Pyspark Virtual Environment - A Story

Spark Community has been working on Virtual Environment Support for many years (+ 2 years) & **Its still Incomplete !**

It will be incorrect to claim that Pyspark doesn't have a Virtual Environment, it does have, but it's still incomplete!
There are many discussions & comments like an ocean, finally I managed to draft following 2 major approaches. There are many interesting points to learn and I am sure there are still arguments which are harder argue.

ABC | https://www.linkedin.com/in/iamabhishekchoudhary/

Episode 1: Wheelhouse Support For PySpark

Description

Rational

Is it recommended, in order to deploying Scala packages written in Scala, to build big fat jar files. This allows to have all dependencies on one package so the only "cost" is copy time to deploy this file on every Spark Node.

On the other hand, Python deployment is more difficult once you want to use external packages, and you don't really want to mess with the IT to deploy the packages on the virtualenv of each nodes.

This ticket proposes to allow users the ability to deploy their job as "Wheels" packages. The Python community is strongly advocating to promote this way of packaging and distributing Python application as a "standard way of deploying Python App". In other word, this is the "Pythonic Way of Deployment".

Previous approaches

I based the current proposal over the two following bugs related to this point:

- SPARK-6764 ("Wheel support for PySpark")
- SPARK-13587("Support virtualenv in PySpark")

First part of my proposal was to merge, in order to support wheels install and virtualenv creation

Virtualenv, wheel support and "Uber Fat Wheelhouse" for PySpark

In Python, the packaging standard is now the "wheels" file format, which goes further that good old ".egg" files. With a wheel file (".whl"), the package is already prepared for a given architecture. You can have several wheels for a given package version, each specific to an architecture, or environment.

For example, look at https://pypi.python.org/pypi/numpy all the different version of Wheel available.

The pip tools knows how to select the right wheel file matching the current system, and how to install this package in a light speed (without compilation). Said otherwise,

package that requires compilation of a C module, for instance "numpy", does **not** compile anything when installing from wheel file.

pypi.pypthon.org already provided wheels for major python version. It the wheel is not available, pip will compile it from source anyway. Mirroring of Pypi is possible through projects such as http://doc.devpi.net/latest/ (untested) or the Pypi mirror support on Artifactory (tested personnally).

pip also provides the ability to generate easily all wheels of all packages used for a given project which is inside a "virtualenv". This is called "wheelhouse". You can even don't mess with this compilation and retrieve it directly from pypi.python.org.

Use Case 1: no internet connectivity

Here my first proposal for a deployment workflow, in the case where the Spark cluster does not have any internet connectivity or access to a Pypi mirror. In this case the simplest way to deploy a project with several dependencies is to build and then send to complete "wheelhouse":

- you are writing a PySpark script that increase in term of size and dependencies. Deploying on Spark for example requires to build numpy or Theano and other dependencies
- to use "Big Fat Wheelhouse" support of Pyspark, you need to turn his script into a standard Python package:

write a requirements.txt. I recommend to specify all package version. You can use pip-tools to maintain the requirements.txt

```
astroid==1.4.6 # via pylint
autopep8==1.2.4
click==6.6 # via pip-tools
colorama==0.3.7 # via pylint
enum34==1.1.6 # via hypothesis
findspark==1.0.0 # via spark-testing-base
first==2.0.1 # via pip-tools
hypothesis==3.4.0 # via spark-testing-base
lazy-object-proxy==1.2.2 # via astroid
linecache2==1.0.0 # via traceback2
pbr==1.10.0
pep8==1.7.0 # via autopep8
pip-tools==1.6.5
py==1.4.31 # via pytest
pyflakes==1.2.3
pylint==1.5.6
pytest==2.9.2 # via spark-testing-base
six==1.10.0 # via astroid, pip-tools, pylint, unittest2
spark-testing-base==0.0.7.post2
traceback2==1.4.0 # via unittest2
```

```
unittest2==1.1.0 # via spark-testing-base
wheel==0.29.0
wrapt==1.10.8 # via astroid
```

- write a setup.py with some entry points or package. Use PBR it makes the jobs of maitaining a setup.py files really easy
- create a virtualenv if not already in one:

```
virtualenv env
```

    - Work on your environment, define the requirement you need in requirements.txt, do all the pip install you need.

- create the wheelhouse for your current project

```
pip install wheelhouse
pip wheel . --wheel-dir wheelhouse
```

    - This can take some times, but at the end you have all the .whl required **for your current system** in a directory wheelhouse.
    - zip it into a wheelhouse.zip.

Note that you can have your own package (for instance 'my_package') be generated into a wheel and so installed by pip automatically.

Now comes the time to submit the project:

```
bin/spark-submit --master master --deploy-mode client --files
/path/to/virtualenv/requirements.txt,/path/to/virtualenv/wheelhouse.zip --conf
"spark.pyspark.virtualenv.enabled=true" ~/path/to/launcher_script.py
```

You can see that:

- no extra argument is add in the command line. All configuration goes through --conf argument (this has been directly taken from SPARK-13587). According to the history on spark source code, I guess the goal is to simplify the maintainance of the various command line interface, by avoiding too many specific argument.
- The wheelhouse deployment is triggered by the --conf "spark.pyspark.virtualenv.enabled=true" }} argument. The {{requirements.txt and wheelhouse.zip are copied through --files. The names of both files can be

changed through --conf arguments. I guess with a proper documentation this might not be a problem
- you still need to define the path to requirement.txt and wheelhouse.zip (they will be automatically copied to each node). This is important since this will allow pip install, running of each node, to pick only the wheels he needs. For example, if you have a package compiled on 32 bits and 64 bits, you will have 2 wheels, and on each node, pip will only select the right one

I have choosen to keep the script at the end of the command line, but for me it is just a launcher script, it can only be 4 lines:

```
/#!/usr/bin/env python

from mypackage import run
run()
```

-

on each node, a new virtualenv is created **at each deployment**. This has a cost, but not so much, since the pip install will only install wheel, no compilation nor internet connection will be required. The command line for installing the wheel on each node will be like:

```
pip install --no-index --find-links=/path/to/node/wheelhouse -r
requirements.txt
```

advantages

- quick installation, since there is no compilation
- no Internet connectivity support, no need to mess with the corporate proxy or require a local mirroring of pypi.
- package versionning isolation (two spark job can depends on two different version of a given library)

disadvantages

- creating a virtualenv at each execution takes time, not that much but still it can take some seconds
- and disk space
- slighly more complex to setup than sending a simple python script, but this feature is not lost
- support of heterogenous Spark nodes (ex: 32 bits, 64 bits) is possible but one has to send all wheels flavours and ensure pip is able to install in every environment. The complexity of this task is on the hands of the developer and no more on the IT persons! (TMHO, this is an advantage)

Use Case 2: the Spark cluster has access to Pypi or a mirror of Pypi

This is the more elegant situation. The Spark cluster (each node) can install the dependencies of your project independently from the wheels provided by Pypi. Your internal dependencies and your job project can also comes in independent wheel files as well. In this case the workflow is much simpler:

- Turn your project into a Python module
- write requirements.txt and setup.py like in Use Case 1
- create the wheel with pip wheels. But now we will not send **ALL** the dependencies. Only the one that are not on Pypi (current job project, other internal dependencies, etc).
- no need to create a wheelhouse. You can still copy the wheels either with --py-files (will be automatically installed) or inside a wheelhouse named wheelhouse.zip

Deployment becomes:

Now comes the time to submit the project:

```
bin/spark-submit --master master --deploy-mode client --files
/path/to/project/requirements.txt --py-files
/path/to/project/internal_dependency_1.whl,/path/to/project/internal_dependency_2.whl,/path/
to/project/current_project.whl --conf "spark.pyspark.virtualenv.enabled=true" --conf
"spark.pyspark.virtualenv.index_url=http://pypi.mycompany.com/" ~/path/to/launcher_script.py
```

or with a wheelhouse that only contains internal dependencies and current project wheels:

```
bin/spark-submit --master master --deploy-mode client --files
/path/to/project/requirements.txt,/path/to/project/wheelhouse.zip --conf
"spark.pyspark.virtualenv.enabled=true" --conf
"spark.pyspark.virtualenv.index_url=http://pypi.mycompany.com/" ~/path/to/launcher_script.py
```

or if you want to use the official Pypi or have configured pip.conf to hit the internal pypi mirror (see doc bellow):

```
bin/spark-submit --master master --deploy-mode client --files
/path/to/project/requirements.txt,/path/to/project/wheelhouse.zip --conf
```

```
"spark.pyspark.virtualenv.enabled=true" ~/path/to/launcher_script.py
```

On each node, the deployment will be done with a command such as:

```
pip install --index-url http://pypi.mycompany.com --find-links=/path/to/node/wheelhouse -r
requirements.txt
```

Note:

- --conf "spark.pyspark.virtualenv.index_url=http://pypi.mycompany.com/" allows to specify a Pypi mirror, for example a mirror internal to your company network. If not provided, the default Pypi mirror (pypi.python.org) will be requested
- to send a wheelhouse, use --files. To send individual wheels, use --py-files. With the latter, all wheels will be installed. For multiple architecture cluster, prepare all needed wheels for all architecture and use a wheelhouse archive, this allows pip to choose the right version of the wheel automatically.

Important notes about some complex package such as numpy

Numpy is the kind of package that take several minutes to deploy and we want to avoid having all nodes install it each time. Pypi provides several precompiled wheel but it may occurs that the wheel are not right for your platform or the platform fo your cluster.

Wheels are **not** cached for pip version < 7.0. From pip v7.0 and +, wheel are automatically cached when built (if needed), so the first installation might take some time, but after the installation will be straight forward.

On most of my machines, numpy is installed without any compilation thanks to wheels

Certificate

pip does not use system ssl certificate. If you use a local pypi mirror behind https with internal certificate, you'll have to setup pypi correctly with the following content in ~/.pip/pip.conf:

```
[global]
cert = /path/to/your/internal/certificates.pem
```

First creation might take some times, but pip will automatically cache the wheel for your system in /.cache/pip/wheels. You can of course recreate the wheel with pip wheel or find the wheel in /.cache/pip/wheels. You can use pip -v install numpy to see where it has placed the wheel in cache.

If you use Artifactory, you can upload your wheels at a local, central cache that can be shared accross all your slave. See this documentation to see how this works. This way, you can insert wheels in this local cache and it will be seens as if it has been uploaded to the official repository (local cache + remote cache can be "merged" into a virtual repository with artifactory)

Set use of internal pypi mirror

Ask your IT to update the ~/.pip/pip.conf of the node to point by default to the internal mirror:

[global]

```
; Low timeout
timeout = 20
index-url = https://&lt;user&gt;:&lt;pass&gt;@pypi.mycompany.org/
```

Now, no more need to specify the --conf "spark.pyspark.virtualenv.index_url=http://pypi.mycompany.com/" in your Spark submit command line

Note: this will not work when installing package with python setup.py install syntax. In this case you need to update ~/.pypirc and use the -r argument. This syntax is not used in spark-submit

Episode2: Using VirtualEnv with PySpark

Author: Jianfeng Zhang

Introduction

For simple PySpark application, we can use `--py-files` to specify its dependencies. A large PySpark application, usually you will have many dependencies & may also

have transitive dependencies. Sometimes a large application may need some python package that has C code to be compiled before install. And there are times when you want to run different versions of python for different app. For such scenarios with large PySpark application, `--py-files` is not so convenient. Luckily, in Python world, we can create virtual environment as an isolated Python runtime environment. Recently we have enabled virtual environment with PySpark in distributed environment. This makes the transition from local environment to distributed environment with PySpark smooth.
In this article, I will talk about how to use virtual environment in pyspark (This feature is only supported in yarn mode for now).

Prerequisites

- virtualenv and conda should be installed in the same location across the cluster. We support 2 approaches to set up virtual environment: virtualenv and conda. So each node must have either virtualenv or conda installed.
    - How to install virtualenv https://virtualenv.pypa.io/en/stable/installation/
    - How to install conda https://docs.continuum.io/anaconda/install

- Each node is internet accessible (for purpose of downloading packages)

- Python 2.7 or Python 3.x installed (pip is also installed)

Now I will talk about how to set up virtual environment in pyspark using virtualenv and conda. There're 2 scenario for using virtualenv in pyspark. One is batch mode where you launch pyspark app through spark-submit, another is interactive mode such as pyspark-shell or zeppelin pyspark interpreter.

Batch mode

For batch mode, I will follow the pattern of first developing in local environment and then moving to distributed environment, so that you can follow the same pattern for your development.

Use virtualenv

E.g. The following piece of code is what we'd like to develop. This piece of code use numpy.

```
from pyspark import SparkContext

if __name__ == "__main__":
    sc = SparkContext(appName="Simple App")
    import numpy as np
    sc.parallelize(range(1,10)).map(lambda x : numpy.__version__).collect()
```

At the first step, we will do it in local environment. We highly recommend user to create an isolated virtual environment locally first so that moving to distributed virtualenv would be much more smooth. We use the following command to create and setup env_1 in local

```
virtualenv env_1 -p /usr/local/bin/python3   # create virtual environment env_1. Folder
env_1 will be created under current working directory, you'd better to specify the python in
case you have multiple python installed.
source env_1/bin/activate                              # activate virtualenv
```

After that you can run pyspark in local mode where pyspark will run under this virtual environment env_1 and you will hit No module error. Because numpy is not installed in this virtual environment. So, now let's install numpy through pip

```
pip install numpy                                       # install numpy
```

After that you can use numpy in pyspark app launched by spark-submit in local environment

Now let's move this into distributed environment. There're 2 steps for moving from local development to distributed environment.

1. Create requirement file which contains the spec of your third party python dependencies. The following command will put all the installed python packages info of the current virtual environment into this file, so keep to stay in the virtual environment you created above.

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

   Here's sample output of requirement file.

   ```
   numpy==1.12.0
   ```

2. Run the pyspark app through spark-submit. Use the following commands to launch pyspark in yarn-client mode

```
bin/spark-submit --master yarn-client
--conf spark.pyspark.virtualenv.enabled=true
--conf spark.pyspark.virtualenv.type=native
--conf spark.pyspark.virtualenv.requirements=/Users/jzhang/github/spark/requirements.txt
--conf spark.pyspark.virtualenv.bin.path=/Users/jzhang/anaconda/bin/virtualenv
--conf spark.pyspark.python=/usr/local/bin/python3
spark_virtualenv.py
```

Use conda

Next, I will talk about how to do that via conda. It is very similar with virtualenv except using different commands.

This the command to create virtual environment in local

```
conda create --prefix env_conda_1 python=2.7   // create virtual environment env_conda_1
with python 2.7, folder env_conda_1 will be created under current working directory.
```

And use the following command to activate it.

```
source activate env_conda_1  // activate this virtual environment
```

The next thing is to install numpy via command

```
conda install numpy
```

Use the following command to create the requirement file. This command will put all the installed python packages info into this file, so keep to stay in this virtual environment you created above.

```
conda list --export > requirements_conda.txt
```

Run pyspark job in yarn-client mode

```
bin/spark-submit --master yarn-client
--conf spark.pyspark.virtualenv.enabled=true
--conf spark.pyspark.virtualenv.type=conda
--conf
spark.pyspark.virtualenv.requirements=/Users/jzhang/github/spark/requirements_conda.txt
--conf spark.pyspark.virtualenv.bin.path=/Users/jzhang/anaconda/bin/conda
```

```
Spark_virtualenv.py
```

Interactive mode

Interactive mode means you don't need to specify the requirement file when launching pyspark app, and can install packages in your virtualenv at runtime. The following command launch pyspark shell with virtualenv enabled. Both in driver & executor it will create an isolated virtual environment instead of using the default python of that host.

```
bin/pyspark --master yarn-client
--conf spark.pyspark.virtualenv.enabled=true
--conf spark.pyspark.virtualenv.type=native
--conf spark.pyspark.virtualenv.bin.path=/Users/jzhang/anaconda/bin/virtualenv
--conf spark.pyspark.python=/Users/jzhang/anaconda/bin/python
```

After you launch this pyspark shell, you will have clean python runtime environment on both driver and executors. You can use sc.install_packages to install any python packages that could could be installed by pip.

```
sc.install_packages("numpy")               # install the latest numpy
             sc.install_packages("numpy==1.11.0")        # install a specific version of
numpy
             sc.install_packages(["numpy", "pandas"])    # install multiple python packages
```

After that, you can use the packages that you just installed.

```
       import numpy
       sc.range(4).map(lambda x: numpy.__version__).collect()
```

Interactive mode with conda is almost the same. One exception is that you need to specify spark.pyspark.virtualenv.python_version because conda need to specify python version to create virtual environment.

```
bin/pyspark --master yarn-client
--conf spark.pyspark.virtualenv.enabled=true
--conf spark.pyspark.virtualenv.type=conda
```

```
--conf spark.pyspark.virtualenv.bin.path=/Users/jzhang/anaconda/bin/conda
--conf spark.pyspark.virtualenv.python_version=3.5
```

PySpark VirtualEnv Configurations

Property	Description
spark.pyspark.virtualenv.enabled	Property flag to enable virtualenv
spark.pyspark.virtualenv.type	Type of virtualenv. Valid values are “native”, “conda”
spark.pyspark.virtualenv.requirements	Requirement file (optional, not required for interactive mode)
spark.pyspark.virtualenv.bin.path	The location of virtualenv executable file for type native or conda executable file for type conda
spark.pyspark.virtualenv.python_version	Python version for conda. (optional, only required when you use conda in interactive mode)

Penalty of virtualenv

For each executor, it needs to take some time to setup the virtualenv (installing the packages), and for the first time, it may be very slow. e.g. The first time I install numpy on each node it takes almost 3 minutes, because it needs to download it and compiling it to wheel format. But for the next time, it only takes 3 seconds to install numpy, because it would install the numpy from the cached wheel file.