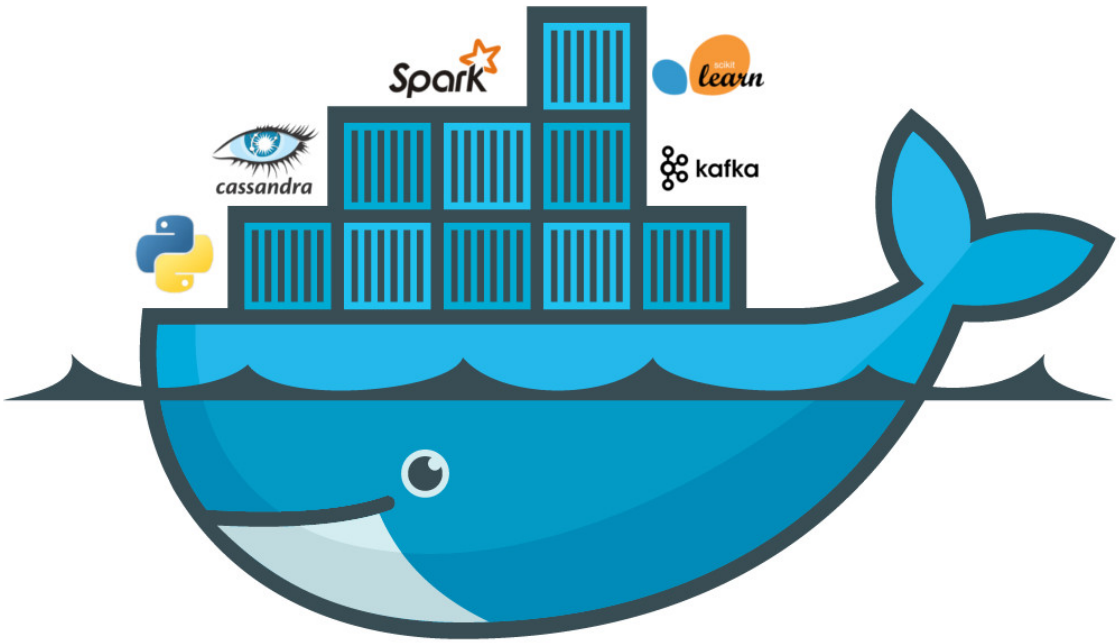


How Docker Can Help You Become A More Effective Data Scientist

Hamel Husain Follow
Dec 17, 2017 · 14 min read

By Hamel Husain

For the past 5 years, I have heard lots of buzz about docker containers. It seemed like all my software engineering friends are using them for developing applications. I wanted to figure out how this technology could make me more effective but I found tutorials online either too detailed: elucidating features I would never use as a data scientist, or too shallow: not giving me enough information to help me understand how to be effective with Docker quickly.

I wrote this quick primer so you don't have to parse all the information out there and instead can learn the things you need to know to quickly get started.

What is Docker?

You can think of Docker as lightweight virtual machines — that contain everything you need to run an application. A docker container can capture a snapshot of the state of your system so that someone else can quickly re-create your compute environment. That's all you need to know for this tutorial, but for more detail you can head here.

Why should you use docker?

1. **Reproducibility**: It is really important that your work is reproducible as a professional data scientist. Reproducibility not only facilitates peer review, but ensures the model, application or analysis you build can run without friction which makes your deliverables more robust and withstand the test of time. For example, if you built a model in python it is often not enough to just run just pip-freeze and send the resulting requirements.txt file to your colleague as that will only encapsulate python specific dependencies — whereas there are often dependencies that live outside python such as operating systems, compilers, drivers, configuration files or other data that are required for your code to run successfully. Even if you can get away with just sharing python dependencies, wrapping everything in a Docker container reduces the burden on others of recreating your environment and makes your work more accessible.

2. **Portability of your compute environment**: As a data scientist, especially in machine learning, being able to rapidly change your compute environment can drastically effect your productivity. Data science work often begins with prototyping, exploring and research — work that doesn't necessarily require specialized computing resources right off the bat. This work often occurs on a laptop or personal computer. However, there often comes a moment where different compute resources would drastically speed up your workflow —for example a machine with more CPUs or a more powerful GPU for things like deep learning. I see many data scientists limit themselves to their local compute environment because of a perceived friction of re-creating their local environment on a remote machine. Docker makes the process of porting your environment (all of your libraries, files, etc.) very easy. Porting your compute environment quickly is also a huge competitive advantage in Kaggle competitions because you can take advantage of precious compute resources on AWS in a cost effective manner. Lastly, creating a docker file allows you to port many of

the things that you love about your local environment — such as bash aliases or vim plugins.

3. **Strengthen your engineering chops**: being comfortable with Docker can allow you to deploy your models or analyses as applications (for example as a REST API endpoint that can serve predictions) that make your work accessible to others. Furthermore, other applications that you may need to interact with as part of your data science workflow may exist in a Docker container such as a database or other application.

Docker Terminology

Before we dive in, its helpful to be familiar with Docker terminology:

- **Image**: Is a blueprint for what you want to build. Ex: Ubuntu + TensorFlow with Nvidia Drivers and a running Jupyter Server.
- **Container**: Is an instantiation of an image that you have brought to life. You can have multiple copies of the same image running. It is really important to grasp the difference between an image and a container as this is a common source of confusion for new comers. If the difference between an image and a container isn't clear, STOP and read again.
- **Dockerfile**: Recipe for creating an Image. Dockerfiles contain special Docker syntax. From the official documentation: A `Dockerfile` is a text document that contains all the commands a user could call on the command line to assemble an image.
- **Commit**: Like git, Docker containers offer version control. You can save the state of your docker container at anytime **as a new image** by committing the changes.
- **DockerHub / Image Registry**: Place where people can post public (or private) docker images to facilitate collaboration and sharing.
- **Layer**: modification to an existing image, represented by an instruction in the Dockerfile. Layers are applied in sequence to the base image to create the final image.

I will be using this terminology throughout the rest of the post so refer back to this list if you get lost! It is easy to get confused between these terms, especially between images and containers — so be vigilant while you are reading!

Install Docker

You can download and install Docker Community Edition for free. You can follow the instructions here.

Create Your First Docker Image

Before you create a docker container, it is useful to create a Dockerfile that will define the image. Let's go through the below Dockerfile slowly. *You can find this file on the accompanying Github repo for this tutorial.*

```
# reference: https://hub.docker.com/_/ubuntu/
FROM ubuntu:16.04

# Adds metadata to the image as a key value pair example LABEL version="1.0"
LABEL maintainer="Hamel Husain <youremail@gmail.com>"

##Set environment variables
ENV LANG=C.UTF-8 LC_ALL=C.UTF-8

RUN apt-get update --fix-missing && apt-get install -y wget bzip2 ca-certificates \
    build-essential \
    byobu \
    curl \
    git-core \
    htop \
    pkg-config \
    python3-dev \
    python-pip \
    python-setuptools \
    python-virtualenv \
    unzip \
    && \
apt-get clean && \
rm -rf /var/lib/apt/lists/*

RUN echo 'export PATH=/opt/conda/bin:$PATH' > /etc/profile.d/conda.sh && \
    wget --quiet https://repo.continuum.io/archive/Anaconda3-5.0.0.1-Linux-x86_64.sh -O
```

```
    /bin/bash ~/anaconda.sh -b -p /opt/conda && \
    rm ~/anaconda.sh

ENV PATH /opt/conda/bin:$PATH

RUN pip --no-cache-dir install --upgrade \
        altair \
        sklearn-pandas

# Open Ports for Jupyter
EXPOSE 7745

#Setup File System
RUN mkdir ds
ENV HOME=/ds
ENV SHELL=/bin/bash
VOLUME /ds
WORKDIR /ds
ADD run_jupyter.sh /ds/run_jupyter.sh
RUN chmod +x /ds/run_jupyter.sh

# Run a shell script
CMD  ["./run_jupyter.sh"]
```

Dockerfile hosted with ❤️ by GitHub view raw

The FROM statement

```
FROM ubuntu:16.04
```

The **FROM** statement encapsulates the most magical part of Docker. This statement specifies the base image you want to build on top of. Upon specifying a base image with **FROM,** Docker will look in your local environment for an image named **ubuntu:16.04** — and if it cannot find it locally it will search your designated Docker Registry which by default is DockerHub. This layering mechanism is convenient as you often want to install your programs on top of an operating system such as Ubuntu. Rather than worrying about how to install Ubuntu from scratch, you can simply build on top of the official Ubuntu image! There are a wide variety of Docker images hosted on Dockerhub,

including those that provide more than an operating system, for example if you want a container with Anaconda already installed you can build a container on top of the official anaconda docker image. Most importantly, you can also publish an image you have built at anytime even if that image has been created by layering on top of another image! The possibilities are endless.

In this example, we are specifying that our base image is **ubuntu:16.04** which will look for a DockerHub repo called ubuntu. The part of the image name after the colon — 16.04 is the **tag** which allows you to specify what version of the base image you want to install. If you navigate to the Ubuntu DockerHub repo, you will notice that different versions of Ubuntu correspond with different tags:

Screenshot of the official Ubuntu DockerHub repo as of December 2017.

For example **ubuntu:16.04**, **ubuntu:xenial-20171201**, **ubuntu:xenial**, and **ubuntu:latest** all refer to Ubuntu version 16.04 and are all aliases for the same image. Furthermore, the links provided in this repository link you to the corresponding Dockerfiles that were used to build the images for each version. You will not always find Dockerfiles on DockerHub repositories as it is optional for the maintainer to include the Dockerfile on how they made the image. I personally found it useful to look at several of these Dockerfiles to understand Dockerfiles more (but wait until you are finished with this tutorial!)

There is one tag that deserves special mention — the **:latest** tag. This tag specifies what you will pull by default if you do not specify a tag in your **FROM** statement. For example, if your FROM statement looks like this:

```
FROM ubuntu
```

Then you will just end up pulling the ubuntu:16.04 image. Why? — If you look closely at the above screenshot, you will see the :latest tag is associated with 16.04

One last note about Docker images: exercise sensible judgment when pulling random Docker images from DockerHub. Docker images created by a nefarious actor could potentially contain malicious software.

The LABEL statement

This statement adds metadata to your image, and is completely optional. I add this such that others know who to contact about the image and also so I can search for my docker containers, especially when there are many of them running concurrently on a server.

```
LABEL maintainer="Hamel Husain <youremail>"
```

The ENV statement

```
ENV LANG=C.UTF-8 LC_ALL=C.UTF-8
```

This allows you to change environment variables and is pretty straightforward. You can read more about this here.

The RUN statement

This is usually the workhorse of accomplishing what you want to in building a Docker image. You can run arbitrary shell commands like *apt-get* and *pip install* to install the packages and dependencies you want.

```
RUN apt-get update --fix-missing && apt-get install -y wget bzip2
    build-essential \
    ca-certificates \
    git-core \

...
```

In this case, I'm installing some utilities that I like such as curl, htop, byobu and then installing anaconda, followed by other libraries that do not come in the base anaconda install (scroll up to the full Dockerfile to see all of the RUN statements).

The commands after the **RUN** statement have nothing to do with Docker but are normal linux commands that you would run if you were installing these packages yourself, so do not worry if you aren't familiar with some of these packages or linux commands. Also, as a further piece of advice — when I first started learning about docker I looked at other Dockerfiles on Github or DockerHub and copy and pasted relevant parts that I wanted in my Dockerfile.

One thing you may notice about the RUN statement is the formatting. Each library or package is neatly indented and arranged in alphabetical order for readability. This is a prevalent convention for Dockerfiles so I suggest you adopt this as it will ease collaboration.

The EXPOSE statement

This statement is helpful if you are trying to expose a port — for example if you are serving a jupyter notebook from inside the container or some kind of web-service. Docker's documentation is quite good in explaining the **EXPOSE** statement:

> *The `EXPOSE` instruction does not actually publish the port. It functions as a type of documentation between the person who builds the image and the person who runs the container, about which ports are intended to be published. To actually publish the port when running the container, use the `-p` flag on `docker run` to publish and map one or more ports, or the `-P` flag to publish all exposed ports and map them to to high-order ports.*

The VOLUME statement

```
VOLUME /ds
```

This statement allows you to share data between your docker container and the host computer. The VOLUME statements allow you to mount externally mounted volumes. The host directory is declared only when a container is run (because you might run this

container on different computers), not when the image is being defined*. For right now, you only specify the name of the folder within the docker container that you would like to share with the host container.

From the docker user guide:

> *__The host directory is declared at container run-time__: The host directory (the mountpoint) is, by its nature, host-dependent. This is to preserve image portability. since a given host directory can't be guaranteed to be available on all hosts.For this reason, you can't mount a host directory from within the Dockerfile. The `VOLUME` instruction does not support specifying a `host-dir` parameter. You must specify the mountpoint when you create or run the container.*

Furthermore, these volumes are meant to persist data outside the filesystem of a container, which often useful if you are working with large amounts of data that you do not want to bloat the docker image with. When you save a docker image, any data in this **VOLUME** directory will not be saved as part of the image, however data that exists outside this directory in the container will be saved.

The WORKDIR statement

```
WORKDIR /ds
```

This statement sets the working directory incase you want to reference a specific file without absolute paths in another command. For example, the last statement in the Dockerfile is

```
CMD [“./run_jupyter.sh”]
```

Which assumes the working directory is /ds

The ADD statement

```
ADD run_jupyter.sh /ds/run_jupyter.sh
```

This command allows you to copy files from the host computer into the docker container when the docker container is run. I use this to execute bash scripts and import useful things into the container such as .bashrc files.

Notice how the path of the host container is not fully specified here, as the host path is relative to the *context directory* that you specify when the *container* is run (which is discussed later).

It just so happens that I will have the file *run_jupyter.sh* in the root of the context directory when I run this container, so that is why there is no path in front of the source file.

From the user guide:

> `ADD <src>... <dest>`
>
> *The* `ADD` *instruction copies new files, directories or remote file URLs from* `<src>` *and adds them to the filesystem of the image at the path* `<dest>`.

The CMD statement

Docker containers are designed with the idea that they are ephemeral and will only stay up long enough to finish running the application you want to run. However, for data science we often wish to keep these containers running even though there is nothing actively running in them. One way that many people accomplish this by simply running a bash shell (which doesn't terminate unless you kill it).

```
CMD [“./run_jupyter.sh”]
```

In the above command I am running a shell script that instantiates a Jupyter notebook server. However, if you do not have any specific application you want to run but you want your container to run without exiting — you can simply run the bash shell instead with the following command:

```
CMD ["/bin/bash"]
```

This works because the bash shell does not terminate until you exit out of it, thus the container stays up and running.

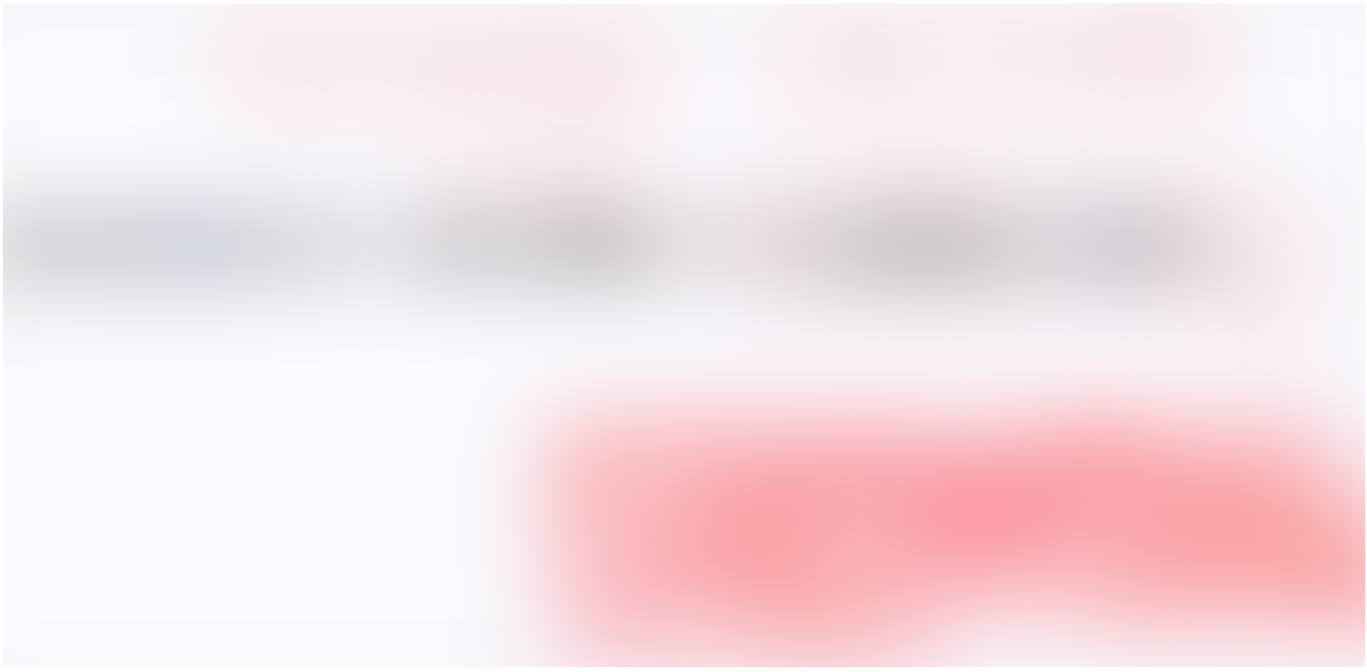
From the user guide:

> *There can only be one* `CMD` *instruction in a* `Dockerfile`*. If you list more than one* `CMD` *then only the last* `CMD` *will take effect.*
>
> ***The main purpose of a*** `CMD` ***is to provide defaults for an executing container.*** *These defaults can include an executable, or they can omit the executable, in which case you must specify an* `ENTRYPOINT` *instruction as well.*

. . .

Build your Docker image

Phew, that was lots of information about Dockerfiles. Don't worry, everything else is fairly straightforward from here. Now that we have created our recipe in the form of a DockerFile, its time to build an image. You can accomplish this via the following command:



Also available on Github

This will build a docker image (not a container, read the terminology in the beginning of this post if you don't remember what the difference is!), which you can then run at a later time.

. . .

Create and run a container from your Docker image

Now you are ready to put all this magic to work. We can bring up this environment by executing the following command:

Also available on Github

After you run this, your container will be up and running! The jupyter server will be up in running because of the

```
CMD ["./run_jupyter.sh"]
```

command at the end of the Dockerfile. Now you should be able to access your jupyter notebook on the port it is serving on — in this example it should be accessible from http://localhost:7745/ with the password *tutorial*. If you are running this docker

container remotely, you will have to setup local port forwarding so that you can access the jupyter server from your browser.

. . .

Interacting With Your Container

Once your container is up and running, these commands will come in handy:

- **Attach a new terminal session to a container**. This is useful if you need to install some software or use the shell.

. . .

- **Save the state of your container as a new image**. Even though you started out with a Dockerfile with all the libraries you wanted to install, over time you may significantly change the state of the container by adding more libraries and packages interactively. It is useful to save the state of your container as an image that you can later share or layer on top of. You can accomplish this by using the **docker commit** CLI command:

```
docker commit <container_name> new_image_name:tag_name(optional)
```

For example, if I wanted to save the state of the container called *container1* as an image called hamelsmu/tutorial:v2, I would simply run this command:

```
docker commit container_1 hamelsmu/tutorial:v2
```

You might be wondering why *hamelsmu/* is in front of the image name — this just makes it easier to push this container to DockerHub later on, as hamelsmu is my DockerHub username (more about this later). If you are using Docker at work, it is likely that there is an internal private Docker repo that you can push your Docker images to instead.

. . .

- **List running containers**. I often use this when I have forgot the name of the container that is currently running.

```
docker ps –a –f status=running
```

If you run the above command without the status=running flag, then you will see a list of all the containers (even if they are no longer running) on your system. This can be useful for tracking down an old container.

. . .

- **List all images** that you have saved locally.

```
docker images
```

. . .

- **Push your image to DockerHub (or another registry)**. This is useful if you want to share your work with others, or conveniently save an image in the cloud. Be careful that you do not share any private information while doing this (there are private repos available on DockerHub, too).

First create a DockerHub repository and name your image appropriately, as described here. This will involve running the command **docker login** to first connect to your account on DockerHub or other registry. For example, to push an image to this container, I first have to name my local image as hamelsmu/tutorial (I can choose any tag name) For example, the CLI command:

```
docker push hamelsmu/tutorial:v2
```

Pushes the aforementioned docker image to this repository with the tag **v2**. It should be noted that if you make your image publicly available **others can simply layer on top of your image** just like we added layers to the **ubuntu** image in this tutorial. This is quite useful for other people seeking to reproduce or extend your research.

. . .

Now You Have Superpowers

Now that you know how to operate Docker, you can perform the following tasks:

- Share reproducible research with colleagues and friends.
- Win Kaggle competitions without going broke, by moving your code to larger compute environments temporarily as needed.
- Prototype locally inside a docker container on your laptop, and then seamlessly move the same computation to a server without breaking a sweat, while taking many of

the things you love about your local environment with you (your aliases, vim plugins, bash scripts, customized prompts, etc).

- Quickly instantiate all the dependencies required to run Tensorflow, Pytorch or other deep learning libraries on a GPU computer using Nvidia-Docker (which can be painful if you are doing this from scratch). See the bonus section below for more info.

- Publish your models as applications, for example as a rest api that serves predictions from a docker container. When your application is Dockerized, it can be trivially replicated as many times as needed.

Further Reading

We only scratched the surface of Docker, and there is so much more you can do. I focused on the areas of Docker that I think you will encounter most often as a Data Scientist and hopefully gave you enough confidence to start using it. Below are some resources that helped me on my Docker journey:

- Helpful Docker commands
- More helpful Docker commands
- Dockerfile reference
- How to create and push to a repository on DockerHub

Bonus: Nvidia-Docker

The original motivation for me to learn Docker in the first place was to prototype deep learning models on a single GPU and move computation to AWS once I needed more horsepower. I was also taking the excellent course Fast.AI by Jeremy Howard and wanted to share prototypes with others.

However, to properly encapsulate all the dependencies such as drivers for Nvidia GPUs you will need to use Nvidia-Docker instead of Docker. This requires a little bit more work than using vanilla Docker but is straight forward once you understand Docker.

I have placed my Nvidia-Docker setup in this repo, and will leave this as an exercise for the reader.

. . .

Get In Touch!

I hope this tutorial is useful for others. You can reach out to me in the following ways: Github, Twitter, Linkedin.

Docker Machine Learning Data Science Towards Data Science