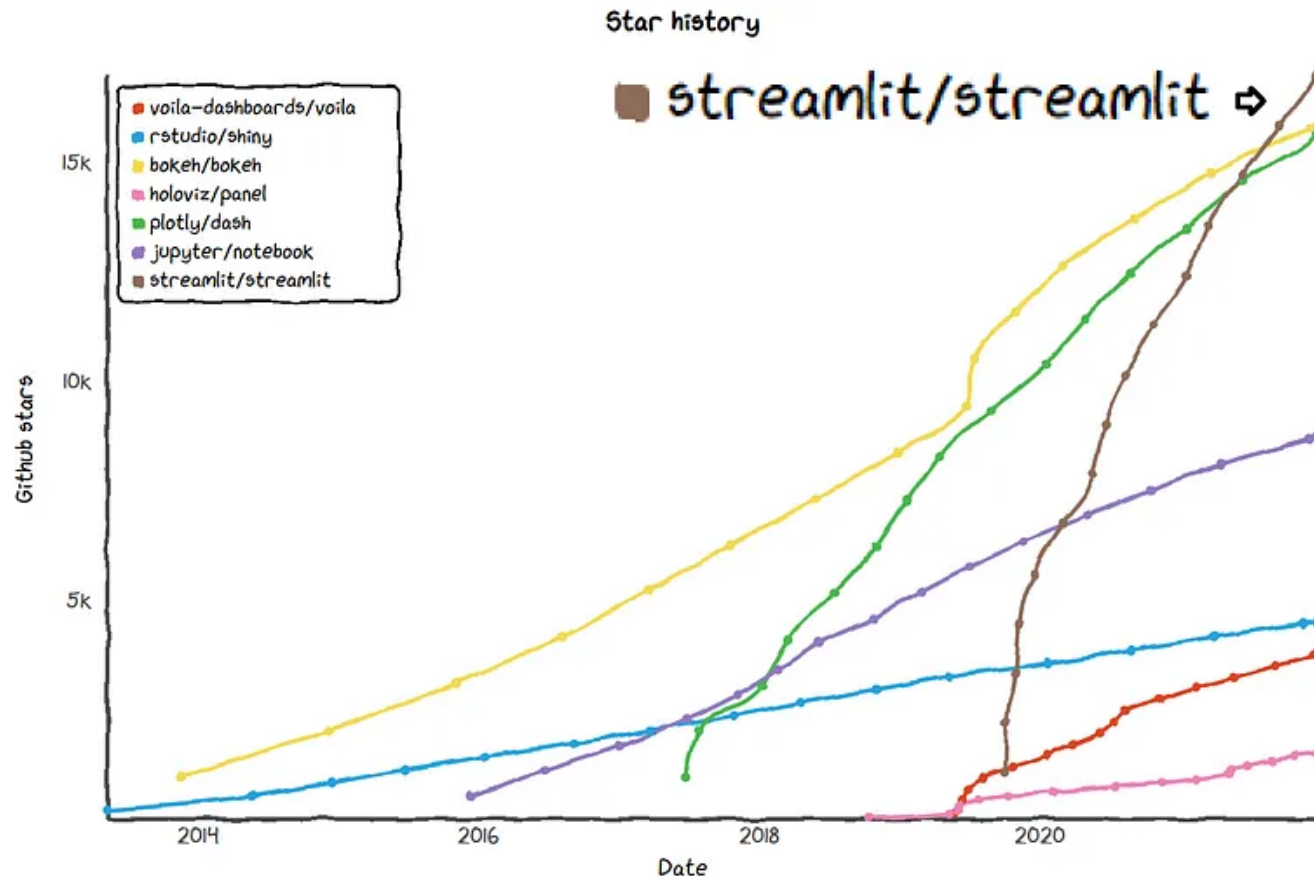


Image by [Star history](#), edited by author

However, developing an object segmentation app with the CRISP-DM phases in Streamlit made me realize many basic features are missing from the platform. These features are rarely mentioned in Streamlit articles, probably since they do “dirty work” in the background and are, simply put, not that glamorous. Even though they are not discussed, I’m confident that without those features your data app won’t cut it. In my case, some of the features were brought up as requirements by the app’s users, while others are built in the R-shiny platform, where I joined developing another AutoML app.

Many basic app features are missing from Streamlit.
Without them, your app won’t cut it.

So why am I writing this article? Setting aside the missing features, Streamlit is an awesome platform (see [here](#)), maybe one of its kind for developing apps in *Python* with almost no experience in apps development. Thus, I did everything I can to scavage those missing features online, googling dozens of solutions and trying to implement them. Furthermore, when there was no good solution online, I developed features on my own. And if I’ve already spent the time on gathering those features — you shouldn’t. You’re welcome to sit back, relax, and enjoy the ride...

I’ve already spent the time gathering those features—
so you shouldn’t. You’re welcome to sit back, relax,
and enjoy the ride...

Before diving into the code, let me stress most of the features here are dependent on each other. Don’t worry — they’re mentioned in order of dependence, so each former feature is independent of the a latter one. In addition, it’s worth mentioning I’m a Data Scientist in my core, and not an app developer. Therefore, I might not use the proper jargon for concepts in the app development domain. Nevertheless, I excuse myself from being precise since I believe most of you come from a similar background to mine, meaning you’re proficient in Python and in building Machine Learning models, yet you have little to no knowledge in developing

apps. If, however, you ARE an experienced developer, I hereby apologize for desecrating your profession :)

So, without further ado, here are the 10 features:

1. Preserving work on refresh and lost connection
2. Saving work for later use
3. Files explorer
4. Login screen
5. Multi-page App
6. Running long parallel calculations
7. Using models of different environments
8. Dynamically plotting training results
9. Embedding Jupyter and Serving a license file on new tab
10. Deploying a trained model

. . .

1. Preserving work on refresh and lost connection

Problem: The Streamlit user loses her work on refresh or connection timeouts.

Solution: Auto-save and auto-load the user's session with a session file.

Data Labelling

This is one for all machine learning fans: Label some images and all of your annotations are preserved in `st.session_state` !



Annotated: 2 – Remaining: 20

This is a dog! 🐶

This is a cat! 🐱

Annotations

```
{  
  "cat.1.jpg" : "dog"  
  "dog.5.jpg" : "dog"  
}
```

The app “remembers” user choices, Image by [Streamlit v0.84 Demos](#),

One of the most important things in any app is a session management capability, i.e., the app must “remember” any setting the user have configured whenever the user switches pages, refreshes the web browser and, of course, saves the model and loads it later on. If you know how a ML model class looks like, a saved session is basically the same concept only on a broader context. A session will not only save the hyper parameters and trained model weights but also the entire dataset (or its file path), the predictions, the file paths of the training log, the plots, the scripts the user has written, and any other work the user has made in his session. This means that without keeping her session persistent, the user will lose all her work whenever she refreshes the browser or just temporarily loses connection to the app.

Without keeping her session persistent the user will lose all her work whenever she refreshes the browser or just temporarily loses connection to the app.

Even though this feature was constantly asked for in the Streamlit forum right from the early days of the platform, session management was introduced to the platform only two years after. Still and all, **the official Streamlit solution has a major downside: it only preserves the user's work when she switches between pages, whilst it doesn't preserve a user's work if a session ends expectedly**. My solution does, and it's using a session management solution of the user okld instead.

First, the app's script should have these lines:

Second, this functionality only works when Streamlit is started in debug mode and writes the debug messages to a log file:

```
streamlit run app.py --logger.level=debug log_file.log 2>&1
```

The log file should be stored in the folder `configs['APP_BASE_DIR'] + '/logs/streamlit_logs/'` as it is used in the `was_session_shutdown()` function.

Third, this is how you auto-save a session:

```
autosave_session(state)
```

When should we auto-save and auto-load? I call the autosave function whenever the user changes a configuration or even just changes a page in the app — that way when a session reload occurs, the user can immediately continue his work without having to browse back to the page he was working on. The loading part should be one of the first commands to be called in the app's script, since an app refresh re-runs the entire script from the beginning, and we want the user's downtime to be minimal.

When will a session NOT be auto-loaded? when a user has logged out (see [Multi-pages section below](#)), or when the Streamlit app restarted.

Is it possible to add the auto-save feature to the official Streamlit session solution? it very well might, do tell me if you succeed on doing that!

2. Saving work for later use

Problem: Streamlit has no option to save a user's work for later use.

Solution: Save, load & delete buttons

	Model	Last modified	Size in MB
0	awesome_model.pickle	2021-12-14T17:02:28.394725	0.018
1	~session_auto_save.pickle	2021-12-14T17:01:28.551557	0.018

Select Model to save/load/delete

<save_new_model>

New Model's name

awesome_model.pickle

Saved awesome_model.pickle

Save

Saving a model, image by author

	Model	Last modified	Size in MB
0	awesome_model.pickle	2021-12-14T17:02:28.394725	0.018
1	~session_auto_save.pickle	2021-12-14T17:03:21.752634	0.018

Select Model to save/load/delete

awesome_model.pickle

Load

Delete

Loading/deleting a chosen model, image by author

Letting the user save and load his work has the same exact logic as in the [section above](#), only this time save and load are triggered when the user is pressing action buttons. It goes without saying you'll also have to present a table showing all saved

models, as well as some kind of indication of whether a model was successfully saved/loaded/deleted.

3. Files explorer

Problem: The Streamlit user can't create, delete, or browse folders

Solution: A recursive function for folder manipulation

Models' folder: level 1

<create folder>

New folder name

classic_models

Create new folder

Show saved_models folder tree

	Model	Last modified	Size in MB
0	awesome_model.pickle	2021-12-14T17:02:28.394725	0.018
1	BERT_models	2021-12-14T17:29:37.496445	0.004
2	classic_models	2021-12-14T17:45:11.322368	0.000
3	~session_auto_save.pickle	2021-12-14T17:44:10.154027	0.018

Creating a folder, image by author

Models' folder: level 1

classic_models

Models' folder: level 2

logistic_models

Models' folder: level 3

<current folder>

Show logistic_models folder tree

Model	Last modified	Size in MB
empty		

Browsing sub-folders, image of author

Models' folder: level 1

<delete folder>

Folder to delete

BERT_models

BERT_models is not empty. Are you sure you want to delete it?

Show BERT_models folder tree

├── first_BERT_model.pickle

└── second_BERT_model.pickle

Yes

Deleting a folder, image by author

A files explorer is an absolute necessity for any CRISP-DM app as it is used for saving files and organizing them in directories in almost all of the work's phases. For instance, you might decide to have a folder for each user's trained models and a folder for the datasets of each specific project. Even though it's a very basic requirement, if you google this issue you'll probably find some not-so-functionating [Streamlit users' solutions](#).

This solution enables the user to create, delete, and browse folders, and see the folders' tree structure. **Most importantly, it'll work even if your Streamlit app is running on a remote server where the user has no direct access to the file system.**

4. Login screen

Problem: Streamlit has no built-in user authentication screen

Solution: If statement before `main()` which verifies the user and password

Awesome App

broguht to you by great company

User

Password



Login screen, image by author

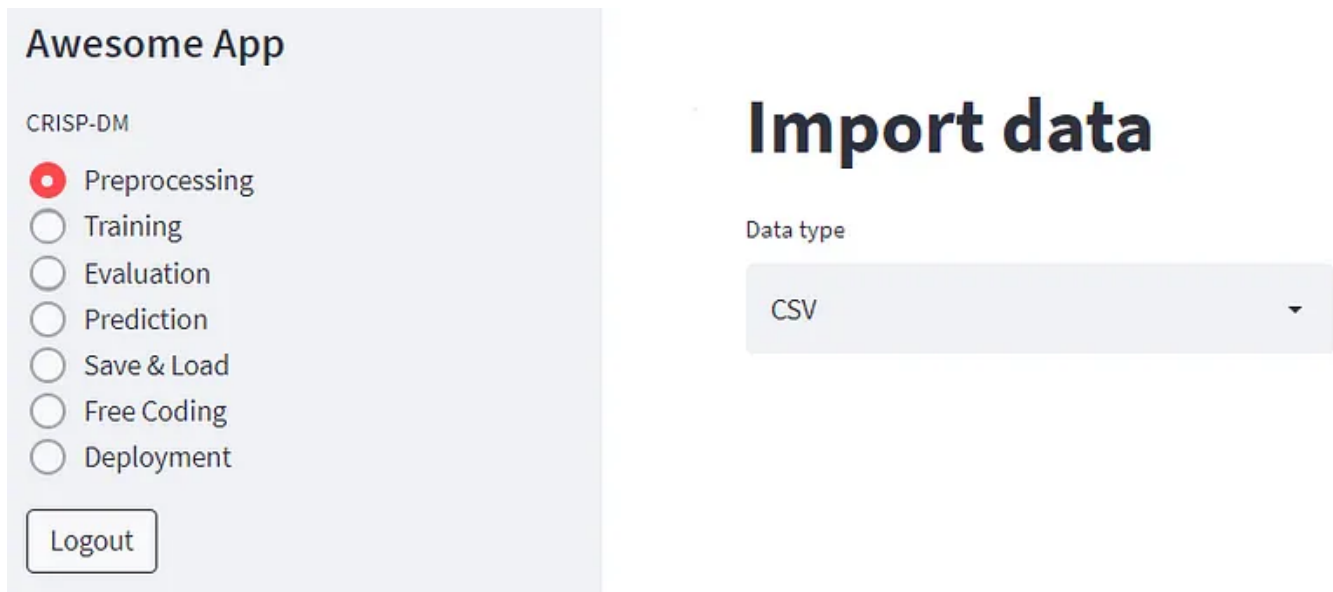
Any professional app needs a “bouncer” guarding the entrance. Streamlit doesn’t have one, even though this was also asked for by Streamlit users a long time ago.

Bear in mind that my implementation uses hard-coded user and password but adding LDAP or MongoDB authentication should be quite easy.

5. Multi-page App

Problem: Streamlit has no built-in multi-page capability

Solution: Add radio menu in the sidebar linked to functions for each page



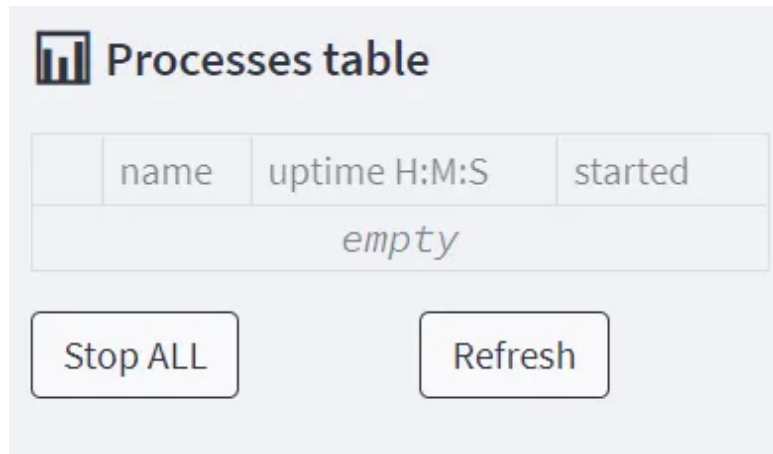
Multi-page app, image by author

Working with CRISP-DM requires moving back and forth between the different phases. This work process is best represented in the app in the form of ordered options of a menu, each option linked to a different function. The concept of a separate function for each phase also makes perfect sense when you code those phases.

6. Running long parallel calculations

Problem: Streamlit can't run two operations at once and stop them in the middle

Solution: Run sub-processes and create a process control table in the sidebar



Processes table, image by author

Whether you're brute forcing your way in feature engineering or training heavy deep learning models, you'll need a way to run things in parallel and, in some cases, kill them prematurely. One way to add this functionality is to run each calculation as a sub-process, while having a processes table that is both dynamically updating and has action buttons to kill each process separately or shut them all at once. **Running independent sub-processes not only let's you use parallel computing but also ensures that even if the Streamlit session is suddenly killed, the sub-processes will continue running without interruption on the server.** So, whatever happens to the Streamlit session, the user can login again and see the results. Another approach to this solution would be to send these sub-processes to work on a cloud service instead of on your own machine - this will probably be of value to you if you need an extra computing power.

Running a sub-process is rather easy with *Popen*:

```
try:
    subprocess.Popen(command)
except:
    st.write(f"Couldn't run: {command}")
    e = sys.exc_info()
    st.error(e)
updated_processes_table(process_table_sidebar, state)
# see updated_processes_table() implementation on the gist below
```

It gets a bit trickier if the command you're running asks the user for input, but it's also manageable with *PIPE*:

```
process = subprocess.Popen(command, stdin=subprocess.PIPE,  
stdout=subprocess.PIPE,  
                             universal_newlines=True)  
try:  
    process.communicate("\n".join(user_arguments), timeout=0.2)  
except:  
    pass  
updated_processes_table(process_table_sidebar, state)
```

And here's how you can implement the processes control table:

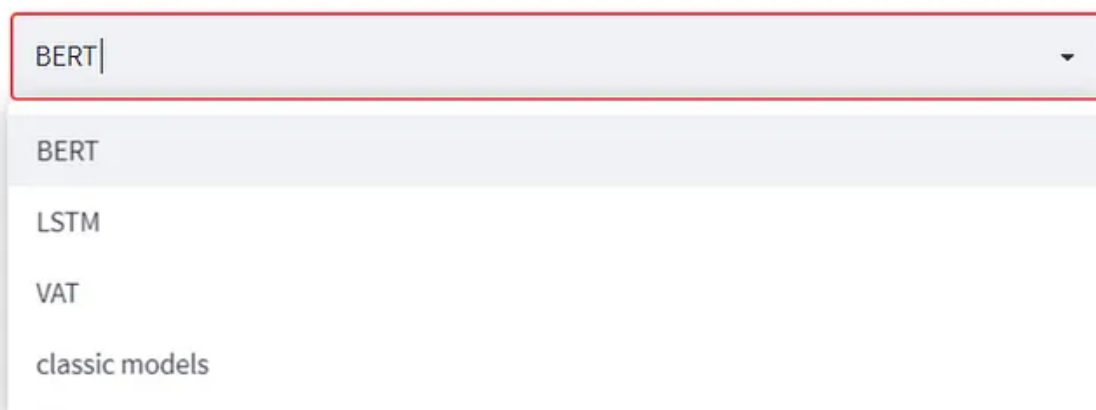
7. Using models of different environments

Problem: Streamlit has no option for using models on different environments

Solution: Activate Conda environments when running sub-processes

Choose model

switch conda environment



The image shows a UI element for selecting a model. It consists of a dropdown menu with the label 'switch conda environment'. The dropdown is currently open, displaying a list of options: 'BERT', 'LSTM', 'VAT', and 'classic models'. The 'BERT' option is highlighted, indicating it is the selected model.

UI for switching python environments, image by author

When working with heavy CV and NLP models you often need to install different coding environments for each model, since many-a-time they rely on specific *Tensorflow* or *Pytorch* versions which aren't compatible with other installations. In that case, you'll want to call each model command from its own

environment, and you should also capture all of the environments outputs to a log (the variable *output_to_log* in the gist below). Of course, *Conda* is just one multi-env solution, one might also consider *Pyenv* or even use *Docker* images instead.

Now that we know how to run sub-processes (see [section above](#)), it's quite straightforward to run them on different conda environments:

8. Dynamically plotting training results

Problem: Streamlit can't show training outputs during training

Solution: Tail models logs and plot them on intervals

- ☒ Logs
- ☐ Graphs

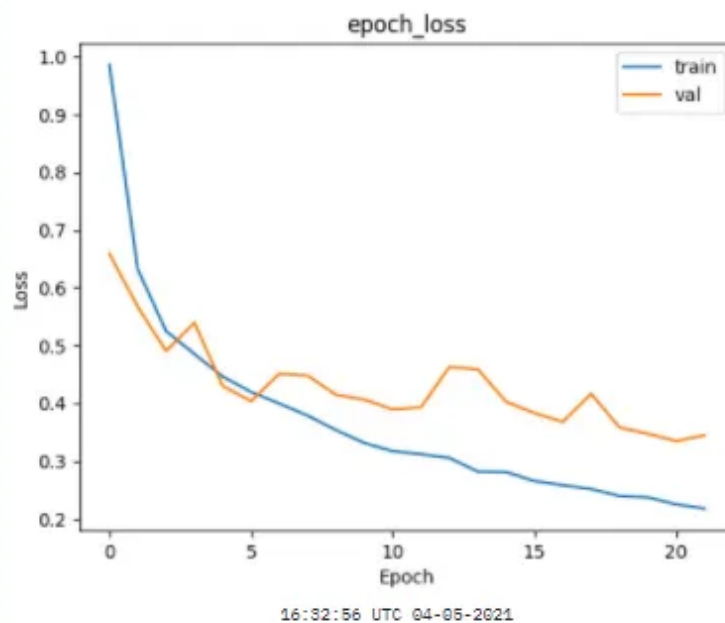
Logs

```
2021-08-11 18:37:35.564541: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1746] Addi
```

Dynamic log output of a training model , image by author

- ☐ Logs
- ☒ Graphs

Graphs

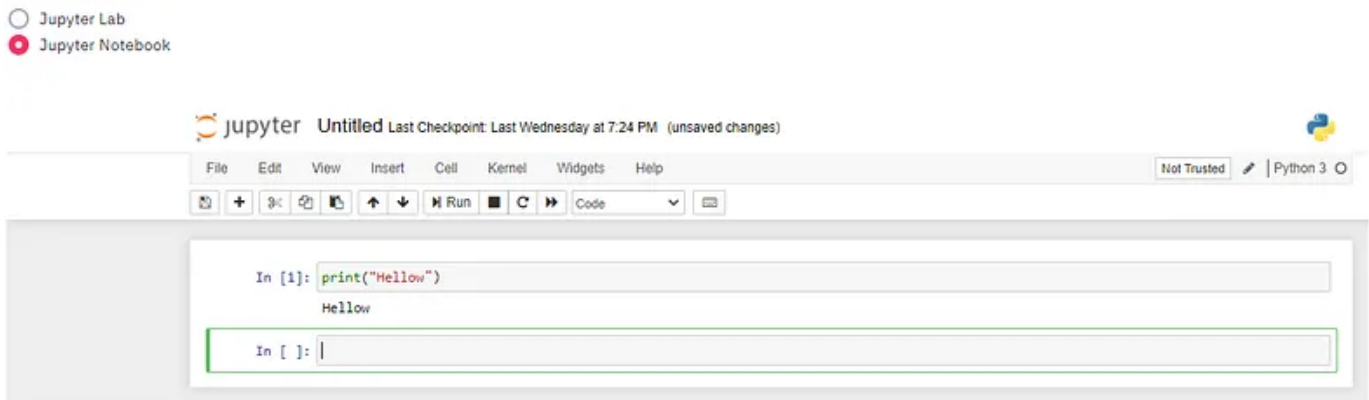


Are you underfitting, overfitting or just-fitting? When dealing with deep learning models (as in the [section above](#)) it's especially hard to know this without plotting the models' metrics during training. It might also be useful to see the textual output of the commands you're running, and to have an option to download it to a file. finally, it might be convenient to set the interval for refreshing the metrics plots, since they usually update on the finish of one training epoch and the duration of an epoch varies for different models and datasets.

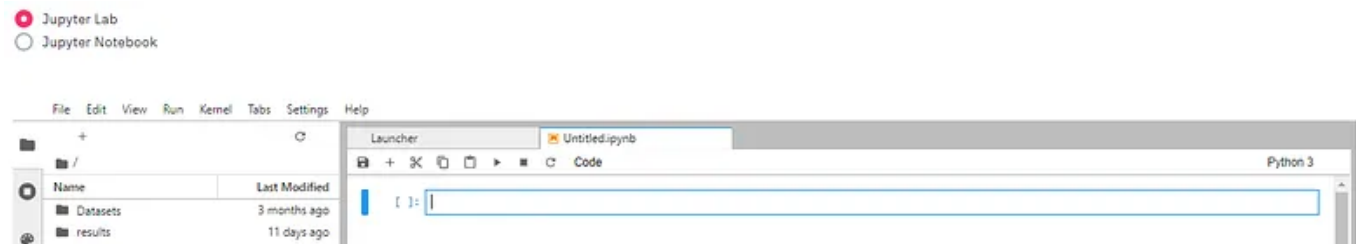
9. Embedding Jupyter & Serving license on a new tab

Problem: Streamlit has no free coding *and* can't redirect to a file on a new tab

Solution: Embed Jupyter as HTML iframe *and* add a Jupyter link to the file



Free coding in Jupyter notebook, image by author



Free coding and files exploring in Jupyter lab, image by author

Embedding a Jupyter notebook/lab instance on a page of the app is one of the hardest challenges I faced. There are multiple theoretical ways and tools for doing that but from my experience, “stars need to align” for the configurations to work. In the end I succeeded embedding Jupyter to Streamlit only with a rather promiscuous security configuration. If that’s a concern for you, start with my configurations and try adding restrictions while still having everything work.

Jupyter was tough implementing, but worth the effort. It’s a very powerful tool as it lets the knowledgeable user do basically anything he wants with

the models already trained, and even add some phases which are not yet implemented like post-processing, explainability, etc. But from the exact same reason, **it is also a dangerous tool which gives the user a terminal access to the server**, i.e., a capability to destroy it with a command — so use with caution!

Jupyter configuration

First, you should create a jupyter configuration file, with:

```
jupyter notebook --generate-config
```

Next, edit the file so that it'll have these configurations:

```
c.NotebookApp.allow_origin = '*'
c.NotebookApp.disable_check_xsrf = True
c.NotebookApp.ip = '*'
c.NotebookApp.notebook_dir = <directory of datasets and outputs>
c.NotebookApp.open_browser = False
c.NotebookApp.port = 8502
c.NotebookApp.token = ''
c.NotebookApp.tornado_settings = {'headers': {
    'Content-Security-Policy': "frame-ancestors http://<Server-IP>:8501/ "
}}
c.NotebookApp.use_redirect_file = True
```

The most important configuration is *c.NotebookApp.tornado_settings*, which tells Jupyter to trust your Streamlit app when presenting an iframe of Jupyter. Also note that you should set *c.NotebookApp.port* to a number that is not already occupied by the Streamlit port. The default Streamlit port number is 8501, and any new Streamlit session will be opened with the next free port, i.e. 8502, 8503, and so on.

Streamlit code

Starting Jupyter is straightforward, all you should do is add these lines to the beginning of your app:

```
# in if __name__ == "__main__":
command = rf"""jupyter lab --allow-root"""
```

```
try:
    subprocess.Popen(command, stdout=subprocess.PIPE, shell=True)
except:
    st.error(f"Couldn't run {script_type}:")
    e = sys.exc_info()
    st.error(e)
```

But embedding Jupyter to Streamlit is a different and longer story:

Serving a license file on new tab using Jupyter

Awesome App

brought to you by great company

[license](#)



The license link leads to a license file in a new tab, image by author

If you wish the users of your app's to be redirected to a new separate tab when opening the LICENCE file, you need, in other words, some web app to serve your LICNESE file. Lucky for you, Jupyter can do this exactly, as it's built upon a Torando web server.

```
with col9:
    linespace_generator(2)
    while not is_jupyter_up():
        time.sleep(0)
    link = f"[license] (http://{state.jupyter_ip}:
{state.jupyter_port}/files/Licence.html) "
    st.markdown(link, unsafe_allow_html=True)
```

10. Deploying a trained model

Problem: Streamlit has no option for delivering a trained model

Solution: Create a REST API flask web service

```
* Serving Flask app "api" (lazy loading)
* Environment: production
WARNING: This is a development server. Do not use it in a production deployment.
Use a production WSGI server instead.
2021-12-24 16:14:39.765 INFO werkzeug: * Running on http://0.0.0.0:8503/ (Press CTRL+C to quit)
2021-12-24 16:17:33.605 INFO werkzeug: 127.0.0.1 - - [24/Dec/2021 16:17:33] "[37mGET /models HTTP/1.0m" 200 -
2021-12-24 16:17:38.789 INFO werkzeug: 127.0.0.1 - - [24/Dec/2021 16:17:38] "[37mPOST /models HTTP/1.0m" 200 -
2021-12-24 16:17:45.638 INFO werkzeug: 127.0.0.1 - - [24/Dec/2021 16:17:45] "[37mPOST /models HTTP/1.0m" 200 -
2021-12-24 16:17:55.724 INFO werkzeug: 127.0.0.1 - - [24/Dec/2021 16:17:55] "[37mGET /configs HTTP/1.0m" 200 -
(streamlit_conda_env) C:\AwesomeApp\FastFCN\FastFCN-master>conda.bat activate FastFCN
Working on GPU
Loading Model C:\AwesomeApp\FastFCN\FastFCN-master\logs\coco\FastFCN_coco.h5
0%|          | 0/2 [00:00<?, ?it/s]
50%|#####   | 1/2 [00:00<00:00, 1.53it/s]
100%|#####  | 2/2 [00:00<00:00, 2.27it/s]
100%|#####  | 2/2 [00:00<00:00, 2.12it/s]
Prediction finished
2021-12-24 16:21:12.291 INFO werkzeug: 127.0.0.1 - - [24/Dec/2021 16:21:12] "[37mPOST /predict HTTP/1.0m" 200 -
```


Building a model is easy but serving the model so that people can use it directly and on a daily basis — that's a totally different challenge. **The solution I'm presenting here has nothing to do with Streamlit, nonetheless it's an essential part of any CRISP-DM app and so I chose to include it in the article.** My solution is using Flask, which is a rather straightforward platform that will probably suffice for your POC needs. Still, when things get serious and you need to manage multiple sessions, security and more, you might want to switch to Django or Node.js.

My Image segmentation API have several independent paths:

www.AwsomeApp.com:8503

/models: Query trained models

/configs: Set prediction arguments

/base64: Convert image files formats to base64 strings

/predict: Feed model with data and get predictions

Here's is the entire API implementation, along with instructions on how to use it (right after the gist):

Query trained models

- Query which algorithms are available for prediction:

```
import requests
IP = "<IP address>"
res = requests.get(f"http://{IP}:8503/models"); res.json()
```

- Query which trained weights are available for a specific model:

```
params = {"algorithm": "FastFCN", "Model_path": "coco",
"return_weights": "True"}
res = requests.post(f"http://{IP}:8503/models", data=params);
res.json()
```

- Query which classes a specific model was trained on:

```
params = {"algorithm": "FastFCN", "Model_path": "coco",
"return_classes": "True"}
res = requests.post(f"http://{IP}:8503/models", data=params);
res.json()
```

Set prediction configuration

First, create a json file containing the prediction configurations. Those configurations will be used for all subsequent predictions until a new configuration file is sent to the API.

Example of *configs.json* file:

```
{'algorithm': 'FastFCN',
'Model_path': 'coco',
'weight': 'FastFCN_coco1',
'IMAGE_MIN_DIM': '1024',
'IMAGE_MAX_DIM': '2048',
'DETECTION_MIN_CONFIDENCE': '0.7',
'mode': '2',
'class_show': 'None'}
```

Second, Send the file and evaluate the current configurations:

```
IP = "<IP>"
files = {'file': open('configs.json', 'rb')};
json.load(files['file'])
files = {'file': open('configs.json', 'rb')}
res = requests.post(f"http://{IP}:8503/configs", files=files)
res = requests.get(f"http://{IP}:8503/configs"); res.json()
```

Convert images to base64

Optional: Preprocess a zip of images to a list of base64 strings, to be later fed into the model for prediction:

```
files = {'file': open('image_to_predict.zip', 'rb')};
res = requests.post(f"http://{IP}:8503/base64", files=files);
list_of_base64=res.json()
params['images'] = list_of_base64
```

Make predictions

First, define a dictionary called *params* with an *images* key of which its value is a list of base64 strings representing the images for prediction:

```
params = {'images': [<image0_base64>, <image1_base64>, <image2_base64>]}
```

Set this config to True to get the results in the form of image files instead of base64 strings:

```
params['create_images'] = True
```

Second, make a prediction request, and evaluate results:

```
IP = "<IP>"
res = requests.post(f"http://{IP}:8503/predict", data=params)
res = res.json()['Responses']
predicted_image0 = res[0][1]
json_result0 = res[0][0]
print(f"Classes found: {json_result0['results']['class_name']}")
print(f"Prediction scores: {json_result0['results']['scores']}")
print(f"Bounding Boxes: {json_result0['results']['rois [y_min, x_min, y_max, x_max]']})")
```

Summary

Developing an ML app with Streamlit is like taking a ride on the wildest roller-coaster of the amusement park: it's super quick, extremely exciting, dizzying at times but certainly a lot of fun. Yet after a speedy implementation in Streamlit of your brilliant Machine Learning ideas, it's almost inevitable your app will not really be usable. That's a result of Streamlit lacking some basic features: [Preserving work on refresh and lost connection](#), [Saving work for later use](#), [Files explorer](#), [Login screen](#), [Multi-page App](#), [Running long parallel calculations](#), [Using models of different environments](#), [Dynamically plotting training results](#), [Embedding Jupyter and Serving a license file on new tab](#), [Deploying a trained model](#).

You could try to implement those features yourself or even “roll the dice” on google, but why waste the time? All those features' codes are above - so sit back, relax, and enjoy the ride :)

. . .

Feel free to share your feedback and contact me on [LinkedIn](#).

Thank you for reading, and good luck! 🍀

Streamlit

Implementation

Crisp Dm

Python

Jupyter

