Absolutely! 🚀 Here's the **fully-commented code** with a **complete directory structure** for your **ML and backend pipeline**. This version makes it **easy to follow**, **deploy**, and **reuse** for any coding challenge or production task.

---

1. Directory Structure

plaintext
Copy code

```
project/
├── app/
│   ├── __init__.py
│   ├── main.py                    # FastAPI server with ML inference endpoints
│   ├── model.py                   # Model definition and training code
│   ├── preprocess.py              # Data preprocessing logic
│   ├── utils.py                   # Utility functions (logging, validation)
├── tests/
│   ├── test_main.py               # Unit tests for API endpoints
│   ├── test_model.py              # Unit tests for ML model
├── deployment/
│   ├── Dockerfile                 # Docker setup for deployment
│   ├── deployment.yaml            # Kubernetes YAML configuration
├── scripts/
│   ├── train.py                   # Model training pipeline
│   ├── quantize.py                # Model optimization pipeline
│   ├── monitor.py                 # Prometheus monitoring pipeline
├── requirements.txt               # Dependencies
├── README.md                      # Documentation
├── scaler.pkl                     # Preprocessing scaler (saved)
├── model.pth                      # Trained model weights
```

---

2. Model Definition (app/model.py)

python
Copy code

```python
import torch
import torch.nn as nn
import torch.optim as optim
import joblib
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification

# Define the Binary Classifier Neural Network
class BinaryClassifier(nn.Module):
    def __init__(self, input_dim):
        super(BinaryClassifier, self).__init__()
        # Linear layer with 16 neurons
        self.layer1 = nn.Linear(input_dim, 16)
        # Output layer with 1 neuron for binary classification
        self.output = nn.Linear(16, 1)
        # Sigmoid activation for probability output
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Apply ReLU activation in the hidden layer
        x = torch.relu(self.layer1(x))
        # Apply sigmoid activation in the output layer
        x = self.sigmoid(self.output(x))
        return x
```

---

3. Model Training (scripts/train.py)

python
Copy code

```python
import torch
import torch.nn as nn
import torch.optim as optim
from model import BinaryClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import joblib

# Generate synthetic dataset for binary classification
```

```
X, y = make_classification(n_samples=1000, n_features=20,
random_state=42)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize features using Scikit-learn's StandardScaler
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Save the scaler for later use during inference
joblib.dump(scaler, "../scaler.pkl")

# Initialize the PyTorch model
model = BinaryClassifier(input_dim=X_train.shape[1])
criterion = nn.BCELoss()  # Binary Cross-Entropy loss for binary
classification
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Convert data to PyTorch tensors
X_train_tensor = torch.FloatTensor(X_train)
y_train_tensor = torch.FloatTensor(y_train).unsqueeze(1)  # Add
dimension for compatibility

# Train the model for 100 epochs
for epoch in range(100):
    # Zero gradients to prevent accumulation
    optimizer.zero_grad()
    # Forward pass through the model
    outputs = model(X_train_tensor)
    # Compute the loss
    loss = criterion(outputs, y_train_tensor)
    # Backward pass to compute gradients
    loss.backward()
    # Update weights
    optimizer.step()

    # Print loss every 10 epochs
    if epoch % 10 == 0:
```

```
        print(f"Epoch [{epoch}/100], Loss: {loss.item()}")

# Save the trained model
torch.save(model.state_dict(), "../model.pth")
```

---

4. API for Predictions (app/main.py)

python
Copy code

```
from fastapi import FastAPI
import torch
import joblib
import numpy as np
from model import BinaryClassifier

# Create a FastAPI app instance
app = FastAPI()

# Load the saved model and scaler
model = BinaryClassifier(input_dim=20)  # Initialize model structure
model.load_state_dict(torch.load("../model.pth"))  # Load weights
model.eval()  # Set model to evaluation mode

scaler = joblib.load("../scaler.pkl")  # Load the saved scaler

# Define an API endpoint for predictions
@app.post("/predict")
async def predict(data: dict):
    """
    Perform inference for binary classification.
    Input:
        data (dict): A dictionary containing input features.
    Output:
        Prediction result (class 0 or 1).
    """
    # Extract and scale input features
    inputs = np.array(data['inputs'])  # Convert input to NumPy
array
    inputs = scaler.transform([inputs])  # Scale using pre-saved
scaler
```

```
    inputs = torch.FloatTensor(inputs)  # Convert to PyTorch tensor

    # Perform inference with the model
    with torch.no_grad():
        outputs = model(inputs)  # Forward pass through the model
        prediction = (outputs > 0.5).int().item()  # Threshold to
get class (0 or 1)

    # Return prediction result
    return {"prediction": prediction}
```

---

5. Kubernetes Deployment (deployment/deployment.yaml)

yaml
Copy code

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ml-api-deployment
spec:
  replicas: 3  # Set 3 replicas for scalability
  selector:
    matchLabels:
      app: ml-api
  template:
    metadata:
      labels:
        app: ml-api
    spec:
      containers:
      - name: ml-api-container
        image: ml-api:latest  # Docker image for the API
        ports:
        - containerPort: 8000  # Expose port 8000
---
apiVersion: v1
kind: Service
metadata:
  name: ml-api-service
```

```
spec:
  type: LoadBalancer  # Expose the service externally
  ports:
  - port: 80
    targetPort: 8000
  selector:
    app: ml-api  # Connect service to deployment pods
```

---

6. Test Suite (tests/test_main.py)

python
Copy code

```
from fastapi.testclient import TestClient
from main import app

# Initialize the test client for the FastAPI app
client = TestClient(app)

# Test the /predict endpoint
def test_predict():
    # Send sample input for prediction
    response = client.post("/predict/", json={"inputs": [0.5, -1.2,
0.8, 1.5, 0.4, 0.3, 0.7, -0.6, 0.5, 1.0,
                                                         -1.3, 0.9,
-0.5, 0.4, 1.2, -0.8, 0.7, -0.9, 0.3, 1.1]})
    # Verify the response code is 200 (OK)
    assert response.status_code == 200
    # Ensure the prediction key exists
    assert "prediction" in response.json()
```