2.1 OOP in ML Pipeline Refactoring

POSTGRAD
MNA

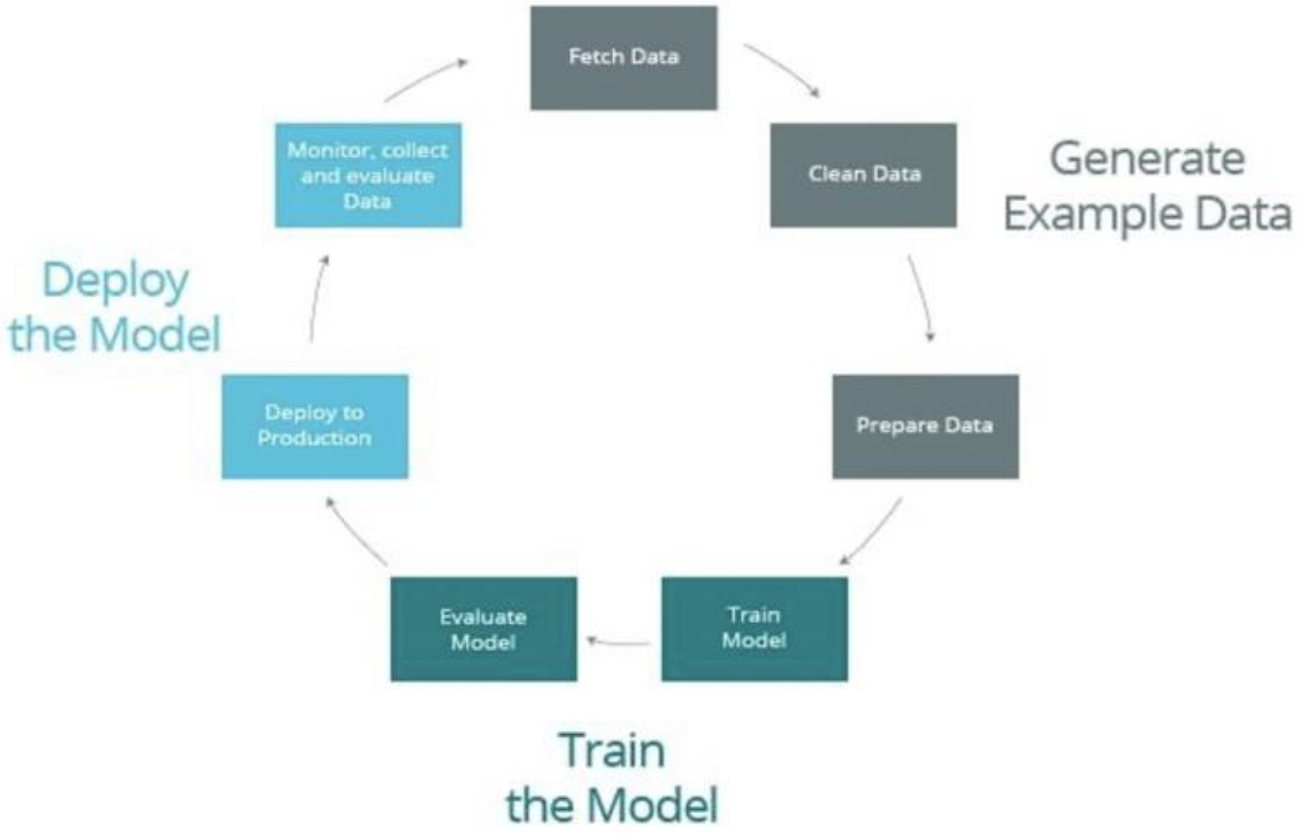
Introduction

Code refactoring is crucial in machine learning projects because it enhances the quality, maintainability, and scalability of the codebase. In ML development, initial models are often created quickly, focusing on experimentation and testing hypotheses. However, as projects grow, this approach can lead to code that is difficult to manage, test, or scale. Refactoring transforms this prototype-level code into modular, reusable, and efficient components, improving performance and making it easier to integrate new features or models. Additionally, clean and structured code simplifies debugging and testing, ensures consistent performance across environments, and enhances collaboration, especially when working with large teams or deploying to production environments. In the context of MLOps, refactoring supports best practices like version control, reproducibility, and automated pipelines, enabling smoother transitions from research to production.

2.1.1 Structure of ML Projects

ML project stages/workflow

- Data preparation
- Exploratory Data Analysis
- Feature Engineering
- Model Training
- Model Evaluation
- Deployment
- Monitoring



ML Cookie Cutter

- Provides structure for ML projects.

```
├── LICENSE          <- Open-source license if one is chosen
├── Makefile         <- Makefile with convenience commands like `make data` or `make train`
├── README.md        <- The top-level README for developers using this project.
├── data
│   ├── external     <- Data from third party sources.
│   ├── interim      <- Intermediate data that has been transformed.
│   ├── processed    <- The final, canonical data sets for modeling.
│   └── raw          <- The original, immutable data dump.
│
├── docs             <- A default mkdocs project; see www.mkdocs.org for details
│
├── models           <- Trained and serialized models, model predictions, or model summaries
│
├── notebooks        <- Jupyter notebooks. Naming convention is a number (for ordering),
│                       the creator's initials, and a short `-` delimited description, e.g.
│                       `1.0-jqp-initial-data-exploration`.
│
├── pyproject.toml   <- Project configuration file with package metadata for
│                       mlops and configuration for tools like black
│
├── references       <- Data dictionaries, manuals, and all other explanatory materials.
│
├── reports          <- Generated analysis as HTML, PDF, LaTeX, etc.
│   └── figures      <- Generated graphics and figures to be used in reporting
│
├── requirements.txt <- The requirements file for reproducing the analysis environment, e.g.
│                       generated with `pip freeze > requirements.txt`
│
├── setup.cfg        <- Configuration file for flake8
│
└── mlops  <- Source code for use in this project.
    │
    ├── __init__.py          <- Makes mlops a Python module
    │
    ├── config.py            <- Store useful variables and configuration
    │
    ├── dataset.py           <- Scripts to download or generate data
    │
    ├── features.py          <- Code to create features for modeling
    │
    ├── modeling
    │   ├── __init__.py
    │   ├── predict.py       <- Code to run model inference with trained models
    │   └── train.py         <- Code to train models
    │
    └── plots.py             <- Code to create visualizations
```

https://github.com/jeannefukumaru/cookiecutter-ml

Importance of repeatable workflows for scaling ML

- **Consistency in Processes:** Ensures data preparation, model training, and evaluation are reproducible.
- **Automation for Scalability:** Automating workflows allows projects to grow without requiring manual intervention.
- **Efficient Experimentation**: Repeatable workflows make it easier to test and compare different models or datasets.
- **Collaboration & Debugging**: Team members can rely on consistent workflows to debug and collaborate effectively.
- **Smoother Production Deployment**: Clear, repeatable workflows reduce risks and errors when moving from development to production.

Challenges of using notebooks in production

- **Lack of Modularity**: Notebooks often combine code, data, and logic in a single place, making it difficult to reuse or modify specific components independently.
- **Poor Version Control**: Tracking changes in notebooks is challenging since outputs and code cells can be modified out of order, complicating collaboration and code history.
- **Difficulty in Testing and Debugging:** Notebooks are designed for interactive exploration, making it hard to write unit tests or systematically debug complex code.
- **Scalability Issues:** Notebooks are not well-suited for handling large-scale data processing or training models at scale due to their linear execution flow and memory management.
- **Hard to Integrate with CI/CD Pipelines:** Continuous integration and deployment pipelines rely on modular, scriptable code, which is harder to implement in the notebook format.
- **Inconsistent Environments:** Running notebooks across different environments can lead to inconsistencies in libraries or dependencies, causing reproducibility problems in production.

2.1.2 OOP Fundamentals

Introduction to Object Oriented Programming (OOP)

- **Programming Paradigm:** OOP is a programming paradigm that organizes software design around objects, which represent real-world entities and data, instead of focusing solely on functions and logic.
- **Invented in the 1960s:** The concept of OOP was introduced in the 1960s with the creation of the programming language Simula, designed for simulations and modeling complex systems.
- **Popularized by Languages like C++, Java:** OOP gained widespread use in the 1980s and 1990s, with languages like C++ and Java, making it the dominant approach in software engineering.
- **Supported on Python:** Allowing developers to create classes and objects and implement OOP principles.
- **Key Concept Objects and Classes:** Objects represent real-world entities or concepts, and classes define the blueprint for these objects. This makes it easier to model complex systems.
- **Shift from Procedural Programming:** OOP emerged as an evolution from procedural programming, which focuses on functions and procedures. OOP allows for better data management and modularity.

Why use OOP?

The larger the programs become, the more difficult they are to keep organized.

OOP helps organize and structure the code:

- By grouping data and behaviors within one place.
- It promotes the modularization of programs.
- It allows you to isolate different parts of a program from others.
- Promotes reusability
- Simplifies debugging and maintenance

Basic Concepts of OOP

- **Class**: This is a template for creating objects of a particular type.
- **Methods**: functions that are part of the class.
- **Attributes**: Variables that contain data that are part of the class.
- **Object**: Specific instances of a class.
- **Inheritance**: means that one class can inherit capabilities from another.
- **Polymorphism**: obtaining different results depending on the class of an object.

Object

- The object is an entity that has a **state** and **behavior** associated with it.
- It resembles a real-world object abstraction.
- Consists of state, behavior and ID.
- It is the instance of a class.



```
# Python program to
# demonstrate defining
# a class and
# instantiating it

class SomeClass:
    pass

object = SomeClass()
```

Class

- A class contains the **blueprints, templates or the prototype** from which the objects are being created.
- It is a logical entity that contains some attributes and methods.
- In Python, classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. i.e. Class.Attribute
- In Python, class methods must have an extra self first parameter in the method definition. No value for this parameter is given when the method is called, it is provided by Python. It is an auto reference.
- Clases have __init__ method similar to constructors in C++. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization for the object.

```
#Python Class declaration
#example:

class ClassName:
  # Statement-1
  .
  .
  .
```

Attributes

They are variables that encode the **state** of an object.

That is, they are **characteristics** that apply to the object, for example:

- The look it shows
- The color
- Size
- Position
- Whether or not it is enabled

```
class Auto: # 1)
  kilometraje = 1000 # 2)
  marca = "Nissan" # 3)
  modelo = "March" # 4)
```

Methods

- Methods are those **functions** that the object can perform and that generate some type of service or results through the course of the program.
- They are part of the **functionality** we give to an object.

```
class Auto: # 1)
  kilometraje = 1000 # 2)
  marca = "Nissan" # 3)
  modelo = "March" # 4)

  def encender_auto(self): # 5)
    print("Auto encendido")

  def apagar_auto(self): # 6)
    print("Auto apagado")
```

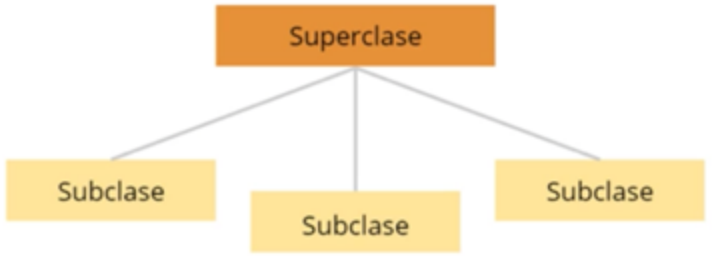
self

- The ***self*** parameter is a reference to the current instance of the class and is used to access the variables that belong to the object.
- It doesn't have to be called self, you can call it whatever you want, but it has to be the first parameter of any method in the class.

__init__()

- All classes have a method called ***__init__()***, which is always executed when the class is started.
- The ***__init__()*** method is used to assign values to object properties or other operations that are required when the object is created.

Inheritance



- Allowing new classes to derive from existing ones, promoting code reuse and extensibility.
- The class that derives properties is called the derived class or **child** class and the class from which the properties are being derived is called the base class or **parent**.
- Inheritance could be simple (inherit from a single class) or multiple (inherit from multiple classes simultaneously).

```
#Python single inheritance
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

```
#Python multiple inheritance
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Multiple Inheritance

- Multiple inheritance allows you to inherit methods and attributes from two or more base classes.
- It is important to properly name the attributes and methods in each class so as not to create conflicts.

```
class ClaseBase1:
  pass

class ClaseBase2:
  pass

class SubClase(ClaseBase1, ClaseBase2):
  pass
```

Polymorphism

- Enabling objects of different types to be accessed through the **same interface**.
- Supporting flexibility in program design, such that it could be possible to obtain different results calling the same method, depending on the object's class.
- Example shows how subclasses can override methods defined in their parent class to provide specific behavior.

```
#Python example of polymorphism and
#Inheritance

class Bird:

    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):

    def flight(self):
        print("Ostriches cannot fly.")
```

https://www.geeksforgeeks.org/python-oops-concepts/

Encapsulation

- Bundling data and methods together in a single class, hiding internal details and exposing only what's necessary.
- It describes the idea of wrapping data and the methods that work on data within one unit.
- Puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data.
- An object’s private variable can only be changed by an object’s method.

```
# Python program to
# demonstrate private members
# "__" double underscore represents private attribute.
# Private attributes start with "__".

# Creating a Base class
class Base:
    def __init__(self):
        self.a = "GeeksforGeeks"
        self.__c = "GeeksforGeeks"

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c) #Will rise error
```

https://www.geeksforgeeks.org/python-oops-concepts/

Abstraction

- Hiding unnecessary complexity by exposing only essential features.
- Reducing the burden on developers to understand implementation details.

```
class Rectangle:
    def __init__(self, length, width):
        self.__length = length  # Private attribute
        self.__width = width    # Private attribute

    def area(self):
        return self.__length * self.__width

    def perimeter(self):
        return 2 * (self.__length + self.__width)


rect = Rectangle(5, 3)

# Output: Area: 15
print(f"Area: {rect.area()}")

# Output: Perimeter: 16
print(f"Perimeter: {rect.perimeter()}")

# print(rect.__length)
# This will raise AttributeError
# as length and width are private attributes
```

https://www.geeksforgeeks.org/python-oops-concepts/

Why is OOP important for refactoring?

- **Modularity**: OOP breaks code into modular components (classes), making it easier to maintain and update.
- **Reusability**: Common functionality is encapsulated in reusable classes, reducing redundancy and simplifying updates.
- **Scalability**: OOP structures allow ML pipelines to scale by separating concerns into distinct objects.
- **Testability**: OOP makes code easier to test by isolating components, enabling targeted unit tests.

Benefits of OOP in ML Projects (Takeaways)

- **Improved Code Organization**: OOP helps organize code by separating concerns into classes, making it more structured and maintainable, following good software development practices.
- **Enhanced Collaboration**: Teams can work on separate modules or classes independently, improving productivity.
- **Easier Transition to Production**: OOP facilitates moving from experimentation to production by offering modular and testable code.
- **Integration with MLOps Tools**: OOP fits well into CI/CD pipelines and integrates smoothly with tools like MLFlow for tracking experiments and models.

2.1.3 Refactoring

Why is important to refactor for ML Pipelines

- Improving structure without changing behavior.
- Cleaner, **modular code**.
- Easier to maintain, test, and deploy.
- Scalability!!!
- When deploying a model, there are several possible scenarios:
  - One model deployed on one server
  - One model deployed on multiple servers
  - Multiple versions of a model deployed on one server
  - Multiple versions of a model deployed on multiple servers
  - Multiple versions of multiple models deployed on multiple servers

Why Object Oriented Programming for Refactoring?

- Modularity, flexibility, reusability, and scalability naturally provided by OOP.
- Code organization: Preprocessing, training, evaluation in separate classes.
- Cleaner, more resilient code.
- Easier maintenance, testing.
- Version control for models and different pipeline components.
- Improve reproducibility and experiment management.

Challenges of Refactoring Jupyter Notebooks

- **Non-modular Code**: Jupyter notebooks often mix data, logic, and presentation, making refactoring challenging.
- **Linear Execution Flow**: Notebooks are designed for interactive workflows, which don’t align well with production needs.
- **Difficult to Debug and Test**: Debugging and unit testing are more complex due to the lack of structured code in notebooks.
- **Dependency Management**: Notebooks can suffer from inconsistent dependencies, making reproducibility difficult.

Converting Notebooks into Functions

- Refactor reusable code blocks into standalone Python functions.
- Separating by functionality based on ML Workflow:
  - Data preparation
  - Exploratory Data Analysis
  - Feature Engineering
  - Model Training
  - Model Evaluation
  - Deployment
  - Monitoring

Modular Design: Breaking Down into ML Pipeline Components

- Use different code clases for different ML pipeline components: preprocessing, training, evaluation, etc.
- Modular design allows easy extension, scaling and reuse.

Identifying Code Smells in ML Notebooks

- **Common anti-patterns of Jupyter Notebooks**:
  - Redundant cells
  - Hardcoding
  - Copy-pasting
  - Complex execution sequence (individual cells)
  - Possible mixed dependencies (in different cells)
  - Sequential programming paradigm (in many cases)

- **Problems**: Difficult to scale, maintain, and debug

Appling OOP concepts to ML pipelines

- Designing a Simple OOP Structure:
  - Sample class design for ML pipeline: DataLoader, Preprocessor, Model, Evaluator, etc.
  - Relationships between classes and pipeline stages.
- Applying Encapsulation:
  - Encapsulating data preprocessing inside a specific class.
  - Encapsulating model training inside a specific class.
  - Improved code readability, separation of logic.
- Handling Configuration with OOP:
  - Use a Config class to manage hyperparameters and file paths
  - I.e. refactor a hardcoded script to use a configuration object that contains hyperparameters for hyperparameter tunning.

Clean and Modular Code
Good Practices

Clean and modular code

Regardless of your profile, the code you write can be taken to production.

- Code in production: Software that runs on production servers to manage live users and intended audience data
- Clean code: Refers to readable, simple, and concise code.
- Modular code: more organized, efficient, and reusable code.

Writing Modular Code

- DRY (Don't Repeat Yourself)
- Abstract logic to improve readability
- Minimize the number of entities (functions, classes, module, etc.)
- Code cohesion
- Try to use fewer than three arguments per function

Writing Modular Code

Measuring Execution Times with Python Timeit()

Python timeit() is a method in the Python library for measuring the execution time taken by the given piece of code.
It is a useful method that helps to check the performance of your code.

```
python -m timeit -n 1000 -s 'import math' 'math.sqrt(144)'
```

Writing Modular Code

Measuring Execution Times with Python Timeit()

- Using the timeit module directly in a script.
- Using timeit on a line with stmt and setup.
- Comparing different implementations.

```python
import timeit

# Código a evaluar
def suma_lista():
    return sum(range(1000))

# Medir tiempo de ejecución
tiempo = timeit.timeit(suma_lista, number=10000)
print(f"Tiempo de ejecución: {tiempo} segundos")
```

```python
import timeit

tiempo = timeit.timeit(stmt="sum(range(1000))", number=10000)
print(f"Tiempo de ejecución: {tiempo} segundos")
```

```python
import timeit

# Método con sum()
t1 = timeit.timeit("sum(range(1000))", number=10000)

# Método con bucle for
t2 = timeit.timeit("""
total = 0
for i in range(1000):
    total += i
""", number=10000)

print(f"Suma con sum(): {t1} segundos")
print(f"Suma con bucle for: {t2} segundos")
```