Lab 06: Deploying a Titanic Passenger Survival Predictor as a FastAPI Service

Overview

This lab converts the *application phase* of a trained Titanic passenger survival model into a production-ready FastAPI service. Students will learn how to persist preprocessing artifacts, create a typed prediction API, expose metadata and health endpoints, and containerize the application for deployment.

Learning Objectives

- Persist scikit-learn preprocessing artifacts (LabelEncoders or mappings) alongside the trained model.
- Build a FastAPI application that loads artifacts at startup and serves predictions via a JSON API.
- Create Pydantic request/response schemas for input validation and automatic API documentation.
- Add operational endpoints: `/health` and `/metadata` and basic error handling.
- Run and test the API locally and containerize it with Docker.

Required Libraries

Install the following libraries in your environment:

```
pip install fastapi uvicorn[standard] pandas numpy scikit-learn joblib
```

Purpose

Convert an interactive, script-based application phase (which reads `input()` values, encodes them, and runs `model.predict`) into a networked API that can be queried by frontends or other services. The API must:

- Validate inputs and return meaningful HTTP status codes.
- Use the exact same encodings the model saw during training.
- Load model artifacts once at startup for performance.

Project Layout (recommended)

```
titanic_fastapi/
├─ model_files/
│  ├─ svc_trained_model.pkl
│  ├─ pclass_encoder.pkl
│   ├─ gender_encoder.pkl
│   ├─ sibling_encoder.pkl
│   └─ embarked_encoder.pkl
├─ app/
│   ├─ main.py          # FastAPI application
│   └─ schemas.py       # Pydantic models
├─ requirements.txt
└─ Dockerfile
```

Step-by-step Lab Tasks

Step 1 — Persist preprocessing artifacts in training script

Modify your training script so it saves the `LabelEncoder` instances (or equivalent mappings) used in preprocessing. This ensures the API uses identical encodings at inference time.

Example (add after encoders are fit):

```
import pickle

# assuming these objects exist in your training script
with open('model_files/pclass_encoder.pkl','wb') as f:
    pickle.dump(pclass_label_encoder, f)
with open('model_files/gender_encoder.pkl','wb') as f:
    pickle.dump(gender_label_encoder, f)
with open('model_files/sibling_encoder.pkl','wb') as f:
    pickle.dump(sibling_label_encoder, f)
with open('model_files/embarked_encoder.pkl','wb') as f:
    pickle.dump(embarked_label_encoder, f)
# save the trained model too
with open('model_files/svc_trained_model.pkl','wb') as f:
    pickle.dump(svc_model, f)
```

Tip: Alternatively save a single `sklearn.Pipeline` that includes preprocessing + model (joblib recommended) to simplify serving.

Step 2 — Create Pydantic schemas

Create `app/schemas.py` and define request/response types. Use `Literal` to tightly restrict categories (helps validation and generated docs).

```
# app/schemas.py
from pydantic import BaseModel
from typing import Literal

class PredictRequest(BaseModel):
    PClass: Literal["First","Second","Third"]
    Gender: Literal["Male","Female"]
    Sibling: Literal["Zero","One","Two","Three"]
    Embarked: Literal["Southampton","Cherbourg","Queenstown"]
```

```
class PredictResponse(BaseModel):
    predicted_label: int
    prediction: Literal["SURVIVED","NOT SURVIVED"]
```

Step 3 — Implement the FastAPI application

Create `app/main.py`. Responsibilities:

- load artifacts once at startup (`@app.on_event("startup")`),
- expose `/predict` POST endpoint that validates, encodes, predicts and returns JSON,
- expose `/metadata` GET endpoint to return allowed categorical values,
- expose `/health` GET endpoint for basic readiness checks,
- catch `LabelEncoder` errors (unknown category) and return `400` with a helpful message.

Example skeleton (complete file):

```
# app/main.py
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
import pickle
import pandas as pd
import numpy as np
import logging
from .schemas import PredictRequest, PredictResponse

app = FastAPI(title="Titanic Survival Predictor")

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Optional: CORS (tighten origins in production)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Artifact paths
MODEL_PATH = "model_files/svc_trained_model.pkl"
PCLASS_ENC_PATH = "model_files/pclass_encoder.pkl"
GENDER_ENC_PATH = "model_files/gender_encoder.pkl"
SIBLING_ENC_PATH = "model_files/sibling_encoder.pkl"
EMBARKED_ENC_PATH = "model_files/embarked_encoder.pkl"


def load_artifacts():
    global model, pclass_le, gender_le, sibling_le, embarked_le
    with open(MODEL_PATH, 'rb') as f:
        model = pickle.load(f)
    with open(PCLASS_ENC_PATH, 'rb') as f:
        pclass_le = pickle.load(f)
    with open(GENDER_ENC_PATH, 'rb') as f:
        gender_le = pickle.load(f)
    with open(SIBLING_ENC_PATH, 'rb') as f:
        sibling_le = pickle.load(f)
    with open(EMBARKED_ENC_PATH, 'rb') as f:
        embarked_le = pickle.load(f)


@app.on_event('startup')
def startup_event():
    try:
        load_artifacts()
        logger.info('Model and encoders loaded')
    except Exception as e:
        logger.exception('Failed to load artifacts')
        raise


@app.get('/health')
def health():
    return {'status': 'ok'}


@app.get('/metadata')
def metadata():
    return {
        'PClass': list(pclass_le.classes_),
        'Gender': list(gender_le.classes_),
        'Sibling': list(sibling_le.classes_),
        'Embarked': list(embarked_le.classes_)
    }


@app.post('/predict', response_model=PredictResponse)
def predict(req: PredictRequest):
    try:
        df = pd.DataFrame([{
            'PClass': req.PClass,
            'Gender': req.Gender,
            'Sibling': req.Sibling,
            'Embarked': req.Embarked
        }])

        # Encode
        df['PClass'] = pclass_le.transform(df['PClass'])
        df['Gender'] = gender_le.transform(df['Gender'])
        df['Sibling'] = sibling_le.transform(df['Sibling'])
        df['Embarked'] = embarked_le.transform(df['Embarked'])

        features =
df[['PClass','Gender','Sibling','Embarked']].to_numpy()
        pred = model.predict(features)
        pred_value = int(np.ravel(pred)[0])
        label = 'SURVIVED' if pred_value == 1 else 'NOT SURVIVED'

        return PredictResponse(predicted_label=pred_value,
prediction=label)
```

```
    except ValueError as ve:
        # Unknown category passed to LabelEncoder.transform
        raise HTTPException(status_code=400, detail=str(ve))
    except Exception as e:
        logger.exception('Prediction error')
        raise HTTPException(status_code=500, detail=str(e))
```

Step 4 — requirements.txt

```
fastapi
uvicorn[standard]
pandas
numpy
scikit-learn
joblib
```

Step 5 — Run the API locally

From the project root run:

```
uvicorn app.main:app --reload --host 0.0.0.0 --port 8000
```

Docs (Swagger UI) will be available at `http://localhost:8000/docs`.

Step 6 — Test the API with `curl`

Successful example:

```
curl -X POST "http://localhost:8000/predict" \
  -H "Content-Type: application/json" \
  -d
'{"PClass":"First","Gender":"Male","Sibling":"Zero","Embarked":"Southam
pton"}'
```

If a categorical value is invalid the API returns `400` with the encoder error message.

Step 7 — Dockerize (optional)

Create a `Dockerfile` for deployment:

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY app ./app
COPY model_files ./model_files
EXPOSE 8000
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Build and run:

```
docker build -t titanic-fastapi:latest .
docker run -p 8000:8000 titanic-fastapi:latest
```

Lab Exercises

1. **Bundle preprocessing:** Re-train and export a `sklearn.Pipeline` that includes your encoders and the classifier, then modify the API to load only that pipeline artifact. Explain pros and cons.
2. **Unknown category handling:** Change the API to accept unknown categories by mapping them to a reserved index (e.g., `-1`) and document the behavior. Measure prediction differences.
3. **Versioning:** Add a `/version` endpoint that returns `model_version` and `artifact_checksums`. Demonstrate how you would use this in A/B testing.
4. **Monitoring:** Add basic Prometheus metrics (request count, latency, error count) to the FastAPI app.

Deliverables

- `app/main.py` and `app/schemas.py` source files
- `model_files/` containing `svc_trained_model.pkl` and encoder pickles
- `requirements.txt` and `Dockerfile` (if used)
- A short report (1 page) describing: artifact saving strategy, API contract (`/predict` schema), and test results from at least three example requests.

References

- FastAPI docs: https://fastapi.tiangolo.com/
- scikit-learn persistence: https://scikit-learn.org/stable/modules/model_persistence.html
- Uvicorn: https://www.uvicorn.org/