Developing Dockerized Data Apps

Datapalooza Austin 2016, #RockYourData, @dwhitena

Daniel Whitenack
Data Scientist at Telnyx, Freelancer, Mentor with Thinkful

Outline

1. Docker intro.

2. Challenges facing the data science community.

3. How Docker helps us overcome these challenges.

4. Examples (including building a dockerized data app start to finish).

5. Resources.

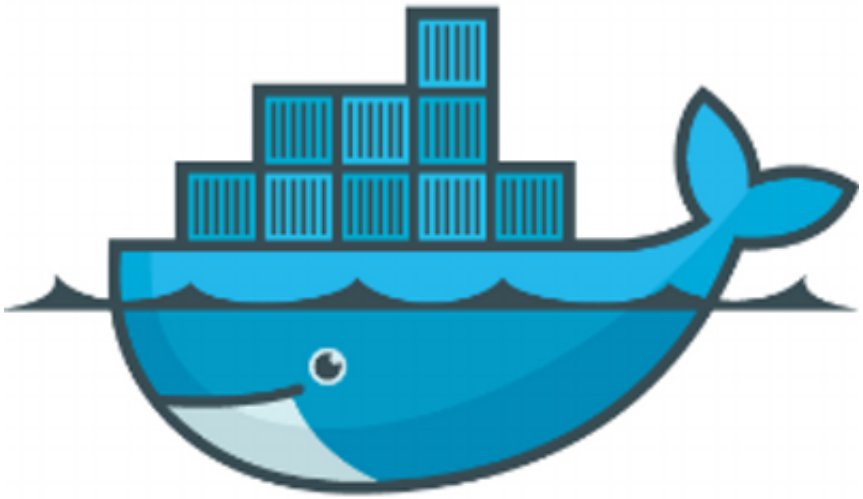
(slides are here: https://github.com/dwhitena/slides (https://github.com/dwhitena/slides/tree/master/datapalooza2016))

Docker Intro

Docker Intro:

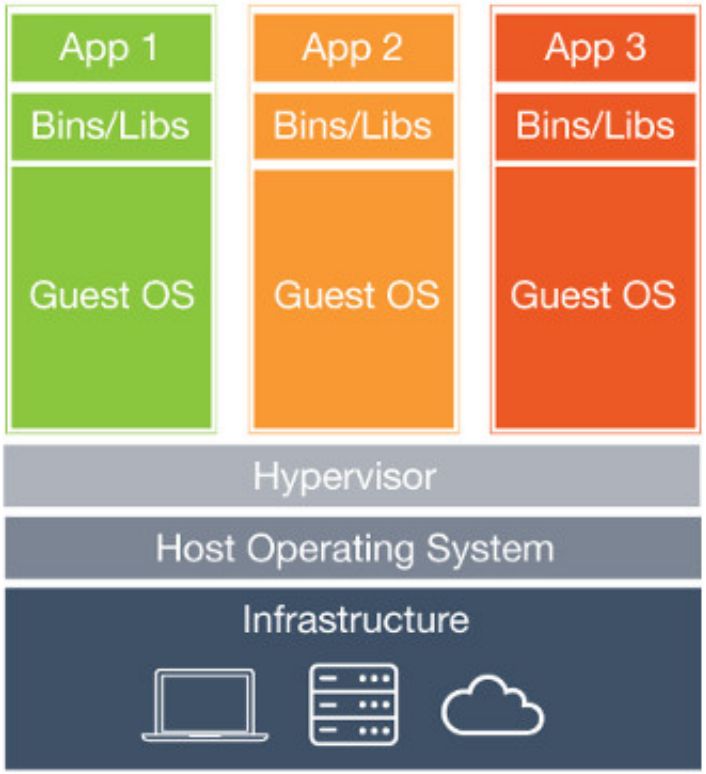
Docker is an open source project that:

- wraps a piece of software
- in a complete filesystem
- that contains everything it needs to run (i.e., anything you can install on a server)



Docker Intro:

Isn't that a virtual machine? Not quite.



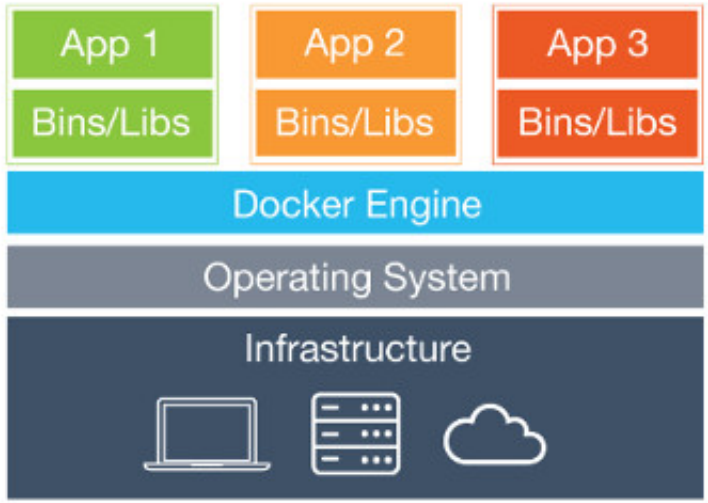


image from Docker (https://www.docker.com/what-docker)

Docker Intro:

A virtual machine includes:

- the application
- necessary binaries and libraries
- an entire guest operating system
- 10's of GB

A Docker "container" includes:

- the application
- its dependencies
- 10's of MB

Docker Intro:

A Docker container:

- runs as an isolated process in userspace on the host OS
- shares the kernel with other containers

Thus, Docker containers have similar isolation and allocation benefits as virtual machines but are much more portable and efficient.

Docker Intro:

Docker jargon:

- **docker engine -** builds and runs your Docker containers
- **image -** the basis of containers
- **container -** an instance of an image
- **docker compose -** a tool for defining and running multi-container Docker applications
- **docker swarm, kubernetes, mesos, ... -** orchestration tools that help start containers on appropriate hosts and connect them together.
- **registry -** a repository for images (note Docker provides a hosted "registry" called Docker Hub)

Docker Intro:

Cue example...

Docker Intro:

Installation:

- general instructions (https://docs.docker.com/engine/installation/)
- Ubuntu (https://docs.docker.com/engine/installation/linux/ubuntulinux/)
- OSX (https://docs.docker.com/engine/installation/mac/)

Challenges facing the data science community

Challenges facing the data science community:

"There was only one problem—all of my work was done in my local machine in R. People appreciate my efforts but **they don't know how to consume my model because it was not 'productionized'** and the infrastructure cannot talk to my local model. Hard lesson learned!" *-Robert Chang, data scientist at Twitter*

Challenges facing the data science community:

"Data engineers are often frustrated that data scientists produce **inefficient and poorly written code, have little consideration for the maintenance cost of productionizing ideas,** demand unrealistic features that skew implementation effort for little gain… The list goes on, but you get the point." *-Jeff Magnusson, director of data platform at Stitchfix*

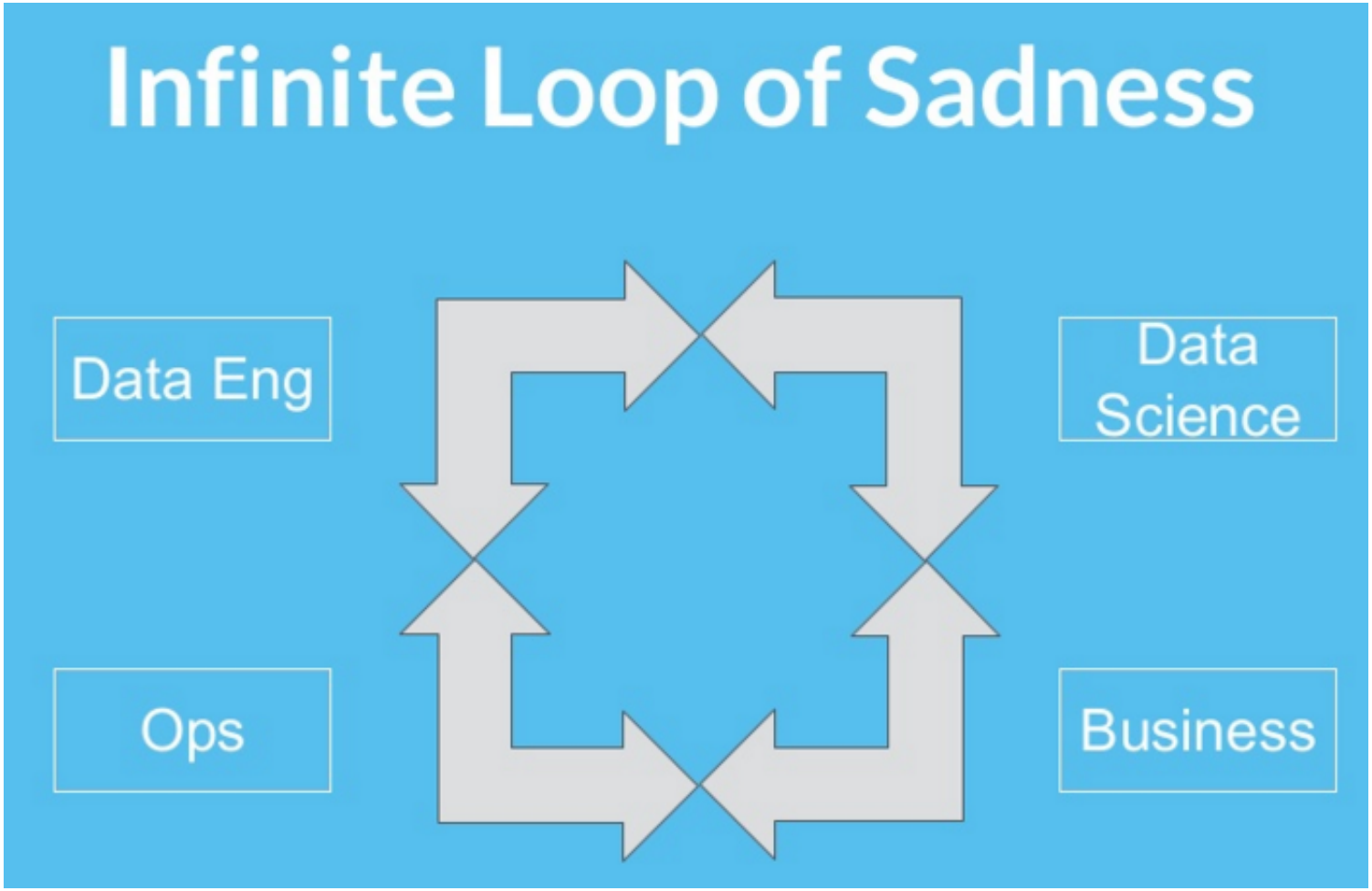
Challenges facing the data science community:

"Ever **tried to reproduce an analysis** that you did a few months ago or even a few years ago? You may have written the code, but it's now **impossible to decipher** whether you should use *makefigures.py.old*, *makefiguresworking.py* or *newmakefigures01.py* to get things done."

- "Are we supposed to go in and join the "region" column to the data before we get started or did that come from one of the notebooks?"
- "Come to think of it, which notebook do we have to run first before running the plotting code: was it 'process data' or 'clean data'?"
- "Where did the shapefiles get downloaded from for the geographic plots you made?"

from the Cookiecutter Data Science (http://drivendata.github.io/cookiecutter-data-science/) project

Challenges facing the data science community:



via Josh Wills (https://twitter.com/josh_wills), Head of Data Engineering, Slack

Challenges facing the data science community:

1. Deployment and dependencies

2. Reproducibility

3. Scaling, efficiency, orchestration

How Docker helps us overcome these challenges

How Docker helps: Deployment and dependencies

Instead of saying to Devops, "I need a box with these versions of these dependencies installed" (which I assure you will only contribute the the infinite loop of sadness)...

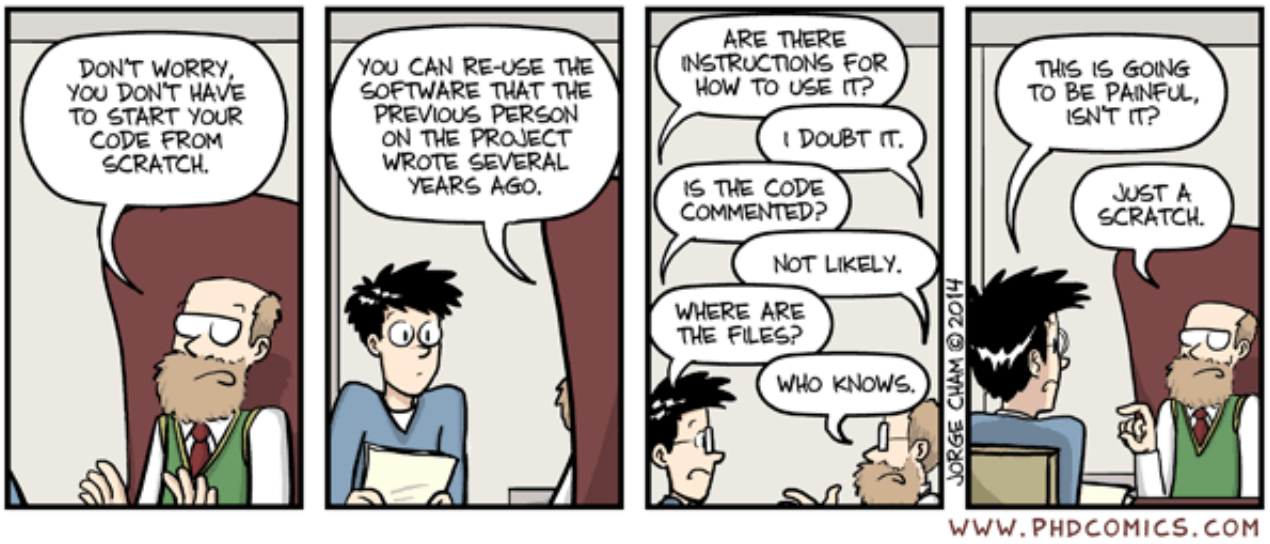
How Docker helps: Deployment and dependencies

say "run this container"

```
docker run -d registry:5000/myservice:latest
```

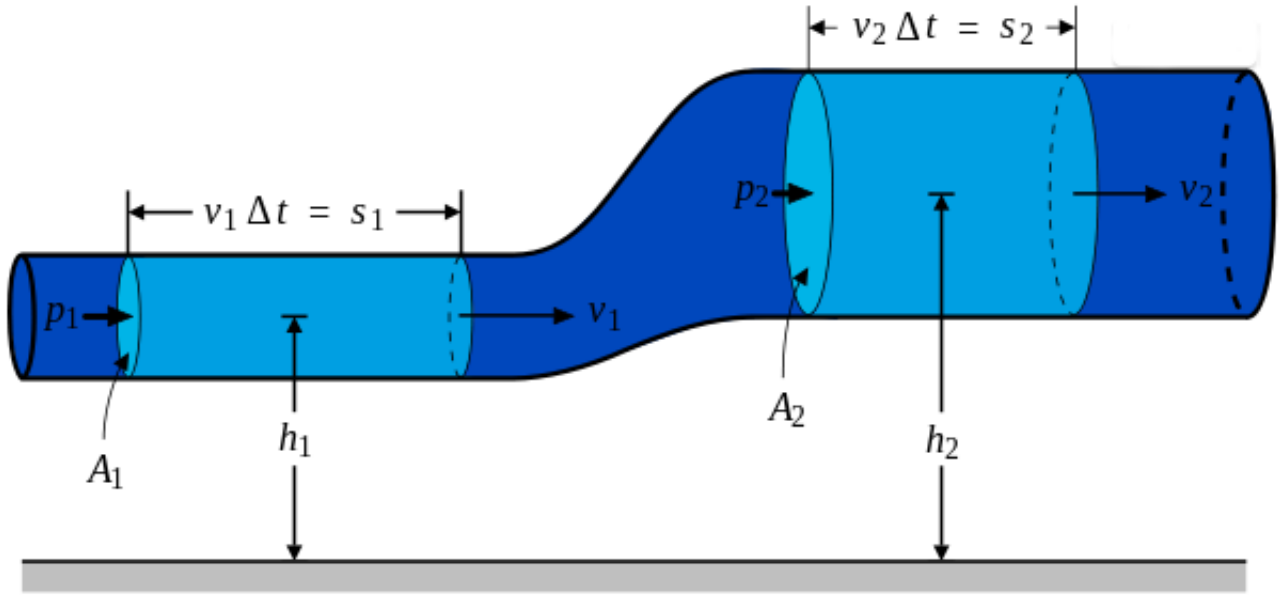
How Docker helps: Reproducibility

"This is the real value of containers in data science: the ability to capture an experiment's state (data, code, results, package versions, parameters, etc) at a point in time, **making it possible to reproduce an experiment at any stage** in the research process." *-Daniel Chalef, Domino Data Lab*



How Docker helps: Scaling, efficiency, orchestration

By virtue of the portability and efficiency of docker containers, they enable the orchestration of truly innovative, reliable, and efficient data science pipelines.



Examples

Example 1: Deployment and dependencies

Example 1: Deployment and dependencies

We are going to use:

- python (https://www.python.org/)
- scikit-learn (http://scikit-learn.org/stable/)
- flask-restful (http://flask-restful-cn.readthedocs.org/en/0.3.4/)

to develop a dockerized data science app that makes predictions based on the famous Iris Data Set (https://en.wikipedia.org/wiki/Iris_flower_data_set).

Sepal length	Sepal width	Petal length	Petal width	Species
5.1	3.5	1.4	0.2	*I. setosa*
4.9	3.0	1.4	0.2	*I. setosa*
4.7	3.2	1.3	0.2	*I. setosa*
4.6	3.1	1.5	0.2	*I. setosa*

Example 1: Deployment and dependencies

Let's start with the machine learning piece:

```
from sklearn import datasets
from sklearn.neighbors import KNeighborsClassifier

def predict(inputFeatures):

    iris = datasets.load_iris()
    knn = KNeighborsClassifier()
    knn.fit(iris.data, iris.target)

    predictInt = knn.predict(inputFeatures)
    if predictInt[0] == 0:
        predictString = 'setosa'
    elif predictInt[0] == 1:
        predictString = 'versicolor'
    elif predictInt[0] == 2:
        predictString = 'virginica'
    else:
        predictString = 'null'

    return predictString
```

Example 1: Deployment and dependencies

Now, how are we going to request predictions and receive the results?

Example 1: Deployment and dependencies

JSON API of course.

Example 1: Deployment and dependencies

So let's build the JSON API piece.

First, imports flask-restful piecies and define our "app:"

```
from flask import Flask
from flask_restful import Resource, Api
from flask_restful import reqparse
from utils import makeprediction

app = Flask(__name__)
api = Api(app)
```

Example 1: Deployment and dependencies

Then implement a class for handling requests to a prediction endpoint:

```
class Prediction(Resource):
    def get(self):

        parser = reqparse.RequestParser()
        parser.add_argument('slength', type=float, help='slength cannot be converted')
        parser.add_argument('swidth', type=float, help='swidth cannot be converted')
        parser.add_argument('plength', type=float, help='plength cannot be converted')
        parser.add_argument('pwidth', type=float, help='pwidth cannot be converted')
        args = parser.parse_args()

        prediction = makeprediction.predict([args['slength'], args['swidth'],
            args['plength'], args['pwidth']])

        print "THE PREDICTION IS: " + str(prediction)

        return {'slength': args['slength'], 'swidth': args['swidth'],
            'plength': args['plength'], 'pwidth': args['pwidth'],
            'species': prediction}
```

Example 1: Deployment and dependencies

And add the resource to our "app:"

```
api.add_resource(Prediction, '/prediction')
```

Now if we GET,

```
http://<host>:5000/prediction?slength=1.5&swidth=0.7&plength=1.3&pwidth=0.3
```

we should get a response that looks like this:

```
{
    "pwidth": 0.3,
    "plength": 1.3,
    "slength": 1.5,
    "species": "setosa",
    "swidth": 0.7
}
```

Example 1: Deployment and dependencies

To "Docker-ize" our app we need a "Dockerfile." In this case, it looks like this:

```
FROM ubuntu:12.04

# get up pip, vim, etc.
RUN apt-get -y update --fix-missing
RUN apt-get install -y python-pip python-dev libev4 libev-dev
RUN apt-get install -y gcc libxslt-dev libxml2-dev libffi-dev vim curl
RUN pip install --upgrade pip

# get numpy, scipy, scikit-learn and flask and add our project
RUN apt-get install -y python-numpy python-scipy
RUN pip install scikit-learn
RUN pip install flask-restful
ADD . /

# expose the port for the API
EXPOSE 5000

# run the API
CMD [ "python", "/api.py" ]
```

Example 1: Deployment and dependencies

From the root of the repo containing this work we can build a **Docker image** of our app (assuming Docker is installed) by running:

```
docker build -t pythoniris .
```

where:

- **docker build** tells docker-engine to build a Docker image,
- **-t pythoniris** tells us to "tag" our Docker image as "pythoniris," and
- **.** tells docker-engine to build the image based on the contents of the current directory as specified by whatever "Dockerfile" it finds.

Example 1: Deployment and dependencies

Remember however that a Docker image just wraps up our project with a file system, dependencies, etc. We still need to "run" this Docker image as a **container** somewhere. In this case we will run it with:

```
docker run --net host -p 5000:5000 --name pythoniris -d pythoniris
```

where,

- **docker run** tells docker-engine to run a container based on the "pythoniris" image (as specified by the last argument),
- **--net host** specifies that the network adapter of the container should be mapped to the network adapter of the host,
- **-p 5000:5000** maps port 5000 of the container to port 5000 of the host, and
- **--name pythoniris** names our container "pythoniris."

Example 1: Deployment and dependencies

There you have it! you could "pull" down this image to any machine running Docker (regardless of OS or installed dependencies) and run our data science app as specified above.

We can deploy anywhere in one command.

Example 1: Deployment and dependencies

Assuming we ran this command on localhost, if we GET

```
http://localhost:5000/prediction?slength=1.5&swidth=0.7&plength=1.3&pwidth=0.3
```

we will receive the response:


```
{
    "pwidth": 0.3,
    "plength": 1.3,
    "slength": 1.5,
    "species": "setosa",
    "swidth": 0.7
}
```

Example 1: Deployment and dependencies

Let's try it out...

Example 2: Reproducibility

Example 2: Reproducibility

Well, doing what we just did with our data science app already moves us quite a ways towards reproducibility.

That is, our Docker image captures our experiment's state (data, code, results, package versions, parameters, etc) as long as those things are included in our container.

We can even utilize the "tags" mentioned before to tag different stages of experimentation,

```
dwhitena/pythoniris:latest

dwhitena/pythoniris:staging

dwhitena/pythoniris:production
```

Example 2: Reproducibility

But what if,

- we want to keep our image size small, or we don't want to keep our training data in our container for any other reason.
- we don't want to rely our correct/understandable human tagging of images.

What we really want is a Git-like repository for data, so we know what at our data looked at any stage of research and experimentation, or at any given time of a production run.

Example 2: Reproducibility

Well, there are a couple of Docker-based projects that provide exactly this!

My favorite is called Pachyderm (http://pachyderm.io).

Example 2: Reproducibility

Pachyderm provides:

- PFS - a file system that supports versioned commits of data (no matter how big that data is), and
- PPS - a pipelining system that supports automated analyses based on changes in data that is under PFS version control.

And it all runs inside of Docker containers of course!

Example 2: Reproducibility

Let's say that we want to store a verion of the iris data set. Let's create a repository for the iris data in PFS (at "/pfs" by default, and using pachyderm's CLI tool pachctl):

```
pachctl create-repo data
```

then open a commit,

```
pachctl start-commit data
```

which will return a commit ID:

```
c766f66801c5477e8a4c7e4088949d90
```

Example 2: Reproducibility

We can then push data into the repo

```
cp iris.csv /pfs/data/c766f66801c5477e8a4c7e4088949d90/irisdataset
```

and close the commit

```
pachctl finish-commit data c766f66801c5477e8a4c7e4088949d90
```

Now that version of "iris.csv" is both stored and versioned. We can commit new versions of "iris.csv" to the same repo, in the same fashion, over and over, with each change being tracked and recoverable.

Example 2: Reproducibility

That sort of data versioning is pretty useful in and of itself, but the real power comes from the fact that you can also version results and tie together commits of new training/prediction data with commits of corresponding results.

Pachyderm orchestrates and automates all of this (via pachyderm running inside of Docker and Kubernetes).

Example 2: Reproducibility

Imagine:

- receive new data -> commit data (with commit_id_1)
- transform data into features -> commit features (with commit_id_2)
- calculate prediction based on the features -> commit prediction (with commit_id_3)

Now at any point in the future you immediately see what data, produced what features, and resulted in what prediction by getting commit_id_(1,2,3)

(Aside)

Also, with Pachyderm, you can automatically spawn this pipeline (i.e., make it a streaming pipeline) upon a commit of new data.

When the pipeline is spawned it will only run stages of the pipeline that need to be rerun to recompute the result.

Thus providing huge boosts in efficiency in addition to reproducibility.

Example 3: Scaling, efficiency, orchestration

Example 3: Scaling, efficiency, orchestration

Now, in another scenario, let's image that our business is scaling and our data engineers have implemented a high throughput data pipeline that will send us millions of requests for prediction on a queue such as Kafka.

How are we going to correspondingly scale our Dockerized data science app?

Example 3: Scaling, efficiency, orchestration

Simple. Assuming you have modified our app to consume from Kafka, just spin up a bunch of docker containers based on your *pythoniris* image.

```
docker run --net host --name pythoniris1 -d pythoniris
docker run --net host --name pythoniris2 -d pythoniris
docker run --net host --name pythoniris3 -d pythoniris
docker run --net host --name pythoniris4 -d pythoniris
```

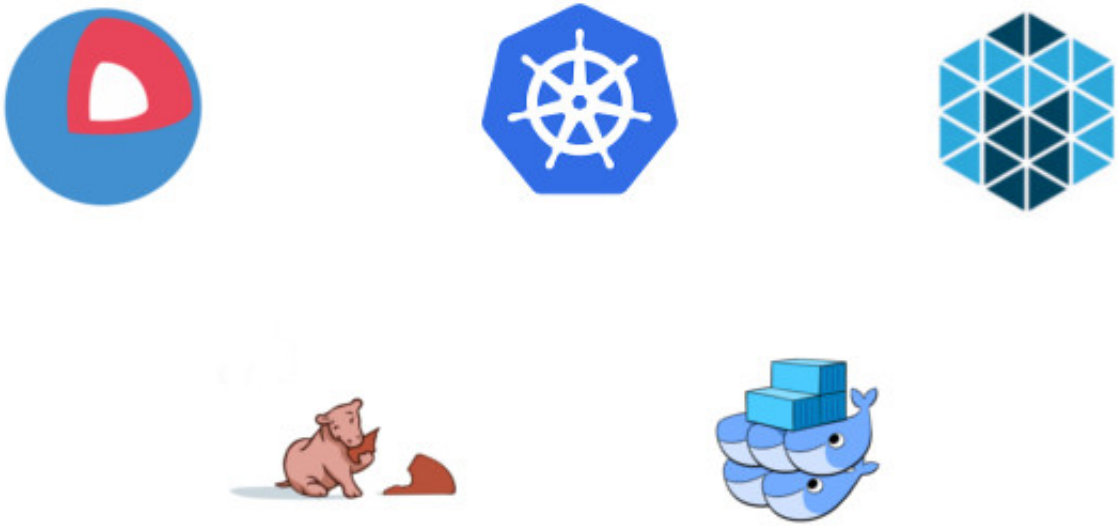
These containers are lightweight, and it's so easy to create more instances.

Plus, as the load scales back down, it's easy to remove the containers.

```
docker rm -f pythoniris2
docker rm -f pythoniris3
docker rm -f pythoniris4
```

Example 3: Scaling, efficiency, orchestration

Actually, if you want these sorts of things to be automated, there are tools for that as well: Kubernetes, Docker Swarm, Mesos, Pachyderm, etc.



Resources

Resources

Getting started with Docker:

- install (https://docs.docker.com/engine/installation/)
- CLI tools walkthrough (https://coreos.com/os/docs/latest/getting-started-with-docker.html)
- Docker Hub (https://docs.docker.com/docker-hub/)
- the Docker ecosystem (https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components)

Resources

Getting started with Pachyderm:

- general info (http://pachyderm.io/)
- PFS (http://pachyderm.io/pfs.html)
- PPS (http://pachyderm.io/pps.html)
- quick start guide (https://github.com/pachyderm/pachyderm/blob/master/examples/fruit_stand/GUIDE.md)

Resources

Pre-dockerized data science and big data tools/frameworks:

- python data science (https://www.dataquest.io/blog/data-science-quickstart-with-docker/)
- Anaconda (https://hub.docker.com/r/continuumio/anaconda/)
- Jupyter (https://hub.docker.com/r/jupyter/notebook/)
- Spark (https://hub.docker.com/r/sequenceiq/spark/)
- TensorFlow (https://hub.docker.com/r/tensorflow/tensorflow/)

Thank you

Daniel Whitenack
Data Scientist at Telnyx, Freelancer, Mentor with Thinkful
http://datadan.io/ (http://datadan.io/)
@dwhitena (http://twitter.com/dwhitena)