Module 3: AST-2

TITLE: Testing the Modules & Packaging

LEARNING OBJECTIVES:

At the end of the experiment, you will be able to include testing aspects in the project and write test cases continuing from AST1. Finally, you will be able to create a python package of the model which can be easily consumed by any API.

You will be able to understand and implement the following aspects:

1. Testing concept and automated testing using pytest
2. Packaging of model

INTRODUCTION

Testing: Software testing is a crucial part of the software development process. It involves executing a program or system with the intention of finding errors or verifying its compliance with specified requirements. The goal of testing is to identify defects and ensure that the software functions as intended, meets user expectations, and operates reliably in various scenarios.

Testing provides several benefits to software development:

- (i) **Error Detection**: Testing helps identify bugs, errors, and unexpected behavior in the software
- (ii) **Verification and Validation**: Testing validates that the software meets the specified requirements and performs as expected. Verification involves checking if the software conforms to the design and development specifications, while validation ensures that it meets the user's needs
- (iii) **Risk Mitigation**: Testing helps mitigate risks associated with software failure. By uncovering and fixing bugs early in the development cycle, potential risks such as system crashes, data loss, security vulnerabilities, or financial losses can be minimized
- (iv) **Code Maintenance and Refactoring**: Testing facilitates code maintenance and refactoring. When tests are in place, developers can confidently modify or refactor existing code without the fear of introducing new defects.
- (v) **Documentation and Understanding**: Tests serve as executable documentation of the software's behavior. They help developers understand the codebase better by acting as living examples and serving as a guide for future development.

Pytest is a popular testing framework in Python that simplifies the process of writing and executing tests. It provides a clean and concise syntax for defining test cases, fixtures for test setup and teardown, test discovery, and powerful assertions. Pytest integrates well with other tools and frameworks, making it a valuable tool for boosting testing productivity.

Types of Tests: Three types of tests can be designed: **Unit test, Integration test** and **system test. A unit test** is the smallest and simplest form of software testing. These tests are employed to assess a separable unit of software such as a class or a function for correctness, independent of the larger software system composed of many such unit components. A unit test verifies and passes individual units which are assembled into larger components. The **integration tests** are run on an assembled component to verify that it functions correctly, and finally, the **system tests** cover the end-to-end functionality of the system.

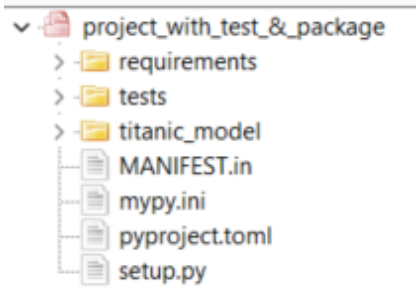
How much should we test? Prioritizing test for code base - functions or classes that are absolutely mission critical. This reduces uncertainty about the system's functionality when there is any change in the system.

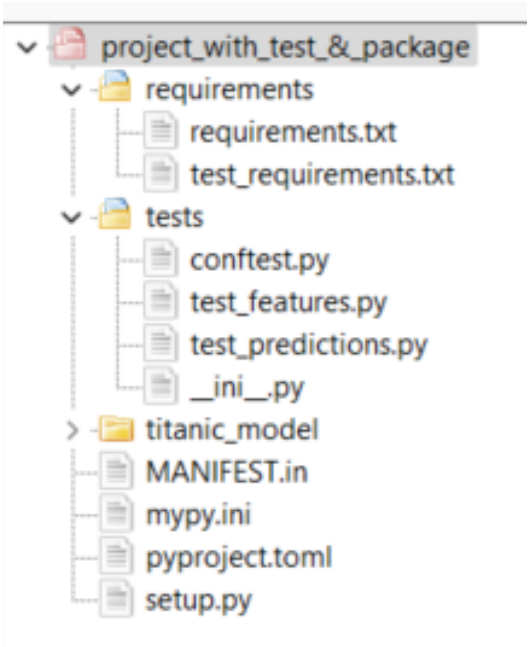
Testing design for machine learning systems is complex as compared to a traditional software system. The system behavior is controlled by software code only in a traditional software system. The ML model and the data are additional new elements in machine learning systems, which impact the behavior of the ML system

Package

A package is a collection of Python modules, and it is a useful way for us to publish related functionality so that different project applications can install our package and make use of the Python modules. The files that are related to packaging are **pyproject.toml**, **setup.py**, **manifest.in** and **mypy.ini** file.

Understanding the folder structure in production environment:





Understanding the functionality of each file

Files for Testing:

1. **conftest.py:** Here we are defining a fixture for **pytest** and then **sample_input_data** function is passed to each test case. The **sample_input_data function** returns the test set with the target which is used further for feature testing and prediction.
2. **test_features.py**: A sample test for one of the feature engineering codes. Inside the test, we have the **test_features.py** file for testing **age_col_tfr** class functionality.
3. **test_prediction.py:** A sample test for prediction and accuracy test.

Files for Packaging:

1. **pyproject.toml :**

   The key lines to do with packaging are one to six, where we specify what the basic dependencies for installing the packages are i.e. setuptools and wheel. We might

see some pyproject.toml lines which specify alternative tools for building. These are the standard tools that you'll come across the most. The rest of the pyproject.toml is dedicated to configuring our **tooling**, like **pytest** settings. And then further down, we have the configuration for our **linter**, which is called **black** and the import sorting tool called **isort**.
Be rest assured, as rarely, we will need to write something like a pyproject.toml file from scratch. Most of the time, we will be using a template, pulling and modifying a pyproject.toml from another project, or generated with the tool. The same applies to the setup file, manifest.in, and the tooling configurations. So, we don't have to write a file like this line by line.

2. **setup.py**

   The majority of the packaging functionality resides inside the Setup file. You can see the package metadata. We're going to call the package '**titanic_model**'. The setup file contains metadata about the package, such as its description, author, and compatibility with a specific version of Python. The version of the package is determined by reading a version file, which contains a single value specifying the version. This version value is then assigned to the metadata dictionary in the Setup file. The requirements for the package are also specified, usually by using a list of requirements from a requirements directory. All these values are passed as arguments to the setup function from the setuptools library, which is crucial for creating the package.

3. **Manifest.in**

   The Manifest.in file is used to specify the files that should be included or excluded in the package. This file is responsible for ensuring that important files like the pickle file, as well as the train and test CSVs, are included in the package. The inclusion of the CSVs allows other applications that depend on the package to use them for testing purposes.

4. **mypy.in:** The mypy.ini file is used to configure the specific type hints that we want to focus on and pay attention to in our code. However, this is manageable and effortless.

Tooling: Tools that are used to manage our package: In addition to our pytest library, which we're using for testing, we may have a few other libraries viz. black, flake, mypy, isort.

- **black** is a code-styling enforcement.

- **flake8** is a linting tool to tell us where we are not adhering to good Python conventions.
- **mypy** is a type-checking tool and
- **isort** is a tool for ensuring our imports are in the correct order.

The prediction of the test set and calculation of accuracy previously kept inside the **train_pipeline.py** is shifted to the **test_predictions.py** file**.**

PROCEDURE

The following steps involve downloading the project folder and uploading it to VS code, creating a virtual environment within the project, installing necessary dependencies, and executing specific scripts for training, testing, and packaging. It allows us to effectively configure the project and execute it within the VS code environment.

Steps: Download the **project_with_test_and_package** folder and upload in your VS code and follow the steps below:

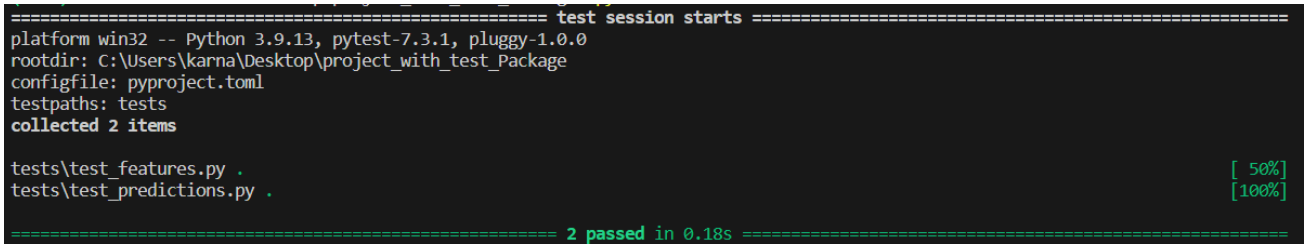
1. Upload the **project_with_test_and_package** folder in VS code

2. Inside the **project_with_test_and_package,** create a virtual environment as in AST-1.

3. Go inside requirements[cd requirements] folder and run :
   …> pip install -r test_requirements.txt
4. Navigate out of requirements folder [cd..] and go inside titanic_model folder and run :
   py train_pipeline.py

5. Navigate out from the titanic_model folder and be inside the **project_with_test_and_package** folder.
   Now, run: ...> pytest

   You will see result like this:



Which indicates that tests have been passed and the model is working as intended

6. For building the package from the model with its remaining functionality:

   Run: ...> py -m pip install --upgrade build

   And then
   Run: ...> py -m build

   You will see result like this upon successful building of package:

   You will notice, two additional folders will be created:

   1. **dist:** This contains the distributable '**whl'** file and '**gz tar'** file. The '**whl'** file is easy to install.
   2. **titanic_model.egg-info** : This contains metadata and information of the packages along with the requirements information.

Note: For the **technical question** and to understand how test cases are written inside the **test_features.py**, open the **Experimentation_Phase_2_with_test.ipynb** file.