Starting @ 03:15 PM

Volunteers? For chocolate!

- **Questions Typist:** quickly type interesting questions & answers in a google doc for everyone's benefit.
- **Note Taker:** Take notes of important things said in the lecture & are not present in the slides.
- **Linker:** Help keep us connected with online folks.
- **Coder:** Write simple yet effective lab code, will be contributing to the course repo through a PR.

2 chocolates each.

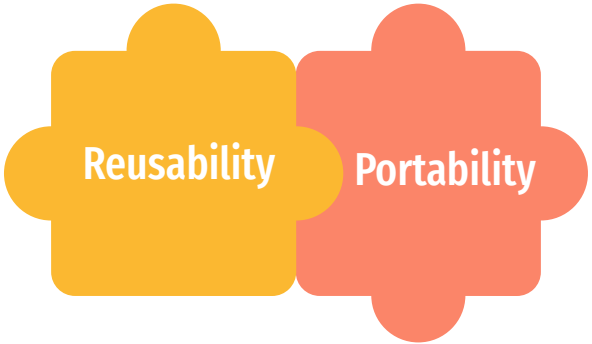
- More to come…

Packaging, Serving & Unit Testing

Pack It Up!

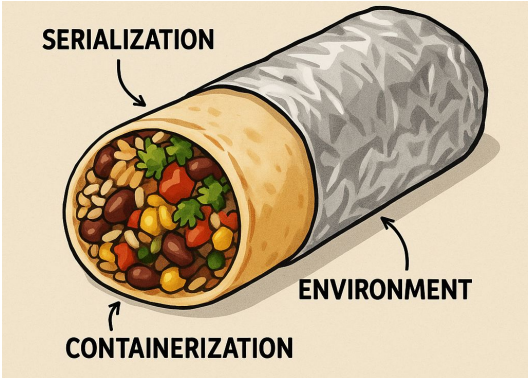
Model Optimization & Packaging in MLOps

Why Model Packaging?



The Three-Layer Packaging Burrito

- Serialization (Pickle, ONNX)
- Environment Packaging (Conda, Poetry)
- Containerization (Docker, Kubernetes)



Serialization Formats

- **Pickle, Joblib**: Ideal for serializing traditional machine learning models in Python-exclusive environments.
- **Torch, TensorFlow**: Best suited when models are trained and deployed within their respective frameworks.
- **ONNX, PMML**: Useful for model interoperability across different platforms and tools.

ONNX = Open Neural Network Exchange
PMML = Predictive Model Markup Language

Serialization with Pickle & Joblib

Serializing an sklearn model using pickle:

```
import pickle
from sklearn.ensemble import
RandomForestClassifier

model = ...

# Save
with open("model.pkl", "wb") as f:
    pickle.dump(model, f)

# Load
with open("model.pkl", "rb") as f:
    model = pickle.load(f)
```

Serialization with PyTorch & Tensorflow

Serializing a Tensorflow model:

```
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(10, activation="relu"),
    tf.keras.layers.Dense(1)
])
model.compile(optimizer='adam', loss='mse')
model.fit(X_train, y_train, epochs=3)

# Save
model.save('my_model.keras')
# Or
model.save('my_model.h5')

# Load
loaded_model =
tf.keras.models.load_model('my_model.keras')
# Or load in mobile/web apps
```

HD5 = HDF5 = Hierarchical Data Format version 5

Serialization with ONNX & PMML

```
# Torch
onnx_program = torch.onnx.export(...
onnx_program.save(...

# Sklearn
onx = to_onnx(...
with open("model.onnx", "wb") as f:
    f.write(...

# Tensorflow
onnx_model, _ = tf2onnx.convert.from_keras(...
onnx.save(...
```

Game Time!!

TODO: [Quizizz Link]

Why Serve?

- Make predictions via HTTP
- Integrate into other apps
- Monitor usage

Q: Why do we need to separate research & production?

Poetry

- Conda + Poetry => Magic!!
- Unlike req.txt, we need to separate dev & prod dependencies
- Bonus question: (chocolate) what is setup.py / lockfile?
- We need a lock file & we NEED to commit it
    - “Lockfiles are files that capture the exact versions of each dependency and their sub-dependencies in a project. They ensure uniformity in dependency versions across all instances of a project, preventing "dependency hell" and potential security risks.”

Serving Methods

- FastAPI (lightweight, async)
- Flask (classic, simple)
- MLflow Serve (model registry integration)
- TorchServe (for PyTorch models at scale)
- Tensorflow.js

Q: When to use MLflow Serve?

FastAPI Example

```python
from fastapi import FastAPI


app = FastAPI()
model = ...

@app.post("/predict")
def predict(data: dict):
    return {"prediction": model.predict([list(data.values())])[0]}
```

- Why is it my fav?

Extensions

- ThunderClient/Postman => communicating with API
- Black & isort => beautifying

Logging

- Why log? & Where?
- Std, file, db?
- Bonus: use hyperdx

Logging Levels (yesterday’s question)

- Thank you Heba
- Python's logging levels in increasing order of severity are:
  - DEBUG (10)
  - INFO (20)
  - WARNING (30)
  - ERROR (40)
  - CRITICAL (50)
- level=logging.INFO: Only log messages at level INFO and above will be shown.

Logging Levels (yesterday’s question)

- level=logging.INFO: Only log messages at level INFO and above will be shown.

● logger.debug(...)	● Not printed
● logger.info(...)	● Printed
● logger.warning(...)	● Printed
● logger.error(...)	● Printed
● logger.critical(...)	● Printed

Custom Level?

```
import logging

NOTICE_LEVEL_NUM = 25

def notice(self, message, *args, **kwargs):
    self._log(NOTICE_LEVEL_NUM, message, args, **kwargs)

logging.addLevelName(NOTICE_LEVEL_NUM, "NOTICE")
logging.Logger.notice = notice
logging.NOTICE = NOTICE_LEVEL_NUM

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger("my_logger")

logger.notice("This is a NOTICE level message")
logger.log(NOTICE_LEVEL_NUM, "This is a NOTICE level message")
logger.log(logging.NOTICE, "This is a NOTICE level message")
```

Game Time!!

TODO: [Quizizz Link]

Why Unit Test?

Unit Testing = Ensures each component works as expected

Q: What to unit test?

What to Unit Test?

- Data processing functions (e.g. feature scaling)
- Model prediction logic (e.g. output shapes, value ranges)
- API endpoints (input validation, output format)

Functions Testing

```python
# model_utils.py
def predict_dummy(x):
    return [i * 2 for i in x]

# test_model_utils.py
from model_utils import predict_dummy

def test_predict_dummy():
    input_data = [1, 2, 3]
    expected = [2, 4, 6]
    assert predict_dummy(input_data) == expected
```

Endpoints Testing

```python
from fastapi.testclient import TestClient
from main import app  # your FastAPI app

client = TestClient(app)

def test_predict_endpoint():
    response = client.post("/predict", json={"feature1": 5, "feature2": 10})
    assert response.status_code == 200
    assert "prediction" in response.json()
```

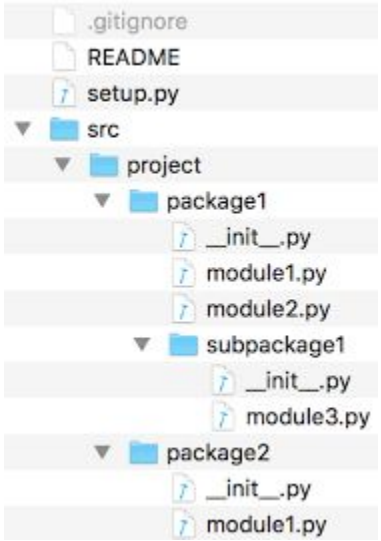
Running Tests & Best Practices

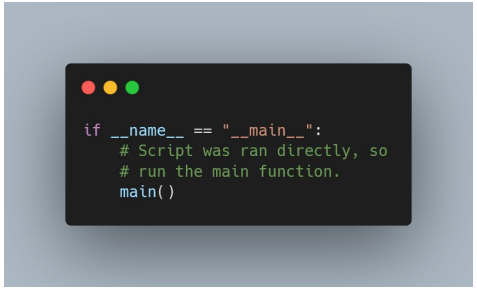
- Use pytest for simple, readable tests: pytest tests/
- pytest.ini (thank you Heba)
- Organize tests in a /tests folder parallel to your code
- Use mocks for complex dependencies
- TDD? [Q]

Past Questions

- Packaging & sub packages & modules?
- Why we have the if __name__ thingy in modules?
- Git LFS? [Q]

- $ git lfs track "*.onnx"
- model_path = "s3://my-bucket/models/my_model.onnx"
- Hugging Face Hub
- ML flow model URI (dedicated server)





Game Time!!

TODO: [Quizizz Link]

Download Docker (for tomorrow)

1. Download Docker:
   - Go to https://www.docker.com/get-started and download Docker Desktop.
2. Install Docker:
   - Run the installer and follow the on-screen instructions.
   - On Windows, enable WSL 2 if prompted (for Windows 10/11).
3. Start Docker:
   - Launch Docker Desktop.
   - Wait until Docker is running (check the whale icon in system tray).
4. Verify Installation:
   - Open terminal/command prompt.
   - Run: docker --version
   - Run: docker run hello-world (to test Docker setup).

Done! Docker is ready for use.

Lab

Your task is as follows:

- Pick your best model (chosen in lab 1)
- Create a fastAPI for it with three endpoints: home (/), health (/health), predict (/predict)
- Log thoroughly.
- Write at least 1 test function.
- Upload a screenshot of the swagger ui (http://localhost:8000/docs)
- Upload the link to your repo
- Bonus: Use HyperDX for live logs
- Bonus: Use commit convention & branching convention.
- Bonus: Write 2 more test functions