**Testing Seldon Microservice deployed in Docker using Python**

Created on: 09th March, 2022

**Overview and Prerequisites:**

In this document, we walk you through developing a test script to test your seldon microservice deployed in a docker container. Before we discuss about different tests, it is assumed that following pre-requisites on your part are met:

1. System with Docker and Python installed.
2. System with PyCharm IDE installed is preferable.
3. A docker image has been created for seldon microservice you wish to containerize. This document briefly describes how this dockerfile is useful in our task. If the images are not already present, execute the following two commands a terminal opened at the root directory of this project:

* $docker build -f docker/train-dockerfile/Dockerfile -t wine-classifier-image .
* $docker run --name wine-classifier-container -v $(pwd):/app -d wine-classifier-image:latest
* $docker build -f docker/seldon-core-dockerfile/Dockerfile -t wine-classifier-seldon-image .

**Setup:**

**Creating WineClassifierProject**

The wine-classification code was downloaded from this [GitHub](https://github.com/kumargauravsingh14/wine-classification) link just for implementational purposes, with the following modifications made to the files in the downloaded folder:

1. The predict function that returns class probabilities of data points has been moved to a file called WineClassifierModel.py
2. Packages necessary for this project have all been listed in requirements.txt file. They are installed when docker image is created for our model using Dockerfile.
3. A json file called predict\_requests has been created which holds 5 test data-points that are passed to the test script.
4. A folder called docker where there are two sub-folders – seldon-core-dockerfile and train-dockerfile. These folders each host a Dockerfile that are run to create docker container images required for our task.

*Note: The location of project is /Users/<UserName>/PycharmProjects/WineClassifierProject in your system which can be obtained by shift+command+F*

*By default, every new project in Pycharm is created in the folder PycharmProjects, and it is initialized with a main.py file*

**WineClassifierModel.py**

The seldon microservice for this project comprises of a wine classifier model that takes data-point with 4 numeric features and returns prediction probabilities of three classes of wine to which the data-point may belong to.

The predict method is defined in WineClassifierModel class in which the ML model ‘wine-classifier-model.pkl’ is loaded and predict\_proba method of this model is invoked on the new data-point whose class needs to be predicted.

Text

Description automatically generated

**Graphical user interface, text, application, chat or text message

Description automatically generated**

**Building images using docker files**

In this project, there are two Dockerfile scripts from which two images are built. A brief purpose of these files has been explained below:

* **Train-dockerfile/Dockerfile:** This is the first dockerfile which is run to build wine-classifier-image. A snapshot of this dockerfile is given below:

A screenshot of a computer

Description automatically generated with medium confidence

When a container is created using this image, this dockerfile instructs the docker to run the TrainWineClassifier.py script. TrainWineClassifier.py is a python script that trains a model on training data and writes the resultant model to wine-classifier-model.pkl file. Hence, the model training happens inside a docker container, instead of local directory, and the resultant model pkl file is also copied into the local directory. This approach of training the model in container –

1. saves one the hassle of creating a virtual python environment in local system
2. eliminates version-compatibility issues while working with seldon-core and python
3. allows easy code portability across different python versions just by changing the base python image
4. ensures replicability and scalability of results irrespective of OS and other system settings

Hence, now we have a wine-classifier-model.pkl file that is in our local directory, which leads us to the second dockerfile’s purpose.

* **Seldon-core-dockerfile/Dockerfile:** This is the second dockerfile used to build wine-classifier-seldon-image which offers the seldon microservice in a containerized form. A snapshot of this dockerfile is seen below:

Text

Description automatically generated

As can be seen, this image builds on top of the first docker-image. A container created using this image communicates with external ports through ‘9000/tcp’. i.e. microservice requests can be placed to the container through port 9000. Here, [MODEL\_NAME](https://docs.seldon.io/projects/seldon-core/en/latest/python/python_wrapping_docker.html) is the model class containing the mode, which is WineClassifierModel.

Hence when a container is built from wine-classifier-seldon-image and running, it allows one to post HTTP requests with data-point information to this container at port 9000, by invoking the predict method in the HTTP url. The container, in-turn, invokes this method in the model class, and responds with a response (i.e. the classification result for the data-point).

Hence, we now have two images (wine-classifier-image and wine-classifier-seldon-image), we have a model class with predict function that will predict data-point’s class. Time to start our testing!

**Test Setup**

Tests have all been implemented using pytest module in the script called test\_script.py, which has integration and unit tests (described in detail in Functional Tests section).

Pytest fixtures have been used to create global variables that are accessed by the tests written in test\_script. These fixtures are in conftest.py file

When pytest goes to run a test, it looks at the parameters in that test function’s signature, and then searches for fixtures that have the same names as those parameters. Once pytest finds them, it runs those fixtures, captures what they returned (if anything), and passes those objects into the test function as arguments. More details about how pytest fixtures work can be found [here](https://docs.pytest.org/en/6.2.x/fixture.html)

*Note: To ensure that fixtures are created only once, scope = ‘module’ has been defined. This creates fixtures once for the whole test\_script module, instead of the default setting which creates fixtures for each test function defined in the module. This is crucial because we want the docker container to be created only once for the whole test\_script*

*Note: All the tests functions in this script. start with the name test\_ to help distinguish between fixtures and tests. In addition, each fixture function has @pytest.fixture identifier above it.*

To run all the tests, open the terminal in project root directory and type the following command:

$pytest

When this command is run, pytest automatically calls all the files that start with test\_ or end with \_test.py.

**Client\_helper.py**

This script uses docker SDK for python to build and run docker container from docker image using python.

Text

Description automatically generated

It requires installation of docker module. The create\_container() function defined below –

1. builds and runs the docker for the wine-classifier-seldon-image in detached mode
2. establishes a connection between the container’s app directory with the root directory of our project in our local system, so that any files created in container reflect in our directory and vice-versa.
3. Exposes the classification model inside the container for outside communication through port 9000/tcp. This 9000/tcp port of container is connected to <http://localhost:3000>. So any (POST) predict requests placed to <http://localhost:3000> are directed to our classifier’s predict method through port 9000/tcp.

**Functional Tests**

This section contains details about the tests implemented in this test\_script.py file.

**Integration Tests:** The components tested are the container, communication between container and local host, and communication between python and docker.

1. **test\_container\_creation:** This test checks whether a container has been created successfully or not.

It takes container fixture as input.

1. **test\_container\_running:** This test checks
2. whether the container is running properly (status of container)
3. the container's exposed port
4. the host port of communication
5. whether container is created for right image
6. the container's working directory, where all files created while running application are stored.

It takes container fixture as the input.

1. **test\_model\_used:** This test checks whether the right classification model is used by the container for predicting new instances.

It takes container fixture as the input.

1. **test\_connection\_to\_model:** This test checks whether the responses received from the docker container for the (POST) predict requests to the localhost URL are successful or not.

It takes all\_responses fixture as the input.

1. **test\_container\_stopping:** This test checks whether the container stopped successfully.

It takes stop\_container and container fixtures as the inputs.

**Units Tests:** Schema of request and response json files has been tested.

1. **test\_request\_schema**: This test checks whether the request data-point comprises of four numeric values passed as an array or not.

It takes predict\_requests fixture as the inputs.

1. **test\_response\_schema**: This test checks whether the response received from container has the expected fields or not, and whether the sum of all class probabilities is equal to 1

*Note on capturing runtime metrics:*

*Currently Seldon does not support the collection of run-time metrics outside Kubernetes cluster. Typically, Seldon metrics are exposed inside Kubernetes for Prometheus to scrape them. Hence the “meta” field in the API response is empty here.*