

Lab 1: Converting Jupyter Notebooks to Production-Ready Python Projects

Overview

Welcome to the first lab of your ML Deployment journey! In this lab, you'll transform your exploratory Jupyter notebook from your previous ML course into a well-structured, modular Python project that's ready for deployment.

In your previous course, you assembled comprehensive Jupyter Notebooks that included exploratory data analysis (EDA), data preprocessing, feature engineering, and model development—culminating in the validation of your chosen approaches. Whether your project focused on classification, regression, time-series prediction, or any other ML task, all these Jupyter Notebook experiments share a common need to be **reproducible, maintainable, and ready for deployment**.

While the notebook stage is perfect for prototyping and rapid iteration, production projects demand best practices for code organization, versioning, testing, and environment management. By converting your existing work into a well-structured machine learning project, you ensure that others (or future versions of you!) can easily understand, reproduce, and extend the solution.

This lab asks you to apply software engineering and MLOps principles—modularizing your code, automating pipelines, and setting up environments that can be deployed across different systems. In doing so, you will transform your specialized ML work into an industrial-strength project ready for real-world usage.

Learning Objectives

By the end of this lab, you will:

1. **Project Structuring:** Apply best practices to organize your project files and folders
2. **Software Engineering Principles:** Incorporate modular, well-documented code with proper separation of concerns
3. **MLOps Understanding:** Practice the principles of the machine learning lifecycle, from data ingestion to model evaluation, and set up a project structure ready for continuous integration and deployment
4. **Configuration and Extensibility:** Demonstrate how to use configuration files to manage different training or preprocessing parameters
5. **Notebook Organization:** Create clear, focused Jupyter notebooks for exploration, prototyping, or demonstration
6. **Version Control Proficiency:** Understand Git best practices for ML projects
7. **Foundation for Advanced Tools:** Prepare your project structure for future integration with DVC (Data Version Control) and MLflow (experiment tracking)

What We Did in the Tutorial

In the tutorial session, we convert the **Telecom Customer Churn Prediction** notebook into a structured project called `cmpt2500f25-project-tutorial` (available at: <https://github.com/NorQuest-MLAD-Courses/cmpt2500f25-project-tutorial.git>). You can reference this as an example while working on your own project.

Part 1: Understanding the Why

Why Convert Notebooks to Python Modules?

Jupyter notebooks are **excellent for**:

- 📊 Exploratory Data Analysis (EDA)
- 🛠️ Experimenting with models
- 📈 Visualizing results
- 📖 Teaching and documentation
- 🔬 Research and prototyping

However, notebooks are **poor for**:

- 🚀 Production deployment
- 🔄 Code reusability
- 🛠️ Automated testing
- 👥 Team collaboration
- 🔧 Version control
- 📦 Packaging and distribution
- ⚡ Performance optimization

The Production Mindset

Moving from research/learning to production requires a shift in thinking:

Research/Learning	Production
Get it working	Get it working reliably
Run once	Run thousands of times
One dataset	Many datasets
Manual execution	Automated pipelines
Jupyter notebooks	Python modules
Print statements	Proper logging
Hard-coded values	Configuration files
"It works on my machine"	"It works everywhere"

The Journey Ahead

This lab is **Step 1** of your deployment journey:

1. ✅ **Lab 1:** Convert notebook → Modular code (You are here!)
2. **Lab 2:** Add dependencies, CLI, DVC for data versioning, MLflow for experiment tracking, and testing
3. **Lab 3:** Build REST API (with Flask/FastAPI)
4. **Lab 4:** Containerize (with Docker)

5. **Lab 5:** Deploy to cloud (AWS/GCP/Azure)

6. **Lab 6:** Add monitoring (using Grafana and Prometheus) and CI/CD pipelines

Each step builds on the previous one, so getting the foundation right is crucial!

Part 2: Project Structure

Industry-Standard ML Project Layout

Here's an example of a comprehensive directory structure we'll create:

```

your-project-name/
├── data/                                # Data directory (will be managed by DVC
in Lab 2)
│   ├── raw/                            # Original, immutable data
│   ├── processed/                      # Cleaned, processed data
│   └── external/                       # External data sources (optional)
├── models/                             # Saved model files (.pkl, .h5, .joblib,
.pt)
├── notebooks/                          # Jupyter notebooks for EDA and
exploration
│   ├── proof_of_concept.ipynb
│   ├── 01_exploratory_data_analysis.ipynb
│   ├── 02_feature_engineering.ipynb
│   └── 03_model_prototyping.ipynb
├── src/                                # Source code for production
│   ├── __init__.py                    # Makes src a Python package
│   ├── preprocess.py                  # Data loading & preprocessing
│   ├── train.py                       # Model training functions
│   ├── predict.py                    # Prediction functions
│   ├── evaluate.py                   # Model evaluation metrics
│   ├── feature_engineering.py        # Feature creation & selection (if needed)
│   └── utils/                         # Shared helper functions/classes
│       ├── __init__.py
│       ├── config.py                 # Configuration constants
│       ├── model_utils.py            # Model-related utilities
│       └── helpers.py                # General helper functions
├── configs/                            # Configuration files (optional but
recommended)
│   ├── train_config.yaml              # Training hyperparameters
│   └── preprocess_config.yaml         # Preprocessing parameters
├── tests/                              # Unit tests (we'll add these in Lab 2)
│   ├── __init__.py
│   ├── test_preprocess.py
│   ├── test_train.py
│   └── test_predict.py
├── outputs/                            # Plots, reports, results
├── logs/                              # Log files (optional)
├── experiments/                        # Experiment tracking (will use MLflow in
Lab 2)
└── docs/                              # Project documentation

```

```

├── README.md
├── requirements.txt      # Python dependencies
├── Makefile             # Automation tasks
├── .gitignore           # Git ignore rules
└── README.md            # Main project documentation

```

Why This Structure?

- **data/**: Separates raw and processed data. Raw data is **immutable** (never modify the original!). In Lab 2, we'll use DVC to version control this data without storing it in Git.
- **models/**: Centralized location for saved models. Makes it easy to load/deploy. Will be tracked by DVC in Lab 2.
- **notebooks/**: Keeps exploratory work separate from production code. Use numbered prefixes (01_, 02_) for logical ordering. This also stores our original proof of concept notebook.
- **src/**: All production code goes here. Each file has a single responsibility.
- **configs/**: Store YAML or JSON configuration files for preprocessing parameters, training hyperparameters, or environment-specific settings.
- **tests/**: Automated tests ensure code works correctly (we'll add these in Lab 2).
- **experiments/**: Keep records from your experiments. In Lab 2, MLflow will automatically track experiments here.
- **logs/**: Capture logs or run outputs for debugging and monitoring.
- **docs/**: Use for documentation (e.g., usage instructions, developer notes, architecture diagrams).
- **outputs/**: Organized location for results, plots, reports.

The **.gitkeep** Pattern

You'll notice **.gitkeep** files in empty directories. This is because **Git doesn't track empty directories**. The **.gitkeep** files are empty placeholders that ensure the directory structure is preserved when others clone your repository.

Example:

```

data/
├── raw/
│   └── WA_Fn-UseC_-Telco-Customer-Churn.csv
├── processed/
│   └── .gitkeep      # Preserves the directory structure
│                   # even when no data files are present yet

```

Once you add actual files to these directories, you can delete the **.gitkeep** files (or just leave them - they're harmless).

Part 3: Breaking Down Your Notebook

Step 1: Analyze Your Notebook

Open your ML notebook and identify these sections:

1. Imports and Setup

- Library imports
- Random seeds
- Configuration values

2. Data Loading

- Reading files (for example, data files like CSV/Excel files)
- API calls
- Database queries

3. Data Preprocessing

- Handling missing values
- Encoding categorical variables
- Feature scaling
- Train-test splitting

4. Feature Engineering (if applicable)

- Creating new features
- Feature selection
- Dimensionality reduction

5. Model Training

- Model initialization
- Training loops
- Hyperparameter tuning

6. Model Evaluation

- Accuracy, precision, recall
- Confusion matrices
- ROC curves

7. Predictions

- Making predictions on new data

8. Visualization (keep in notebooks!)

- Plots and charts for EDA

Step 2: Map Notebook Sections to Modules

Here's how to organize your code:

Notebook Section	Target Module	Purpose
------------------	---------------	---------

Notebook Section	Target Module	Purpose
Imports & Config	<code>utils/config.py</code>	Central configuration
Data Loading	<code>preprocess.py</code>	Load data function
Preprocessing	<code>preprocess.py</code>	Clean and prepare data
Feature Engineering	<code>feature_engineering.py</code>	Create and select features
Model Training	<code>train.py</code>	Train different models
Evaluation	<code>evaluate.py</code>	Calculate metrics
Predictions	<code>predict.py</code>	Make predictions
Visualization	Keep in <code>notebooks/</code>	EDA stays in notebooks!

Note: Exploratory Data Analysis (EDA) and visualizations will stay in notebooks. They're for understanding data, not for production deployment.

Part 4: Creating Your Modules

Module 1: `src/utils/config.py`

This file contains all configuration constants. **Never hard-code values in your functions!**

What goes here:

- File paths
- Random seeds
- Train-test split ratios
- Model hyperparameters (default values)
- Feature names
- Column names to drop

Example from tutorial project:

```
"""
Configuration file for the ML project.
"""

import os

# Paths
# Get the project root directory by going up three levels from the current
# file's location
# os.path.abspath(__file__) returns the absolute path of the current file
# Each os.path.dirname() removes one level from the path (goes up one
# directory)
BASE_DIR =
os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
```

```

))
# os.path.join() combines path components into a complete path using the
# OS-appropriate separator
# Example: if BASE_DIR = "/project", this creates "/project/data/raw"
# (Mac/Linux) or "\\project\\data\\raw" (Windows)
DATA_RAW_PATH = os.path.join(BASE_DIR, "data", "raw")
DATA_PROCESSED_PATH = os.path.join(BASE_DIR, "data", "processed")
MODELS_PATH = os.path.join(BASE_DIR, "models")
OUTPUTS_PATH = os.path.join(BASE_DIR, "outputs")

# Model parameters
# RANDOM_STATE controls random number generation for reproducibility (same
# results every time)
# The value 42 is arbitrary but conventionally used; any integer works
RANDOM_STATE = 42
TEST_SIZE = 0.2

DATA_FILENAME = "WA_Fn-UseC_-Telco-Customer-Churn.csv"

# Feature columns
CATEGORICAL_FEATURES = [
    'customerID', 'gender', 'Partner', 'Dependents',
    'PhoneService', 'MultipleLines', 'InternetService',
    'OnlineSecurity', 'OnlineBackup', 'DeviceProtection',
    'TechSupport', 'StreamingTV', 'StreamingMovies',
    'Contract', 'PaperlessBilling', 'PaymentMethod',
    'TotalCharges', 'Churn'
]

NUMERICAL_FEATURES = ['SeniorCitizen', 'tenure', 'MonthlyCharges']

TARGET = 'Churn'

# Columns to drop
DROP_COLUMNS = ['customerID']

```

Your task: Extract all hard-coded values from your notebook into this config file.

Module 2: `src/preprocess.py`

This module handles all data loading and preprocessing.

Key functions to create:

1. `load_data(filepath: str) -> pd.DataFrame`
 - Loads data from CSV/Excel/database
 - Returns a pandas DataFrame
 - Includes error handling
2. `handle_missing_values(df: pd.DataFrame) -> pd.DataFrame`
 - Fills or drops missing values

- Documents the strategy used

3. `encode_categorical_features(df: pd.DataFrame) -> pd.DataFrame`

- Label encoding or one-hot encoding
- Handles the target variable separately

4. `drop_unnecessary_columns(df: pd.DataFrame) -> pd.DataFrame`

- Removes ID columns, timestamps, etc.

5. `split_features_target(df: pd.DataFrame) -> Tuple[pd.DataFrame, pd.Series]`

- Separates features (X) from target (y)

6. `split_train_test(X, y) -> Tuple`

- Creates train-test split
- Uses random state from config

7. `scale_features(X_train, X_test) -> Tuple`

- Applies StandardScaler or MinMaxScaler
- Fits on training data, transforms both
- Returns scaler for later use

8. `preprocess_pipeline(filepath: str) -> Tuple`

- Orchestrates all preprocessing steps
- One function to rule them all!

Understanding Logging and Type Hints

Before we dive into the code examples, let's understand two important concepts you'll see throughout this lab: **logging** and **type hints** (typing).

What is Logging?

Logging is Python's built-in system for recording messages about what your program is doing. Think of it as a professional replacement for `print()` statements.

Why not just use `print()`?

In your notebooks, you probably used `print()` statements like this:

```
print("Loading data...")
print(f"Data shape: {df.shape}")
print("Training complete!")
```

This works fine for notebooks and quick scripts, but in production code, `print()` has major limitations:

- ❌ All messages go to the same place (the screen)

- **✗** No way to control which messages appear (can't filter by importance)
- **✗** Can't save messages to a file for later review
- **✗** No timestamps or context about where the message came from
- **✗** Hard to debug issues that happened in the past

Logging solves all of these problems:

```
import logging

# Set up logging (do this once at the top of each module)
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Instead of print(), use logger methods
logger.info("Loading data...")
logger.info(f"Data shape: {df.shape}")
logger.warning("Missing values detected - filling with mean")
logger.error("Failed to load file!")
```

Logging has different levels of severity:

1. **logger.debug()** - Detailed information for debugging (only shown when debugging)
2. **logger.info()** - General informational messages (replaces most **print()** statements)
3. **logger.warning()** - Something unexpected happened, but the program can continue
4. **logger.error()** - A serious problem occurred
5. **logger.critical()** - A very serious error - the program might crash

Example: Converting **print()** to logging:

```
# ✗ Before (using print in notebook)
def load_data(filepath):
    print("Loading data...")
    df = pd.read_csv(filepath)
    print(f"Loaded {len(df)} rows")
    return df

# ✓ After (using logging in production code)
def load_data(filepath: str) -> pd.DataFrame:
    logger.info("Loading data...")
    df = pd.read_csv(filepath)
    logger.info(f"Loaded {len(df)} rows")
    return df
```

Benefits in production:

- Can save logs to files: **logging.basicConfig(filename='app.log')**
- Can control verbosity: Set **level=logging.WARNING** to only see warnings and errors
- Includes timestamps: **2025-10-25 14:30:15 - INFO - Loading data...**

- Shows where the message came from: `preprocess.py:42 – INFO – Loading data...`

For this lab: Replace all `print()` statements in your notebook code with appropriate `logger` calls when converting to modules.

What are Type Hints (Typing)?

Type hints (also called type annotations) are a way to specify what type of data a function expects and returns. They were introduced in Python 3.5+ to make code more readable and catch errors early.

In your notebooks, you probably wrote functions like this:

```
def calculate_accuracy(y_true, y_pred):
    correct = sum(y_true == y_pred)
    total = len(y_true)
    return correct / total
```

This works, but it raises questions:




- What type should `y_true` and `y_pred` be? Lists? NumPy arrays? Pandas Series?
- What does the function return? A float? An int?
- How would someone else (or future you) know how to use this function?

Type hints make this explicit:

```
import numpy as np

def calculate_accuracy(y_true: np.ndarray, y_pred: np.ndarray) -> float:
    """Calculate classification accuracy."""
    correct = sum(y_true == y_pred)
    total = len(y_true)
    return correct / total
```

Now it's crystal clear:

-  `y_true` should be a NumPy array
-  `y_pred` should be a NumPy array
-  The function returns a float

Common type hints you'll use:

```
from typing import Tuple, Dict, List, Any
import pandas as pd
import numpy as np

# Basic types
def greet(name: str) -> str:
    return f"Hello, {name}"
```

```
def add_numbers(x: int, y: int) -> int:
    return x + y

def calculate_average(numbers: list) -> float:
    return sum(numbers) / len(numbers)

# NumPy and Pandas types
def load_data(filepath: str) -> pd.DataFrame:
    return pd.read_csv(filepath)

def predict(model: Any, X: np.ndarray) -> np.ndarray:
    return model.predict(X)

# Multiple return values (Tuple)
def split_data(df: pd.DataFrame) -> Tuple[pd.DataFrame, pd.Series]:
    X = df.drop('target', axis=1)
    y = df['target']
    return X, y

# Dictionaries
def get_metrics(y_true: np.ndarray, y_pred: np.ndarray) -> Dict[str, float]:
    return {
        'accuracy': accuracy_score(y_true, y_pred),
        'precision': precision_score(y_true, y_pred)
    }

# Lists of specific types
def get_model_names() -> List[str]:
    return ['random_forest', 'logistic_regression', 'xgboost']

# Any type (when you don't know or it can be anything)
def save_model(model: Any, filepath: str) -> None:
    joblib.dump(model, filepath)
    # None means this function doesn't return anything
```

Important **typing** module types:

- **Tuple**: For functions that return multiple values

```
def split_train_test(X, y) -> Tuple[np.ndarray, np.ndarray,
np.ndarray, np.ndarray]:
    return X_train, X_test, y_train, y_test
```

- **Dict**: For dictionaries with specific key/value types

```
def get_config() -> Dict[str, Any]:
    return {'learning_rate': 0.01, 'epochs': 100}
```

- **List**: For lists of specific types

```
def get_feature_names() -> List[str]:  
    return ['age', 'income', 'education']
```

- **Any**: When the type can be anything

```
def load_model(filepath: str) -> Any:  
    return joblib.load(filepath)
```

- **None**: When function doesn't return anything

```
def save_to_file(data: pd.DataFrame, filepath: str) -> None:  
    data.to_csv(filepath)
```

Why use type hints?

1. **Documentation**: Type hints are self-documenting - you can see what types are expected just by looking at the function signature
2. **IDE Support**: Your IDE (VS Code, PyCharm) can provide better autocomplete and catch errors before you run the code
3. **Error Prevention**: Tools like **mypy** can check your code for type errors before runtime
4. **Professional Standard**: Type hints are expected in production Python code
5. **Easier Collaboration**: Other developers (and future you) will understand your code faster

Example: Before and After

```
# ❌ Before (no type hints - from notebook)  
def preprocess_data(df):  
    df = df.dropna()  
    X = df.drop('target', axis=1)  
    y = df['target']  
    return X, y  
  
# ✅ After (with type hints - production code)  
def preprocess_data(df: pd.DataFrame) -> Tuple[pd.DataFrame, pd.Series]:  
    """  
    Preprocess data by removing missing values and splitting  
    features/target.  
  
    Args:  
        df: Input DataFrame with features and target column  
  
    Returns:
```

```
    Tuple of (features DataFrame, target Series)
    """
    df = df.dropna()
    X = df.drop('target', axis=1)
    y = df['target']
    return X, y
```

For this lab: Add type hints to all function parameters and return values when converting your notebook code to modules.

Putting It All Together

Here's how logging and type hints work together in production code:

```
import logging
import pandas as pd
from typing import Tuple

# Set up logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def load_and_split_data(filepath: str) -> Tuple[pd.DataFrame, pd.Series]:
    """
    Load data from CSV and split into features and target.

    Args:
        filepath: Path to the CSV file

    Returns:
        Tuple of (features DataFrame, target Series)
    """
    logger.info(f"Loading data from {filepath}")

    try:
        df = pd.read_csv(filepath)
        logger.info(f"Successfully loaded {len(df)} rows")
    except FileNotFoundError:
        logger.error(f"File not found: {filepath}")
        raise






    if df.empty:
        logger.warning("Loaded DataFrame is empty!")

    X = df.drop('target', axis=1)
    y = df['target']

    logger.info(f"Split data into {X.shape[1]} features and target")
    return X, y
```

Notice how the type hints (`filepath: str, -> Tuple[pd.DataFrame, pd.Series]`) make it clear what goes in and out, while logging (`logger.info()`, `logger.error()`) provides runtime feedback about what's happening.

Key Takeaways:

-  **Use logging instead of print()** for all informational messages
-  **Add type hints to all functions** for clarity and error prevention
-  **Logging = runtime feedback** (what's happening when the code runs)
-  **Type hints = compile-time documentation** (what types the code expects)
-  Together, they make your code **professional, maintainable, and production-ready**

Now let's see these concepts in action with our preprocessing module!

Module 2: `src/preprocess.py` (Continued)

Example function structure:

```
import pandas as pd
import logging
from typing import Tuple

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







def load_data(filepath: str) -> pd.DataFrame:
    """
    Load data from CSV file.

    Args:
        filepath: Path to the CSV file

    Returns:
        DataFrame containing the loaded data

    Raises:
        FileNotFoundError: If file doesn't exist
        pd.errors.EmptyDataError: If file is empty
    """
    try:
        df = pd.read_csv(filepath)
        logger.info(f"Data loaded successfully. Shape: {df.shape}")
        return df
    except FileNotFoundError:
        logger.error(f"File not found: {filepath}")
        raise
    except pd.errors.EmptyDataError:
        logger.error(f"File is empty: {filepath}")
        raise
    except Exception as e:
        logger.error(f"Error loading data: {e}")
        raise
```

Key principles:

-  One function = one responsibility
-  Type hints on all parameters and returns
-  Docstrings explaining purpose, args, returns, and exceptions
-  Use logging instead of print statements
-  Error handling with try-except
-  Return copies, don't modify in place (unless explicitly stated)

Module 3: `src/train.py`

This module contains all model training functions.

Key functions to create:**1. One function per model type:**

- `train_logistic_regression(X_train, y_train, **kwargs)`
- `train_random_forest(X_train, y_train, **kwargs)`
- `train_xgboost(X_train, y_train, **kwargs)`
- etc.

2. `train_all_models(X_train, y_train) -> Dict[str, Any]`

- Trains all your models
- Returns dictionary of {model_name: trained_model}

3. `save_model(model, model_name: str, output_dir: str) -> str`

- Saves model using pickle/joblib
- Adds timestamp to filename for versioning
- Returns path to saved file

4. `save_all_models(models: Dict, output_dir: str) -> Dict[str, str]`

- Saves multiple models
- Returns dictionary of {model_name: filepath}

Example:

```
import logging
from typing import Any
from datetime import datetime
from sklearn.ensemble import RandomForestClassifier
from .utils.config import RANDOM_STATE

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

def train_random_forest(X_train: np.ndarray, y_train: np.ndarray,
```

```

**kwargs) -> RandomForestClassifier:
    """
    Train Random Forest Classifier.

    Args:
        X_train: Training features
        y_train: Training target
        **kwargs: Additional parameters for RandomForestClassifier

    Returns:
        Trained RandomForestClassifier model
    """
    logger.info("Training Random Forest...")

    # Set default random state if not provided
    if 'random_state' not in kwargs:
        kwargs['random_state'] = RANDOM_STATE

    model = RandomForestClassifier(**kwargs)
    model.fit(X_train, y_train)

    logger.info("Random Forest training completed")
    return model

def save_model(model: Any, model_name: str, output_dir: str = 'models') ->
str:
    """
    Save a trained model to disk with timestamp for versioning.

    Args:
        model: Trained model object to save
        model_name: Name for the model (e.g., 'random_forest',
'logistic_regression')
        output_dir: Directory where model will be saved (default:
'models')

    Returns:
        String path to the saved model file

    Example:
        >>> model = RandomForestClassifier()
        >>> model.fit(X_train, y_train)
        >>> filepath = save_model(model, 'random_forest')
        >>> print(filepath)
        'models/random_forest_20251025_143022.pkl'
    """
    import joblib
    from pathlib import Path

    # Create output directory if it doesn't exist
    Path(output_dir).mkdir(parents=True, exist_ok=True)

    # Add timestamp to filename for versioning
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')

```



```

filename = f"{model_name}_{timestamp}.pkl"
filepath = Path(output_dir) / filename

# Save the model using joblib (optimized for sklearn models)
joblib.dump(model, filepath)
logger.info(f"Model saved successfully to: {filepath}")

return str(filepath)

def save_all_models(models: Dict[str, Any], output_dir: str = 'models') ->
Dict[str, str]:
    """
    Save multiple trained models to disk.

    Args:
        models: Dictionary mapping model names to trained model objects
            Example: {'random_forest': rf_model,
'logistic_regression': lr_model}
        output_dir: Directory where models will be saved (default:
'models')

    Returns:
        Dictionary mapping model names to their saved file paths

    Example:
        >>> models = {
        ...     'random_forest': rf_model,
        ...     'logistic_regression': lr_model,
        ...     'xgboost': xgb_model
        ... }
        >>> paths = save_all_models(models)
        >>> print(paths)
        {'random_forest': 'models/random_forest_20251025_143022.pkl',
'logistic_regression':
'models/logistic_regression_20251025_143025.pkl',
'xgboost': 'models/xgboost_20251025_143028.pkl'}
    """
    saved_paths = {}

    for model_name, model in models.items():
        filepath = save_model(model, model_name, output_dir)
        saved_paths[model_name] = filepath

    logger.info(f"Successfully saved {len(models)} models to
{output_dir}")
    return saved_paths

```

Why separate functions for each model?

- Easier to test individual models
- Can train models in parallel later
- Clear separation of concerns

- Easy to add new models

Understanding Model Serialization: Saving Models with Pickle and Joblib

In your previous ML courses, you might have trained models and used them immediately in the same notebook session. But what happens when you close the notebook? Your trained model is **gone** - all that training time wasted! In production, we need to **save trained models to disk** so we can load and use them later without retraining.

This process is called **serialization** (saving an object to a file) and **deserialization** (loading an object from a file).

Why Save Models?

The Problem:

```
# In your notebook - you probably did this:
model = RandomForestClassifier()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
# Model is lost when you close the notebook! 🤖
```

The Solution:

```
# Train once
model = RandomForestClassifier()
model.fit(X_train, y_train)

# Save to disk
joblib.dump(model, 'models/my_model.pkl')

# Later (or in a different script/API):
model = joblib.load('models/my_model.pkl')
predictions = model.predict(X_new) # Model still works!
```

Real-world scenarios where you need to save models:

1. **Training takes hours/days** - You don't want to retrain every time you need to make predictions
2. **Deployment** - Your web API needs to load a pre-trained model to make predictions
3. **Model versioning** - Save different versions to compare performance over time
4. **Sharing** - Send your trained model to colleagues or deploy to production
5. **Reproducibility** - Save the exact model used in your research/experiments
6. **Production pipelines** - Train models in one script, use them in another

Pickle vs. Joblib: What's the Difference?

Python offers two main ways to serialize machine learning models: **pickle** and **joblib**. Let's understand both.

Pickle: Python's Built-in Serialization

Pickle is Python's built-in module for serializing Python objects. It can save almost any Python object to a file.

Basic usage:

```
import pickle

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

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

What else can pickle save?

Pickle isn't just for models - it can serialize almost any Python object:

```
import pickle

# Save a dictionary
config = {'learning_rate': 0.01, 'epochs': 100}
with open('config.pkl', 'wb') as f:
    pickle.dump(config, f)

# Save a list
feature_names = ['age', 'income', 'education']
with open('features.pkl', 'wb') as f:
    pickle.dump(feature_names, f)

# Save a custom class instance
class DataProcessor:
    def __init__(self):
        self.scaler = StandardScaler()

    def process(self, X):
        return self.scaler.fit_transform(X)

processor = DataProcessor()
with open('processor.pkl', 'wb') as f:
    pickle.dump(processor, f)

# Save multiple objects at once
with open('bundle.pkl', 'wb') as f:
    pickle.dump({'model': model, 'scaler': scaler, 'features':
feature_names}, f)
```

Pickle use cases beyond ML:

- Caching expensive computations
- Saving game state
- Serializing complex data structures
- Inter-process communication
- Session management in web apps

Joblib: Optimized for NumPy Arrays

Joblib is a library that's part of the scikit-learn ecosystem. It's specifically optimized for objects that contain large NumPy arrays (like most ML models).

Basic usage:

```
import joblib

# Save a model (simpler syntax!)
joblib.dump(model, 'model.pkl')

# Load a model
model = joblib.load('model.pkl')
```

What else can joblib do?

Besides serialization, joblib provides powerful utilities for:

1. **Parallel computing** - Run functions in parallel

```
from joblib import Parallel, delayed

# Train multiple models in parallel
def train_model(params):
    model = RandomForestClassifier(**params)
    model.fit(X_train, y_train)
    return model

param_sets = [
    {'n_estimators': 100},
    {'n_estimators': 200},
    {'n_estimators': 300}
]

# Train all 3 models in parallel using 3 CPU cores
models = Parallel(n_jobs=3)(delayed(train_model)(params) for params in
                             param_sets)
```

2. **Caching function results** - Avoid recomputing expensive operations

```

from joblib import Memory

memory = Memory(location='cache_dir', verbose=0)

@memory.cache
def expensive_preprocessing(filepath):
    # This will only run once per unique filepath
    # Subsequent calls will load from cache
    df = pd.read_csv(filepath)
    # ... expensive operations ...
    return processed_data

```

Pickle vs. Joblib: Which Should You Use?

Feature	Pickle	Joblib
Speed for large arrays	Slower	Much faster
File size for arrays	Larger	Smaller
Syntax	Verbose (<code>with open(...)</code>)	Simple (<code>joblib.dump()</code>)
Best for	Simple objects, built-in types	ML models, NumPy arrays
Part of Python	Built-in	Requires installation
Compression	Manual	Automatic (can compress)
scikit-learn models	Works	Optimized for this

Recommendation for ML projects:

✅ Use joblib for:

- Scikit-learn models (RandomForest, LogisticRegression, etc.)
- Any model with large NumPy arrays
- Scalers (StandardScaler, MinMaxScaler)
- Encoders (LabelEncoder, OneHotEncoder)
- Vectorizers (CountVectorizer, TfidfVectorizer)

⚠ Use pickle for:

- Simple Python objects (dicts, lists, strings)
- Custom classes without large arrays
- When joblib isn't available

Practical Examples for Your Lab

Example 1: Save a single model with joblib

```

import joblib
import logging
from datetime import datetime
from pathlib import Path

logger = logging.getLogger(__name__)

def save_model(model: Any, model_name: str, output_dir: str = 'models') ->
str:
    """
    Save a trained model to disk with timestamp.

    Args:
        model: Trained model object
        model_name: Name for the model (e.g., 'random_forest')
        output_dir: Directory to save the model

    Returns:
        Path to the saved model file
    """
    # Create output directory if it doesn't exist
    Path(output_dir).mkdir(parents=True, exist_ok=True)

    # Add timestamp for versioning
    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    filename = f"{model_name}_{timestamp}.pkl"
    filepath = Path(output_dir) / filename

    # Save the model
    joblib.dump(model, filepath)
    logger.info(f"Model saved to: {filepath}")

    return str(filepath)

# Usage
model = RandomForestClassifier()
model.fit(X_train, y_train)
model_path = save_model(model, 'random_forest')
# Saves to: models/random_forest_20251025_143022.pkl

```

Example 2: Save multiple models

```

def save_all_models(models: Dict[str, Any], output_dir: str = 'models') ->
Dict[str, str]:
    """
    Save multiple trained models to disk.

    Args:
        models: Dictionary of {model_name: trained_model}
        output_dir: Directory to save models
    """

```

```

Returns:
    Dictionary of {model_name: filepath}
    """
    saved_paths = {}

    for model_name, model in models.items():
        filepath = save_model(model, model_name, output_dir)
        saved_paths[model_name] = filepath

    logger.info(f"Saved {len(models)} models to {output_dir}")
    return saved_paths

# Usage
models = {
    'random_forest': rf_model,
    'logistic_regression': lr_model,
    'xgboost': xgb_model
}
paths = save_all_models(models)
# Returns: {'random_forest': 'models/random_forest_20251025_143022.pkl',
...}

```

Example 3: Save model with preprocessing objects

```

def save_model_bundle(model: Any, scaler: Any, encoder: Any,
                      model_name: str, output_dir: str = 'models') -> str:
    """
    Save model along with preprocessing objects.

    This is important because you need the SAME scaler and encoder
    used during training to preprocess new data for predictions!
    """
    Path(output_dir).mkdir(parents=True, exist_ok=True)

    timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
    filename = f"{model_name}_bundle_{timestamp}.pkl"
    filepath = Path(output_dir) / filename

    # Bundle everything together
    bundle = {
        'model': model,
        'scaler': scaler,
        'encoder': encoder,
        'timestamp': timestamp
    }

    joblib.dump(bundle, filepath)
    logger.info(f"Model bundle saved to: {filepath}")

    return str(filepath)

```

```
# Usage
bundle_path = save_model_bundle(model, scaler, encoder,
'complete_pipeline')

# Later, load everything back:
bundle = joblib.load(bundle_path)
model = bundle['model']
scaler = bundle['scaler']
encoder = bundle['encoder']
```

Example 4: Load model with error handling

```
def load_model(filepath: str) -> Any:
    """
    Load a trained model from disk with error handling.

    Args:
        filepath: Path to the saved model file

    Returns:
        Loaded model object

    Raises:
        FileNotFoundError: If model file doesn't exist
    """
    try:
        model = joblib.load(filepath)
        logger.info(f"Model loaded successfully from: {filepath}")
        return model
    except FileNotFoundError:
        logger.error(f"Model file not found: {filepath}")
        raise
    except Exception as e:
        logger.error(f"Error loading model: {e}")
        raise

# Usage
model = load_model('models/random_forest_20251025_143022.pkl')
predictions = model.predict(X_new)
```

Example 5: Compressed saving for large models

```
# Joblib can compress models to save disk space
# Level 0-9: 0 = no compression, 9 = maximum compression

# Save with compression
joblib.dump(model, 'model.pkl', compress=3) # Good balance of speed/size
```



```

joblib.dump(model, 'model.pkl.gz', compress=9) # Maximum compression

# Automatic based on file extension
joblib.dump(model, 'model.pkl.gz') # Automatically compressed
joblib.dump(model, 'model.pkl')    # Not compressed

# Load works the same way regardless of compression
model = joblib.load('model.pkl.gz')

```

Important Considerations

⚠ Security Warning:

Both pickle and joblib can execute arbitrary code when loading files. **Never load pickle/joblib files from untrusted sources!** A malicious pickle file can execute harmful code on your machine.

```

# ❌ DANGEROUS – Don't load models from unknown sources
model = joblib.load('random_internet_model.pkl') # Could be malicious!

# ✅ SAFE – Only load your own models or from trusted sources
model = joblib.load('models/my_model.pkl') # Your own model

```

⚠ Version Compatibility:

Models saved with one version of scikit-learn might not load correctly with another version. Always document which versions you used:

```

import sklearn
import joblib

def save_model_with_version(model, filepath):
    """Save model with library version info."""
    bundle = {
        'model': model,
        'sklearn_version': sklearn.__version__,
        'python_version': sys.version
    }
    joblib.dump(bundle, filepath)

# Also save a text file with version info
with open(f"{filepath}.info.txt", 'w') as f:
    f.write(f"scikit-learn: {sklearn.__version__}\n")
    f.write(f"Python: {sys.version}\n")

```

⚠ File Naming Best Practices:

```
# ❌ Bad – will overwrite previous versions
joblib.dump(model, 'model.pkl')

# ✅ Good – includes timestamp for versioning
timestamp = datetime.now().strftime('%Y%m%d_%H%M%S')
joblib.dump(model, f'model_{timestamp}.pkl')

# ✅ Even better – includes model name and timestamp
joblib.dump(model, f'random_forest_{timestamp}.pkl')

# ✅ Best – includes all important info
joblib.dump(model, f'random_forest_accuracy_0.95_{timestamp}.pkl')
```

Summary Table: When to Use What

Task	Tool	Code
Save ML model	joblib	<code>joblib.dump(model, 'model.pkl')</code>
Load ML model	joblib	<code>model = joblib.load('model.pkl')</code>
Save with compression	joblib	<code>joblib.dump(model, 'model.pkl', compress=3)</code>
Save simple dict	pickle	<code>pickle.dump(obj, open('file.pkl', 'wb'))</code>
Save model + scaler	joblib	<code>joblib.dump({'model': m, 'scaler': s}, 'bundle.pkl')</code>
Parallel processing	joblib	<code>Parallel(n_jobs=-1)(...)</code>
Cache results	joblib	<code>@memory.cache</code> decorator

Key Takeaways

- ✅ **Always save trained models** - Don't waste training time by losing models when scripts end
- ✅ **Use joblib for ML models** - It's faster and more efficient for scikit-learn models
- ✅ **Use pickle for simple objects** - Good for configs, lists, dicts without large arrays
- ✅ **Include timestamps** - Never overwrite models; version them instead
- ✅ **Save preprocessing objects too** - Save scalers, encoders with your models
- ✅ **Add error handling** - Always handle `FileNotFoundError` and other exceptions
- ✅ **Document versions** - Track which library versions were used
- ✅ **Never load untrusted pickles** - Security risk!
- ✅ **Joblib has other uses** - Parallel processing and caching beyond serialization

For this lab: Use joblib to save your trained models with timestamps, and implement proper loading functions with error handling.

Module 4: `src/predict.py`

This module handles predictions and serves as an **independent software component**.

Key functions to create:

1. **load_model(filepath: str) -> Any**
 - Loads a saved model from disk
 - Returns the model object
2. **predict(model, X) -> np.ndarray**
 - Makes predictions using the model
 - Returns array of predictions
3. **predict_proba(model, X) -> np.ndarray**
 - Gets prediction probabilities (for classifiers that support it)
 - Returns probability array
4. **predict_single(model, features: dict) -> tuple**
 - Predicts for a single sample
 - Takes a dictionary of features
 - Returns (prediction, probability)
 - Useful for API endpoints later!
5. **batch_predict(model, X, batch_size: int) -> np.ndarray**
 - Makes predictions in batches
 - Useful for large datasets

Example with class-based approach (recommended for production):

```
"""
Prediction module for making inferences with trained models.
This module can be used independently for production predictions.
"""

import pickle
import joblib
import numpy as np
import logging
from typing import Any, Union
import pandas as pd

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

class ModelPredictor:
    """
    Independent predictor class for loading and using trained models.

    This class is designed to be self-contained and can be used
    independently of the rest of the project for making predictions.
    """
```

```

"""

def __init__(self, model_path: str):
    """
    Initialize the predictor with a trained model.

    Args:
        model_path: Path to the saved model file (.pkl, .joblib, etc.)

    Raises:
        FileNotFoundError: If model file doesn't exist
    """
    self.model = self._load_model(model_path)
    self.model_path = model_path
    logger.info(f"ModelPredictor initialized with model from:
{model_path}")

def _load_model(self, filepath: str) -> Any:
    """Load a trained model from disk."""
    try:
        # Try joblib first (more efficient for sklearn models)
        model = joblib.load(filepath)
        logger.info(f"Model loaded successfully from: {filepath}")
        return model
    except:
        # Fall back to pickle
        try:
            with open(filepath, 'rb') as f:
                model = pickle.load(f)
            logger.info(f"Model loaded successfully (pickle) from:
{filepath}")
            return model
        except Exception as e:
            logger.error(f"Error loading model: {e}")
            raise

def predict(self, input_data: Union[np.ndarray, pd.DataFrame]) ->
np.ndarray:
    """
    Make predictions using the loaded model.

    Args:
        input_data: Features for prediction (numpy array or pandas
DataFrame)

    Returns:
        Array of predictions

    Example:
        >>> predictor = ModelPredictor("models/my_model.pkl")
        >>> X_new = np.array([[5.1, 3.5, 1.4, 0.2]])
        >>> predictions = predictor.predict(X_new)
    """
    try:

```

```

        predictions = self.model.predict(input_data)
        logger.info(f"Generated predictions for {len(predictions)}
samples")
        return predictions
    except Exception as e:
        logger.error(f"Error making predictions: {e}")
        raise

    def predict_proba(self, input_data: Union[np.ndarray, pd.DataFrame]) -
> np.ndarray:
        """
        Get prediction probabilities (for classifiers).

        Args:
            input_data: Features for prediction

        Returns:
            Array of prediction probabilities
        """
        if not hasattr(self.model, 'predict_proba'):
            logger.warning("Model does not support probability
predictions")
            return None

        try:
            probabilities = self.model.predict_proba(input_data)
            logger.info(f"Generated probabilities for {len(probabilities)}
samples")
            return probabilities
        except Exception as e:
            logger.error(f"Error generating probabilities: {e}")
            raise

    def predict_single(self, features: dict) -> tuple:
        """
        Make prediction for a single sample.

        Args:
            features: Dictionary of feature values

        Returns:
            Tuple of (prediction, probability)

        Example:
            >>> predictor = ModelPredictor("models/my_model.pkl")
            >>> features = {'feature1': 5.1, 'feature2': 3.5}
            >>> prediction, probability =
predictor.predict_single(features)
            """
            # Convert dict to DataFrame
            X = pd.DataFrame([features])

            # Get prediction
            prediction = self.predict(X)[0]

```

```

        # Get probability if available
        proba = None
        if hasattr(self.model, 'predict_proba'):
            proba = self.predict_proba(X)[0]

        logger.info(f"Single prediction: {prediction}")
        return prediction, proba

# Standalone functions for flexibility
def load_model(filepath: str) -> Any:
    """
    Load a trained model from disk.

    Args:
        filepath: Path to the saved model file

    Returns:
        Loaded model object
    """
    try:
        with open(filepath, 'rb') as f:
            model = pickle.load(f)
        logger.info(f"Model loaded successfully from: {filepath}")
        return model
    except Exception as e:
        logger.error(f"Error loading model: {e}")
        raise

def predict(model: Any, X: Union[np.ndarray, pd.DataFrame]) -> np.ndarray:
    """
    Make predictions using a trained model.

    Args:
        model: Trained model object
        X: Features for prediction

    Returns:
        Array of predictions
    """
    try:
        predictions = model.predict(X)
        logger.info(f"Predictions generated for {len(predictions)} samples")
        return predictions
    except Exception as e:
        logger.error(f"Error making predictions: {e}")
        raise

# Example usage demonstration
if __name__ == "__main__":

```

```

# Example 1: Using the ModelPredictor class
predictor =
ModelPredictor(model_path="../../../models/my_trained_model.joblib")

# Create sample input
sample_input = np.array([[5.1, 3.5, 1.4, 0.2]])






# Get predictions
preds = predictor.predict(sample_input)
print("Predictions:", preds)

# Get probabilities
probas = predictor.predict_proba(sample_input)
print("Probabilities:", probas)

# Example 2: Single prediction with feature dict
features = {
    'sepal_length': 5.1,
    'sepal_width': 3.5,
    'petal_length': 1.4,
    'petal_width': 0.2
}
pred, proba = predictor.predict_single(features)
print(f"Single prediction: {pred}, Probability: {proba}")

```

Key Points about ModelPredictor:

-  Independent software component - doesn't require the rest of your project
-  Can handle model versioning
-  Includes input validation
-  Works with both function calls and class instances
-  Ready for API integration in Lab 3

Module 5: `src/evaluate.py`

This module handles model evaluation.

Key functions to create:

1. `calculate_accuracy(y_true, y_pred) -> float`
 - Calculates accuracy score
2. `calculate_metrics(y_true, y_pred) -> Dict[str, float]`
 - Calculates multiple metrics at once
 - Returns dict with accuracy, precision, recall, f1
3. `generate_classification_report(y_true, y_pred) -> str`
 - Generates sklearn's classification report
4. `get_confusion_matrix(y_true, y_pred) -> np.ndarray`

- Computes confusion matrix

5. `calculate_roc_auc(y_true, y_proba) -> float`

- Calculates ROC-AUC score

6. `evaluate_model(model, X_test, y_test) -> Dict[str, Any]`

- Comprehensive evaluation of a model
- Returns all metrics in a dictionary

7. `compare_models(models: Dict, X_test, y_test) -> pd.DataFrame`

- Compares multiple models
- Returns DataFrame with results sorted by performance

Example:

```
import logging
from typing import Any, Dict
import numpy as np
import pandas as pd
from sklearn.metrics import (
    accuracy_score,
    classification_report,
    confusion_matrix,
    roc_auc_score,
    f1_score,
    precision_score,
    recall_score
)

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

def evaluate_model(model: Any, X_test: np.ndarray, y_test: np.ndarray) -> Dict[str, Any]:
    """
    Comprehensive evaluation of a model.

    Args:
        model: Trained model
        X_test: Test features
        y_test: Test labels

    Returns:
        Dictionary containing all evaluation metrics
    """
    logger.info("Evaluating model...")

    # Make predictions
    y_pred = model.predict(X_test)
```



```
# Calculate metrics
results = {
    'accuracy': accuracy_score(y_test, y_pred),
    'precision': precision_score(y_test, y_pred, average='binary'),
    'recall': recall_score(y_test, y_pred, average='binary'),
    'f1_score': f1_score(y_test, y_pred, average='binary'),
    'confusion_matrix': confusion_matrix(y_test, y_pred),
    'classification_report': classification_report(y_test, y_pred)
}

# Add ROC-AUC if model supports probability predictions
if hasattr(model, 'predict_proba'):
    y_proba = model.predict_proba(X_test)
    results['roc_auc'] = roc_auc_score(y_test, y_proba[:, 1])

logger.info("Model evaluation completed")
return results
```

Module 6: `src/feature_engineering.py` (Optional but Recommended)

If you have significant feature engineering in your notebook, create this module.

Possible functions:

- `create_interaction_features(df) -> pd.DataFrame`
- `create_polynomial_features(df) -> pd.DataFrame`
- `create_aggregated_features(df) -> pd.DataFrame`
- `select_features_by_importance(X, y, model) -> pd.DataFrame`
- `get_feature_importance(model, feature_names) -> pd.DataFrame`

Note: Not all projects need this module. If your feature engineering is minimal, you can include it in `preprocess.py`.

Part 5: Object-Oriented Programming (OOP)

Where possible, encapsulate functionality into classes. This makes your code more modular and reusable.

Why Use OOP in ML Projects?

In your class sessions, you learned about the **MVC (Model-View-Controller) design pattern**, which is a perfect example of how Object-Oriented Programming helps organize code by separating concerns:

- **Model:** Handles data and business logic
- **View:** Manages presentation and user interface
- **Controller:** Coordinates between Model and View

While ML projects don't always follow MVC exactly, the same **separation of concerns principle** applies! By using classes, we can organize our ML pipeline into logical components:

- **Data Layer** (like Model in MVC): Classes that handle data loading, validation, and preprocessing
- **Training Layer** (like Controller in MVC): Classes that orchestrate model training, hyperparameter tuning
- **Prediction Layer** (like View in MVC): Classes that handle inference and output formatting

Benefits of OOP in ML projects:

1. **Encapsulation**: Group related data and functions together (e.g., `DataLoader` class bundles data path with loading methods)
2. **Reusability**: Create a class once, use it across multiple projects
3. **Maintainability**: Changes to implementation don't affect the interface
4. **State Management**: Objects maintain their state (e.g., `Trainer` remembers the trained model)
5. **Clear Interfaces**: Classes provide clear contracts for how to interact with functionality
6. **Separation of Concerns**: Just like MVC separates presentation from logic, classes separate data processing from model training from prediction

Example: ML Pipeline as Separation of Concerns

```
# Data Layer – handles all data operations
class DataProcessor:
    def __init__(self, config):
        self.config = config
        self.scaler = None

    def load_and_preprocess(self, filepath):
        # All data logic encapsulated here
        pass

# Training Layer – handles model training
class ModelTrainer:
    def __init__(self, model_type, params):
        self.model_type = model_type
        self.params = params
        self.model = None

    def train(self, X_train, y_train):
        # All training logic encapsulated here
        pass

# Prediction Layer – handles inference
class ModelPredictor:
    def __init__(self, model_path):
        self.model = self.load_model(model_path)

    def predict(self, X):
        # All prediction logic encapsulated here
        pass

# Usage: Clean separation like MVC!
data_processor = DataProcessor(config)
X_train, y_train = data_processor.load_and_preprocess('data.csv')
```

```
trainer = ModelTrainer('random_forest', params)
trainer.train(X_train, y_train)

predictor = ModelPredictor('model.pkl')
predictions = predictor.predict(X_new)
```

Just like MVC makes web applications maintainable, OOP makes ML projects professional and scalable!

Example Classes to Consider

1. DataLoader Class:

```
class DataLoader:
    """Class for loading and basic validation of datasets."""

    def __init__(self, data_path: str):
        """Initialize with path to data directory."""
        self.data_path = data_path

    def load_csv(self, filename: str) -> pd.DataFrame:
        """Load CSV file from data directory."""
        filepath = os.path.join(self.data_path, filename)
        return pd.read_csv(filepath)

    def validate_columns(self, df: pd.DataFrame, required_cols: list) ->
bool:
        """Validate that DataFrame has required columns."""
        missing = set(required_cols) - set(df.columns)
        if missing:
            raise ValueError(f"Missing required columns: {missing}")
        return True
```

2. FeatureEngineer Class:

```
class FeatureEngineer:
    """Class for feature engineering operations."""

    def __init__(self, config: dict):
        """Initialize with configuration."""
        self.config = config

    def create_features(self, df: pd.DataFrame) -> pd.DataFrame:
        """Create new features based on configuration."""
        # Implementation here
        pass

    def select_features(self, df: pd.DataFrame, method: str =
'importance') -> pd.DataFrame:
```

```
"""Select features using specified method."""  
# Implementation here  
pass
```



3. Trainer Class:




```
class Trainer:  
    """Class for training machine learning models."""  
  
    def __init__(self, model_type: str, params: dict):  
        """Initialize trainer with model type and parameters."""  
        self.model_type = model_type  
        self.params = params  
        self.model = None  
  
    def train(self, X_train, y_train):  
        """Train the model."""  
        # Implementation here  
        pass  
  
    def save(self, filepath: str):  
        """Save trained model."""  
        # Implementation here  
        pass
```

4. Evaluator Class:

```
class Evaluator:  
    """Class for evaluating model performance."""  
  
    def __init__(self, model):  
        """Initialize with trained model."""  
        self.model = model  
  
    def evaluate(self, X_test, y_test) -> dict:  
        """Evaluate model and return metrics."""  
        # Implementation here  
        pass  
  
    def generate_report(self) -> str:  
        """Generate formatted evaluation report."""  
        # Implementation here  
        pass
```

Benefits of OOP:

-  Encapsulation: Related functionality grouped together
-  Reusability: Classes can be reused across projects

-  Maintainability: Easier to modify and extend
-  Testing: Easier to write unit tests for classes
-  State management: Objects maintain state across operations

Part 6: Configuration Files (Optional but Recommended)

Using YAML for Configuration

Instead of hard-coding parameters, use YAML configuration files:

configs/train_config.yaml:

```
model:
  type: random_forest
  params:
    n_estimators: 100
    max_depth: 10
    random_state: 42

training:
  test_size: 0.2
  cv_folds: 5

paths:
  data: data/processed/train_data.csv
  output: models/
```

configs/preprocess_config.yaml:

```
preprocessing:
  missing_values:
    strategy: mean # Options: mean, median, drop
    threshold: 0.5

  scaling:
    method: standard # Options: standard, minmax, robust

  encoding:
    categorical: label # Options: label, onehot

features:
  drop_columns:
    - id
    - timestamp

  categorical:
    - gender
    - category
```

```
numerical:
  - age
  - income
```

Loading config in Python:

```
import yaml

def load_config(config_path: str) -> dict:
    """Load YAML configuration file."""
    with open(config_path, 'r') as f:
        config = yaml.safe_load(f)
    return config

# Usage
config = load_config('configs/train_config.yaml')
n_estimators = config['model']['params']['n_estimators']
```

Part 7: Notebook Organization

Best Practices for Notebooks

Place all Jupyter notebooks in the **notebooks/** folder. You keep your original notebook(s) where you developed your proof of concept here. On top of that, we may create additional notebooks. Instead of one monolithic notebook, create focused notebooks:



Suggested notebook structure:




```
notebooks/
├── proof_of_concept.ipynb
├── 01_exploratory_data_analysis.ipynb
├── 02_data_cleaning.ipynb
├── 03_feature_engineering.ipynb
├── 04_model_prototyping.ipynb
├── 05_model_comparison.ipynb
└── 06_final_visualizations.ipynb
```

Naming conventions:

- Use numeric prefixes for ordering (01_, 02_, etc.)
- Use descriptive names that indicate the purpose
- Use underscores, not spaces or hyphens

Each notebook should:

-  Have a clear purpose stated at the top
-  Include markdown cells explaining each step

-  Show outputs (don't clear cells before committing)
-  Be self-contained (imports at top)
-  Run from top to bottom without errors

Notebook vs. Module decision:

- **Notebooks:** Exploration, visualization, one-off analysis, documentation
 - **Modules:** Reusable code, production pipelines, automated tasks
-

Part 8: Best Practices

1. Function Design Principles

Single Responsibility Principle (SRP)

- Each function should do ONE thing well
- If a function does multiple things, split it up

Bad:

```
def preprocess_and_train(filepath):  
    df = pd.read_csv(filepath)  
    df = df.dropna()  
    X = df.drop('target', axis=1)  
    y = df['target']  
    model = RandomForestClassifier()  
    model.fit(X, y)  
    return model
```

Good:

```
def load_data(filepath):  
    return pd.read_csv(filepath)  
  
def handle_missing_values(df):  
    return df.dropna()  
  
def split_features_target(df):  
    X = df.drop('target', axis=1)  
    y = df['target']  
    return X, y  
  
def train_random_forest(X, y):  
    model = RandomForestClassifier()  
    model.fit(X, y)  
    return model
```

2. Type Hints

Always include type hints for function parameters and return values:

```
from typing import Tuple, Dict, Any
import pandas as pd
import numpy as np

def split_train_test(
    X: pd.DataFrame,
    y: pd.Series,
    test_size: float = 0.2
) -> Tuple[pd.DataFrame, pd.DataFrame, pd.Series, pd.Series]:
    """Split data into train and test sets."""
    # implementation
```

Benefits:

- Code is self-documenting
- IDEs provide better autocomplete
- Catches type errors early
- Makes code review easier

3. Docstrings

Every function needs a docstring explaining:

- What it does
- Parameters (type and description)
- Return value (type and description)
- Exceptions raised (if any)

Use Google-style docstrings:

```
def calculate_metrics(y_true: np.ndarray, y_pred: np.ndarray) -> Dict[str, float]:
    """
    Calculate multiple classification metrics.

    Args:
        y_true: True labels as numpy array
        y_pred: Predicted labels as numpy array

    Returns:
        Dictionary containing accuracy, precision, recall, and f1-score

    Raises:
        ValueError: If arrays have different lengths

    Example:
```



```

>>> y_true = np.array([0, 1, 1, 0])
>>> y_pred = np.array([0, 1, 0, 0])
>>> metrics = calculate_metrics(y_true, y_pred)
>>> print(metrics['accuracy'])
0.75
"""
if len(y_true) != len(y_pred):
    raise ValueError("Arrays must have same length")

return {
    'accuracy': accuracy_score(y_true, y_pred),
    'precision': precision_score(y_true, y_pred),
    'recall': recall_score(y_true, y_pred),
    'f1_score': f1_score(y_true, y_pred)
}

```

4. Logging vs Print Statements

Never use `print()` in production code! Use the `logging` module instead.

Setup at the top of each module:

```

import logging

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

```

Usage:

```

# Instead of:
print("Loading data...")
print(f"Data shape: {df.shape}")

# Do this:
logger.info("Loading data...")
logger.info(f"Data shape: {df.shape}")

# For errors:
logger.error(f"Failed to load file: {e}")

# For warnings:
logger.warning("Missing values detected")

# For debugging (won't show in production):
logger.debug(f"Current value: {x}")

```

Why logging is better:

- Can control verbosity (DEBUG, INFO, WARNING, ERROR)
- Can log to files for later analysis
- Includes timestamps automatically
- Can route to different destinations
- Production systems can aggregate logs

5. Error Handling

Always handle potential errors gracefully:

```
def load_data(filepath: str) -> pd.DataFrame:
    """Load data from CSV file."""
    try:
        df = pd.read_csv(filepath)
        logger.info(f"Successfully loaded {len(df)} rows")
        return df
    except FileNotFoundError:
        logger.error(f"File not found: {filepath}")
        raise
    except pd.errors.EmptyDataError:
        logger.error(f"File is empty: {filepath}")
        raise
    except Exception as e:
        logger.error(f"Unexpected error loading data: {e}")
        raise
```

6. Configuration Over Hard-Coding

 **Bad:**

```
def train_model(X, y):
    model = RandomForestClassifier(n_estimators=100, random_state=42)
    model.fit(X, y)
    return model
```

 **Good:**

```
# In config.py
RANDOM_STATE = 42
N_ESTIMATORS = 100

# In train.py
def train_model(X, y, n_estimators=N_ESTIMATORS):
    model = RandomForestClassifier(
        n_estimators=n_estimators,
        random_state=RANDOM_STATE
    )
```

```
model.fit(X, y)
return model
```

7. DRY Principle (Don't Repeat Yourself)

If you're writing the same code twice, create a function!

 **Bad:**

```
# In multiple places:
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

 **Good:**

```
def scale_features(X_train, X_test):
    """Scale features using StandardScaler."""
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)
    return X_train_scaled, X_test_scaled, scaler
```

Part 9: Documentation

README.md Structure

Your main README.md should include:

```
# Project Title

Brief description of what the project does.

## Project Overview

More detailed explanation of the problem and solution.

## Project Structure

...

directory tree here

...
```

Installation

Prerequisites

- Python 3.8+
- pip or conda

Setup

```
# Clone repository
git clone <your-repo-url>
cd your-project

# Create virtual environment
python -m venv venv
source venv/bin/activate # On Windows: venv\Scripts\activate

# Install dependencies
pip install -r requirements.txt
```

Usage

Preprocessing

```
from src.preprocess import preprocess_pipeline

X_train, X_test, y_train, y_test, scaler =
preprocess_pipeline('data/raw/data.csv')
```

Training

```
from src.train import train_random_forest, save_model

model = train_random_forest(X_train, y_train)
save_model(model, "random_forest")
```

Prediction

```
from src.predict import ModelPredictor

predictor = ModelPredictor('models/random_forest_20241025.pkl')
predictions = predictor.predict(X_new)
```

Model Performance

Model	Accuracy	Precision	Recall	F1-Score
Random Forest	0.85	0.82	0.88	0.85
Logistic Regression	0.78	0.75	0.80	0.77

Key Insights

- 1. Feature X is the most important predictor
- 2. Model performs best on class Y
- 3. Consider collecting more data for class Z

Future Improvements

- ☐ Hyperparameter tuning with GridSearchCV
- ☐ Add cross-validation
- ☐ Deploy as REST API
- ☐ Add model monitoring

Contributing

Instructions for contributors (if applicable)

Docstring Coverage

Ensure every module, class, and function has docstrings:

```
"""
Module docstring: Brief description of what this module does.

This module handles data preprocessing for the ML pipeline,
including data loading, cleaning, and feature engineering.
"""

class DataProcessor:
    """
    Class for processing raw data into ML-ready format.

    This class handles all data preprocessing steps including
    missing value imputation, encoding, and scaling.

    Attributes:
        config: Configuration dictionary
        scaler: Fitted StandardScaler object
    """

    def __init__(self, config: dict):
        """
        Initialize DataProcessor with configuration.
        """
```

```

    Args:
        config: Dictionary containing preprocessing parameters
    """
    pass

```

Part 10: Makefile for Automation

Create a **Makefile** to automate common tasks:

```

.PHONY: setup clean train test lint format help

# Default target
help:
@echo "Available targets:"
@echo "  setup    - Create virtual environment and install dependencies"
@echo "  clean    - Remove build artifacts and cache files"
@echo "  train    - Train all models"
@echo "  test     - Run tests (Lab 2)"
@echo "  lint     - Check code style"
@echo "  format   - Format code with black"

# Setup virtual environment and install dependencies
setup:
python -m venv venv
./venv/bin/pip install -r requirements.txt
@echo "Setup complete! Activate environment with: source venv/bin/activate"

# Clean build artifacts
clean:
find . -type f -name "*.pyc" -delete
find . -type d -name "__pycache__" -delete
find . -type d -name "*.egg-info" -exec rm -rf {} +
rm -rf .pytest_cache
rm -rf dist build

# Train models
train:
python src/train.py

# Run tests (will add in Lab 2)
test:
pytest tests/ -v

# Lint code
lint:
flake8 src/ --max-line-length=100
pylint src/

```

```
# Format code
format:
  black src/ --line-length=100
  isort src/

# Install as package
install:
  pip install -e .
```

Usage:

```
make setup      # Set up environment
make train      # Train models
make clean      # Clean up
make help       # See all options
```

Part 11: Requirements.txt

Document all your dependencies with specific versions:

```
# Core dependencies
numpy==1.24.3
pandas==2.0.3
scikit-learn==1.3.0

# Visualization (if used in notebooks)
matplotlib==3.7.2
seaborn==0.12.2

# Model saving
joblib==1.3.2

# Configuration
pyyaml==6.0.1

# Logging
python-dotenv==1.0.0

# Development dependencies
jupyter==1.0.0
ipykernel==6.25.0

# Testing (Lab 2)
pytest==7.4.0
pytest-cov==4.1.0

# Code quality (Lab 2)
black==23.7.0
```

```
flake8==6.1.0  
pylint==2.17.5
```

Generate requirements.txt:

```
pip freeze > requirements.txt
```

Install from requirements.txt:

```
pip install -r requirements.txt
```

Part 12: Git Setup and Version Control

.gitignore for ML Projects

Create a comprehensive **.gitignore**:

```
# Byte-compiled / optimized / DLL files  
__pycache__/  
*.py[cod]  
*$py.class  
  
# Virtual environments  
venv/  
env/  
ENV/  
.venv  
  
# IDEs  
.vscode/  
.idea/  
*.swp  
*.sw0  
  
# OS files  
.DS_Store  
Thumbs.db  
  
# Jupyter Notebook checkpoints  
.ipynb_checkpoints  
*/.ipynb_checkpoints/*  
  
# Python egg files  
*.egg-info/  
dist/  
build/
```



```
# pytest
.pytest_cache/
.coverage
htmlcov/

# Logs
logs/
*.log

# TODO: Will add these in Lab 2 when introducing DVC
# For Lab 1, we'll commit data and models to keep things simple
# data/raw/*.csv
# data/processed/*.csv
# models/*.pkl
# models/*.h5
# models/*.joblib
```

Git Best Practices

Commit Often, Commit Meaningfully

✗ Bad commit messages:

```
git commit -m "fixed stuff"
git commit -m "update"
```

✓ Good commit messages:

```
git commit -m "feat: Add data preprocessing pipeline"
git commit -m "fix: Handle missing values in categorical encoding"
git commit -m "docs: Update README with usage examples"
git commit -m "refactor: Split preprocessing into smaller functions"
```

Commit message format:

```
<type>: <short description>

<optional longer description>
```

Types:

- **feat**: New feature
- **fix**: Bug fix
- **docs**: Documentation changes

- **refactor**: Code refactoring
- **test**: Adding tests
- **chore**: Maintenance tasks

GitHub Classroom Workflow

For this course, you'll use GitHub Classroom:

1. **Accept the assignment** - Click the GitHub Classroom link
2. **Clone your repository**

```
git clone <your-repo-url>
cd <your-repo-name>
```

3. **Create project structure** - Follow this lab's instructions
4. **Commit regularly**:

```
git add .
git commit -m "feat: Add preprocessing module"
git push
```

Typical workflow:

```
# Make changes to your files
git status                    # See what changed
git add src/preprocess.py     # Stage specific files
git commit -m "feat: Add preprocessing" # Commit with message
git push origin main          # Push to GitHub
```

Part 13: Looking Ahead - Lab 2 Preview

What's Coming: DVC and MLflow

In Lab 2, we'll introduce two critical tools for ML projects:

DVC (Data Version Control)

Why DVC?

- Data files are too large for Git
- Need to track data versions alongside code
- Want reproducible pipelines
- Share data without bloating Git repos

What DVC does:

- Tracks large files (data, models) separately from Git
- Creates **.dvc** files (small pointers) that Git tracks
- Stores actual data in remote storage (S3, GCS, etc.)
- Enables data pipelines and reproducibility

Quick preview:

```
# Initialize DVC (Lab 2)
dvc init

# Track data file
dvc add data/raw/dataset.csv
# This creates dataset.csv.dvc (tracked by Git)
# The actual dataset.csv goes in .gitignore

# Add remote storage
dvc remote add -d myremote s3://mybucket/dvcstore

# Push data to remote
dvc push

# Others can pull data
dvc pull
```

Project structure with DVC:

```
your-project/
├── data/
│   ├── raw/
│   │   ├── dataset.csv      # Ignored by Git, tracked by DVC
│   │   └── dataset.csv.dvc  # Tracked by Git
│   └── processed/
│       ├── train.csv        # Ignored by Git
│       └── train.csv.dvc    # Tracked by Git
├── .dvc/                    # DVC configuration
└── .dvcignore               # DVC ignore rules
```

MLflow (Experiment Tracking)

Why MLflow?

- Track all experiments automatically
- Compare model performance across runs
- Manage model lifecycle
- Package models for deployment

What MLflow does:

- Logs parameters, metrics, and artifacts
- Creates UI for comparing experiments
- Manages model registry
- Enables model deployment

Quick preview:

```
import mlflow
import mlflow.sklearn

# Start MLflow run (Lab 2)
with mlflow.start_run():
    # Log parameters
    mlflow.log_param("n_estimators", 100)
    mlflow.log_param("max_depth", 10)

    # Train model
    model = train_model(X_train, y_train)

    # Log metrics
    accuracy = evaluate_model(model, X_test, y_test)
    mlflow.log_metric("accuracy", accuracy)

    # Log model
    mlflow.sklearn.log_model(model, "model")
```

MLflow UI:

```
# View experiments in browser
mlflow ui
```

Project structure with MLflow:

```
your-project/
├── mlruns/                # MLflow experiment tracking
│   ├── 0/                # Experiment ID
│   │   ├── run1/         # Run artifacts
│   │   └── run2/
│   └── models/           # Model registry
├── experiments/          # Your experiment notes
└── mlflow.db             # MLflow backend
```

Testing (Also in Lab 2)

We'll add proper unit tests using pytest:

Example test structure:

```
# tests/test_preprocess.py
import pytest
from src.preprocess import load_data, handle_missing_values

def test_load_data():
    """Test data loading function."""
    df = load_data('tests/fixtures/sample_data.csv')
    assert len(df) > 0
    assert 'target' in df.columns

def test_handle_missing_values():
    """Test missing value handling."""
    df = create_sample_df_with_missing()
    df_clean = handle_missing_values(df)
    assert df_clean.isnull().sum().sum() == 0
```

Running tests:

```
pytest tests/ -v                # Run all tests
pytest tests/test_preprocess.py # Run specific test file
pytest --cov=src tests/         # With coverage report
```

Why This Progression Matters

Lab 1 (Current):

- Focus: Code structure and modularity
- Tools: Git only
- Data: Committed to Git (simple but not scalable)

Lab 2 (Next):

- Focus: Data versioning and experiment tracking
- Tools: Git + DVC + MLflow + pytest
- Data: Tracked by DVC (scalable and professional)

Lab 3+:

- Focus: Deployment, APIs, containers, cloud
- Tools: Everything from Labs 1-2 plus Flask/Docker/AWS

Each lab builds on the previous, teaching you industry-standard ML engineering practices!

Part 14: Your Assignment

Required Deliverables

Submit your GitHub repository URL containing:

1. Complete directory structure:

- ✓ data/raw/, data/processed/, data/external/ (if applicable)
- ✓ src/
- ✓ src/utils/
- ✓ models/
- ✓ notebooks/
- ✓ outputs/
- ✓ configs/ (optional but recommended)
- ✓ docs/

2. All required Python modules:

- ✓ src/__init__.py
- ✓ src/utils/__init__.py
- ✓ src/utils/config.py
- ✓ src/preprocess.py
- ✓ src/train.py
- ✓ src/predict.py (with ModelPredictor class)
- ✓ src/evaluate.py
- ✓ src/feature_engineering.py (if applicable)

3. Documentation:

- ✓ README.md (comprehensive project documentation)
- ✓ Docstrings in all functions and classes
- ✓ .gitignore (properly configured)

4. Automation files:

- ✓ requirements.txt (all dependencies with versions)
- ✓ Makefile (at minimum: setup target)

5. Working example:

- ✓ stage_01_example_usage.py (demonstrates your workflow; you need to have a python environment with the required packages installed.)

6. Organized notebooks:

- ✓ All notebooks in notebooks/ folder
- ✓ Clear naming convention (numbered and descriptive)
- ✓ Each notebook serves a specific purpose

7. Object-Oriented Design:

- ✓ At least one substantial class (DataLoader, FeatureEngineer, Trainer, etc.)
- ✓ ModelPredictor class for independent predictions
- ✓ Proper use of encapsulation and methods

Quality Checklist

Before submitting, verify:

Code Quality:

- ☐ All functions have type hints
- ☐ All functions have Google-style docstrings
- ☐ Using logging instead of print statements
- ☐ Proper error handling with try-except
- ☐ No hard-coded values (use config.py or config files)
- ☐ Following DRY principle (no code duplication)
- ☐ Each function has a single responsibility
- ☐ At least one well-designed class demonstrating OOP

Functionality:

- ☐ Can load and preprocess data end-to-end
- ☐ Can train all your models
- ☐ Can evaluate models and compare them
- ☐ Can make predictions on new data
- ☐ Can save and load models
- ☐ example_usage.py runs without errors
- ☐ ModelPredictor class works independently

Documentation:

- ☐ README.md is complete and well-formatted
- ☐ Project structure is clearly explained
- ☐ Installation instructions are provided
- ☐ Usage examples are provided
- ☐ Model performance is documented
- ☐ All modules have module-level docstrings

Organization:

- ☐ Notebooks are in notebooks/ folder
- ☐ Notebooks have descriptive names
- ☐ Each notebook focuses on specific task
- ☐ Original notebook preserved for reference
- ☐ Data files organized in appropriate folders
- ☐ Models saved in models/ folder

Configuration:

- ☐ Config values centralized (config.py or YAML)
- ☐ No hard-coded paths or parameters
- ☐ Environment-specific settings separated

Git:

- ☐ All code is committed
- ☐ Commit messages are meaningful
- ☐ No unnecessary files committed
- ☐ .gitignore properly configured
- ☐ Latest changes pushed to GitHub

Optional Enhancements (Bonus Points)

Consider adding these for extra credit:

- ☐ YAML configuration files with examples
- ☐ Command-line argument parsing (argparse)
- ☐ Multiple classes demonstrating OOP principles
- ☐ Comprehensive example_usage.py with multiple scenarios
- ☐ Additional helper utilities beyond basic requirements
- ☐ Well-organized notebook series (01_, 02_, etc.)
- ☐ Performance comparison visualizations
- ☐ Model interpretability functions
- ☐ Cross-validation implementation
- ☐ Hyperparameter tuning framework

Part 15: Common Pitfalls and Solutions

Pitfall 1: Overly Complex Functions

✗ Problem: Creating one giant function that does everything

Solution: Break it down! If a function is more than 30-40 lines, consider splitting it.

Pitfall 2: Not Handling Edge Cases

✗ Problem: Code crashes on empty dataframes, missing columns, etc.

Solution: Add validation and error handling:


```
def preprocess_data(df):  
    if df.empty:  
        raise ValueError("DataFrame is empty")  
  
    required_columns = ['feature1', 'feature2', 'target']  
    missing_cols = set(required_columns) - set(df.columns)  
    if missing_cols:  
        raise ValueError(f"Missing required columns: {missing_cols}")  
  
    # ... rest of preprocessing
```

Pitfall 3: Not Testing with Fresh Data

✗ Problem: Code works on the exact data you developed with, but fails on new data

Solution:

- Test with a small subset of data
- Test with edge cases (empty values, outliers)
- Have a classmate test your code

Pitfall 4: Forgetting About the Scaler/Encoder

✗ Problem: Training a scaler/encoder on training data but not saving it for prediction time

Solution: Always return and save preprocessing objects:

```
def preprocess_pipeline(filepath):  
    # ... preprocessing ...  
    scaler = StandardScaler()  
    X_train_scaled = scaler.fit_transform(X_train)  
    X_test_scaled = scaler.transform(X_test)  
  
    # Save scaler for later use!  
    joblib.dump(scaler, 'models/scaler.pkl')  
    return X_train_scaled, X_test_scaled, y_train, y_test, scaler
```

Pitfall 5: Poor Error Messages

✗ Bad:

```
raise Exception("Error")
```

✓ Good:

```
raise ValueError(
    f"Expected {expected_features} features, got {X.shape[1]}. "
    f"Please ensure your data has columns: {FEATURE_NAMES}"
)
```

Pitfall 6: Not Documenting Assumptions

✗ Problem: Code assumes certain things but doesn't document them

Solution: Document assumptions in docstrings:

```
def encode_categorical(df):
    """
    Encode categorical variables using Label Encoding.

    Assumptions:
    - All categorical columns are strings
    - No new categories at prediction time
    - Target is binary (0/1)

    Args:
        df: Input DataFrame with categorical columns

    Returns:
        DataFrame with encoded categorical columns
    """
```

Pitfall 7: Mixing OOP and Functional Styles Inconsistently

✗ Problem: Some parts use classes, others use functions with no clear pattern

Solution: Be consistent:

- Use classes for stateful operations (preprocessing pipelines, trainers)
- Use functions for stateless operations (metrics calculation, data loading)
- Document your design decision in README

Pitfall 8: Not Using Configuration Files

✗ Problem: Parameters scattered throughout code

Solution: Centralize in config.py or YAML:

```
# config.py
class Config:
    """Central configuration for the project."""
    RANDOM_STATE = 42
    TEST_SIZE = 0.2
```

```
MODEL_PARAMS = {  
    'n_estimators': 100,  
    'max_depth': 10  
}
```

Part 16: Testing Your Conversion

Create an Example Usage Script

Create `example_usage.py` to demonstrate your complete workflow:

```
"""  
Example usage of the ML project modules.  
Demonstrates the complete workflow from data to predictions.  
"""  
  
import logging  
from src.preprocess import preprocess_pipeline  
from src.train import train_all_models, save_model  
from src.evaluate import evaluate_model, compare_models  
from src.predict import ModelPredictor, predict  
  
logging.basicConfig(level=logging.INFO)  
logger = logging.getLogger(__name__)  
  
def main():  
    """Main workflow demonstrating project usage."""  
  
    # Step 1: Preprocess data  
    logger.info("="*60)  
    logger.info("Step 1: Preprocessing data...")  
    logger.info("="*60)  
  
    X_train, X_test, y_train, y_test, scaler = preprocess_pipeline(  
        'data/raw/your_data.csv',  
        scale=True  
    )  
  
    # Step 2: Train models  
    logger.info("\n" + "="*60)  
    logger.info("Step 2: Training models...")  
    logger.info("="*60)  
  
    models = train_all_models(X_train, y_train)  
  
    # Step 3: Compare models  
    logger.info("\n" + "="*60)  
    logger.info("Step 3: Comparing models...")  
    logger.info("="*60)
```

```

comparison_df = compare_models(models, X_test, y_test)
print("\nModel Comparison:")
print(comparison_df)

# Step 4: Evaluate best model
logger.info("\n" + "="*60)
logger.info("Step 4: Evaluating best model...")
logger.info("="*60)

best_model_name = comparison_df.iloc[0]['Model']
best_model = models[best_model_name]
results = evaluate_model(best_model, X_test, y_test)

print(f"\nBest model: {best_model_name}")
print(f"Accuracy: {results['accuracy']:.4f}")
print(f"Precision: {results['precision']:.4f}")
print(f"Recall: {results['recall']:.4f}")
print(f"F1-Score: {results['f1_score']:.4f}")

# Step 5: Save best model
logger.info("\n" + "="*60)
logger.info("Step 5: Saving model...")
logger.info("="*60)

model_path = save_model(best_model, best_model_name)
print(f"Model saved to: {model_path}")

# Step 6: Demonstrate prediction with ModelPredictor
logger.info("\n" + "="*60)
logger.info("Step 6: Making predictions with ModelPredictor...")
logger.info("="*60)

predictor = ModelPredictor(model_path)
predictions = predictor.predict(X_test[:5])
probabilities = predictor.predict_proba(X_test[:5])

print(f"Sample predictions: {predictions}")
if probabilities is not None:
    print(f"Sample probabilities: {probabilities}")

logger.info("\n" + "="*60)
logger.info("✅ Complete workflow executed successfully!")
logger.info("="*60)

if __name__ == "__main__":
    main()

```

Verification Commands

Run these commands to verify your project:

```
# 1. Check directory structure exists
ls -la data/raw data/processed src/utils models notebooks outputs

# 2. Check all Python files exist
ls src/*.py src/utils/*.py

# 3. Verify you can import your modules
python -c "from src import preprocess, train, predict, evaluate; print('✅ All imports successful')"

# 4. Check for syntax errors
python -m py_compile src/*.py src/utils/*.py

# 5. Run your example usage
python example_usage.py

# 6. Check Git status
git status

# 7. Verify requirements.txt
pip install -r requirements.txt --dry-run
```

Part 17: Getting Help

Resources

1. Tutorial Project:

- Repository: <https://github.com/NorQuest-MLAD-Courses/cmpt2500f25-project-tutorial.git>
- Use as reference for structure and implementation

2. Python Documentation:

- Type hints: <https://docs.python.org/3/library/typing.html>
- Logging: <https://docs.python.org/3/library/logging.html>
- OOP: <https://docs.python.org/3/tutorial/classes.html>

3. Scikit-learn Documentation:

- <https://scikit-learn.org/stable/>

4. Git Documentation:

- <https://git-scm.com/doc>

5. YAML:

- <https://pyyaml.org/wiki/PyYAMLDocumentation>

Troubleshooting

Problem: Import errors

```
ModuleNotFoundError: No module named 'src'
```

Solution: Make sure you're running Python from your project root directory, not from inside `src/`

Problem: Circular imports

```
ImportError: cannot import name 'X' from partially initialized module 'Y'
```

Solution: Check if your modules are importing each other. Refactor to avoid circular dependencies.

Problem: Can't find data file

```
FileNotFoundError: data/raw/dataset.csv
```

Solution: Use absolute paths from `config.py` or make sure you're running from project root

Problem: Git won't commit

```
fatal: pathspec 'file.py' did not match any files
```

Solution: Make sure the file exists and you've run `git add file.py`

Problem: Class instantiation errors

```
TypeError: __init__() missing required positional argument
```

Solution: Check your class `__init__` method and ensure you're passing all required parameters

Questions to Ask When Stuck

1. **What is the exact error message?** (Copy the full traceback)
2. **What were you trying to do?** (Describe the goal)
3. **What did you expect to happen?** (Expected behavior)
4. **What actually happened?** (Actual behavior)
5. **What have you tried?** (Show debugging attempts)
6. **Can you reproduce it?** (Consistent or random?)

Office Hours and Support

- Attend lab sessions for hands-on help
 - Use GitHub Issues for questions about the assignment
 - Collaborate with classmates (but write your own code!)
 - Check course discussion board for common questions
 - Reference the tutorial project repository
-

Part 18: Submission

What to Submit

1. **GitHub Repository URL**

- Your GitHub Classroom repository
- Should contain all required files
- Latest changes pushed
- Repository should be public or accessible to instructors

2. **Brief Project Summary** (in README.md)

- What ML problem did you solve?
- What models did you implement?
- What were your key findings?
- Any unique aspects of your implementation?

Grading Criteria

Your lab will be evaluated on:

1. **Project Structure (20%)**

- Proper directory organization
- All required folders present
- Logical file placement
- Clean and organized

2. **Code Quality (30%)**

- Type hints and docstrings
- Proper error handling
- Logging instead of prints
- Following best practices (SRP, DRY, etc.)
- Code is readable and maintainable

3. **Object-Oriented Design (15%)**

- At least one substantial class implemented
- ModelPredictor class for predictions
- Proper use of encapsulation

- Methods are cohesive and well-designed

4. **Modularity and Reusability (15%)**

- Functions are single-purpose
- Code is reusable
- Proper separation of concerns
- Configuration management

5. **Documentation (15%)**

- Comprehensive README
- All functions documented
- Clear usage examples
- Installation instructions
- Module-level docstrings

6. **Functionality (10%)**

- `example_usage.py` runs successfully
- All modules work as intended
- Can load data, train, evaluate, predict
- No critical bugs

7. **Git Usage (5%)**

- Meaningful commit messages
- Proper `.gitignore`
- Clean repository
- Regular commits showing progress

Bonus Points (Up to +10%)

- Comprehensive YAML configuration with examples
- Multiple well-designed classes beyond requirements
- Extensive example usage script with edge cases
- Well-organized notebook series (numbered, focused)
- Command-line argument parsing
- Additional utility functions
- Cross-validation implementation
- Particularly clean and elegant code
- Going above and beyond requirements

Due Date

[To be announced by instructor]

Late Policy: [To be specified by instructor]

Part 19: Final Checklist

Before you submit, go through this complete checklist:

Structure

- ☐ All required directories created and organized
- ☐ .gitkeep files in empty directories
- ☐ Logical organization of all files
- ☐ No unnecessary files or artifacts

Code Files

- ☐ src/**init**.py exists
- ☐ src/utis/**init**.py exists
- ☐ src/utis/config.py implemented
- ☐ src/preprocess.py implemented with all key functions
- ☐ src/train.py implemented with training functions
- ☐ src/predict.py implemented with ModelPredictor class
- ☐ src/evaluate.py implemented with evaluation functions
- ☐ src/feature_engineering.py (if applicable)

Code Quality

- ☐ All functions have type hints
- ☐ All functions have Google-style docstrings
- ☐ Using logging, not print statements
- ☐ Proper error handling throughout
- ☐ No hard-coded values (configuration centralized)
- ☐ DRY principle followed
- ☐ Single responsibility per function
- ☐ Code is readable and well-formatted

Object-Oriented Programming

- ☐ At least one substantial class implemented
- ☐ ModelPredictor class works independently
- ☐ Classes have proper **init** methods
- ☐ Methods are well-designed and cohesive
- ☐ Proper use of encapsulation

Functionality

- ☐ Can load and preprocess data end-to-end
- ☐ Can train all models
- ☐ Can evaluate and compare models
- ☐ Can make predictions on new data
- ☐ Can save and load models
- ☐ ModelPredictor works standalone
- ☐ example_usage.py runs without errors

Documentation

- ☐ README.md is comprehensive
- ☐ Installation instructions clear
- ☐ Usage examples provided
- ☐ Project structure explained
- ☐ Model performance documented
- ☐ All modules have docstrings
- ☐ Key design decisions documented

Notebooks

- ☐ All notebooks in notebooks/ folder
- ☐ Descriptive names (preferably numbered)
- ☐ Each notebook has clear purpose
- ☐ Original notebook preserved
- ☐ Notebooks run without errors

Configuration

- ☐ requirements.txt complete with versions
- ☐ Makefile present with at least setup target
- ☐ Config values centralized
- ☐ Optional: YAML configs provided

Git

- ☐ All required files committed
- ☐ Meaningful commit messages
- ☐ .gitignore properly configured
- ☐ No unnecessary files committed
- ☐ Latest changes pushed to GitHub
- ☐ Repository accessible

Testing

- ☐ Ran example_usage.py successfully
- ☐ Tested import statements
- ☐ Checked for syntax errors
- ☐ Verified file structure
- ☐ Tested on fresh clone (if possible)

Congratulations! 🎉

You've completed the foundational step in your ML deployment journey! You've transformed exploratory notebook code into production-ready Python modules with:

- ✅ **Proper project structure** for scalability
- ✅ **Modular, reusable code** following best practices

- ✓ **Object-oriented design** for maintainability
- ✓ **Comprehensive documentation** for collaboration
- ✓ **Version control proficiency** with Git
- ✓ **Foundation for advanced tools** (DVC, MLflow coming in Lab 2)

Key Takeaways:

1. ✓ Notebooks are for exploration, modules are for production
2. ✓ One function = one responsibility
3. ✓ Type hints and docstrings are essential
4. ✓ Configuration files beat hard-coded values
5. ✓ Logging beats print statements
6. ✓ OOP provides structure and reusability
7. ✓ Good structure today = easy deployment tomorrow

You're now ready to add:

- **Lab 2:** DVC for data versioning, MLflow for experiment tracking, and proper testing
- **Lab 3:** REST APIs with Flask/FastAPI
- **Lab 4:** Docker containerization
- **Lab 5:** Cloud deployment
- **Lab 6:** Monitoring and CI/CD

Each lab builds on this foundation. The structure you've created will support all future enhancements!

Questions? Issues? Stuck?

- Check the tutorial project for examples
 - Review this document
 - Ask in lab sessions
 - Use GitHub Issues for technical questions
 - Collaborate with classmates (but write your own code!)
-

Happy Coding! 🚀

Lab prepared for CMPT 2500 - ML/AI Deployment
NorQuest College

Appendix A: Quick Reference

Essential Commands

```
# Git
git status
git add .
git commit -m "message"
git push
```

```
# Virtual environment
python -m venv venv
source venv/bin/activate # Mac/Linux
venv\Scripts\activate    # Windows

# Makefile
make setup
make train
make clean

# Python
python file.py
python -m pytest tests/
python -c "from src import preprocess"
```

Project Templates

Minimal config.py:

```
import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.dirname(__file__)))
DATA_RAW_PATH = os.path.join(BASE_DIR, "data", "raw")
MODELS_PATH = os.path.join(BASE_DIR, "models")

RANDOM_STATE = 42
TEST_SIZE = 0.2
TARGET = 'target_column'
```

Minimal train.py:

```
from sklearn.ensemble import RandomForestClassifier

def train_model(X, y):
    model = RandomForestClassifier(random_state=42)
    model.fit(X, y)
    return model
```

Minimal predict.py:

```
import joblib

class ModelPredictor:
    def __init__(self, model_path):
        self.model = joblib.load(model_path)
```

```
def predict(self, X):  
    return self.model.predict(X)
```

Common Patterns

Loading config:

```
from src.utils.config import DATA_RAW_PATH, RANDOM_STATE
```

Error handling:

```
try:  
    result = risky_operation()  
except SpecificError as e:  
    logger.error(f"Error: {e}")  
    raise
```

Type hints:

```
def function(x: int, y: str) -> float:  
    return float(x)
```

Appendix B: Example ModelPredictor Variations

For Classification with Preprocessing

```
class ModelPredictor:  
    def __init__(self, model_path, scaler_path):  
        self.model = joblib.load(model_path)  
        self.scaler = joblib.load(scaler_path)  
  
    def predict(self, X):  
        X_scaled = self.scaler.transform(X)  
        return self.model.predict(X_scaled)
```

For Regression

```
class RegressionPredictor:  
    def __init__(self, model_path):  
        self.model = joblib.load(model_path)
```

```
def predict(self, X):
    predictions = self.model.predict(X)
    return predictions

def predict_with_intervals(self, X, confidence=0.95):
    # Add prediction intervals if model supports
    pass
```

For Multi-Model Ensemble

```
class EnsemblePredictor:
    def __init__(self, model_paths):
        self.models = [joblib.load(p) for p in model_paths]

    def predict(self, X):
        predictions = [m.predict(X) for m in self.models]
        return np.mean(predictions, axis=0)
```
