Building a Reproducible Data Science Environment with Nix

David Josephs

2019-11-06

Back to Navigation

Why Nix?

Data science environments are a massive headache. We depend on lots of *languages*, including python, R, sometimes javascript, and many more. We also depend on lots of *packages* within each language, some of which have dependencies outside of the language, for example if we are using tensorflow or pytorch with CUDA, or almost any fast R package. This creates for each project more or less a complex web of dependencies, to the point where our entire project, or any old project, can be more or less ruined by a simple update, or some issue when we install a new package. This is especially true in Python.

Luckily for Python users everywhere, we have some workarounds. We can use conda, which resolves dependencies itself, but is very difficult to rollback and oftentimes mysterious and frustrating. We can also develop an elaborate setup with pipenv or virtualenv and pip or whatever, but what about dependencies outside of python? How do we intend to deploy our model?

Enter nix. Nix is a 100% reproducible package manager, for all languages and all things. This means your python environment, your R environment, your models, your ***entire computer*** can be completely reproduced, all using the magic of nix. In this article, we will walk through setting up a simple, reproducible, and failproof data science stack with nix, including importing packages not found on nixpkgs and caching the builds online

How does Nix work?

Let us first think about how an environment is managed on a normal system. For that, the easiest way to think of it is to imagine a cobweb

Each time you add a new package to your environment, it goes into /usr/bin or somewhere similar, and connects itself to all the other packages it depends on and the ones which may depend on it, through a crazy web of symlinks and fresh hell. You are adding another connected node to the cobweb.

Now, try updating your computer, or working on a project that needs an older version of one of these dependencies. Suddenly, you are applying a force to the cobweb, moving nodes around. Unsurprisingly, this breaks the cobweb, and ***nothing*** works anymore.

Now, lets discuss how nix does it. Instead of /usr/bin, or whatever, a package managed by nix goes in /nix/store, named by a cryptographic hash for that specific version of that package. So this means that for example python 3.7 is stored in /nix/store/someverylongandcomplexhash-python37, while python 3.6 is stored in /nix/store/someotherlongcomplexhash-python36. Within these directories, the dependencies are stored for each package. So instead of a densely connected node in a cobweb, we can visualize each package as a tree in a forest.

Each tree minds its own business, and has its own roots, branches and leaves. If one tree moves or falls down or anyting like that, it does not affect the other trees. This is the same as /nix/store. Similarly, if we were to "update" our system, none of the old trees would die. Instead, new trees would grow, still separate from the old trees. This means that A) it is literally impossible to enter dependency hell, or break packages by changing other packages, and B) we can work with multiple versions of the same package.

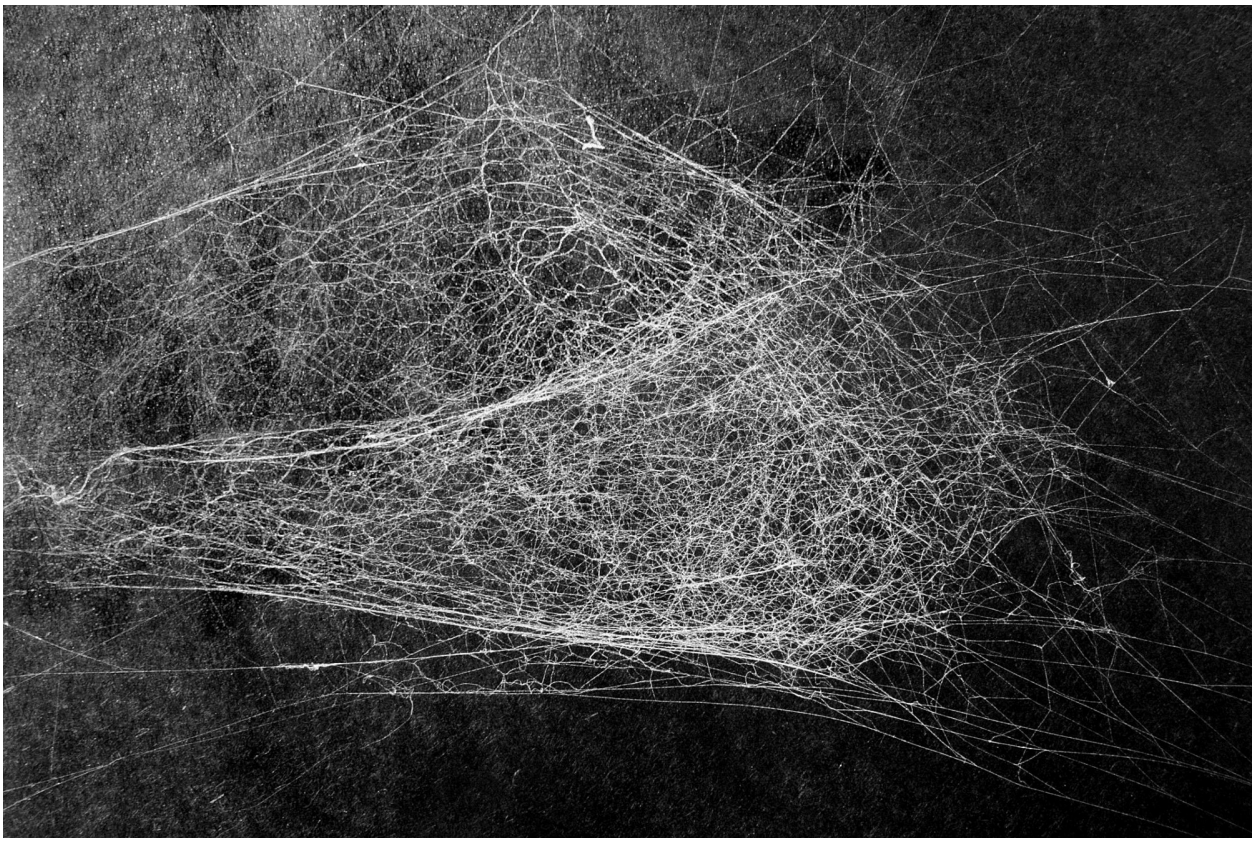

Figure 1: /usr/bin...

Figure 2: /nix/store :)

Let us now, for the purposes of data science, ask a question. Can I represent my environment ***for a single project*** as a package? Of course!!!! We can *easily* wrap up our entire data science environment as a completely isolated tree in our forest of packages. This means we can work on one project without any fear of it messing up our other projects, or anything on our computer!!

Making a reproducible, isolated environment

First, I will show the code used to make a solid working environment, and then we will walk through it line by line. Name this file `shell.nix`

```
let
  pkgs = import <nixpkgs> {};
in
  pkgs.mkShell {
    name = "simpleEnv";
    buildInputs = with pkgs; [
    # basic python dependencies
      python37
      python37Packages.numpy
      python37Packages.scikitlearn
      python37Packages.scipy
      python37Packages.matplotlib
    # a couple of deep learning libraries
      python37Packages.tensorflowWithCuda # note if you get rid of WithCuda then you will not be using
      python37Packages.Keras
      python37Packages.pytorchWithCuda

    # Lets assume we also want to use R, maybe to compare sklearn and R models
      R
      rPackages.mlr
      rPackages.data_table # "_" replaces "."
      rPackages.ggplot2
    ];
   shellHook = ''
      '';
  }
```

Now, lets walk through the code.

let ... in

First, we define I would say really our imports. I am not sure what the technical word for this is but its the stuff in between let and in.

```
let
  pkgs = import <nixpkgs> {};
```

On the left hand half of the equation (this is simple review, but it is intimidating seeing a new language), we have the name of the variable, which can be called after `in`. Next we have the trickier statement:

```
import <nixpkgs> {};
```

What this is doing is naming our connection to the nix package repository (`<nixpkgs>`) online. The thing in the `<>` is the name of the nix channel we are connecting to. This is really boilerplate and will never change for any of our use case. We will see later on what the `{}` does. Basically it is a space for any options or specific parts of `<nixpkgs>` we want to import. For now, it is just boilerplate.

pkgs.mkShell

Here is the first important bit. We are building what is called a `nix shell`. What this is, is when you type `nix-shell` in this directory on the command line, your computer is going to drop you into a new shell with the little environment we are building in its $PATH. This means you can still access your computer and all your files, but also access these include packages, without having it affect your system or touch any of your other environments.

The `name = "simpleEnv"` line is very simple, we are specifying what the name of our environment package will be in /nix/store. It is also the name it will appear as in our shell.

Next we come to the buildInputs. ***This is where you put the packages***. Note that `with pkgs; [...]` essentially means we dont have to type

```
pkgs.python37
pkgs.python37Packages.numpy
...
...
```

thankfully.

Finally, after this bit, we have an optional `shellHook = '' ''` line. This allows us to run any scripts, set up any services, or anything like that we may want to do. For me this line is useful when developing R packages, but otherwise I do not use.

And that is it :) You have made your first isolated, reproducible environment in Nix. To activate it, simply cd into the directory where the shell.nix is located, and type `nix-shell`. Note that if this is slow, you may want to try `nix-shell -j 8`, or however many cores you have, to make it build faster.

But what if my python package isnt in nixpkgs?

Let us say we are also interested in the cool data science visualization package yellowbrick. First, lets search the nix repository for it, using a convenient command line tool:

```
nix search yellowbrick
```

We can search for anything this way, with results automatically popping out in less. Unfortunately, this gives us no results. That means yellowbrick, which is in pypi, is not available in nipkgs by default. This is not the end of the world by any means, and we have a few ways to deal with it:

Method one: Cheating and using virtualenv like a baby

This time, we can just cheat, and use virtualenv in our shellHook along with requirements.txt. This is super lame, impure, and not advised, but sometimes we are all lazy. Please note the simple changes we made to wrap our nix environment in a virtualenv:

```
let
  pkgs = import <nixpkgs> {};
in
  pkgs.mkShell {
    name = "simpleEnv";
    buildInputs = with pkgs; [
    # basic python dependencies
      python37
      python37Packages.numpy
      python37Packages.scikitlearn
      python37Packages.scipy
      python37Packages.matplotlib
```

```
    # a couple of deep learning libraries
      python37Packages.tensorflow
    # python37Packages.tensorflowWithCuda # note if you get rid of WithCuda then you will not be using
      python37Packages.Keras
      python37Packages.pytorch # used for speedy examples
    # python37Packages.pytorchWithCuda

    # Lets assume we also want to use R, maybe to compare sklearn and R models
      R
      rPackages.mlr
      rPackages.data_table # "_" replaces "."
      rPackages.ggplot2

    # dirty virtualenv dependencies
      python37Packages.pip
      python37Packages.virtualenv
    ];
   shellHook = ''
    echo "yellowbrick" > requirements.txt
    virtualenv simpleEnv
    source simpleEnv/bin/activate
    pip install -r requirements.txt
      '';
  }
```

This will always work, but it is incredibly lame, and we cant do cooler things with this. We are simply putting yellowbrick in requirements.txt, activating a virtualenv, and then installing it, after we enter the nix shell. This is two layers of abstraction, but it is definitely possible. Not recommended at all.

Doing it the Nix way

Doing this the nix way is very simple. We can pull the package from PyPi (or GitHub/GitLab/wherever) using two lovely Nix builtins: `fetchFromPyPi` and `fetchFromGithub` (very appropriately named). Lets now proceed to fetch yellowBrick from PyPi in the let ... in section of our `shell.nix`:

```
let
  pkgs = import <nixpkgs> {};


  # bring in yellowbrick from pypi, building it with a recursive list
    yellowbrick = pkgs.python37.pkgs.buildPythonPackage rec {
      pname = "yellowbrick";
      version = "1.0.1" ;

      src = pkgs.python37.pkgs.fetchPypi {
        inherit pname version;
        sha256 = "1q659ayr657p786gwrh11lw56jw5bdpnl6hp11qlckvh15haywvk";
      };

# no tests because this is a simple example
      doCheck = false;
```

```
# dependencies for yellowbrick
      buildInputs = with pkgs.python37Packages; [
      pytest
      pytestrunner
      pytest-flakes
      numpy
      matplotlib
      scipy
      scikitlearn
    ];
  };
in
  pkgs.mkShell {
    name = "simpleEnv";
    buildInputs = with pkgs; [
    # basic python dependencies
      python37
      python37Packages.numpy
      python37Packages.scikitlearn
      python37Packages.scipy
      python37Packages.matplotlib
    # a couple of deep learning libraries
      python37Packages.tensorflow
    # python37Packages.tensorflowWithCuda # note if you get rid of WithCuda then you will not be using
      python37Packages.Keras
      python37Packages.pytorch # used for speedy examples
    # python37Packages.pytorchWithCuda

    # Lets assume we also want to use R, maybe to compare sklearn and R models
      R
      rPackages.mlr
      rPackages.data_table # "_" replaces "."
      rPackages.ggplot2

      yellowbrick
    ];
   shellHook = ''
      '';
  }
```

Lets now walk through this code, which seems very scary but is actually very simple.

First important bit is `pkgs.python37.pkgs.buildPythonPackage`, which is calling from nixpkgs (online repo), the buildPythonPackage function, which clearly allows us to build a python package. As an input, this function takes in a recursive set `rec {...}`.

The first thing we need to specify is the package name (`pname`), and the package version. Next, we need to specify the source (`src`) of the package, or where it comes from. This calls the name and the version of the package mentioned above, as well as a SHA256 hash sent in from Pypi. To get this hash, we can either do it an elegant and pure way, or the lazy David way. To get the proper hash the lazy way, simply put the wrong hash in and let nix correct you.

After that, we need to construct our tree. For this simple example. we will not run any python checkPhase, so we will say `doCheck = false`. After this example we will desimplify the recipe a bit.

The final step is we need to specify what dependencies are needed to build our package. In this case we use

`buildInputs`. In this case, the package will not build without pytest et al, matplotlib, numpy, scipy, and sklearn. And that is it. Go ahead and try running the following code, to test that it worked:

```
import sklearn.datasets
X,Y = sklearn.datasets.load_breast_cancer(return_X_y=True)

from yellowbrick.features import ParallelCoordinates

visualizer = ParallelCoordinates()
visualizer.fit_transform(X, Y)
visualizer.show()
```

Desimplifying buildInputs

Lets assume we are building production code. Obviously, we will probably want to run the checks on our package. So the first thing we do is we delete the `doCheck = false` line. The next thing we need to do is move the test dependencies away from buildInputs. These are actually only used during the check phase, so we write a new set of inputs, `checkInputs`

In this case we will now have

```
checkInputs = with pkgs.python37Packages; [
  pytest
  pytestruner
  pytest-flakes
];
buildInputs = with pkgs.python37Packages; [
  numpy
  matplotlib
  scipy
  scikitlearn
];
```

So pytest and friends will be called during the check phase, and the rest will be called when actually constructing the package. Finally, if there were some other packages that were `runtime dependencies` for the python package we were to install, we would put them in `propogatedBuildInputs`. For more extensive and technical documentation on doing this, please refer to the excellent nix python documentation

How can I make this quicker??

One thing you may notice, especially if you are compiling your environment to run with Cuda, is this build can take forever. Sometimes you end up having to compile very big packages, and this is never quick business. Thankfully, there is an amazing workaround to that. We can actually construct our environment as an honest to goodness package, in binary form, and then put that binary in an online cache, so we can simply download our tree whenever we want to use it. This is also an incredible tool for working in teams when you want total reproducibility. Every member of the team will have the exact same environment compiled under the exact same system.

For this, we will use cachix. The first thing we are going to do is write a new file, `default.nix`, which we build a nix ***derivation***(read: recipe) which compiles to a binary which we can dump into the online cache.

We will need to change the boilerplate code slightly to get this to work, but in general this is simple. As with the other examples, I will show it first then walk through what we did:

```
let
  pkgs = import <nixpkgs> {}; # standard
  # bring in yellowbrick from pypi, building it with a recursive list
```

```
    yellowbrick = pkgs.python37.pkgs.buildPythonPackage rec {
      pname = "yellowbrick";
      version = "1.0.1";

      src = pkgs.python37.pkgs.fetchPypi {
        inherit pname version;
        sha256 = "1q659ayr657p786gwrh11lw56jw5bdpnl6hp11qlckvh15haywvk";
      };

# no tests because this is a simple example
      doCheck = false;

# dependencies for yellowbrick
      buildInputs = with pkgs.python37Packages; [
      pytest
      pytestrunner
      pytest-flakes
      numpy
      matplotlib
      scipy
      scikitlearn
    ];
  };
  in with import <nixpkgs> {};
stdenv.mkDerivation rec { # new boilerplate
  name = "simpleEnv";

  # Mandatory boilerplate for buildable env
  # this boilerplate is courtesy of Asko Soukka
  env = buildEnv { name = name; paths = buildInputs; };
  builder = builtins.toFile "builder.sh" ''
    source $stdenv/setup; ln -s $env $out
  '';

  buildInputs = [

      python37
      python37Packages.numpy
      python37Packages.scikitlearn
      python37Packages.scipy
      python37Packages.matplotlib
      yellowbrick

      python37Packages.tensorflowWithCuda
      python37Packages.Keras
      python37Packages.pytorchWithCuda

      R
      rPackages.mlr
      rPackages.data_table # "_" replaces "."
      rPackages.ggplot2
  ];
```

```
}
```

And there we have it. We have built our first nix package/recipe. Lets now talk through the boilerplate.

The first thing we notice is instead of mkShell, we are using stdenv.mkDerivation, which makes a nix derivation (recipe). It is defined with a recursive set `rec`. Next, just as in the shell.nix, we give it a name. Then, in the `env = ...` bit, we define where nix should look to build the environment. In this case, it should look for the name we made, and the build inputs we define next.

We also have to define what tool is going to build it, in this case we are using a builtin (to nix) build script, which again is just boilerplate.

Finally, we have our buildInputs as usual, and then we are done. To see what we have done, try `nix-build` in the directory containing `default.nix` (you can also `nix-build default.nix`, but that is not needed). we can now look around the tree for the package which we are made, which is symlinked to `/nix/store/longhash-simpleEnv`.

To get this on an online cache, so we dont have to build our environment over and over, first you must sign up for cachix. Then, try out `cachix create simpleEnv`, so you create room in the online cache for you.

Next, enter in `nix-build | cachix push simpleEnv`. This builds the derivation and pipes the binary into the cachix push command and stores it.

Now, simply type `cachix use simpleEnv`. This connects your computer to your lovely cache, and any time you are using a package that is part of this environment, in nix shell or otherwise, it will instead of being built slowly, simply pull down that cached binary onto your computer.

Until next time

Hopefully, you now can quickly create nix environments for your data science workflow, and a bit of the boilerplate nix is demystified. Next time, we will look at building a container from nix expressions, fully reproducible Jupyter Notebooks and kernels, and packaging a machine learning model using nix (and deploying it as a service).

References

For more info from people significantly smarter than I, check out:

- nix python documentation
- Asko Soukka's Brilliant Blog

part 2