Building a Dummy ML Data Pipeline

Lab Outline

Objective:

To create a simple, modular data pipeline using best practices in Python packaging, configuration, documentation, and testing, while applying object-oriented programming (OOP) principles and design patterns.

Lab Structure:

Section 1: Project Setup and Python Packaging

1. **Create a New Project with Poetry:**
    - Initialise a new project using Poetry
    - Configure dependencies.
2. **Set Up Python Package Structure:**

Create a `src/` directory with the following subdirectories and modules:

```
src/
├── data_pipeline/
│   ├── __init__.py
│   ├── main.py           # entry point for the pipeline
│   └── ...               # rest of the package
└── tests/
    └── ...               # tests for each module
```

Section 2: Implementing Configuration Management

1. **Set Up YAML Configuration:**
    - Define a `config.yaml` with basic pipeline parameters (e.g., paths, data source, transformation steps).
    - Include sections like `data` (for data loading config) and `model` (for model-specific configurations).
2. **Load Configuration with OmegaConf and Pydantic:**
    - Use OmegaConf to read `config.yaml`.
    - Create `config.py` to validate and structure configuration data using Pydantic.

Section 3: Developing the Pipeline Modules (OOP and Design Patterns)

1. **Implement a Dummy Data Loader (Factory Pattern):**
    - Define a `DataLoader` base class with methods like `load_data`.
    - Use the Factory pattern to create different types of data loaders (e.g., `CSVLoader`, `JSONLoader`).
2. **Build a Data Transformation Module (Strategy Pattern):**
    - Define a `DataTransformer` base class with a transform method.
    - Implement strategies (subclasses) like `NormalizeTransformer` and `StandardizeTransformer` to perform different transformations.
3. **Create a Model Module (Adapter Pattern):**
    - Implement a simple `Model` interface and create dummy ML models like `LinearModel` and `TreeModel`.
    - Use an Adapter pattern if models require different input formats or structures, allowing them to interact seamlessly with the pipeline.

Section 4: Documentation, Typing, Linting, and Formatting

1. **Document Classes and Methods with Docstrings:**
    - Write comprehensive docstrings for all classes and methods following Google or NumPy style.
    - Generate HTML documentation using a tool like `pdoc` or `mkdocs`.
2. **Add Typing and Type Checking:**
    - Annotate all function parameters and return types with Python type hints.
    - Use `mypy` to check type consistency.
3. **Lint and Format with Ruff:**
    - Set up `ruff` for linting, following conventions (e.g., PEP 8, docstring conventions).
    - Ensure formatting consistency using `black`.

Section 5: Testing and Logging

1. **Write Unit Tests with Pytest:**
    - Write unit tests for each module (`data_loader.py`, `data_transform.py`, `model.py`) in `tests/`.
    - Use pytest fixtures to handle reusable test setups.
2. **Add Logging with Loguru:**
    - Integrate `loguru` in the main pipeline and modules to log information, warnings, and errors.
    - Configure logging to save logs to a file in addition to console output.

Section 6: Task Automation with PyInvoke

1. **Define PyInvoke Tasks:**
   - Set up `tasks.py` in the root project directory to automate tasks like:
     - Running tests: `invoke test`
     - Linting and formatting: `invoke lint`
     - Generating documentation: `invoke docs`
     - Running the pipeline: `invoke run`

Section 7: Final Assembly and Execution

1. **Assemble and Run the Pipeline:**
   - Using `poetry`, install the package locally.
   - Execute the pipeline using the defined entry point (`data-pipeline`) and verify correct data flow from loading to transformation and model application.
2. **Extend for Bonus:**
   - Students can be encouraged to extend this pipeline by adding more model strategies, more complex data transformations, or additional config validations.

Section 1: Project Setup and Python Packaging

Objective:

To initialise the project and set up a basic Python package structure using Poetry, while installing only the foundational data science libraries. Create your first entry point and get familiar with poetry.

Instructions:

Step 1: Initialise the Project with Poetry

1. **Create a New Poetry Project:**
    - Open your terminal and navigate to a working directory for this lab.
    - Use `poetry new ml_data_pipeline --src` to create a new project with a `src/` directory structure.
2. **Explore the Generated Structure:**

Inside the `ml_data_pipeline/` folder, you'll see a structure generated by Poetry:

```
ml_data_pipeline/
├── README.md
├── pyproject.toml
├── src/
│   └── ml_data_pipeline/
│       └── __init__.py
└── tests/
    └── __init__.py
```

- This structure will form the foundation of your data pipeline. Familiarise yourself with the `pyproject.toml` file, where your project dependencies and configuration are managed.

Step 2: Install Basic Data Science Libraries

1. **Install Packages Using Poetry:**
    - Add essential data science packages by running: `poetry add pandas numpy scikit-learn`
    - Poetry will add these packages to your environment, allowing you to use them in your pipeline.

2. **Verify the Virtual Environment:**
   - Activate the Poetry shell to enter your project's virtual environment:
     `poetry shell`
   - Verify your installed packages by opening a Python shell within this environment:

```
import pandas as pd
import numpy as np
import sklearn
```

   - Exit poetry shell by running `exit`
3. **Check your Projects Dependencies**:
   - List installed packages with
     `poetry show`
   - Check the dependencies in a tree structure to see inter-libraries dependencies
     `poetry show –tree`

Step 3: Create Initial Pipeline Files

1. **Add Basic Modules:**
   - Inside the `src/ml_data_pipeline` directory, create two new files: `data_loader.py` and `main.py`.
2. **Define a Simple Data Loader in `data_loader.py`:**
   - Implement a basic function to load a sample dataset using `pandas`:

```
# src/ml_data_pipeline/data_loader.py
import pandas as pd

def load_data():
    # Example: Create a dummy DataFrame
    data = pd.DataFrame({
        "feature1": [1, 2, 3],
        "feature2": [4, 5, 6],
        "target": [0, 1, 0]
    })
    return data
```

3. **Write a Simple Main Script in `main.py`:**
   - Create an initial script to load and print the data:

```python
# src/ml_data_pipeline/main.py
from ml_data_pipeline.data_loader import load_data

def main():
    data = load_data()
    print("Loaded Data:")
    print(data)

if __name__ == "__main__":
    main()
```

Step 4: Run the Pipeline

1. **Run `main.py` as a Script:**
   - Execute `main.py` from the command line:
     `poetry run python src/ml_data_pipeline/main.py`
   - You should see the dummy data printed in the terminal.
   - Try to run the same command but from a poetry shell (hint: you can run python directly)

Step 5: Set Up an Entry Point

1. **Define the Entry Point in `pyproject.toml`:**
   - Open the `pyproject.toml` file and locate the `[tool.poetry.scripts]` section (if it doesn't exist, you can add it).
   - Add a new entry point under `[tool.poetry.scripts]` to make `ml_data_pipeline` executable from the command line:

```toml
[tool.poetry.scripts]
ml-data-pipeline = "ml_data_pipeline.main:main"
```

   - Here, `"ml_data_pipeline.main:main"` specifies that the entry point function (`main`) is located in `src/ml_data_pipeline/main.py`.
2. **Install the Project to Register the Entry Point:**
   - For the new entry point to take effect, reinstall the project:
     `poetry install`

3. **Run the Pipeline Using the Entry Point:**
    - Now, you can run the data pipeline directly from the command line without specifying the Python file:
      `poetry run ml-data-pipeline`
    - You should see the output from the `main` function, which loads and prints the dummy data.
4. **Explanation:**
    - This setup makes it easier to run the pipeline as a standalone application, without needing to navigate to specific directories or specify script paths.

Section 2: Implementing Configuration Management

Objective:

To add configuration management to the pipeline using YAML files for storing configurations, OmegaConf for loading the YAML configurations, and Pydantic for validating configuration data. This will introduce the basics of configuration handling in data pipelines.

Instructions:

Step 1: Add YAML Configuration File

1. **Create a `config.yaml` File:**
   - Inside `src/ml_data_pipeline`, create a file named `config.yaml`.
   - Define basic configuration settings that will be used in the pipeline. Start with settings for data loading and transformation. For example:

```
data_loader:
  file_path: "data/sample_data.csv"
  file_type: "csv"
transformation:
  normalize: true
  scaling_method: "standard"  # Options: "standard", "minmax"
```

2. **Explanation:**
   - The `data_loader` section can include parameters for loading data, such as the file path and type.
   - The `transformation` section specifies settings for data transformations.

Step 2: Install OmegaConf and Pydantic

1. **Add OmegaConf and Pydantic to the Project:**
   - Run the following command to install OmegaConf and Pydantic with poetry
   - These packages will allow you to load and validate configurations within the pipeline.
   - Check if the packages were installed

Step 3: Implement Configuration Management

1. **Create a `config.py` Module:**
    - Inside `src/ml_data_pipeline`, create a new file named `config.py`.
    - This module will contain the code for loading and validating configuration data.
2. **Define Configuration Models with Pydantic:**
    - In `config.py`, use Pydantic to create configuration models that match the structure of `config.yaml`.
    - You should use nested configurations:
        i. A configuration class for DataLoader
        ii. Another for transformation
        iii. A final one that combines first two
    - Add a validation step for file_type where only “csv” is allowed
    - Add a validation step for scaling method where the value can be `“std”` or `“minmax”`

```python
# src/ml_data_pipeline/config.py
from pydantic import BaseModel, validator
from omegaconf import OmegaConf
import os

class DataLoaderConfig(BaseModel):
    ...

class TransformationConfig(BaseModel):
    ...

class Config(BaseModel):
    data_loader: DataLoaderConfig
    transformation: TransformationConfig
```

3. **Load and Validate Configurations Using OmegaConf:**
    - Add a function in `config.py` to load the YAML configuration using OmegaConf, and then parse it into Pydantic models:

```python
# src/ml_data_pipeline/config.py
def load_config() -> Config:
    config_path = os.path.join(os.path.dirname(__file__), "config.yaml")
    raw_config = OmegaConf.load(config_path)
    config_dict = OmegaConf.to_container(raw_config, resolve=True)
    return Config(**config_dict)
```

4. **Explanation:**
    - The `load_config` function reads `config.yaml` using OmegaConf, converts it to a dictionary, and passes it to Pydantic's `Config` model for validation.
    - This setup ensures that any misconfigurations are caught before they impact the pipeline.

Step 4: Modify the Pipeline to Use Configuration Settings

1. **Update `main.py` to Load Configurations:**
    - Modify `main.py` to use the `load_config` function to read the configuration and apply it in the pipeline:

```
# src/ml_data_pipeline/main.py
from ml_data_pipeline.data_loader import load_data
from ml_data_pipeline.config import load_config

def main():
    # Load configuration
    ...

    # Use configuration in the pipeline
    ...
```

2. **Adjust `data_loader.py` to Use Configuration:**
    - Update `load_data` to accept the file path as a parameter:

```
# src/ml_data_pipeline/data_loader.py
import pandas as pd

def load_data(file_path: str):
    ...
```

3. **Explanation:**
    - Now, `main.py` loads and validates configurations from `config.yaml` before running the pipeline. The configuration settings are then passed to `data_loader.py` as needed.

Step 5: Run and Test the Updated Pipeline

1. **Create a Sample Data File:**
   - To test the configuration, create a `data` directory at the root of the project and save a file named `sample_data.csv` with the following dummy data:

```
feature1,feature2,target
1,4,0
2,5,1
3,6,0
```

2. **Run the Pipeline:**
   - Execute the pipeline from the command line using the entry point: `poetry run ml-data-pipeline`
   - You should see the configuration settings printed, followed by the loaded dummy data.
3. **Verify Configuration Validation:**
   - Test the configuration validation by changing `file_type` in `config.yaml` to an unsupported type (e.g., `xml`) and re-running the pipeline.
   - The pipeline should raise a validation error, demonstrating that Pydantic is catching misconfigurations.

Step 6: Better organisation

Having configuration files next to source code is not always a good idea. This becomes cumbersome when the project grows and we end up having a lot of different configurations that we want to keep.

In this step, we will improve the current structure to accommodate this problem.

1. **Create the `config` Folder:**
   - In the root of the project directory, create a new folder named `config`. This folder will store all YAML configuration files, making it easier to manage different configurations.
2. **Move `config.yaml` to the `config` Folder:**
   - Move the existing `config.yaml` file from `src/ml_data_pipeline` to the new `config` folder.
   - After this change, your project structure should look like this:

```
ml_data_pipeline/
├── config/
│   └── config.yaml
├── src/
│   └── ml_data_pipeline/
│       ├── __init__.py
│       ├── config.py
│       ├── data_loader.py
│       └── main.py
└── tests/
```

Step 7: Add Different Configuration Files

1. **Create Additional Configuration Files:**
    - Inside the `config` folder, add a few more configuration files to simulate different configurations, such as:
        - `config_dev.yaml`: Configuration for development.
        - `config_prod.yaml`: Configuration for production.
2. **Customize the Content of Each Configuration File:**
    - For example, update `config_dev.yaml` to include sample settings:

```
data_loader:
  file_path: "data/sample_data_dev.csv"
  file_type: "csv"
transformation:
  normalize: false
  scaling_method: "minmax"
```

    - Customize `config_prod.yaml` similarly with production-specific settings:

```
data_loader:
  file_path: "data/sample_data_prod.csv"
  file_type: "csv"
transformation:
  normalize: true
  scaling_method: "std"
```

3. **Explanation:**
    - These different configuration files allow you to easily switch between development and production settings without modifying the code.

Step 8: Update main.py to Accept a Configuration File Path Argument

1. **Modify `main.py` to Accept an Argument:**
    - Update `main.py` to accept a command-line argument called `config` specifying the path to the configuration file.
    - Use Python's `argparse` library to handle this argument
    - Pass argument to config loader

```python
# src/ml_data_pipeline/main.py
import argparse
from ml_data_pipeline.data_loader import load_data
from ml_data_pipeline.config import load_config

parser = argparse.ArgumentParser(...)
# Add arguments ...

def main(argv: list[str] | None = None):
    args = parser.parse_args(argv)
    ...
```

2. **Update `load_config` to Use the Argument:**
    - In `config.py`, update the `load_config` function to accept a file path parameter instead of using a hardcoded path:

```python
# src/ml_data_pipeline/config.py
from pydantic import BaseModel, validator
from omegaconf import OmegaConf

def load_config(config_path: str) -> Config:
    ...
```

3. **Explanation:**
    - Now, `main.py` accepts a `--config` argument to specify the path to the configuration file. This change allows users to run the pipeline with different configurations by passing the appropriate file path.

Step 9: Run the Pipeline with Different Configurations

1. **Run the Pipeline with the Development Configuration:**
   - Use the entry point to run the pipeline with the development configuration file:
     ```
     poetry run ml-data-pipeline --config config/config_dev.yaml
     ```
2. **Run the Pipeline with the Production Configuration:**
   - Run the pipeline with the production configuration file:
     ```
     poetry run ml-data-pipeline --config config/config_prod.yaml
     ```
3. **Observe the Output:**
   - Based on the configuration passed, the pipeline will load different data and apply different transformations, demonstrating how configuration files help control pipeline behaviour.

Section 3: Building a Dummy ML Data Pipeline

Objective:

To implement core modules of the pipeline using OOP principles, specifically abstract base classes (ABC), and organise them with a modular structure that leverages design patterns (Factory and Strategy). This organisation will improve modularity, readability, and reusability of the pipeline components.

Instructions:

Step 1: Organise the Data Loader Module with Abstract Base Classes

1. **Create the `data_loader` Package Structure:**
    - Inside `src/ml_data_pipeline`, create a new folder named `data_loader`.
    - Inside `data_loader`, create the following structure:

```
data_loader/
├── __init__.py
├── base_loader.py
├── csv_loader.py
├── json_loader.py
└── factory.py
```

2. **Define the Base DataLoader Class with ABC:**
    - In `base_loader.py`, create an abstract base class `DataLoader` with an abstract method `load_data`.

```python
# src/ml_data_pipeline/data_loader/base_loader.py
from abc import ABC, abstractmethod
import pandas as pd

class DataLoader(ABC):
    @abstractmethod
    def load_data(self, file_path: str) -> pd.DataFrame:
        pass
```

3. **Implement Specific Loaders (`CSVLoader` and `JSONLoader`):**
   - Create `csv_loader.py` for the `CSVLoader` class:

```python
# src/ml_data_pipeline/data_loader/csv_loader.py
import pandas as pd
from .base_loader import DataLoader

class CSVLoader(DataLoader):
    def load_data(self, file_path: str) -> pd.DataFrame:
        ...
```

   - Create `json_loader.py` for the `JSONLoader` class:

```python
# src/ml_data_pipeline/data_loader/json_loader.py
import pandas as pd
import json
from .base_loader import DataLoader

class JSONLoader(DataLoader):
    def load_data(self, file_path: str) -> pd.DataFrame:
        ...
```

4. **Create a Factory for Data Loaders in `factory.py`:**
   - Define a `DataLoaderFactory` class to select the appropriate loader based on the file type with a static `get_data_loader` method.

```python
# src/ml_data_pipeline/data_loader/factory.py
from .csv_loader import CSVLoader
from .json_loader import JSONLoader
from .base_loader import DataLoader

class DataLoaderFactory:
    @staticmethod
    def get_data_loader(file_type: str) -> DataLoader:
        ...
```

5. **Update `__init__.py` for Direct Import Access:**
   - In `data_loader/__init__.py`, import `CSVLoader`, `JSONLoader`, and `DataLoaderFactory` so they can be accessed directly from the `data_loader` package.
   - Hint: check what `__all__` does inside `__init__.py`

6. **Explanation:**
    - This setup allows you to import `CSVLoader`, `JSONLoader`, and `DataLoaderFactory` directly from the `data_loader` package:

```
# src/ml_data_pipeline/main.py
from ml_data_pipeline.data_loader import CSVLoader, JSONLoader
```

Step 2: Organise the Data Transformer Module with ABC

1. **Create the `data_transform` Package Structure:**
    - Inside `src/ml_data_pipeline`, create a new folder named `data_transform`.
    - Inside `data_transform`, create the following structure:

```
data_transform/
├── __init__.py
├── base_transformer.py
├── standard_scaler_transformer.py
├── minmax_scaler_transformer.py
└── factory.py
```

2. **Define the Base DataTransformer Class with ABC:**
    - In `base_transformer.py`, define an abstract base class `DataTransformer` with an abstract `transform` method.

```
# src/ml_data_pipeline/data_transform/base_transformer.py
from abc import ABC, abstractmethod
import pandas as pd

class DataTransformer(ABC):
    @abstractmethod
    def transform(self, data: pd.DataFrame) -> pd.DataFrame:
        pass
```

3. **Implement Specific Transformers (`StandardScalerTransformer` and `MinMaxScalerTransformer`):**
    - Create `standard_scaler_transformer.py` for the `StandardScalerTransformer` class:

```
# src/ml_data_pipeline/data_transform/standard_scaler_transformer.py
from .base_transformer import DataTransformer

class StandardScalerTransformer(DataTransformer):
    def transform(self, data: pd.DataFrame) -> pd.DataFrame:
        ...
```

    - Create `minmax_scaler_transformer.py` for the `MinMaxScalerTransformer` class:

```
# src/ml_data_pipeline/data_transform/minmax_scaler_transformer.py
from .base_transformer import DataTransformer

class MinMaxScalerTransformer(DataTransformer):
    def transform(self, data: pd.DataFrame) -> pd.DataFrame:
        ...
```

4. **Create a Factory for Transformers in `factory.py`:**
    - Implement a `TransformerFactory` to select the appropriate transformer based on configuration.

```
# src/ml_data_pipeline/data_transform/factory.py
from .standard_scaler_transformer import StandardScalerTransformer
from .minmax_scaler_transformer import MinMaxScalerTransformer
from .base_transformer import DataTransformer

class TransformerFactory:
    ...
    def get_transformer(scaling_method: str) -> DataTransformer:
        ...
```

5. **Update `__init__.py` for Direct Import Access:**
    - In `data_transform/__init__.py`, import `StandardScalerTransformer`, `MinMaxScalerTransformer`, and `TransformerFactory` for direct access.

6. **Explanation:**
   - This setup allows direct imports for transformers and the factory:

```
from ml_data_pipeline.data_transform import StandardScalerTransformer
```

Step 3: Update main.py to Use Organised Modules

1. **Modify `main.py` to Use the Data Loader and Transformer Factories:**
   - Now, you can import directly from `data_loader` and `data_transform`
   - Update main method to load data using DataLoaderFactory with its config
   - Update main method to apply data transformation using TransformerFactory with config

```
# src/ml_data_pipeline/main.py
import argparse
from ml_data_pipeline.config import load_config
from ml_data_pipeline.data_loader import DataLoaderFactory
from ml_data_pipeline.data_transform import TransformerFactory

def main(config_path):
    # Load config and print it
    ...

    # Use DataLoaderFactory to load data
    ...

    # Use TransformerFactory to transform data
    ...

if __name__ == "__main__":
    # Create parser
    # Parse args
    # Call main function
```

Step 4: Adding the Models package with ABC

Can you create the same structure that we did in step 2 and step 3 to create a package that contains your different ML Models ?

Section 4: Documentation, Typing, Linting

Objective:

To document code effectively, add type hints for improved clarity and error checking, and ensure consistent formatting and linting across the project.

Instructions:

Step 1: Document Classes and Methods with Docstrings

1. **Add Docstrings Using Google Style:**
    - In this section, we will add docstrings to all our classes. Let's start with adding docstrings to a Dummy Linear Model using Google styling convention for a `LinearModel`:

```
# src/ml_data_pipeline/model/linear_model.py
import pandas as pd
from .base_model import Model

class LinearModel(Model):
    """A Linear model with dummy parameters."""

    def predict(self, data: pd.DataFrame) -> pd.DataFrame:
        """Predict using the linear model.

        Args:
            data (pd.DataFrame): A pandas DataFrame containing the data for
prediction.

        Returns:
            pd.DataFrame: A DataFrame with model predictions.
        """
        print("Predicting with Linear Model")
        ...
```

    - This provides clear guidance on the purpose of each function, its parameters, and its return types.

2. **Generate HTML Documentation:**
   - Choose a documentation tool like `pdoc` or `mkdocs` to generate HTML documentation.
   - For this project, let's use `pdoc`. Install it under the dev group with the following command:
     `poetry add --group dev pdoc`
   - Check what changed in your project.toml file. Where did poetry add the dependency ?
   - You can check how poetry can organise dependencies into groups here.
   - Once installed, generate the documentation by running:
     `poetry run pdoc src/ml_data_pipeline --output-dir docs --html`
   - This will create an HTML documentation site in the `docs` folder.
   - Open the generated HTML files in a browser to review the documentation.
3. **Explanation:**
   - Documenting with docstrings improves code readability for both current and future developers, making the codebase easier to maintain.
4. **Add more docs:**
   - For each class and method in `data_loader`, `data_transform`, and `model` packages, add comprehensive docstrings following either Google.
   - Regenerate the docs and check the result.

Step 2: Add Typing and Type Checking

1. **Annotate Function Parameters and Return Types with Type Hints:**
   - Go through each function and method in the project and add type hints for parameters and return types.
   - For instance, in the `CSVLoader` class:

```python
# src/ml_data_pipeline/data_loader/csv_loader.py
import pandas as pd
from .base_loader import DataLoader

class CSVLoader(DataLoader):
    def load_data(self, file_path: str) -> pd.DataFrame:
        ...
```

   - Type annotations improve code clarity, making it clear what types are expected and returned by each function.

2. **Check Type Consistency with `mypy`:**
   - Install `mypy` for static type checking under dev group.
   - Run `mypy` to check for any type inconsistencies:
     `poetry run mypy src/ml_data_pipeline`
   - If `mypy` flags any issues, update the code accordingly. For example, ensure that the types in function signatures match the actual return types in the methods.
3. **Explanation:**
   - Adding types makes the code safer by catching potential type-related errors at an early stage, improving both reliability and readability.

Step 3: Lint and Format with Ruff

1. **Set Up `ruff` for Linting:**
   - Install `ruff` as a development dependency (under dev group).
   - `ruff` can be configured to match your lint/format liking: ruff configuration doc
   - Configure `ruff` by updating `pyproject.toml` configuration file in the project root:

```
[tool.ruff]
line-length = 88
indent-width = 4

[tool.ruff.lint]
select = ["E", "W", "F"] # Specify the types of rules to enforce
ignore = ["E501"]  # Ignore line length warnings

[tool.ruff.format]
quote-style = "double" # Like Black, use double quotes for strings.
indent-style = "space" # Like Black, indent with spaces, rather than tabs.
```

2. **Run `ruff` to Lint the Codebase:**
   - Run `ruff` to check for linting issues across the `src/ml_data_pipeline` folder:
     `poetry run ruff check src/ml_data_pipeline`
   - `ruff` will highlight any PEP 8 violations or other stylistic issues, making the code consistent and clean.
   - You can fix the changes manually if you like, or use `ruff` to automatically fix them:
     `poetry run ruff format src/ml_data_pipeline`

3. **Explanation:**
    - Linting ensures that the code follows a consistent style, which improves readability and maintainability. Ruff is fast and lightweight, making it ideal for enforcing Python's coding conventions.

Section 5: Testing and Logging

Objective:

To write unit tests for each module using `pytest` and implement logging with `loguru` to monitor the pipeline's execution.

Instructions:

Step 1: Write Unit Tests with Pytest

1. **Install `pytest`:**
   - Install `pytest` as a development dependency:
     `poetry add --group dev pytest`
2. **Create Test Files:**
   - Inside the `tests` directory, create individual test files for each main module (`data_loader`, `data_transform`, and `model`).
   - Your folder structure should look like this:

```
tests/
├── test_data_loader.py
├── test_data_transform.py
└── test_model.py
```

3. **Write Tests for `data_loader` Module in `test_data_loader.py`:**
   - In order to test csv loader, we need to
     i. Create a dummy csv file # *given*
     ii. Load it with CsvLoader # *when*
     iii. Validate loaded csv # *then*
   - First, read fixtures doc and understand it in order to write your first test.
   - Write a fixture for generating a dummy csv file that uses `tmp_path`
   - Now create a csv loader test that uses this fixture to test the csv loader.
   - What happens to the csv file created by the fixture ?
4. **Write Tests for `data_transform` Module in `test_data_transform.py`:**
   - Test the transformation logic to ensure it scales data correctly.
   - You can simply test the shape of the transformed data frame
   - You can also have a more detailed test that actually checks if the transformation is done correctly

5. **Write Tests for `model` Module in `test_model.py`:**
    - Test that models can train and predict on data.
    - In this test, just assert that the model's method doesn't throw any exception
    - You can also test the shape of the predictions are correct
6. **Run the Tests:**
    - Use `pytest` to execute all tests:
      `poetry run pytest tests/`

Step 2: Add Logging with Loguru

1. **Install `loguru`:**
    - Add `loguru` as a dependency:
      `poetry add loguru`
2. **Set Up Logging in `main.py`:**
    - Integrate `loguru` into `main.py` to log key events, errors, and information.
    - Configure `loguru` to save logs to a file as well as display them in the console.

```python
# src/ml_data_pipeline/main.py
import argparse
from loguru import logger
from ml_data_pipeline.config import load_config
from ml_data_pipeline.data_loader import DataLoaderFactory
from ml_data_pipeline.data_transform import TransformerFactory
from ml_data_pipeline.model import ModelFactory

# Configure loguru to log to a file and console
logger.add("logs/pipeline.log", rotation="500 MB")  # Log rotation at 500 MB

parser = argparse.ArgumentParser(...)
parser.add_argument(...)

def main(argv: list[str] | None = None):
    args = parser.parse_args(argv)
    logger.info("Pipeline execution started.")
    # ... Add Steps and logging
    logger.info("Pipeline execution completed successfully.")
```

3. **Explanation:**
    - `loguru` logs important events such as successful data loading, transformation, and model prediction, as well as any errors that occur.
    - Logs are saved to `logs/pipeline.log`, which can be used for future troubleshooting and auditing.

4. **Configure Logging in Modules:**
   - Add `loguru` logging to individual modules (e.g., `data_loader.py`, `data_transform.py`, `model.py`) to capture specific events or errors within each component.

```
# src/ml_data_pipeline/data_loader/csv_loader.py
import pandas as pd
from loguru import logger
from .base_loader import DataLoader

class CSVLoader(DataLoader):
    def load_data(self, file_path: str) -> pd.DataFrame:
        logger.info(f"Loading data from CSV file at {file_path}")
        try:
            data = pd.read_csv(file_path)
            logger.info(f"Successfully loaded data from {file_path}")
            return data
        except Exception as e:
            logger.error(f"Error loading data from {file_path}: {e}")
            raise
```

5. **Run the Pipeline to Verify Logging:**
   - Run the pipeline with logging enabled: `poetry run ml-data-pipeline --config config/config_dev.yaml`
   - Check the `logs/pipeline.log` file to review the log output.

Section 6 - Task Automation with PyInvoke

Objective:

To create a `tasks.py` file in the root directory with `PyInvoke` tasks that automate key commands for testing, linting, documentation generation, and pipeline execution.

Instructions:

Step 1: Install PyInvoke

1. **Add `invoke` as a Development Dependency:**
   - Install `PyInvoke` by running:
     `poetry add --dev invoke`
2. **Create `tasks.py` in the Project Root:**
   - In the root directory of the project, create a new file named `tasks.py`. This file will contain automated tasks.

Step 2: Define Tasks in `tasks.py`

1. **Set Up a Task to Run Tests:**
   - Define a `test` task to execute all unit tests using `pytest`.
   - Add the following code to `tasks.py`:

```python
# tasks.py
from invoke.context import Context
from invoke import task

@task
def test(ctx: Context) -> None:
    """Run all tests with pytest."""
    ctx.run("poetry run pytest tests/")
```

   - This task can be invoked with `invoke test`
2. **Define a Task for Linting and Formatting:**
   - Define a `lint` task to check and format code using `ruff`.
   - Define a `format` task that formats the code using `ruff`
   - This will ensure that the code is consistently formatted according to the project's style guide.

```
@task
def lint(ctx: Context) -> None:
    """Run ruff to lint and format the code."""
    ...

@task
def format(ctx: Context) -> None:
    """Run ruff to lint and format the code."""
    ...
```

- Run this tasks with `invoke lint` and `invoke format`

3. **Define a Task for Type Checking:**
   - Define a type task that checks for correct typing using mypy

```
@task
def type(ctx: Context) -> None:
    """Check the types with mypy."""
    ...
```

4. **Define a Task to Generate Documentation:**
   - Define a `docs` task to generate HTML documentation using `pdoc` (assuming `pdoc` is installed as described in Section 4).
   - This task will generate fresh documentation in the `docs/` directory each time it's run.

```
@task
def docs(ctx: Context) -> None:
    """Generate HTML documentation with pdoc."""
    ...
```

   - Run this task with `invoke docs`

5. **Define a Task to Run the Pipeline:**
   - Create a `run` task to execute the pipeline, allowing users to specify the configuration file path.
   - By default, this task can use the `config/config_dev.yaml` file, but users can specify other files if needed.
   - Run this task with different configuration files.

Section 7 - Final Assembly and Execution

Objective:

To install the pipeline as a local package, execute it using the entry point, and verify the end-to-end data flow. Additionally, students will have the option to extend the pipeline with new features.

Instructions:

Step 1: Install the Package Locally with Poetry

1. **Install the Project:**
   - Make sure you're in the project's root directory and use Poetry to install the package locally. This will also install all dependencies listed in `pyproject.toml`.
   - Run: `poetry install`
   - Poetry will create a virtual environment, install the package, and make it accessible via the command line.
2. **Verify Installation:**
   - After installation, verify that the package is installed by listing the installed packages or by simply testing the entry point.

Step 2: Execute the Pipeline Using the Defined Entry Point

1. **Run the Pipeline with the Default Configuration File:**
   - Use the entry point defined in the `pyproject.toml` (e.g., `data-pipeline`) to execute the pipeline. This entry point allows you to specify a configuration file.
   - For example, to run with the development configuration file, use: `poetry run data-pipeline --config config/config_dev.yaml`
2. **Verify Output:**
   - The pipeline should execute in the following order:
     - **Data Loading**: The pipeline loads the data based on the file type and path specified in the configuration.
     - **Data Transformation**: The pipeline applies transformations according to the scaling method specified in the configuration.
     - **Model Application**: The selected model type is initialised, trained, and used to generate predictions on the transformed data.
   - The `loguru` logs will display the progress and results, and any issues will be logged for troubleshooting.

3. **Alternative Configurations:**
   - To test with different configurations, use alternative configuration files:
     `poetry run data-pipeline --config config/config_prod.yaml`
   - This flexibility allows for testing various setups and configurations within the same pipeline.

Step 3: Verify End-to-End Data Flow

1. **Inspect the Console and Log Output:**
   - Check the console output and the log file (e.g., `logs/pipeline.log`) to ensure that:
     - Data is loaded correctly (expected number of rows and columns).
     - Data is transformed according to the specified method (e.g., normalization).
     - Model training and prediction complete without errors, and predictions are returned in the expected format.
2. **Run Automated Tests:**
   - Use the `invoke test` task to run all unit tests and verify the individual components.
     `invoke test`

Step 4: Next Steps

After completing the core pipeline, one can further develop their project with the following enhancements:

1. **Add More Model Strategies**
   a. Extend the pipeline's modeling capabilities by adding new models like `KNNModel` or `RandomForestModel` that inherit from the `Model` base class.
   b. Add these models to the `ModelFactory` to enable easy switching between model types through configuration.
2. **Add More Complex Data Transformations**
   a. Introduce advanced transformations such as `PCA_Transformer` (for Principal Component Analysis) and `PolynomialFeaturesTransformer`.
   b. Add these new transformers to the `TransformerFactory`, allowing the configuration to dictate which transformation strategy to use.
3. **Additional Configuration Validations**
   a. The specified file path exists and is accessible.
   b. The file type matches the supported types.
   c. Transformation types are appropriate for the data.
4. **Add Dockerization and Implement CI/CD**
   a. Details in Next Lab!