Packaging code with PEX — a PySpark example

Fabian Höring Follow
Jan 8, 2019 · 8 min read

Having a distributed computing environment is quite challenging because it is often difficult to know where your code really runs and which code is available. This is particularly true for frameworks like Spark where lot of the complexity is abstracted away.

In this post I'll will be dealing with making sure the code is available everywhere across one single spark job and across multiple environments like notebooks or preprod/prod environments. At Criteo we are using PEX to achieve this.

If you want to execute the code samples please make sure you install the prerequisites via 'source install.sh' first. It will setup a virtual environment with pex, numpy and

pyspark for the dependencies and an editable ‘userlib’ with shared code for our PySpark job.

Let’s first have look on what happens with the code when PySpark starts distributing the workload to the cluster.

Serializing the code that runs on the executors

The default serializer in PySpark is PickleSerializer which delegates all of the work to the pickle module. This applies for the RDDs that are sent to the executors but it also applies to the functions you execute. When you start your job Spark is serializing the functions you used, shipping them to the executors, deserializing and executing them.

Some example in PySpark shell:

```
>>> rdd = sc.parallelize([1, 3], numSlices=2)
>>> import math
>>> def sqrt(a):
…     return math.sqrt(a)
…
>>> def pyth_add(a,b):
…     return sqrt(a ** 2 + b ** 2)
…
>>> rdd.reduce(lambda x,y: pyth_add(x,y))
3.1622776601683795
```

In real life the function wouldn’t just look like that. Production code will use many external libraries.

But pickle doesn’t serialize external libraries:

```
>>> import numpy as np
>>> rdd = sc.parallelize([np.array([1,2,3]), np.array([1,2,3])],
numSlices=2)
>>> rdd.reduce(lambda x,y: np.dot(x,y))
[Stage 0:> (0 + 2) / 2]18/11/09 18:29:17 WARN
scheduler.TaskSetManager: Lost task 0.0 in stage 0.0 (TID 0, xx-xx-
xx-xx-xx-xx.xxx.xxx.criteo.prod, executor 1):
org.apache.spark.api.python.PythonException: Traceback (most recent
call last):
...
ModuleNotFoundError: No module named 'numpy'
```

Different ways to ship code to the executors

So we need somehow to ship our code to the Spark executors. There are already several ways to do this which are more or less documented.

I will present some of them with their advantages and drawbacks.

Using py-files

This is an easy way to ship additional code to the cluster. You can just add individual files or zip whole packages and upload them. This option is also available on the spark context (sc.addPyFile) making it possible to upload code after having started your job.

For packages this option only supports .zip & .egg extensions. So we can't easily ship external packages available as tar.gz source distribution or wheels on pypi (the Python package index). If we want to do that we would need to download the packages and repackage them locally.

Install packages globally

You could just install all needed packages at system level on each node. In reality you will have different packages for each job and the probability is quite high that there will be conflicting version across them. Also updating versions can be a nightmare. You current job might just see the new version of a package and fail.

Using virtual environments

Virtual environments allow to isolate your code and dependencies. They are self contained and can easily be recreated on a different machine to test with the exact same python interpreter and dependencies.

Shipping package names and recreating the virtual environment every time

One solution is to just send the names of the packages and then recreate the virtual environment all the time on each executor. This seems to be already supported by PySpark as suggested by this blogpost. If you have hundreds of executors it will retrieve the packages on each executor and recreate your virtual environment each time. This doesn't seem to be a very scalable solution.

Shipping the whole virtual environment

Another possibility is to ship the whole environment with each job. This is better because you only generate the environment once with all packages on a build server and then download one single consistent package on each executor.

Currently there are two options for this:

- Conda with conda-pack
- Virtual env with venv-pack

Conda is well documented and seems to be what most people use. Disadvantages of Conda are that you have to unzip the environment on each executor before using it. Also it includes the Python interpreter which can be good when new versions come out but creates significant overhead. For packages using native code (like numpy, tensorflow, ..) Conda packages will also ship system system libraries. This will shade the existing ones and might create conflicts on your system.

We could also use Python virtual environments with venv-pack. Venv-pack also has documentation for PySpark but its status seems less clear (see this Spark ticket). Another problem with virtual environments is that your local environment is not easily shippable to another machine. In particular there is the relocatable option (see the doc and this post) which makes it very complicated for the user to ship the virtual env and be sure it works.

Why do we need yet another tool at Criteo ?

At Criteo we have to run thousands of jobs in production with many different environments. The only viable solution for us seems to be to ship the whole environment.

In our first iteration we were using Conda because it was the most documented tool. One issue we had is that our internal packages are only released on our internal pypi package manager. We don't have an internal Conda package manager. Therefore, we had to mix up 'conda install' with 'pip install' commands. This will mess up your Conda environment quite quickly such that conda-pack is unavailable to ship the environment

anymore. It is quite unpredictable and there are many combinations of packages that don't work.

Another issue was that conda overlaps with system libraries. We wanted to have a system that allows to have a clear cut between our application teams and our SRE teams responsible for the platform.

That's why we were looking into a real virtual environment based approach. Just using venv-pack was working but it is always combined with a two-step approach without being sure that it would work when unzipping.

Using PEX to ship code

This is when PEX comes in. A pex file is an executable self contained python environment. It contains the built distribution for all packages needed by the application and a description of the entry-point. When the file is executed the system starts the the Python interpreter with the defined entry point. This takes advantage of PEP-441 where the Python interpreter is able to directly execute zip files (a pex file is actually a zip file). PEX contains a virtual environment. It doesn't include the Python interpreter as Conda does.

PEX also comes with a command line utility to easily generate the pex files.

One advantage is that everything fits in one single file which makes it easy to ship this to the production environment. This is similar to what can be achieved with a Java uber-jar, one uber-jar with the library and all dependencies included.

The good thing is that no Spark patches are necessary. It just works by changing the same environment variables as for Conda/Virtual env.

Here is a working sample with the PySpark shell:

```
$ pex numpy -o myarchive.pex
$ export PYSPARK_DRIVER_PYTHON=`which python`
$ export PYSPARK_PYTHON=./myarchive.pex
$ pyspark \
--conf spark.executorEnv.PEX_ROOT=./.pex \
--master yarn \
--deploy-mode client \
```

```
--files myarchive.pex
>>> import numpy as np
>>> rdd = sc.parallelize([np.array([1,2,3]), np.array([1,2,3])],
numSlices=2)
>>> rdd.reduce(lambda x,y: np.dot(x,y))
14
```

The numpy package is directly included in the pex archive via the pex cli tool and then available when we import numpy inside our Pyspark application.

Developing PySpark applications

When developing applications it can be cumbersome to ship the whole environment all the time. Just changing a single line of code would trigger the regeneration and upload of the whole environment. Therefore, it can be interesting to distinguish between libraries that rarely change and “under development” libraries.

It turns out we can achieve this by automatizing existing pip options.

‘pip list’ command allows to retrieve all installed dependencies.

```
$ pip list --exclude-editable --format json
[{"name": "numpy", "version": "1.15.4"}, {"name": "pex", "version":
"1.5.2"}, {"name": "pip", "version": "10.0.1"}, {"name":
"setuptools", "version": "39.0.1"}, {"name": "wheel", "version":
"0.31.1"}]
```

We have a helper library that parses the json result in Python, dynamically generates the package using pex python apis and uploads it to a shared repository.

The pex package with our dependencies is generated only once. In addition we keep track of all installed versions in a json metadata file. We can then compare with the version of the current virtual environment and just regenerate it when the user updates any dependency.

The downside with including everything in the package is when code being currently under development changes frequently. Every time something changes we need to regenerate and upload the whole pex package.

Fortunately we have the 'pip -e ..' option which permits to install packages in editable mode. All changes are directly taken in account when restarting the python interpreter.

```
$ pip list -e --format json
[{"name": "userlib", "version": "0.0.1"}]
```

We can automatize this in a script that parses the result, zips the folder and then calls sparkContext.addPyFile. This will ship all files each time we launch the application making sure we always use the latest version.

Having the real local installation path of the package was quite challenging. All API's of pip are internal and might change when a new version is released. That's why we are using the pip cli version via the subprocess module. The 'pip freeze' command doesn't return this information in an easily parsable way. The 'pip list' command only provides the path via the -v option which might produce unforeseeable additional output in the future. We decided to import the package again to retrieve its path via __file__.

Running in Prod

In production our Continuous Integration would just generate one single pex file and then push it to a package server. The package contains all libraries, external dependencies and also the entry point of the application (there can be more than one entry point for different use cases). The advantage is that everything is contained in a single file that can be moved, tested and executed easily.

Here is the same code as before packaged inside a startup script (to be added to startup.py module inside userlib/userlib). The spark context creation can easily be factorized in a internal library such that people would only call a method create_spark_context to do the magic.

```
1  import os
2  import sys
3  import numpy as np
4  from pyspark import SparkConf, SparkContext
5
6  def create_spark_context():
7      pex_file = os.path.basename([path for path in sys.path if path.endswith('.pex')][0])
```

```
    conf = SparkConf() \
        .setMaster("yarn") \
        .set("spark.submit.deployMode", "client") \
        .set("spark.yarn.dist.files", pex_file) \
        .set("spark.executorEnv.PEX_ROOT", "./.pex")
    os.environ['PYSPARK_PYTHON'] = "./" + pex_file
    return SparkContext(conf=conf)

if __name__== "__main__":
    sc = create_spark_context()
    rdd = sc.parallelize([np.array([1,2,3]), np.array([1,2,3])], numSlices=2)
    print(rdd.reduce(lambda x,y: np.dot(x,y)))
    sys.exit(0)
```

startup.py hosted with ❤️ by GitHub view raw

Then we can just start the application by executing our pex file with our main entry point and then ship the same file with all libraries to the cluster such that all packages are available. Note that we don't use the pex entry point option at build time as we also want to ship the package to the executors which want to call their own PySpark modules.

```
$ pex pyspark==2.3.2 numpy userlib -o myarchive.pex
$./myarchive.pex -m userlib.startup
14
```

Wrap-up

In this post I presented how PySpark ships the serialized functions to the executors, why it is not easy to execute shared code and what are the different ways to ship code. At Criteo we are using virtual environments with pex to achieve that. A pex file is an executable self contained Python environment containing the built distribution for all packages needed by the application. It also embarks the entry point of our main class for production use cases.

It allows us to rely entirely on our internal Pypi package manager, easily works with PySpark and allows our CI to prepackage single files for production. We have built our

own helper libraries to ease the development mode and not packaging and shipping the whole environment all the time.

I you are interested in these kind of topics Criteo R&D is hiring.

Python Software Development Pyspark Spark