Dockerfiles & Vetiver

Lecture 21

Dr. Colin Rundel

Dockerfile(s)

> Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image.

command	Description
`FROM`	specify a base image
`RUN`	run commands (e.g. apt or yum), changes saved to image
`COPY`	copy a local file into the image
`ENV`	set environment variables for Dockerfile and image
`USER`	set user to use (affects subsequent `RUN`, `CMD`, `ENDPOINT`)
`WORKDIR`	set the working directory
`EXPOSE`	specify which ports will be used (not published automatically)
`CMD`	specify default command run when running the image
…	…

A basic example

ex1/Dockerfile

```
FROM ubuntu:24.04

ENV DEBIAN_FRONTEND=noninteractive

RUN apt update
RUN apt install -y r-base
RUN Rscript -e "install.packages('tibble')"

CMD ["R", "--vanilla"]
```

Building

```
> docker build -t example .
```

```
[+] Building 105.1s (9/9) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 227B
 => [internal] load metadata for docker.io/library/ubuntu:24.04
 => [auth] library/ubuntu:pull token for registry-1.docker.io
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/4] FROM docker.io/library/ubuntu:24.04@sha256:72297848456d5d37d1262630108ab308d3e9ec7ed1c32
 => => resolve docker.io/library/ubuntu:24.04@sha256:72297848456d5d37d1262630108ab308d3e9ec7ed1c32
 => => sha256:72297848456d5d37d1262630108ab308d3e9ec7ed1c3286a32fe09856619a782 6.69kB / 6.69kB
 => => sha256:a3f23b6e99cee41b8fffbd8a22d75728bb1f06af30fc79f533f27c096eda8993 424B / 424B
 => => sha256:c3d1a34325805c22bf44a5157224bcff58dc6a8868558c7746d6a2ea64eb191c 2.31kB / 2.31kB
 => => sha256:5b17151e9710ed47471b3928b05325fa4832121a395b9647b7e50d3993e17ce0 28.89MB / 28.89MB
 => => extracting sha256:5b17151e9710ed47471b3928b05325fa4832121a395b9647b7e50d3993e17ce0
 => [2/4] RUN apt update
 => [3/4] RUN apt install -y r-base
 => [4/4] RUN Rscript -e "install.packages('tibble')"
 => exporting to image
 => => exporting layers
 => => writing image sha256:10932419c2d9dfbf583a04aa8c57fac1a8634261ef53b7f9d5368bbaecf5e978
 => => naming to docker.io/library/example
```

Images

```
> docker images
```

```
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
example      latest    10932419c2d9   56 seconds ago   1.06GB
```

```
> docker run --rm -it example:latest
```

```
R version 4.3.3 (2024-02-29) -- "Angel Food Cake"
Copyright (C) 2024 The R Foundation for Statistical Computing
Platform: aarch64-unknown-linux-gnu (64-bit)

...

> library(tibble)
> tibble(a=1:10,b=letters[1:10])
# A tibble: 10 x 2
       a b
   <int> <chr>
 1     1 a
 2     2 b
 3     3 c
 4     4 d
 5     5 e
 6     6 f
 7     7 g
 8     8 h
 9     9 i
10    10 j
```

Some helpful hints

- Using `ENV DEBIAN_FRONTEND=noninteractive` prevents `apt` from stopping things to prompt for input
  - This is not needed with `rpm` / `dnf` since rpms are not supposed to prompt for input
- Using the `-y` flag with `apt`, `rpm`, or `dnf` skips prompting about installing additional dependencies
- If not specified `docker build` will use the `latest` tag

A slightly different example

ex2/Dockerfile

```
FROM ubuntu:24.04

ENV DEBIAN_FRONTEND=noninteractive

RUN apt update && \
    apt install -y r-base && \
    Rscript -e "install.packages('tibble')" && \
    rm -rf /var/cache/apt/archives /var/lib/apt/lists/*

CMD ["R", "--vanilla"]
```

Building

```
> docker build -t example .
```

```
[+] Building 102.6s (7/7) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 311B
 => [internal] load metadata for docker.io/library/ubuntu:24.04
 => [auth] library/ubuntu:pull token for registry-1.docker.io
 => [internal] load .dockerignore
 => => transferring context: 2B
 => CACHED [1/2] FROM docker.io/library/ubuntu:24.04@sha256:72297848456d5d37d1262630108ab308d3e9ec
 => [2/2] RUN apt update &&     apt install -y r-base &&     Rscript -e "install.packages('tibble'
 => exporting to image
 => => exporting layers
 => => writing image sha256:3ed77d00186595e102de575e37cbb846b4008f9e1c249477d3e0cdf26fcb9dd0
 => => naming to docker.io/library/example
```

```
> docker Images
```

```
REPOSITORY   TAG       IMAGE ID       CREATED              SIZE
example      latest    3ed77d001865   About a minute ago   1.01GB
<none>       <none>    b73484b7a407   8 minutes ago        1.06GB
```

Docker History

```
> docker history 3ed77d001865
```

```
IMAGE          CREATED         CREATED BY                                      SIZE      COMMENT
3ed77d001865   2 minutes ago   CMD ["R" "--vanilla"]                           0B        buildkit.
<missing>      2 minutes ago   RUN /bin/sh -c apt update &&     apt install…   912MB     buildkit.
<missing>      2 minutes ago   ENV DEBIAN_FRONTEND=noninteractive              0B        buildkit.
<missing>      2 months ago    /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>      2 months ago    /bin/sh -c #(nop) ADD file:68158f1ff76fd4de9…   101MB
<missing>      2 months ago    /bin/sh -c #(nop)  LABEL org.opencontainers.…   0B
<missing>      2 months ago    /bin/sh -c #(nop)  LABEL org.opencontainers.…   0B
<missing>      2 months ago    /bin/sh -c #(nop)  ARG LAUNCHPAD_BUILD_ARCH     0B
<missing>      2 months ago    /bin/sh -c #(nop)  ARG RELEASE                  0B
```

```
> docker history b73484b7a407
```

```
IMAGE          CREATED          CREATED BY                                      SIZE      COMMENT
b73484b7a407   9 minutes ago    CMD ["R" "--vanilla"]                           0B        buildkit
<missing>      9 minutes ago    RUN /bin/sh -c rm -rf /var/cache/apt/archive…   0B        buildkit
<missing>      27 minutes ago   RUN /bin/sh -c Rscript -e "install.packages(…   15.3MB    buildkit
<missing>      27 minutes ago   RUN /bin/sh -c apt install -y r-base # build…   897MB     buildkit
<missing>      28 minutes ago   RUN /bin/sh -c apt update # buildkit            46.5MB    buildkit
<missing>      28 minutes ago   ENV DEBIAN_FRONTEND=noninteractive              0B        buildkit
<missing>      2 months ago     /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>      2 months ago     /bin/sh -c #(nop) ADD file:68158f1ff76fd4de9…   101MB
<missing>      2 months ago     /bin/sh -c #(nop)  LABEL org.opencontainers.…   0B
<missing>      2 months ago     /bin/sh -c #(nop)  LABEL org.opencontainers.…   0B
<missing>      2 months ago     /bin/sh -c #(nop)  ARG LAUNCHPAD_BUILD_ARCH     0B
<missing>      2 months ago     /bin/sh -c #(nop)  ARG RELEASE                  0B
```

Dangling images

When an image (and tag) is replaced with a newer versiomn this can result in a dangling image (e.g. `b73484b7a407`)

```
> docker Images
```

```
REPOSITORY   TAG       IMAGE ID       CREATED              SIZE
example      latest    3ed77d001865   About a minute ago   1.01GB
<none>       <none>    b73484b7a407   8 minutes ago        1.06GB
```

This image can be deleted directly via `docker rmi b73484b7a407` or using `docker image prune` to remove *all* dangling images.

```
> docker image prune
```

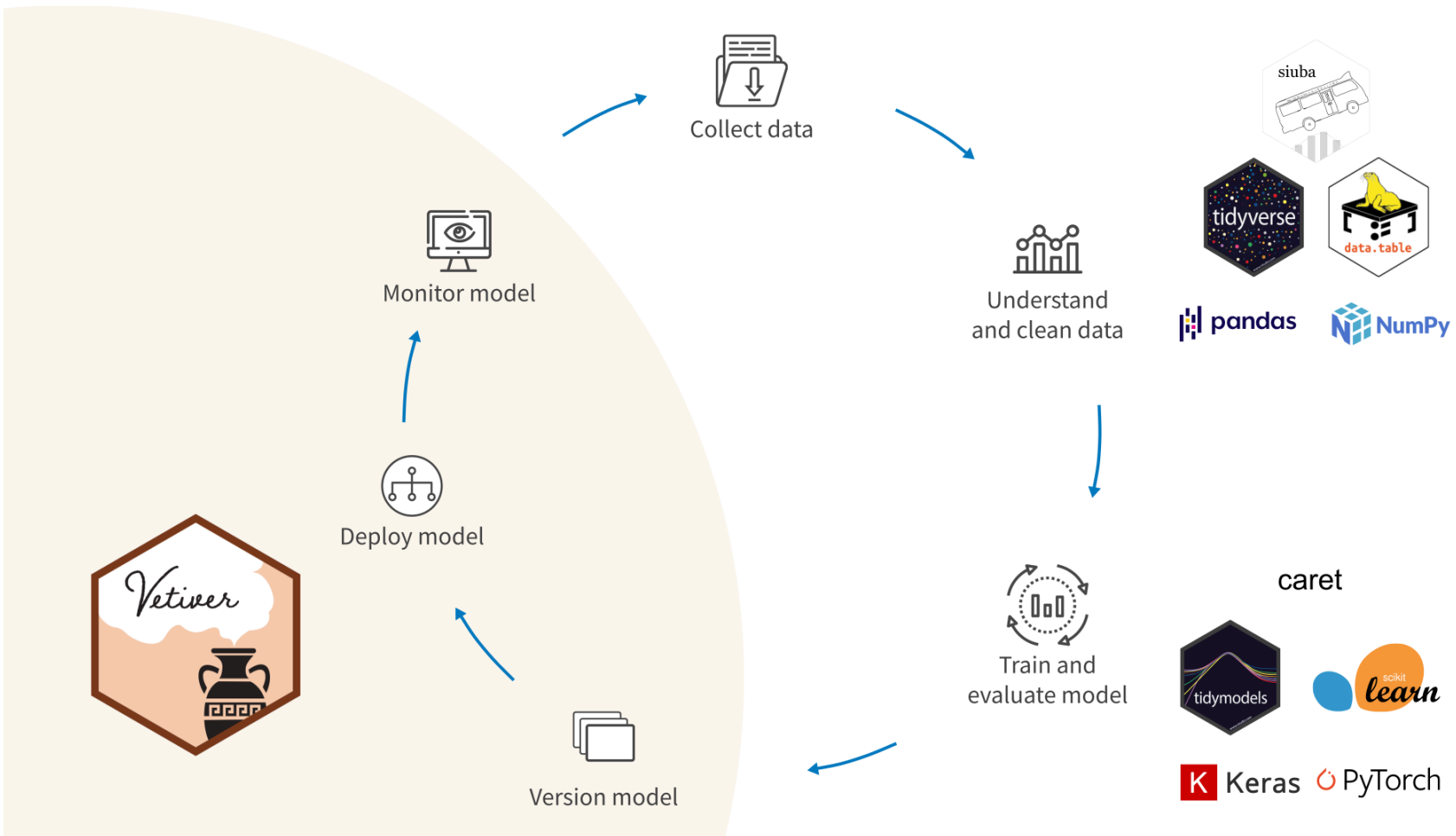
```
WARNING! This will remove all dangling images.
Are you sure you want to continue? [y/N] y
Deleted Images:
deleted: sha256:b73484b7a407bbd2d1e49983bf987147db9531b940d3169ad9b33a0a6101a4a8

Total reclaimed space: 0B
```

Vetiver

MLOps with Vetiver

> The goal of vetiver is to provide fluent tooling to version, deploy, and monitor a trained model. Functions handle both recording and checking the model's input data prototype, and predicting from a remote API endpoint.



Vetiver for R and Python

There are vetiver packages for both R and Python that provide similar functionality. Vetiver supports the following modeling frameworks from each language:

R

- tidymodels workflows
- caret
- mlr3
- XGBoost
- ranger
- lm() and glm()
- GAMS from mgcv

Python

- scikit-learn
- ~~PyTorch~~
- XGBoost
- statsmodels
- spacy

More of this in a bit - but Vetiver works best if your data is a data frame (or at least tabular) - full tensor support is very

Train a model

Back to our tried and true MNIST model, using sklearn's logistic regression model:

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X, y = digits.data, digits.target

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.20, shuffle=True, random_state=1234
)

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

m = LogisticRegression(
  penalty=None
).fit(
  X_train, y_train
)
```

Create a vetiver model

A vetiver model is a light wrapper around a supported model that comes with extra meta data (e.g. data prototype, metrics, etc.)

```
import vetiver
v = vetiver.VetiverModel(
  m, model_name = "mnist_log_reg",
  prototype_data = X_train
)
v.description
```

```
'A scikit-learn LogisticRegression model'
```

Pinning models

Model objects can be saved (and versioned) using a `pins` board

```
1 import pins
```

```
1 board = pins.board_temp(versioned = True, allow_pickle_read = True)
2 board.board
```

```
'/var/folders/v7/wrxd7cdj6l5gzr0191__m9lr0000gr/T/tmpj7lajl6a'
```

```
1 vetiver.vetiver_pin_write(board, v)
```

```
Model Cards provide a framework for transparent, responsible reporting.
 Use the vetiver `.qmd` Quarto template as a place to start,
 with vetiver.model_card()
Writing pin:
Name: 'mnist_log_reg'
Version: 20250402T095413Z-02741
```

```
1 board.pin_versions("mnist_log_reg")
```

```
              created   hash                  version
0 2025-04-02 09:54:13  02741  20250402T095413Z-02741
```

`pins` allows you to publish and share data, models, and other objects to local folders, cloud buckets (S3, gcs, etc.),

Board contents

Structure | data.txt | mnist_log_reg.joblib

```
> tree
```

```
.
└── mnist_log_reg
    └── 20250401T092253Z-02741
        ├── data.txt
        └── mnist_log_reg.joblib
```

Models from boards

Once a model has been saved (pinned) to a board it can be loaded by vetiver using `VetiverModel`s `from_pin()` method.

```
1 v = vetiver.VetiverModel.from_pin(board, "mnist_log_reg")
2 v.description
```

```
'A scikit-learn LogisticRegression model'
```

```
1 print(v.model)
```

```
LogisticRegression(penalty=None)
```

```
1 v.model.intercept_
```

```
array([ 0.0088223 , -0.09615749, -0.00944255,  0.04132351,  0.09290431,
       -0.00333723, -0.02541391,  0.02963178,  0.01658687, -0.0549176 ])
```

One additional pain point currently with loading models is the loading code needs the definitions of any custom classes

Pinning data

Just like models we can also pin (and version) data

```
1 board.pin_write(X_train, "mnist_X_train", type = "joblib")
```

```
Writing pin:
Name: 'mnist_X_train'
Version: 20250402T095413Z-3083a
Meta(title='mnist_X_train: a pinned ndarray object', description=None, created='20250402T095413Z',
```

```
1 board.pin_write(y_train, "mnist_y_train", type = "joblib")
```

```
Writing pin:
Name: 'mnist_y_train'
Version: 20250402T095413Z-07062
Meta(title='mnist_y_train: a pinned ndarray object', description=None, created='20250402T095413Z',
```

```
1 board.pin_write(X_test, "mnist_X_test", type = "joblib")
```

```
Writing pin:
Name: 'mnist_X_test'
Version: 20250402T095413Z-5fbe8
Meta(title='mnist_X_test: a pinned ndarray object', description=None, created='20250402T095413Z',
```

```
1 board.pin_write(y_test, "mnist_y_test", type = "joblib")
```

```
Writing pin:
Name: 'mnist_y_test'
Version: 20250402T095413Z-1cd99
Meta(title='mnist_y_test: a pinned ndarray object', description=None, created='20250402T095413Z',
```

Updated contents

Structure | data.txt

```
> tree
```

```
.
├── mnist_log_reg
│   └── 20250401T092253Z-02741
│       ├── data.txt
│       └── mnist_log_reg.joblib
├── mnist_X_test
│   └── 20250401T094626Z-5fbe8
│       ├── data.txt
│       └── mnist_X_test.joblib
├── mnist_X_train
│   └── 20250401T094620Z-3083a
│       ├── data.txt
│       └── mnist_X_train.joblib
├── mnist_y_test
│   └── 20250401T094626Z-1cd99
│       ├── data.txt
│       └── mnist_y_test.joblib
└── mnist_y_train
    └── 20250401T094624Z-07062
        ├── data.txt
        └── mnist_y_train.joblib

11 directories, 10 files
```

Deploying models

For supported model types, vetiver can generate a basic model api for you using plumber (R) or FastAPI (Python),

```
app = vetiver.VetiverAPI(v, check_prototype=True)
app.run(port = 8080)
```

If everything is working as expected you should see something like the following:

```
INFO:     Started server process [81869]
INFO:     Waiting for application startup.
INFO:     VetiverAPI starting...
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8080 (Press CTRL+C to quit)
```

The predict endpoint can be accessed using the server submodule,

```
endpoint = vetiver.server.vetiver_endpoint("http://127.0.0.1:8080/predict")
res = vetiver.server.predict(endpoint, pd.DataFrame(X_test[:10]))
res.predict.values
```

```
[6 8 5 3 5 6 6 4 5 0]
```

```
y_test[:10]
```

```
[6 8 5 3 5 6 6 4 5 0]
```

Prepare a Dockerfile

```
os.makedirs("docker/", exist_ok=True)
prepare_docker(board, "mnist_log_reg",  path="docker/")
```

app.py | vetiver_requirements.txt | Dockerfile

```
from vetiver import VetiverModel
from dotenv import load_dotenv, find_dotenv
import vetiver
import pins

load_dotenv(find_dotenv())

b = pins.board_folder('board', allow_pickle_read=True)
v = VetiverModel.from_pin(b, 'mnist_log_reg', version = '20250331T101211Z-02741')

vetiver_api = vetiver.VetiverAPI(v)
api = vetiver_api.app
```

Build

```
> cd docker/
> docker build -t mnist .
```

```
[+] Building 31.9s (10/10) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 573B
 => [internal] load metadata for docker.io/library/python:3.12
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [1/5] FROM docker.io/library/python:3.12@sha256:4e7024df2f2099e87d0a41893c299230d2a974c3474e68
 => => resolve docker.io/library/python:3.12@sha256:4e7024df2f2099e87d0a41893c299230d2a974c3474e68
 => => sha256:4e7024df2f2099e87d0a41893c299230d2a974c3474e681b0996f141951f9817 10.04kB / 10.04kB
 => => sha256:4378a6c11dea5043896b9425853a850807e5845b0018fe01ddee56c16245fc3c 23.54MB / 23.54MB
 => => sha256:3340b5550573c063816b90ec36245946fb68ad9780d223ff526cb93279631b21 2.33kB / 2.33kB
 => => sha256:0cac921bfee1587757ce9e51a5480012877e020ba9ad1b0269747b0b78534b81 6.41kB / 6.41kB
 => => sha256:545aa82ec479fb0ff3a196141d43d14e5ab1bd1098048223bfd21e505b70581f 48.30MB / 48.30MB
 => => sha256:140d15be2fea6dcd21c20cadae2601a118c08a938168718b2612ad6aca91f74a 64.36MB / 64.36MB
 => => extracting sha256:545aa82ec479fb0ff3a196141d43d14e5ab1bd1098048223bfd21e505b70581f
 => => sha256:1d9d474cce081e468bc6f85727459852112ba732fbbfe3236fae66c5fa8a5ed5 202.75MB / 202.75MB
 => => extracting sha256:4378a6c11dea5043896b9425853a850807e5845b0018fe01ddee56c16245fc3c
 => => sha256:3ac7d2d82d1801308a369fcd2a58cadcff5e2c88bf92089c61a74309774d76c8 6.24MB / 6.24MB
 => => sha256:7ca5d982b4a5e69d0cfbae58dab889e32fc7601f34888a1047dfddb815d8e3cc 24.91MB / 24.91MB
 => => sha256:5c1ef3e2d60a1e9bb7a0718e08d9e24bf523c6dcc1f8224baf1b42593b587b9e 249B / 249B
 => => extracting sha256:140d15be2fea6dcd21c20cadae2601a118c08a938168718b2612ad6aca91f74a
 => => extracting sha256:1d9d474cce081e468bc6f85727459852112ba732fbbfe3236fae66c5fa8a5ed5
 => => extracting sha256:3ac7d2d82d1801308a369fcd2a58cadcff5e2c88bf92089c61a74309774d76c8
```

Docker run

```
> docker images
```

```
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
mnist        latest    ae6791f6ab84   36 seconds ago   1.51GB
```

```
> docker run --rm mnist:latest
```

```
Traceback (most recent call last):
  File "/usr/local/bin/uvicorn", line 8, in <module>
    sys.exit(main())
             ^^^^^^
  File "/usr/local/lib/python3.12/site-packages/click/core.py", line 1161, in __call__
    return self.main(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/site-packages/click/core.py", line 1082, in main
    rv = self.invoke(ctx)
         ^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/site-packages/click/core.py", line 1443, in invoke
    return ctx.invoke(self.callback, **ctx.params)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/site-packages/click/core.py", line 788, in invoke
    return __callback(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/site-packages/uvicorn/main.py", line 412, in main
    run(
  File "/usr/local/lib/python3.12/site-packages/uvicorn/main.py", line 579, in run
    server.run()
  File "/usr/local/lib/python3.12/site-packages/uvicorn/server.py", line 66, in run
    return asyncio.run(self.serve(sockets=sockets))
```

Failure?

The error messages are a bit convoluted but, the issue amounts to

```
pins.errors.PinsError: Pin mnist_log_reg either does not exist, c
```

which implies that `app.py` in the Docker image wasn't able to find our `mnist_log_reg` pin.

This should not be terribly surpising since the pin lives in the `board/` folder locally and we have not copied that folder or its contents into the Dockerfile.

We have two options to resolve this,

1. Modify our Dockerfile to include a `COPY` to move `board/` into `/vetiver` in the container.
2. Make use of a Docker volume to give our container access to the local `board` folder when we run it.

Docker volumes

are passed to `docker run` via the `-v` or `--volume` flag and uses the syntax:

```
> docker run -v [<volume-name>:]<mount-path>[:opts]
```

where `<volume-name>` is the path on the local system, `<mount-path>` is the path inside the container, and `opts` are options like `ro` for readonly access.

A helpful note:

- Local and container paths must be *absolute paths* - you can use the `` `pwd` `` expansion to simplify things

This is complementary to the Dockerfile `VOLUME` command which is used to enforce persistence for critical paths in case

Corrected run

```
> docker run --rm -v `pwd`/../broad:/vetiver/broad mnist:latest
```

```
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     VetiverAPI starting...
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C to quit)
```

Everything now appears to be running, but can you connect to http://0.0.0.0:8080 or http://localhost:8080?

Docker ports

Similar to volumes, Docker makes a distinction between ports being used within a container and the ports used by the host machine.

Our Dockerfile included `EXPOSE 8080` to expose port 8080 but currently there is no mapping between that container port and our machine's network.

Syntax is similar to volumes and uses the `-p` or `--publish` flags to specify a mapping between the host and container ports

```
> docker run -p HOST_PORT:CONTAINER_PORT
```

The two ports do not need to match, 80:8080 maps the containers port 8080 to the hosts port 80.

Corrected run w/ ports

```
> docker run --rm -v `pwd`/../board:/vetiver/board -p 8080:8080 mnist:latest
```

```
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     VetiverAPI starting...
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C to quit)
```

Hopefully now everything should be running and accessible from your browser.

```
endpoint = vetiver.server.vetiver_endpoint("http://0.0.0.0:8080/predict")
res = vetiver.server.predict(endpoint, pd.DataFrame(X_test[:10]))
res.predict.values
```

```
[6 8 5 3 5 6 6 4 5 0]
```

```
y_test[:10]
```

```
[6 8 5 3 5 6 6 4 5 0]
```

Finalizing deployment

You may notice that when we use `docker run` here we see the `Uvicorn` output but we are not taken back to our prompt. We can use `Ctrl`+`C` to exit but this kills the container. If run remotely (via ssh) the container will also be killed when we disconnect.

A couple of additonal flags are useful with `docker run` here

- `-d`, `--detach` - run container in background
- `--restart` - specifies a container's restart policy, possible policies are:
  - `no` (Default) - container does not restart
  - `on-failure[:max-retries]` - restarts on errors (based on exit code) up to max-retries
  - `always` - restarts unless manually stopped (`docker stop`)
  - `unless-stopped` - similar to `always` but does not restart when daemon restarts

Detached container

```
> docker run -d --restart=on-failure -v `pwd`/../board:/vetiver/board -p 8080:8080 mnist:latest
```

```
998718e4a28a95afd46903df2e5b40642b4fb2245425804f2dd1f4d5f1525f9c
```

We can see our running container with `docker ps`, use the `-a` flag to see all containers (running and stopped),

```
> docker ps
```

```
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS
998718e4a28a   mnist:latest   "uvicorn app.app:api…"   34 seconds ago   Up 34 seconds   0.0.0.0:80
```

If we need to debug or mess with the running container we can run new command inside with `docker exec`,

```
> docker exec -it eager_knuth bash
```

```
root@998718e4a28a:/vetiver# ps -A
  PID TTY          TIME CMD
    1 ?        00:00:02 uvicorn
   31 pts/0    00:00:00 bash
   38 pts/0    00:00:00 ps
```

Stopping and cleaning up

When we're done with the container (or want to replace it) we can stop the container by name or id with `docker stop`, a stopped container can be restarted via `docker start`, and a container can be deleted via `docker rm`.

```
> docker stop eager_knuth
```

```
eager_knuth
```

```
> docker ps -a
```

```
CONTAINER ID   IMAGE          COMMAND                  CREATED         STATUS
998718e4a28a   mnist:latest   "uvicorn app.app:api…"   6 minutes ago   Exited (0) 14 seconds ago
```

```
> docker rm 998718e4a28a
```

```
998718e4a28a
```

```
> docker ps -a
```

```
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

Some comments about HW5

- We will be pushing the deadline for HW5 back to Friday, April 11th at 5 pm
- Parts of Vetiver can be made to work with pytorch but it is clunky at best
  - e.g. Transforming `X_train` to be a flat numpy array lets you create a VetiverModel that can be pinned
- Parts of Vetiver cannot be made to work with pytorch at moment
  - VetiverAPI's predict endpoint is not able to correctly serialize and unserialize the prediction data for torch

The current plan is on Friday is a complete demo involving:

- Briefly introducing FastAPI
- Bootstraping a basic prediction API using a saved torch model
- Combine these pieces into a custom Dockerfile

Container on the vm

Based on the content of the last lecture, you should be able to get Docker up and running on the VM and also be able to install any additional necessary packages to be able to build your image on the VM.

A couple of points,

- Files can be copied to the VM using `scp` via the command line or any number of GUI tools using this protocol
- Feel free to commit your Dockerfile and `pins` board to your GitHub repository, the repo can then be cloned on the VM
    - Do be mindful of the size of your board files - GitHub has a individual file size limit of ~100MB