Practical Packaging For Machine Learning Solutions

Steven Cutting

November 18, 2017

Table of contents

Introduction

What/Why

Why learn about Python Packaging?

- Pythons deployment and sharing tools are general purpose.
  - Not specific to anyone machine learning framework.
  - Not dependent upon a proprietary service.
  - Use with **Docker**. Can **deploy to servers** in the 'Cloud'.
  - Provide a consistent, dependable, and extensible solution across projects.
- Make sharing and deployment easier.

Our target outcome

- We should leave with a basic understanding of the tools and best practices for Python packaging.
- How we can leverage that knowledge for use in our specific domain.

What I assume about you

- You know and *LOVE* **Python**.
- That you have used **pip** or at least conda.
- You have some understanding of the issues facing machine learning projects.
    - This is not strictly necessary, but will help give context.

What is a machine learning solution

For our purposes a machine learning solution is:

- It is a project that provides a model fitted on data for use by others.
- It includes the code necessary to **build the model**, the model, and an interface to use it.
- Nothing else. The solution can later be incorporated into an application, such as a web service.

Packaging in Data-Science

- *"If it's so great why don't I hear more about it?"*
  - Packaging is used, but not commonly talked about.
  - It's not a sexy subject (it's not `TensorFlow`).

Packaging in Data-Science: Things to note

- For the most part I have drawn from general software engineering best practices.
- Therefore I have done my best, by drawing on my industry experience, to extend the general best practices to cover this use case.

Python Packaging Basics

Just Enough

Here we are going to cover just enough of Python packaging basics as they relate to machine learning projects, so we can have the required background needed for the rest of the talk.

Package Vs. Distribution Package

From the Python Packaging Authority (PyPA)

Import Package

A Python module which can contain other modules or recursively, other packages.

Distribution Package

A versioned archive file that contains Python packages, modules, and other resource files that are used to distribute a Release. The archive file is what an end-user will download from the internet and install.

Project Requirements

```
basic_project/
   setup.py
   README.rst
   LICENSE.txt
   package/
        __init__.py
        data/
            model_file.pickle
```

Project Requirements: `setup.py`

```
setup(
    name="iris_classifier",
    version="1.0.0",

    packages=find_packages(),
    python_requires=">=3.6, <4",
    install_requires=['scikit-learn>=0.18,<0.19',
                      'setuptools>=36,<37'],
    package_data={'': ['data/model_file.pickle']},
    )
```

** More arguments are required to upload to PyPI.*

Source Distribution

To create an archive that we can share with others we can create a **Source Distribution** package using everything we setup previously.

(Or a Wheel)

Source Distribution: Definition

From the Python Packaging Authority (PyPA)

> *Source Distribution (or "sdist")*
> *A distribution format (usually generated using python setup.py sdist) that provides metadata and the essential source files needed for installing by a tool like pip, or for generating a Built Distribution.*

Source Distribution: Creation and Installation

For a guide on how to create and install source distribution packages:

ml-pkg-post.blog.stevencutting.com#packing-it-up

Distributing it

- **Email**
  We can simply email our users the distribution package file. Don't always overthink it.
- **PyPI**
  Upload it to PyPI. Make it publicly available.
- **Private package repository**
  Upload it to your client/companies private PyPI instance, so it won't be publicly available.

Deep-dive

For a deep-dive how-to:

ml-pkg-post.blog.stevencutting.com

Project Packaging Schemes

Some suggested packaging schemes

These schemes are not meant to be rigidly followed, but adjusted where needed to better fit unique problems.

Single package scheme

- This is the simplest approach, in which we place everything in a single package.
- The code to fit the model, the model, and the API code.

Single package scheme: structure

```
single_pkg_project/
   pkg/
       __init__.py
       data/
           modle_file.pickle
   setup.py
```

Single package scheme: Pros/Cons

- Pros
  - It's simple.
  - There is only one package to manage and ship and it includes everything.
- Cons
  - When we make updates to our model and any of the code, we must ship everything all over again.
  - We can not version the model itself separately from the API.

Three package scheme

We split the project into three packages:

1. `model_create`
2. `model`
3. `model_api`

Three package scheme: `model_create`

`model_create`

- The code used to create the model.
- Preprocessing code.

Three package scheme: `model`

`model`

- It should be the model file(s) and just enough code to load the model.
- It should only be single model.
- It will allow you to easily version the model.

Three package scheme: `model_api`

`model_api`

- The API
- Includes everything that actually surrounds the use of the model.
  - Data preprocessing/transformation.
  - Transforming model output.
  - Orchestration of application steps.
  - Input/Output validation.
  - etc.

Three package scheme: But why?

What do we get from this?

- We can version our training code, model, and API separately.
- Update and deploy parts independently.
- etc.

Three package scheme: Why - A scenario

Imagine:

- You have a 3Gig model file.
- You make a small bug fix to the API code.
- With the 3 package scheme you won't have to repackage and ship the model. Just the API.

Three package scheme: structure

```
three_pkg_project/
   fit_model_pkg/
       fit_model_pkg/
           __init_.py
       setup.py
   model_api_pkg/
       model_api_pkg/
           __init__.py
       setup.py
   model_pkg/
       model_pkg/
           __init__.py
           data/
               model_file.pickle
       setup.py
```

Three package scheme: Pros/Cons

- Pros
  - The model and API can be versioned separately.
  - User can easily swap in different models without messing with the API package.
- Cons
  - More complex.
  - We will need to build and ship at least two packages.

N-package scheme

- For projects that contain multiple models.
- It can contain multiple `model_create` packages for each model that needs to be created.
- Each model should have its own package that only handles loading the model file(s).
- There should only be one `apply_model` package, otherwise consider not including it.

N-package scheme: topic model with classifier example

Using a topic model as dimensionality reduction step for a classifier.

- The code used to fit the topic model and classifier should go in different packages.
- The models can go in their own packages. They can be released separately.
  - This way if the classifier is updated but the topic model is not, only the classifier has to be shipped. The topic model can be left alone.
  - Note, this does not work in the reverse.
- The classifier training package will depend upon the topic model package.
- The models will be composed in the API package.

N-package scheme: Pros/Cons

- Pros
  - Allows you to have and version multiple models and model fitting code.
- Cons
  - Even more complex.
  - We will need to build and ship a bunch of packages.
  - There may be a lot of duplicate code/effort (hint: there is a solution).

Tips and best practices

Multi-Package Shared Code: Problem

- What to do about code we want to use in multiple packages?
- *e.g. Preprocessing code that may be used in the model training step and when applying the model.*

Multi-Package Shared Code: Solution

- Place shared code in facet specific packages.
- *Example: Place preprocessing code in it's own package and provide it as a dependency to the training package and the API package.*

Multi-Package Shared Code: Suggestions

- Avoid bad dependency patterns
  - e.g. circular dependencies

Multi-Package Shared Code: Suggestions cont.

- Listing our other packages in the project as dependencies:
  - List a lower boundary on the version number.
  - Consider not specifying an upper-boundary.
  - *A lot of the benefit gained from splitting up the project will be lost if you are too restrictive with the dependency versions.*

Fragmented project: Growing number of packages

How are we going to handle the growing number of packages in a sane way?

Fragmented project: Remote repository - Create and Distribute

Creating and Distributing

1. Create a template project with a `setup.py` file and use that as the base for each new subproject.
2. Build and upload the packages as you finish coding them. The same is true for updates.

Fragmented project: Remote repository - Installation - Option 1

Option 1

- We can provide our user with a `requirements.txt` file with all of the required packages listed it.
- This can be very useful as it allow us to provide the locations of each package.
    - e.g. a URI pointing to a git repository or a private package repository.

Fragmented project: Remote repository - Installation - Option 2

Option 2

- Have all of your packages that need to be installed listed as dependencies in the API packages `setup.py` file.
- This will cause pip to automatically fetch and install all of the other packages.

Fragmented project: Remote repository - Installation - Preferred

Using `setup.py` (Option 2) is the preferred method.

- Especially if our project is not an end application, but a distribution package.
- All dependencies are listed in one location.
- We should avoid providing locations of packages. (Leave this to the end application maintainers)

Handling Data

- PyPI
  - Package size limit is around 60MB (increase can be requested).
    - If your model package is over 60MB, don't include the model file in the package.
    - Host the model file somewhere else.
- Using git version control
  - If a package size is over 1GB, consider excluding the model file from git versioning.

Handling Data: Training Data

- Training Data
  - Shouldn’t be included in the package.
  - To make it available to users consider including a way to download it.

Conclusion

Future of packaging in Data Science

We as a community should do a better job of creating general best practices on:

- Delivering, sharing, and deploying solutions.

Questions?

ml-pkg-post.blog.stevencutting.com

@steven_cutting

Things to be aware of

- These topics aren't directly related to packaging but they're very important because they concern the quality of the code within the package you are creating.
- No one will trust to use your package if the contents do not meet certain expectations.

Things to be aware of: Virtualenv

- Basically, ‘virtualenv‘ helps us to deal with multiple projects that have incompatible dependencies.
- For example, if we have two different projects that require different versions of the same package, virtualenv can help us manage this.

Things to be aware of: Automated testing

- Having an automated test suite for our software is important.
- Helps greatly improve our confidence in our code.
- PyTest is an excellent and approachable testing framework.

Things to be aware of: Version control

- Version control will help you to make modifications to your project in a more organized fashion, allowing you to better keep track of changes.
- A popular option is git, but there are other options such as mercurial and subversion.

Questions?

ml-pkg-post.blog.stevencutting.com

@steven_cutting

License