

ETL Tasks in Gapminder

This document is about how to work on ETL projects(dataset repositories) in Gapminder, including creating/managing/manipulating of them¹.

Table of Contents:

- [Structure of a dataset repository](#)
- [Start an ETL Project](#)
- [From Source to DDF](#)
 - [Using Jupyter notebook](#)
 - [ETL script](#)
 - [Checklist for DDF](#)
- [From recipes to DDF](#)
 - [Writing recipes](#)
- [Managing master branch and autogenerated branch](#)
 - [comparing branches using daff](#)
 - [Making patches from diff](#)
- [Other Common Tasks](#)
 - [generate SG dataset](#)
- [One more thing](#)

Structure of a dataset repository

When designing the repository structure, we think that it should allow us to:

1. have a single entry point for ddf files(project root) and a single entry point for all other files(etl/).
2. Allow manipulations
3. Compare datasets for equality

With the help of version control system(git), we achieve above by using 2 branches for each project:

- `edit(master)` branch: the branch that associated with WS server, and to which we can do manual edits
- `autogenerated` branch: the branch that generated with script

All datasets/branches should share same project structure:

¹ A good reference about managing data science projects can be found at [Cookiecutter Data Science](#). We are following some of the advices too.

```

.
|-- README.md
|-- ddf--concepts.csv                <- ddf concepts file
|-- ddf--datapoints--{concept}--{by}.csv  <- ddf datapoints files
|-- ddf--entities--{domain}.csv         <- ddf entities files
|-- datapackage.json                  <- Data Package descriptor
`-- etl
    |-- README.md
    |-- notebooks                    <- Jupyter notebooks
    |-- recipes                       <- recipes, and dictionaries
    |   |-- recipe.yaml
    |   `-- translation_dictionaries
    |-- requirements.txt               <- requirements for the etl script
    |-- scripts                       <- etl scripts
    |   `-- etl.py
    `-- source                        <- raw source data

```

Note that:

1. DDF csvs and datapackage.json should be in the root of the project directory. (For easy access from WS server.)
2. by default, the project_root/etl/source/ folder is excluded from version control
3. if there are libraries used in the project other than python standard library, we should add them to [requirements.txt](#)

Start an ETL Project

A template for ETL project is provided in [ddf_utils](#) for creating projects easily. Just install `ddf_utils` and run `ddf new` in the command line prompt, the script will ask for information about the project and create the new project automatically².

After creating the local project, you should create a github repo and connect the repository to the project directory. The naming of repository should follow this format:

`ddf--{source_provider}--{repo_name}`, in which `source_provider` and `repo_name` should be only consists of lowercase alphanumeric and underscore.

An screencast demonstrates how to set up new project can be found [here](#).

From Source to DDF

Creating DDF from source means to extract components of DDF model (concepts/datapoints/entities) from source files into pandas DataFrame, then save them as CSV files. The source files for each dataset vary in many ways, we have to write different scripts for each dataset. However there are some commonly used functions, for which we created a library [ddf_utils](#).

While doing ETL tasks, you should keep in mind the [DDF data model](#) and [DDF CSV format](#), make sure the result follows the rules.

Using Jupyter notebook

Jupyter notebook is great environment for exploring data, doing ETL tasks and communications. It's recommended to do ETL in notebook first and then put it together into the etl script. Working in the notebook make you able to check and test the output on each step easily, and other people can see how you create the ddf in detail.

An example of using jupyter notebook to create DDF files can be found [here](#).

² The project template can be found [here](#)

ETL script

Once you created the ddf with jupyter notebook, it would be easy for you to create the etl script. The expected api of etl script should be:

- `python etl.py`: create the ddf dataset in project root

An example ETL script can be found [here](#).

Checklist for DDF

Here are some items you should check after creating a DDF from source:

- Check your script, ensure it removes the old ddf files before creating new ones
- Check the concepts, see if they match the concepts you want to extract
- randomly pick some indicators to check if the data is identical to the source
- check if concept names/entities names are in lowercase alphanumeric format
- run the [validate-ddf tool](#) against the DDF repo

From recipes to DDF

[Recipe](#) is an instruction about how to construct new dataset from existing ones. The chef module from ddf_utils is designed to read and run the recipes and create new DDF datasets. For now, chef module will read datasets from your local computer and assuming all datasets are in one directory.

To run a recipe, run following command in commandline:

```
$ ddf run_recipe -i recipe.yaml -o output_dir
```

Then the script will create ddf dataset in `output_dir` base on the `recipe.yaml` file.

Also, you can run the recipe in a script/jupyter notebook. An example can be found [here](#).

Writing recipes

When creating the project, a template recipe can be found at `project_root/etl/recipes/_template.yaml`. You can start writing recipes by make copy of this template file. Inside the template we have provide you the basic structure of a recipe and documents about each components. Learn more about the recipe format in its [document](#).

Managing master branch and autogenerated branch

The `master(edit)` branch should equate with the `autogenerated` branch. But sometimes when we find a problem in the data, we want to fix them as soon as possible in the production server. Then we will modify the master branch manually, and afterward we need to change the script to reflect those changes. The workflow is as below:

1. problem found in dataset, and we fixed them in `master` branch
2. pull all changes in master branch to `autogenerated` branch
3. developer make changes to the script in `autogenerated` branch, and generate new dataset with it.
4. compare `autogenerated` branch to `master` branch, make sure they are the same
5. create pull request to the `master` branch
6. if the dataset is created with recipe, we should also update the source datasets too if necessary.

comparing branches using daff

[daff](#) is a very useful tool for comparing tables, and it works well with git. We can use this handy tool for comparing autogenerated branch and master branch. Here's how:

1. install daff. It is ported to many languages, so we have a many options to install it:

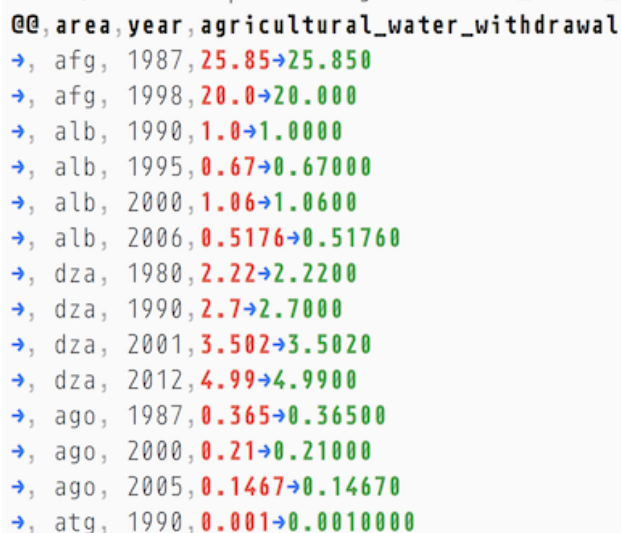
```
npm install daff -g # node/javascript
pip install daff # python
gem install daff # ruby
composer require paulfitz/daff-php # php
install.packages('daff') # R wrapper by Edwin de Jonge
bower install daff # web/javascript
```

2. setup daff for diff csv file for the dataset. (should run in the project root):

```
$ daff git csv
```

3. now you will be able to compare between branches using `git diff`

Here is a screenshot on the diff report:



```
@@, area, year, agricultural_water_withdrawal
->, afg, 1987, 25.85->25.850
->, afg, 1998, 20.0->20.000
->, alb, 1990, 1.0->1.0000
->, alb, 1995, 0.67->0.67000
->, alb, 2000, 1.06->1.0600
->, alb, 2006, 0.5176->0.51760
->, dza, 1980, 2.22->2.2200
->, dza, 1990, 2.7->2.7000
->, dza, 2001, 3.502->3.5020
->, dza, 2012, 4.99->4.9900
->, ago, 1987, 0.365->0.36500
->, ago, 2000, 0.21->0.21000
->, ago, 2005, 0.1467->0.14670
->, atg, 1990, 0.001->0.0010000
```

Making patches from diff

Diff patches is useful for doing arbitrary automatic changes. When we want to make changes that don't have a pattern (usually manual edits), we should consider using patches.

With help of daff, we can create patches for CSV files. The diff format daff uses is [Coopy highlighter diff format for tables](#).

Below is the workflow of patching manual edits:

1. We did manual edits in `master` branch of a dataset
2. in the `master` branch, run

```
$ git diff autogenerated -- the_edited_file.csv --output
patch.csv
```

3. Now the `patch.csv` will contain all manual edits you made to `the_edited_file.csv`

4. Then switch to `autogenerated` branch, you can run the command to apply changes

```
$ daff patch the_edited_file.csv patch.csv
```

or, you can do it programmatically. In `ddf_util` we provide a function `apply_patch(local, patch)` which can help you applying patches

Check example in the [SG repo](#).

Other Common Tasks

generate SG dataset

1. open your SG dataset folder
2. change the `ddf_dir` config in `etl/recipes/recipe_main.yaml` in SG folder to the path of your local computer that have all datasets.
3. go to `etl/scripts` and run `etl.py`.

```
$ cd etl/script && python etl.py
```

the script will run recipes and create all ddf files. If you don't have datasets required by the recipe, the program will raise exception and list the missing datasets.

One more thing

Let's end this document with a quote from Python style guides:

Consistency within a project is more important. Consistency within one module or function is the most important. ... However, know when to be inconsistent -- sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!