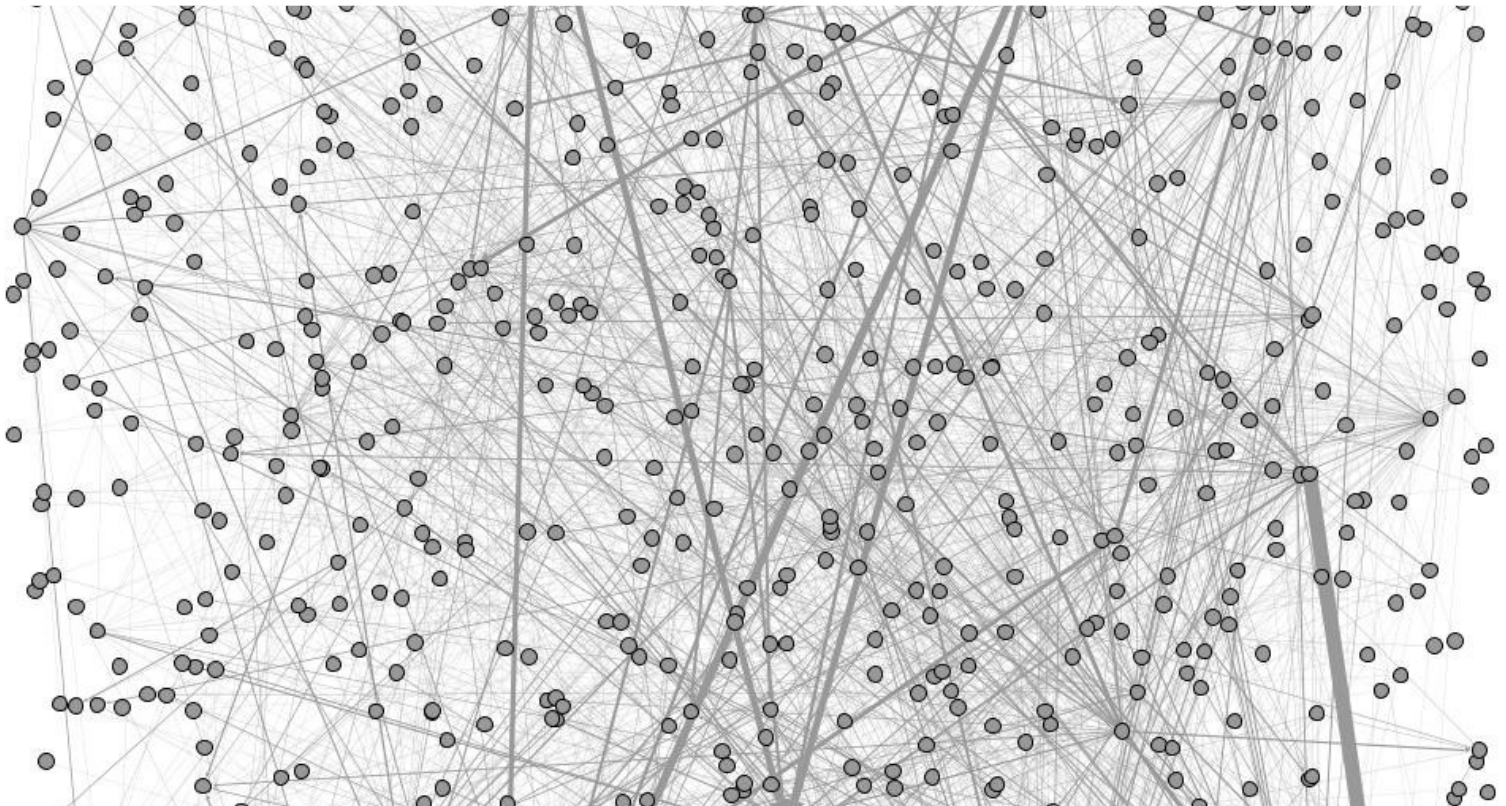


 23 SEPTEMBER 2018 /  @RRFD /  11 MIN READ /  929

Cookiecutter Data Science — Organize your Projects — Atom and Jupyter



One thing prevalent in most data science departments is messy notebooks and messy code. There *are* examples of beautiful notebooks out there, but for the most part notebook code is rough ... really rough. Not to mention all the files, functions, visitations, reporting metrics, etc. scattered through files and folders with no real structure to them.

I have been guilty of going back to a previous projects, searching for the matplotlib function I've written 50 times before that creates a certain graph with a certain metric. And, I found it in cell 200 after searching through a few different notebooks. Not efficient at all.

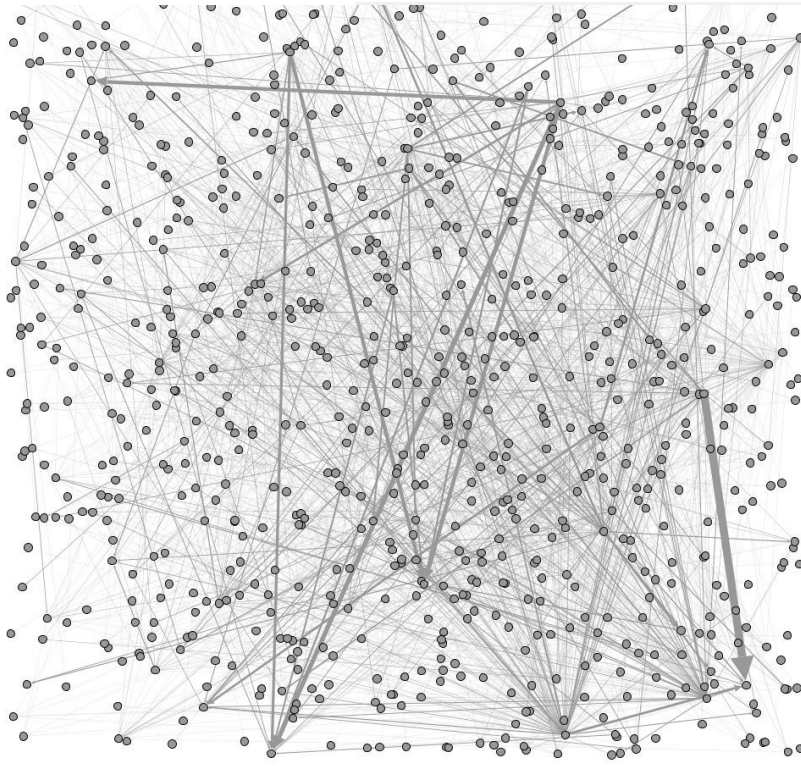


currently doing, and think if are you making this harder or easier for yourself in a few weeks time when you need to come back and review the code, or find something specific like a figure or the original data set?

One of the creeds some software development is not to repeat yourself. So, let's find out how we can do that.

Also, I give a lot of credit to [Go Data Driven's blog](#), which was a huge source of inspiration for this post.





This ... is a mess

How I Think About Projects

Here are some of the main principles of projects I use to keep myself organized (sane) :

1. Implementing version control;
2. Setting up a virtual environment for reproducible results;
3. Separating raw data, intermediate data, and final data;

Now let's get into this.

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

This is normal

I use it because I forget why I've made certain changes to a file, and



annotations and versions.

If you don't have an account, get one [here](#) and start keeping track of your code. I won't go over the entire process, but this [article](#) will show you how to get your first \$ git commit -m "My first commit message" and \$ git push for your project.

Now you can have version control over each project because every commit is simply a new version saved which can be referenced.

Once you start pushing code, the thing that will slow productivity to a halt after reviewing a 6-month project with 100s of commits, will be reading your old commit messages and asking "WTF does this mean?" To solve this problem, make sure all your messages are in the [imperative mood](#).

Taking this from Chris Beams,



subject line should always be able to complete the following sentence:

“If applied, this commit will
your subject line here”

This will help greatly when you're making different models. In the past, I would have modelv1.h5, modelv2.h5, etc. and I could not remember why each of them was different or why I changed the weighting on something.

With Git, I can now just use one name model.h5 and put my update comments in the commit message. E.g. I know I've added a new layer in my Keras model so my commit message would be \$ git commit -m "Add additional layer with 50 neurons".

If I ever need to go back and see what was involved in a certain model I'll be able to view or



see the work I've done.

This saves so much time. Trust me.

\$ virtualenv

Now that we have version control down, let's look at making sure someone else can reproduce our work with a virtual environment.

I was against using virtual environments when I first started learning data science because they seemed like a pain to set up and maintain. However, they will save so much headache as your project grows and you need to start sharing information.

More on them [here](#) and [here](#).

The main reason for using virtual environments is package version control for packages and allowing you to easily import custom functions from different parts of your project, keeping you



pandas 0.23 but you're using pandas 0.19, and you don't want to upgrade because it's going to break something in another project.

By having a virtual environment you'll be able to start with a fresh new version of Python—it's like getting a new car—and install packages with specific versions directly into each project without any conflict from anything you might have previously installed. So, you'll be able to have project Foo using pandas 0.23 and then project Bar using pandas 0.19, without any chance of conflict with each other.

If you don't have the required packages to set up your virtual environment, you can get them by following the instructions below.

```
# Get a virtual environment
```

```
# Python2  
$ pip install virtualenv
```



```
# Already installed with venv
```

Now let's set up the environment.

```
# Python 2:  
$ virtualenv env
```

```
# Python 3.6+  
$ python3 -m venv <<name_of_directory>>
```

If your directory does not exist, specify what you want as `name_of_directory` and it will be created. Else, the `venv` will just be installed into the chosen directory.

Once that is done, activate your environment.

```
$ source name_of_directory/bin/activate
```

If all goes well, you should see a change in your terminal's prompt showing the name of the



And, to deactivate the environment, just run `$ deactivate` in the root of your project.

```
~/Documents
> python3 -m venv cookiecutter

~/Documents
> cd cookiecutter

~/Documents/cookiecutter
> ls
bin      include  lib      pyvenv.cfg

~/Documents/cookiecutter
> source bin/activate

~/Documents/cookiecutter
cookiecutter > |
```

The name of our venv is 'cookiecutter'

Easy.

Next, install a fresh version of Jupyter into your environment.

```
(env_name) $ pip3 install jupyter
```

We have a fresh version of Python and Jupyter to work with for our project.

Try some cookiecutter with that data



your project structure. There is *a lot* with this package that is useful, but I've found when I'm just doing projects on my own I don't need all of the features and packages provided.

And, that's ok. Because every project is different and you'll have different requirements depending on what you work on. The package is structured so that you should only keep the features you need, and if you find a structure that works you're able to import it easily on the next project.

Installation and using it is easy.
Just two lines of code to run.

```
# Install  
$ pip3 install cookiecutter
```

```
# cd to the directory you want to start  
cookiecutter https://github.com/drivend
```



available, as you can look through some additional options [here](#).

Here is what mine looks like:

```
├── README.md          <- Front page of
                        know the majo
├── models             <- Trained and s
                        predictions,
├── notebooks          <- Jupyter noteb
                        E.g. `1.2-rd-
├── reports            <- HTML, PDF, an
    └── figures         <- Generated fig
├── requirements.txt   <- File for repr
                        `$ pip freeze
├── data
    ├── external       <- Third party s
    ├── interim        <- In-progress i
    ├── processed       <- The final dat
    └── raw             <- The original,
├── src                <- Source code f
    ├── __init__.py    <- Makes src a P
    ├── custom_func.py <- Various custo
    ├── data            <- Scripts to do
        └── make_dataset.py
    ├── features        <- Scripts raw d
        └── build_features.py
    └── models          <- Scripts to tr
```



```
├── predict_model.py
├── train_model.py
└── viz                <- Scripts to cr
    └── viz.py
```

If someone wants to recreate your project with the same package versions, all you need to do is export the list with freeze so they can install the same versions as the project.

```
# In project root
```

```
$ pip freeze > requirements.txt
```

```
# And to install the packages
```

```
$ pip install -r requirements.txt
```

Data Folders



import sections of cookiecutter. The data/ folders that allow you to organize yourself and control your data sources. It is so simple that I should have been organizing my projects like this from the start.

There are four standard folders:

- External;
- Interim;
- Processed, and;
- Raw.

By segmenting your data this way you'll be able to save your in-progress work, and ensure that your raw data has an immutable folder to live in. You'll also be able to quickly and logically access your data rather than just importing a single lump .csv which may or may not be immutable.

No more accidentally saving over the raw data because everything



dealing with file names like dataV1.csv or finalV1.csv, because everything will be (should be) put in its own place.

Documenting The Project

Not a huge concern, a lot of data science projects are done in Jupyter which allows the reader to (hopefully) follow the project logically. Allowing others to reproduce the work, should they have access to the correct packages and the same data the author had.

And now that cookiecutter has segmented the project more, the notebooks should be easier to read rather than a mess of code, functions, and results as they have been moved to other parts of the project.

E.g. Your matplotlib functions can now live in ./src/viz and all the



saved to `./reports/figures`.

Get Your Functions In

I used to have cells scattered all over my notebooks with custom functions, that I would later use in the project. And, the more I would write the harder it became to keep track of them all for the project. Let alone try to debug them or test them.

Even finding them for a project in the future to reuse became a major pain.

Later I would write all my function in `.src/funct/Foo.py` and stash them in different folders for some semblance of organization much like an actual software project. This would allow me to partition my project, so my notebook had the structure of the what was being analyzed while the functions lived in `Foo.py`, and were imported into my notebook.



```
└─ funct/                # Sub Folder
   └─ Foo.py              # Class with cus
      └─ __init__.py
```

However, I would need to import the files into my notebook like this.

```
# Meanwhile in my old notebooks ...
```

```
import os
```

```
currDir = os.path.dirname(os.path.realpath(__file__))
rootDir = os.path.abspath(os.path.join(currDir, os.pardir))
sys.path.insert(1, rootDir + '/src/func')
```

```
# Now I can finally import Foo from the
```

```
from func import Foo
```

```
print(Foo.foo()) # 'foo'
```



to checking Stack Overflow on how to use `os.path.dirname` and `os.path.realpath` correctly.

Now, with cookiecutter, you're able to easily link everything in your `./src/` with your notebook so you can import the `Foo.py` directly.

```
# Link ./src/ with your notebook
```

```
# Run this in the project root
```

```
$ pip3 install --editable .  
>>Obtaining file:///Users/foo/cookiecut  
>>Installing collected packages: src  
>>Running setup.py develop for src  
>>Successfully installed src
```

```
$ pip3 list  
Package      Version Location  
-----  
pip          18.0  
setuptools  39.0.1  
src          0.1.0   /Users/foo/cookiecut
```



folders provided with cookiecutter to easily import custom functions into your notebook. And, allow us to have a common location to put custom functions for easy retrieval for later projects.

```
# Back in my notebook
from src.funct import Foo
```

```
print(Foo.foo()) # foo
```

Note, if you do have issues with pip3 install—editable . and you cannot import your /src package, try \$ python setup.py develop—no-deps in the project root.

If that still does not work just copy and paste this code at the top of your notebook. It will allow you to import src as a package. However, this is not suggested.

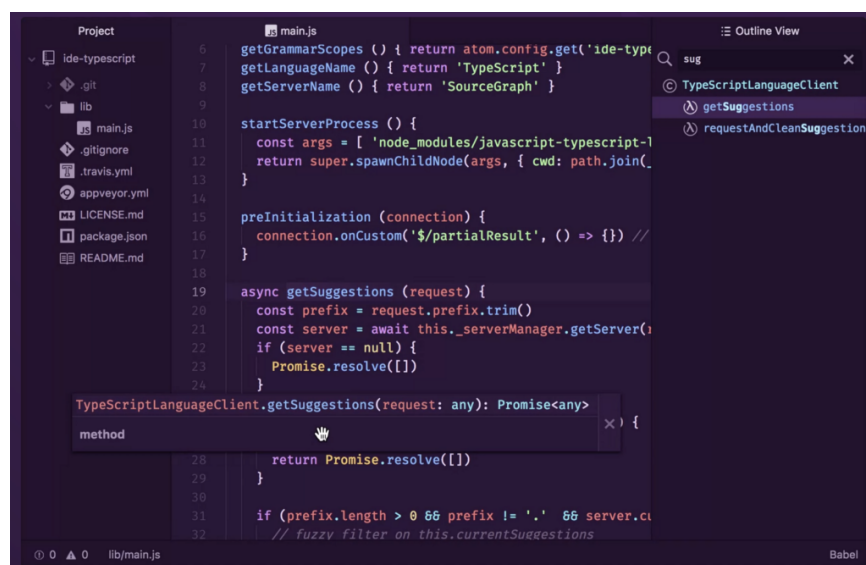
```
if __name__ == '__main__' and __package__
    from os import sys, path
    sys.path.append(path.dirname(path.d
```



Using Jupyter and Atom Together

It's inefficient to write all the *.py files in Jupyter, so instead use an IDE which has a linter, interactive debugger, and other features that make it more preferable to writing *some* code your project. Jupyter has its strengths, but there are some things that only an IDE can do.

Usually, I use IntelliJ or PyCharm. However, because not everyone has access to that editor I'll show the examples in Atom. It also has an IDE feature now, so it will be easier to run debugging within this editor.



So let's run this editor next to our Jupyter notebook. Just split (as shown below) your screen, with one side with Jupyter and one with the IDE.

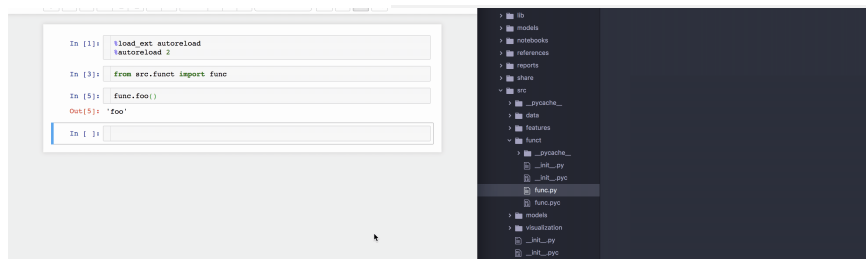
As a note, you need to make sure you have two magic functions imported at the beginning of your notebook.

```
%load_ext autoreload  
%autoreload 2
```

The suffix 2 is used after autoreload, to reload all modules every time *before* executing the Python code typed. This will enable us to edit functions in Atom, debug them, and quickly import the code into our notebook, rather than having to write out a debug the function directly in Jupyter.

I've made a gif to illustrate how this looks in practice.





This also save a lot of time

With the IDE up and running next to Jupyter, not only are you able to easily edit your files but you can also view your entire project easily rather than having to flip back and forth between windows in your browser.

Final Thoughts

At the beginning of the article, I outlined the five things I think about when doing projects, and I hope each point has been explained with the “how” I approach each of them.

1. Implementing version control (Git)
2. Setting up a virtual environment for reproducible

3. Separating raw data, intermediate data, and final data (./data);
4. Documenting work (Jupyter notebooks), and;
5. Separate modules for custom functions(./src).

Of course, all of this is all made possible with cookiecutter. Without this package, I would still be writing mediocre code in a disorganized mess. However, I'm now able to more easily organize myself, and most importantly, find old code to reuse in future projects.

I hope these methods outlined help you on your next project, and you've learned something new.

Cheers.

Additional Reading

- [Home - Cookiecutter Data Science](#)



[Data Science] Projects

(updated for 2018)

- [GoDataDrivenBlog](#)



Like

 Save to Facebook

source: [@rrfd](#)

[@rrfd](#)

[software development](#)

[data science](#)

[jupyter](#)

[python](#)

[cookie cutters](#)



Robert R.F. DeFilippi

Read [more posts](#) by this author.

[Read More](#)

Latest Posts

So You Want My Job? TikTok's Khartoon Weis <

