Member-only story

Docker for data scientists-Part 2

A complete guide to develop a machine learning model, write an API, package it in Docker, run it anywhere, and share it with ease.

Adam Sroka · Follow
Published in Towards Data Science · 9 min read · Dec 30, 2020

169

Docker for data scientists — an end-to-end guide (photo by Rémi Boudousquié on Unsplash).

Overview

In Docker for data scientists — Part 1 we covered some of the basics around using Docker and got a simple Hello World! example up and running. If you missed that post and want to run through the code, you can find it here:

Docker for data scientists — Part 1

The basics. A quick guide for data scientists and machine learning engineers to get started with Docker and...

towardsdatascience.com

Our aim in this post is to move on to an example of how a data scientist or machine learning engineer might want to use Docker to share, scale, and productionise their projects.

We'll work our way through the following topics:

- Model serving — develop and deploy an API for a simple model
- Serialising models — storing and sharing models in Python
- APIs with Flask — a very simple example of a prediction API
- Images — packaging everything up and running in a container
- Repositories — share your work
- Machine learning at scale — Introducing Docker Swarm and Kubernetes
- Next steps — DataOps, MLOps, and machine learning engineering

Model serving — develop a model, an API, and serve using docker

Example: a random forest classifier for the Iris dataset

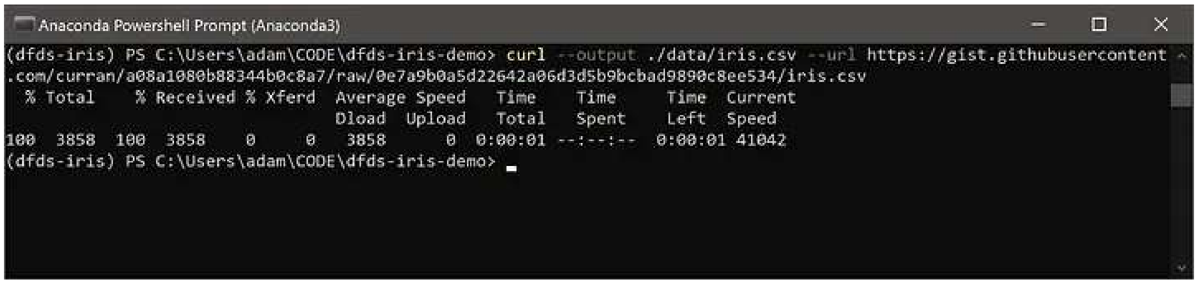
To keep things simple, we'll build a simple model for the well-known Iris dataset. We'll write a straightforward API allowing users to call that model to predict on an array. Then, I'll walk through the process of packaging and sharing the whole thing with Docker.

Get data

First, let’s get the data. As it’s a super-common demo dataset, we can import it directly from scikit-learn within our script using:

Or, if you want the `.csv` you can get it from GitHub using the terminal with:

It should look something like this:



```
(dfds-iris) PS C:\Users\adam\CODE\dfds-iris-demo> curl --output ./data/iris.csv --url https://gist.githubusercontent
.com/curran/a08a1080b88344b0c8a7/raw/0e7a9b0a5d22642a06d3d5b9bcbad9890c8ee534/iris.csv
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  3858  100  3858    0     0   3858      0  0:00:01 --:--:--  0:00:01 41042
(dfds-iris) PS C:\Users\adam\CODE\dfds-iris-demo>
```

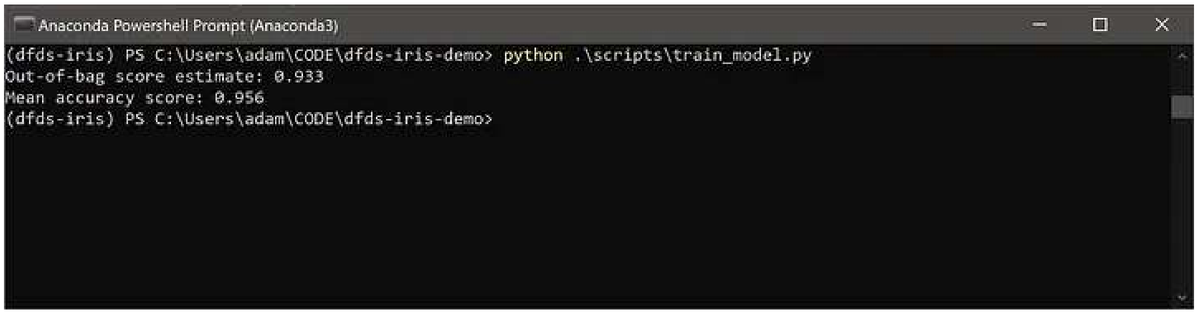
Downloading the Iris dataset as a .csv from GitHub (image by author).

> *Note: If your terminal is Powershell or the Anaconda PowerShell Prompt on Windows this may not work. By default, PowerShell has the* `Invoke-WebRequest`

> *command aliased to `curl` and it may behave differently. Either install cURL using Chocolatey using `choco install curl` or use whatever method you normally do for getting datasets.*

Creating the model

We are going to quickly spin up a Random Forest model. Create a new folder called `scripts` and in it, a new file called `train_model.py` in which we'll write some simple code for building the model, as follows:



```
(dfds-iris) PS C:\Users\adam\CODE\dfds-iris-demo> python .\scripts\train_model.py
Out-of-bag score estimate: 0.933
Mean accuracy score: 0.956
(dfds-iris) PS C:\Users\adam\CODE\dfds-iris-demo>
```

The output from our simple model creation script giving out-of-bag score and mean accuracy (image by author).

We could go further by looking at the confusion matrix, the ROC curve, and tweaking the features and hyperparameters. I won't just here to keep the example simple — but please take the time to tweak this for your application.

Serialising models

Once we have a model we're happy with, we want to store it. There are a few methods for achieving this. Here we're just going to use Pickle to keep things simple. Pickle allows us to convert between Python in-memory objects and streams of bytes — for transfer over networks or storage on a local filesystem.

Pickle syntax is quite straightforward:

For a good introduction to Pickle check out Manu Kalia's post here:

Don't Fear the Pickle: Using Pickle.dump and Pickle.load

Improve your data science workflow with pickled Python objects shared amongst multiple Jupyter notebooks

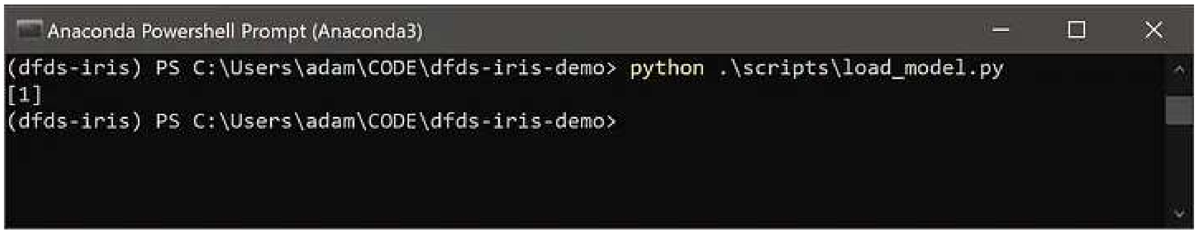
medium.com

Now, to pickle our model, either make the changes to the file we have already `train_model.py` or you can create a new file with a name like `build_model.py` and put your modified code to build and serialise the model in there.

If you're modifying `train_model.py` add the import at the top and replace the `print` statement:

And you should now see the pickled model on your filesystem. To test this out, let's make a new file called `load_model.py` in which we'll load the model from the serialised object and run a prediction. The code will look something like this:

This should return the value `[1]` like this:



Anaconda Powershell Prompt (Anaconda3)

```
(dfds-iris) PS C:\Users\adam\CODE\dfds-iris-demo> python .\scripts\load_model.py
[1]
(dfds-iris) PS C:\Users\adam\CODE\dfds-iris-demo>
```

The output from our script testing the prediction from the pickled model (image by author).

There are a lot of advantages to this approach and you should consider using it in your workflows if you aren't already. You're now able to score with our model without loading the supporting packages, and you can store and share versions of your model.

APIs with Flask

Now that we have our model let's put a quick Flask API around it. We're just going to add a simple score method around it that'll allow users to send an array and get a prediction in return.

We won't delve into error handling or anything here — you can do a lot with Flask and I'd highly recommend taking some time to learn it as adding APIs to your projects is both big and clever! A good place to start is this post by Bharat Choudhary which shows how to build a UI and everything for the Iris dataset:

Deployment of Machine learning project using Flask
machine-learning project in production.
medium.com

Create a new file called `app.py` and let's build a simple API that allows us to adjust the input parameters. In this case, we just need to adjust the values after each of the `=` signs. Our code will look like this:

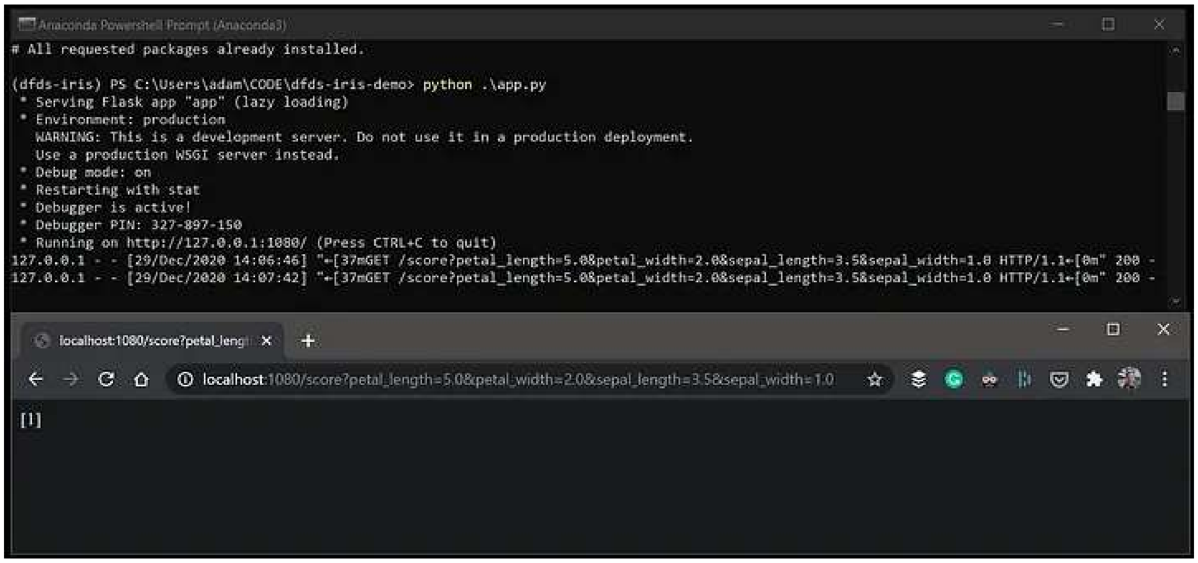
```python
# define imports
from flask import Flask, request
import pickle
from numpy import array2string

# define the path to the pickled model
model_path = "rf-model.pkl"

# unpickle the random forest model
with open(model_path, "rb") as file:
    unpickled_rf = pickle.load(file)

# define the app
app = Flask(__name__)


# use decorator to define the /score input method and define the predict function
@app.route("/score", methods=["POST", "GET"])
def predict_species():
    # create list and append inputs
    flower = []
    flower.append(request.args.get("petal_length"))
    flower.append(request.args.get("petal_width"))
    flower.append(request.args.get("sepal_length"))
    flower.append(request.args.get("sepal_width"))

    # return the prediction
    return array2string(unpickled_rf.predict([flower]))


# run the app
if __name__ == "__main__":
    app.run(host="0.0.0.0", debug=True, port="5001")
```

app.py hosted with ❤ by GitHub view raw

Now, if we run `app.py` we can test that this returns the same prediction as earlier by entering the following URL into our browser:

```
http://localhost:1080/score?
petal_length=5.0&petal_width=2.0&sepal_length=3.5&sepal_width=1.0
```

Your terminal and browser should now look something like this:



```
# All requested packages already installed.

(dfds-iris) PS C:\Users\adam\CODE\dfds-iris-demo> python .\app.py
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 327-897-150
 * Running on http://127.0.0.1:1080/ (Press CTRL+C to quit)
127.0.0.1 - - [29/Dec/2020 14:06:46] "←[37mGET /score?petal_length=5.0&petal_width=2.0&sepal_length=3.5&sepal_width=1.0 HTTP/1.1←[0m" 200 -
127.0.0.1 - - [29/Dec/2020 14:07:42] "←[37mGET /score?petal_length=5.0&petal_width=2.0&sepal_length=3.5&sepal_width=1.0 HTTP/1.1←[0m" 200 -
```

```
localhost:1080/score?petal_length=5.0&petal_width=2.0&sepal_length=3.5&sepal_width=1.0

[1]
```

The Flask prediction wrapper for the Iris random forest model up and running, giving the same result for the test set as seen earlier (image by author).

Images — package, and run

Now the model and app are built and working, we're ready to package them up using Docker. Firstly, we'll need to create a `requirements.txt` to capture all of the packages Python will need to run. This is easily achieved using:

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

```
Flask==1.1.2
numpy==1.19.4
scikit-learn==0.23.2
```

To keep things simple, if you've been working through all of the examples, create a new folder called `production`. In here, create the Dockerfile — remember this is a text file with the name `Dockerfile` and no extension. Then put a final copy of `app.py`, `requirements.txt` and `rf-model.pkl`. The folder should look like this:

```
production
├── app.py
├── Dockerfile
├── requirements.txt
└── rf-model.pkl
```

The contents of the Dockerfile are fairly straightforward as we can utilise existing Python images from Dockerhub. The Dockerfile will look like this:

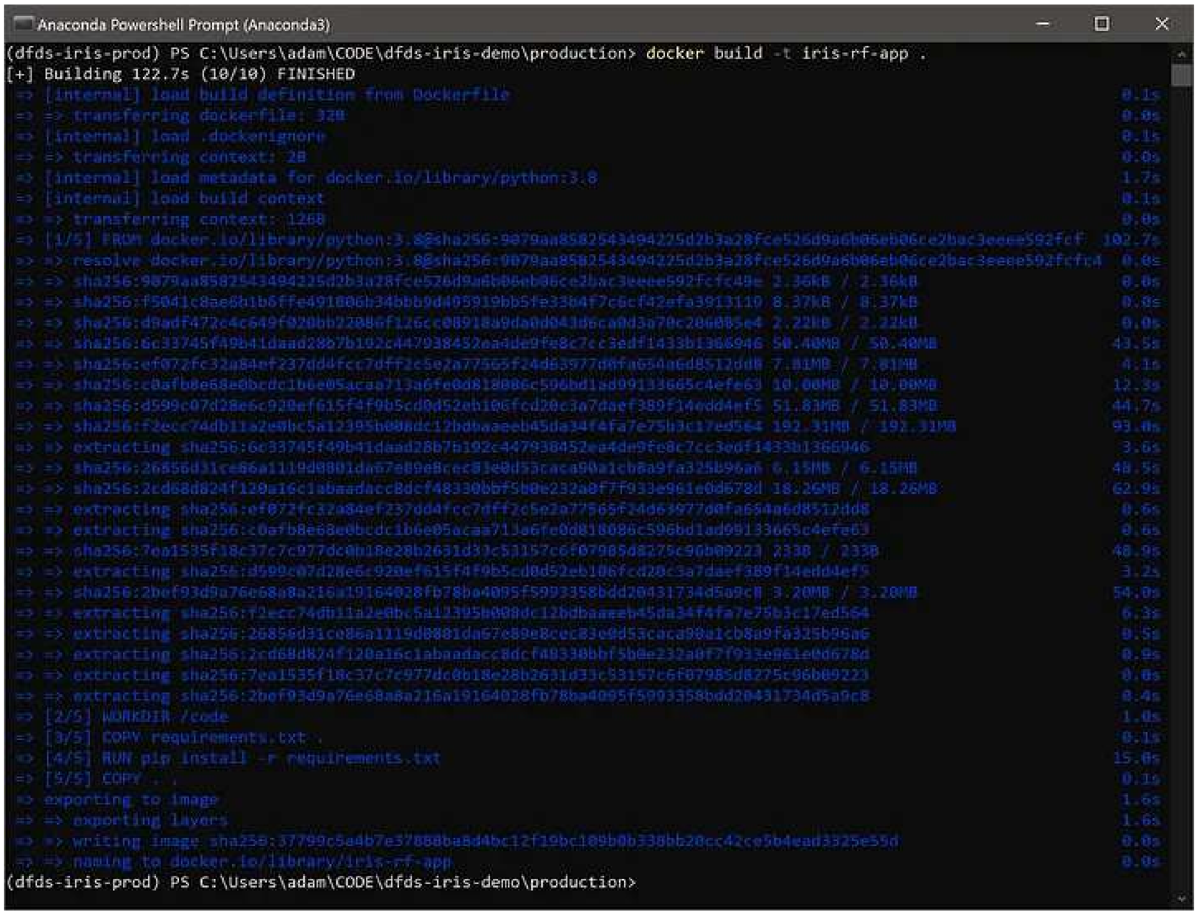
```
# set base image (host OS)
FROM python:3.8

# set the working directory in the container
WORKDIR /code

# copy the dependencies file to the working directory
COPY requirements.txt .

# install dependencies
RUN pip install -r requirements.txt

# copy the content of the directory to the working directory
COPY . .

# command to run on container start
CMD [ "python", "./app.py" ]
```

Dockerfile hosted with ❤ by GitHub view raw

With this all in place we can build our image using:

```
docker build -t iris-rf-app .
```

If you haven't already downloaded the `python:3.8` image, Docker will do so automatically. Overall this took just two minutes on my machine. Your terminal should look something like this:



Anaconda Powershell Prompt (Anaconda3)

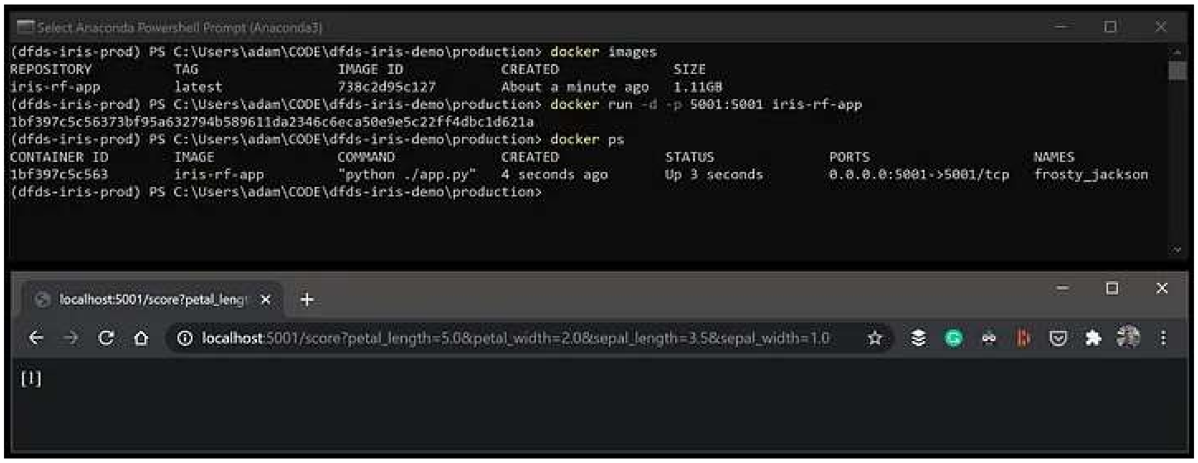
```
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production> docker build -t iris-rf-app .
[+] Building 122.7s (10/10) FINISHED
 => [internal] load build definition from Dockerfile                                                         0.1s
 => => transferring dockerfile: 32B                                                                          0.0s
 => [internal] load .dockerignore                                                                            0.1s
 => => transferring context: 2B                                                                              0.0s
 => [internal] load metadata for docker.io/library/python:3.8                                                1.7s
 => [internal] load build context                                                                            0.1s
 => => transferring context: 126B                                                                            0.0s
 => [1/5] FROM docker.io/library/python:3.8@sha256:9079aa8582543494225d2b3a28fce526d9a6b06eb06ce2bac3eeee592fcf  102.7s
 => => resolve docker.io/library/python:3.8@sha256:9079aa8582543494225d2b3a28fce526d9a6b06eb06ce2bac3eeee592fcfc4  0.0s
 => => sha256:9079aa8582543494225d2b3a28fce526d9a6b06eb06ce2bac3eeee592fcfc49e 2.36kB / 2.36kB               0.0s
 => => sha256:f5041c8ae6b1b6ffe491806b34bbb9d405919bb5fe33b4f7c6cf42efa3913110 8.37kB / 8.37kB               0.0s
 => => sha256:d8adf472c4c649f020bb22086f126cc08918a9da0d043d6ca0d3a70c206085e4 2.22kB / 2.22kB               0.0s
 => => sha256:6c33745f49b41daad28b7b192c447938452ea4de9fe8c7cc3edf1433b1366946 50.40MB / 50.40MB            43.5s
 => => sha256:ef072fc32a84ef237dd4fcc7dff2c5e2a77565f24d63977d0fa654a6d8512dd8 7.81MB / 7.81MB               4.1s
 => => sha256:c0afb8e68e0bcdc1b6e05acaa713a6fe0d818086c596bd1ad09133665c4efe63 10.00MB / 10.00MB            12.3s
 => => sha256:d599c07d28e6c920ef615f4f9b5cd0d52eb106fcd20c3a7daef389f14edd4ef5 51.83MB / 51.83MB            44.7s
 => => sha256:f2ecc74db11a2e0bc5a12395b008dc12bdbaaeeb45da34f4fa7e75b3c17ed564 192.31MB / 192.31MB          93.0s
 => => extracting sha256:6c33745f49b41daad28b7b192c447938452ea4de9fe8c7cc3edf1433b1366946                    3.6s
 => => sha256:26856d31ce86a1119d8801da67e89e8cec83e0d53caca90a1cb8a9fa325b96a6 6.15MB / 6.15MB              48.5s
 => => sha256:2cd68d824f120a16c1abaadacc8dcf48330bbf5b0e232a0f7f933e961e0d678d 18.26MB / 18.26MB            62.9s
 => => extracting sha256:ef072fc32a84ef237dd4fcc7dff2c5e2a77565f24d63977d0fa654a6d8512dd8                    0.6s
 => => extracting sha256:c0afb8e68e0bcdc1b6e05acaa713a6fe0d818086c596bd1ad09133665c4efe63                    0.6s
 => => sha256:7ea1535f18c37c7c977dc0b18e28b2631d33c53157c6f07985d8275c96b09223 233B / 233B                  48.9s
 => => extracting sha256:d599c07d28e6c920ef615f4f9b5cd0d52eb106fcd20c3a7daef389f14edd4ef5                    3.2s
 => => sha256:2bef93d9a76e68a8a216a19164028fb78ba4095f5993358bdd20431734d5a9c8 3.20MB / 3.20MB              54.0s
 => => extracting sha256:f2ecc74db11a2e0bc5a12395b008dc12bdbaaeeb45da34f4fa7e75b3c17ed564                    6.3s
 => => extracting sha256:26856d31ce86a1119d8801da67e89e8cec83e0d53caca90a1cb8a9fa325b96a6                    0.5s
 => => extracting sha256:2cd68d824f120a16c1abaadacc8dcf48330bbf5b0e232a0f7f933e961e0d678d                    0.9s
 => => extracting sha256:7ea1535f18c37c7c977dc0b18e28b2631d33c53157c6f07985d8275c96b09223                    0.0s
 => => extracting sha256:2bef93d9a76e68a8a216a19164028fb78ba4095f5993358bdd20431734d5a9c8                    0.4s
 => [2/5] WORKDIR /code                                                                                      1.0s
 => [3/5] COPY requirements.txt .                                                                            0.1s
 => [4/5] RUN pip install -r requirements.txt                                                               15.0s
 => [5/5] COPY . .                                                                                           0.1s
 => exporting to image                                                                                       1.6s
 => => exporting layers                                                                                      1.6s
 => => writing image sha256:37799c5a4b7e37888ba8d4bc12f19bc109b0b338bb20cc42ce5b4ead3325e55d                 0.0s
 => => naming to docker.io/library/iris-rf-app                                                               0.0s
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production>
```

Docker downloading the base image and sequentially building our Iris Flask app on top of it. All in taking about two minutes (image by author).

We can now check the image exists with `docker images` and run it with:

```
docker run -d -p 5001:5001 iris-rf-app
```

Testing it with our test URL from before we get the same results. The output from the terminal and browser can be observed below:



```
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED              SIZE
iris-rf-app         latest              738c2d95c127        About a minute ago   1.11GB
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production> docker run -d -p 5001:5001 iris-rf-app
1bf397c5c56373bf95a632794b589611da2346c6eca50e9e5c22ff4dbc1d621a
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
1bf397c5c563        iris-rf-app         "python ./app.py"   4 seconds ago       Up 3 seconds        0.0.0.0:5001->5001/tcp   frosty_jackson
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production>
```

```
localhost:5001/score?petal_length=5.0&petal_width=2.0&sepal_length=3.5&sepal_width=1.0
```

```
[1]
```

Running the docker image and checking the output with our test URL as before (image by author).

We now have a working version of our model, with a simple API, that can be run anywhere we can run a Docker container. If we really wanted to we could ramp up our API and start using this approach to build end-to-end machine learning and data science pipelines.

Repositories — sharing your work

Sharing Docker images is also really straightforward. Once you've signed up for an account at hub.docker.com it's really easy to upload your images to the public repo from the command line.

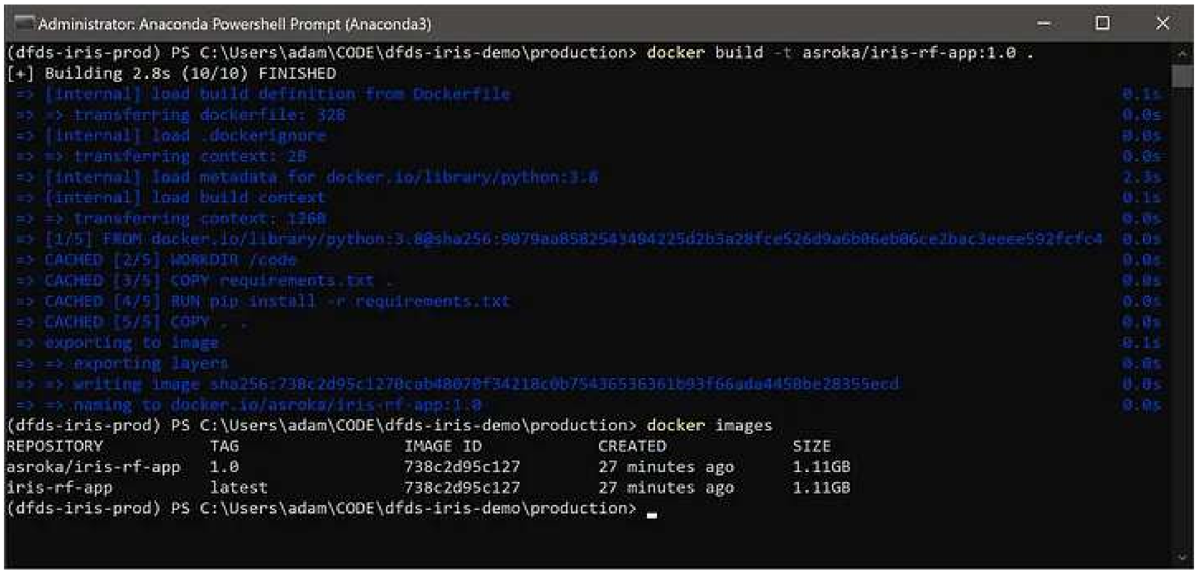
First, you need to log in so your instance of Docker knows your credentials:

```
docker login -u <your_docker_id> --password <your_docker_password>
```

Once logged in you can now upload images to your repo. The only change we need to make is to add tags. This is done in the build stage and follows the format:

```
docker build -t <your_docker_id>/<image_name>:<image_version>
```

If you leave `<image_version>` blank Docker appends the `latest` tag to the image. Rebuilding my image gives the following:

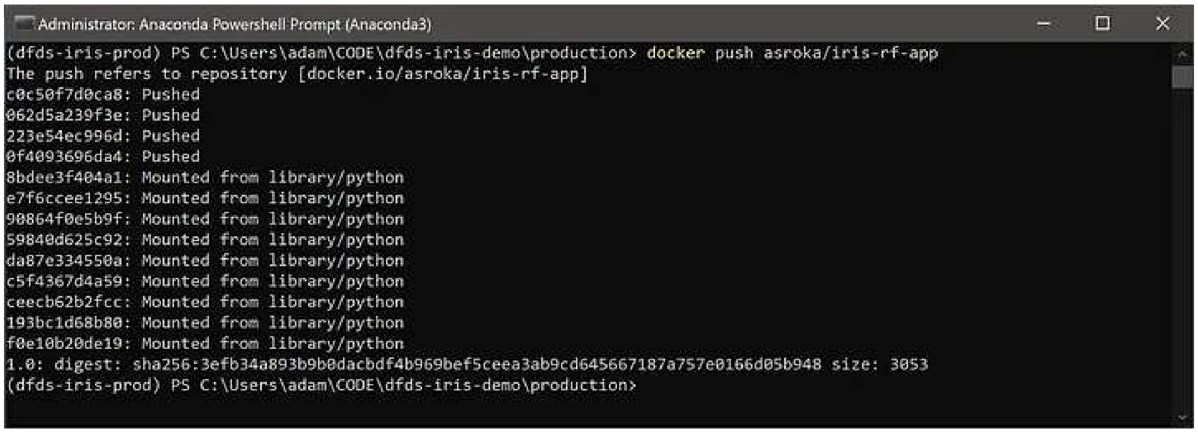


```
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production> docker build -t asroka/iris-rf-app:1.0 .
[+] Building 2.8s (10/10) FINISHED
 => [internal] load build definition from Dockerfile                                                          0.1s
 => => transferring dockerfile: 32B                                                                           0.0s
 => [internal] load .dockerignore                                                                             0.0s
 => => transferring context: 2B                                                                               0.0s
 => [internal] load metadata for docker.io/library/python:3.8                                                 2.3s
 => [internal] load build context                                                                             0.1s
 => => transferring context: 126B                                                                             0.0s
 => [1/5] FROM docker.io/library/python:3.8@sha256:9079aa8582543494225d2b3a28fce526d9a6b06eb06ce2bac3eeee592fcfc4  0.0s
 => CACHED [2/5] WORKDIR /code                                                                                0.0s
 => CACHED [3/5] COPY requirements.txt .                                                                      0.0s
 => CACHED [4/5] RUN pip install -r requirements.txt                                                          0.0s
 => CACHED [5/5] COPY . .                                                                                     0.0s
 => exporting to image                                                                                        0.1s
 => => exporting layers                                                                                       0.0s
 => => writing image sha256:738c2d95c1270cab48070f34218c0b75436536361b93f66ada4458be28355ecd                  0.0s
 => => naming to docker.io/asroka/iris-rf-app:1.0                                                             0.0s
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
asroka/iris-rf-app  1.0                 738c2d95c127        27 minutes ago      1.11GB
iris-rf-app         latest              738c2d95c127        27 minutes ago      1.11GB
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production> _
```

Rebuilt Iris Flask app with my Docker ID, name, and tag (image by author).

Then you just need to push the image to the repo using:

```
docker push <your_docker_id>/<image_name>
```

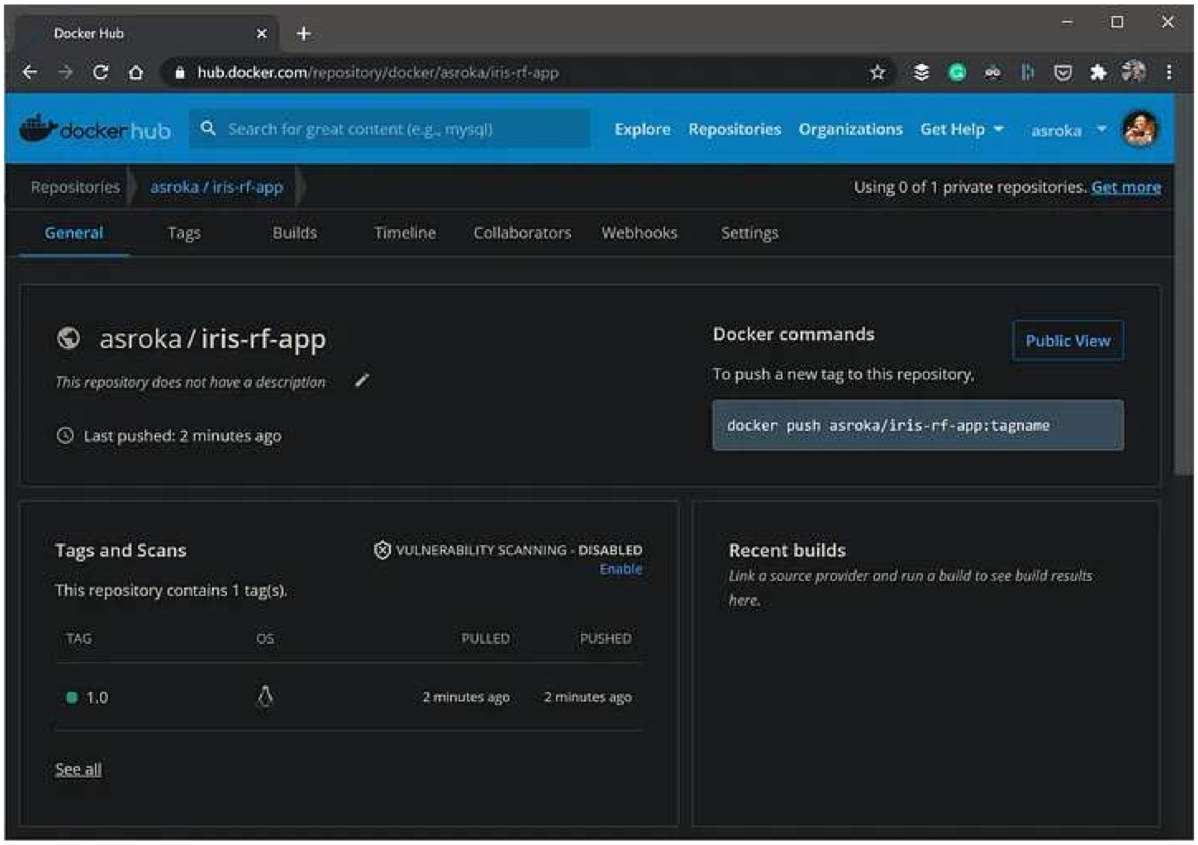


```
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production> docker push asroka/iris-rf-app
The push refers to repository [docker.io/asroka/iris-rf-app]
c0c50f7d0ca8: Pushed
062d5a239f3e: Pushed
223e54ec996d: Pushed
0f4093696da4: Pushed
8bdee3f404a1: Mounted from library/python
e7f6ccee1295: Mounted from library/python
98864f0e5b9f: Mounted from library/python
59840d625c92: Mounted from library/python
da87e334550a: Mounted from library/python
c5f4367d4a59: Mounted from library/python
ceecb62b2fcc: Mounted from library/python
193bc1d68b80: Mounted from library/python
f0e10b20de19: Mounted from library/python
1.0: digest: sha256:3efb34a893b9b0dacbdf4b969bef5ceea3ab9cd645667187a757e0166d05b948 size: 3053
(dfds-iris-prod) PS C:\Users\adam\CODE\dfds-iris-demo\production>
```

Example output of the Docker push command (image by author).

As shown below, our image is now publically viewable by navigating to:

```
hub.docker.com/repository/docker/<your_docker_id>/<image_name>
```



Using 0 of 1 private repositories. Get more

asroka / iris-rf-app

This repository does not have a description

Last pushed: 2 minutes ago

Docker commands

To push a new tag to this repository,

```
docker push asroka/iris-rf-app:tagname
```

Tags and Scans

This repository contains 1 tag(s).

TAG	OS	PULLED	PUSHED
● 1.0		2 minutes ago	2 minutes ago

Recent builds

Link a source provider and run a build to see build results here.

Our Iris Flask app publically available on Docker hub (image by author).

If you want to give it a try, you can navigate to my repo shown above and pull and run that image using:

```
docker pull asroka/iris-rf-app:1.0
docker run -d -p 5001:5001 asroka/iris-rf-app:1.0
```

There are also plenty of ways of hosting private Docker repositories — perfect for sharing within your organisation or team. I won't touch on them in detail here but do check out the documentation as a starter here.

Machine learning at scale

So far we've looked at a very simple case but still, the approach here has many uses, allowing data scientists and machine learning engineers to package up, share, and run projects anywhere Docker will run.

> *There's a lot more to Docker — we haven't touched on multi-container applications. It's possible to define and spin up your entire applications stack — including databases, servers, load balancers, etc. — with Docker and Docker Compose. Once you're in the hang of these approaches, you can start templating and creating complex building blocks for your data pipelines.*

If you're interested in a deeper dive into Docker: Up & Running is great.

Managing many Docker containers at once can get complex and cumbersome. What if your app is in huge demand, how do you go from a handful of users to thousands? Luckily, there are tools available for just that.

Kubernetes

Developed by Google, Kubernetes is an open-source container orchestration platform that automates the management, deployment, scaling, and networking of containers.

I'll be covering Kubernetes for data scientists and machine learning engineers in a future post — so watch this space.

Docker Swarm

Another option is Docker Swarm. I won't be covering Docker Swarm as Kubernetes does the job for me but if you're interested I'd highly recommend checking out this post by Gabriel Tanner to get started:

The Definitive Guide to Docker Swarm

Everything you need to know about Docker Swarm and how to use it to scale and securely maintain your Docker projects

medium.com

Conclusion

This post has enough to get you up and running with Docker and deploying a machine learning model into production. With practice, you'll begin to see the applications in your workflow. There's a lot you can do. Some thoughts for the next steps follow.

Sharing entire project workspaces

Coming from a consultancy background we found Docker was a great way to share a finalised project with a customer without the hassle of environment and dependency management. To get a start on this, check out Cookiecutter Docker Science — a great tool to add to your toolbox.

JupyterLab

Setting up a JupyterLab can sometimes be cumbersome — ensuring credentials and configurations are all in the right place. It's possible to wrap these all up in Docker containers, making it easy to spin up new JupyterLab environments on a range of machines.

MLOps, DataOps, and machine learning engineering

Machine learning engineering is a relatively new field, and the tools and practices are still forming. It looks like containers are here to stay, acting as a fundamental building block of many workflows. The fields of MLOps and DataOps are fascinating and well worth further exploration.

This discussion of MLOps by Cristiano Breuel really works as a first touchpoint:

ML Ops: Machine Learning as an Engineering Discipline

As ML matures from research to applied business solutions, so do we need to improve the maturity of its operation…

towardsdatascience.com

And for an introduction to DataOps, see this great post by DataKitchen on DataOps:

DataOps is NOT Just DevOps for Data

One common misconception about DataOps is that it is just DevOps applied to data analytics. While a little semantically...

medium.com

I hope this was useful, please let me know if you have any feedback reach out at:

- Twitter at www.twitter.com/adzsroka
- LinkedIn at www.linkedin.com/in/aesroka
- Or at www.adamsroka.co.uk

If there are any other topics you're interested in me writing about please let me know!

Please note, this post does contain an affiliate link.

Docker Data Science Machine Learning AI Software Development

Written by Adam Sroka

Follow

809 Followers · Writer for Towards Data Science

Dr Adam Sroka, Head of Machine Learning Engineering at Origami Energy, is an experienced data and AI leader helping organisations unlock value from data.

More from Adam Sroka and Towards Data Science

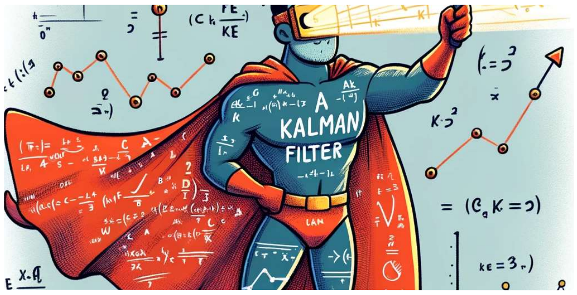
Adam Sroka in The Startup

Why So Many Data Scientists Quit Good Jobs at Great Companies

A look at why the 'sexiest job of the 21st century' has lost its appeal

· 10 min read · Mar 2, 2021

2.5K 27



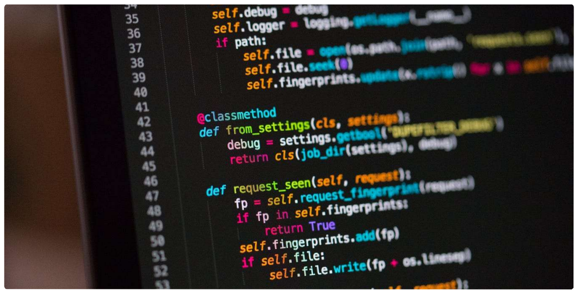
Jimmy Weaver in Towards Data Science

Exposing the Power of the Kalman Filter

As a data scientist we are occasionally faced with situations where we need to model a...

17 min read · Nov 7

816 11



Benjamin Lee in Towards Data Science

The New Best Python Package for Visualising Network Graphs

A guide on who should use it, when to use it, how to use it, and why I was wrong before...

· 10 min read · 6 days ago

659 9

Adam Sroka in Towards Data Science

Why You Shouldn't Hire More Data Scientists

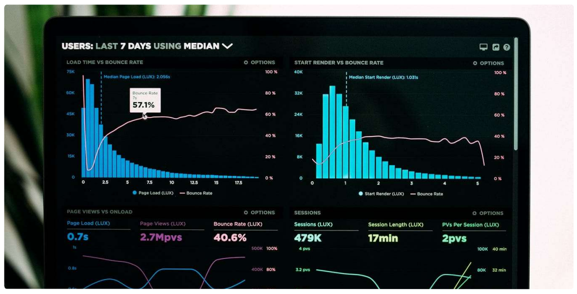
· 8 min read · Apr 26, 2021

457 4

See all from Adam Sroka

See all from Towards Data Science

Recommended from Medium



Jeremy

What I learned after one year of building a Data Platform from...

My key learnings on building a Data platform, from the tech side to the business side

9 min read · Nov 14

1.7K 20

Ayush Thakur

Wait! What are Pipelines in Python?

If you are a Python developer, you might have heard of the term pipeline. But what exactly i...

5 min read · Nov 6

651 13

Lists

Predictive Modeling w/ Python

20 stories · 648 saves

Practical Guides to Machine Learning

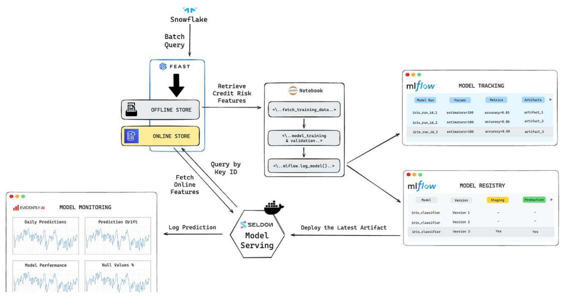
10 stories · 729 saves

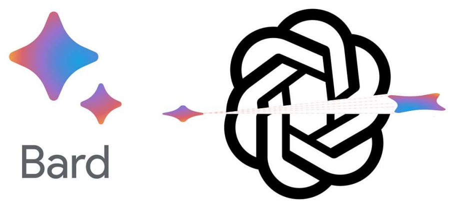
The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 222 saves

Coding & Development

11 stories · 291 saves





Qwak

Building an End-to-End MLOps Pipeline with Open-Source Tools

MLOps Open Source: TL;DR

11 min read · Oct 18

448 4

AL Anany

The ChatGPT Hype Is Over — Now Watch How Google Will Kill…

It never happens instantly. The business game is longer than you know.

· 6 min read · Sep 1

19.7K 603

Sumit Pandey in Towards AI

YOLOv8: The Simple Choice for Better Segmentation on custom…

YOLOv8 is an amazing segmentation model, its easy to train test and deploy. In this tutori…

· 6 min read · Nov 6

63

Paul Rose

I Found A Very Profitable AI Side Hustle

And it's perfect for beginners

6 min read · Oct 18

11.6K 207

See more recommendations