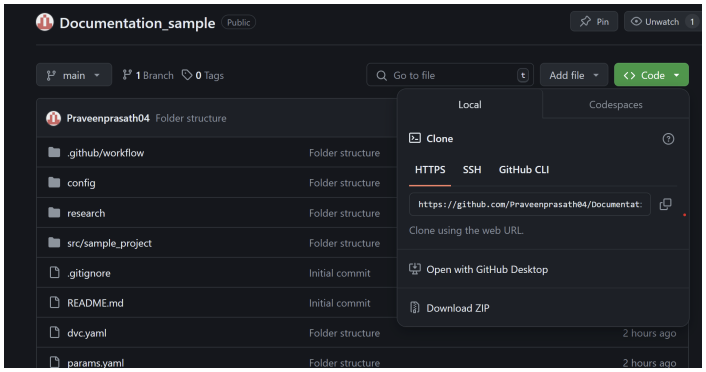
Model Deployment Documentation

Making code Modular

In-order to effectively operate our model we should convert the our model from jupyter notebook to modules.

Creating GIT Repo

First we need create a repository in GitHub and clone it to our PC.
To clone the repository copy the URL from GitHub repo page.



Open the terminal in the file which you want to create the repository and type.

git clone "copied-URL".

Creating a virtual environment

We need to set up a virtual environment separate from the base environment for our project so it can be recreated and isolated from other dependencies.
Create a new python environment in the project directory and make sure to activate the environment.

Creating the Project Template

We need to initialise the structure of the project with multiple parts that work simultaneously. We can use this general template file https://github.com/Praveenprasath04/Documentation_sample/blob/main/template.py for all projects and can additional files that are required.

0.0.1 Requirements.txt

Any libraries wished to be used in model should included in this file. -e. signifies project module (refer setup.py)
To install requirements use.

```
In cmd on workflow

    pip install -r requirements.txt
```

0.0.2 Setup.py

setup.py is a module used to build and distribute Python packages. It typically contains information about the package, such as its name, version, dependencies, and instructions for building and installing the package. This information is used by the pip tool, which is a package manager for Python that allows users to install and manage Python packages from the command line. By running the setup.py file with the pip tool, you can build and distribute your Python package so others can use it.

We are using here inorder to export src/project as a module to use in our project and export to any other project.

to install any function from scripts in src/project in any scripts or notebook

```
from project.{scipt_name} import {function_name}
```

Custom Logging

Inside the project module. we have added a custom logger function which if used logs the action in logs file Inorder to use the Logger funtion

```
from project import logger
```

Utilis

Utils can be used to store utility functions which are used frequently in various modules.

Workflow setup

The workflow for the model should be updated by following steps.

Workflows

Update config.yaml Update params.yaml Update the entity Update the configuration manager in src.config Update the components Update the pipeline Update the main.py Update the dvc.yaml

Example: Data ingestion step

Ensure to have the zip file of data "MURA-v1.1.zip" on workflow.

Step 1: Update config.yaml.

In config.config.yaml. add a directory for artifacts folder which will be used for data storage, training model storage..etc which does not need to be pushed to the git repo (add artifacts to .gitignore). Create a separate section named data ingestion and store every unique metric needed for data ingestion.

```
in workflow\config\config.yaml

    artifacts_root: artifacts

    data_ingestion:
      root_dir: artifacts/data_ingestion
      source_dir: MURA-v1.1.zip
      local_data_file: artifacts/data_ingestion/MURA-v1.1.zip
      unzip_dir: artifacts/data_ingestion
```

Step 2 : Update Params.yaml

Params.yaml is used to store parameters of the model for easier access. For model used in repo params.yaml looks like.

In workflow\params.yaml

```
KEY: VAL
BATCH_SIZE : 17
VALID_SIZE : 1e-5
LEARNING_RATE : 0.0001
MOMENTUM: 0.9
IMAGE_DIM : 224
RANDOM_SEED : 42
EPOCHS: 10
```

Add or change parameters as per your requirement.

Step 3: Update the Entity

In Mura_model.entity.config._entity We will store our config entity for each step. Typically config entity consists of paths, and params used in this particular program. The Data ingestion config entity looks like this.

in src\Mura_model\entity\config_entity.py

```
from dataclasses import dataclass
from pathlib import Path


@dataclass(frozen=True)
class DataIngestionConfig:
    root_dir:Path
    source_dir: Path
    local_data_file : Path
    unzip_dir: Path
```

Step 4: Update Configuration Manager

In Mura_model.config.configuration.py. We create an object Configuration manager. Which stores all configurations and creates folders required throughout the entire process. We can import the configuration manager into our end pipeline. configuration.py for data ingestion looks like

In src\Mura_model\config\configuration.py

```
from Mura_model.constants import *
from Mura_model.utils.common import read_yaml, create_directories
from Mura_model.entity import DataIngestionConfig

class ConfigurationManager:
    def __init__(
        self,
        config_filepath = CONFIG_FILE_PATH,
        params_filepath = PARAMS_FILE_PATH):
```

```
        self.config = read_yaml(config_filepath)
        self.params = read_yaml(params_filepath)

        create_directories([self.config.artifacts_root])



    def get_data_ingestion_config(self) -> DataIngestionConfig:
        config = self.config.data_ingestion

        create_directories([config.root_dir])

        data_ingestion_config = DataIngestionConfig(
            root_dir=config.root_dir,
            source_dir=config.source_dir,
            local_data_file=config.local_data_file,
            unzip_dir=config.unzip_dir
        )

        return data_ingestion_config
```

Step 5: Update Components

In src.Mura_model.components, Create a new file or use an existing file corresponding to your need. For data ingestion, we use data_ingestion.py. Components store the function part of the program. We will define every file and process in the entire model here. For Data ingestion, components look like this.

In src\Mura_model\components\data_ingestion.py

```
    import os
    import urllib.request as request
    import zipfile
    from Mura_model import logger
    from Mura_model.utils.common import get_size
    import shutil
    from Mura_model.entity import DataIngestionConfig

    class DataIngestion:
        def __init__(self, config: DataIngestionConfig):
            self.config = config
        def get_data_file(self):
            shutil.copyfile(self.config.source_dir,self.config.local_data_file)
        def extract_zip_file(self):
            unzip_path = self.config.unzip_dir
            os.makedirs(unzip_path, exist_ok=True)
            with zipfile.ZipFile(self.config.local_data_file, 'r') as zip_ref:
                zip_ref.extractall(unzip_path)
```

Notice this process used Configurations we have set up earlier.

Step 6: Update pipeline

In src.Mura_model.pipeline. We created a new file named stage_01_data_ingestion.py. This file will be used as a connection to the main pipeline and the rest of the process. Notice we have named it stage_1 as data ingestion is the first step in building the model. Subsequent steps should named vice versa.

In src\Mura_model\pipeline\stage_01_data_ingestion.py

```
from Mura_model.config.configuration import ConfigurationManager
from Mura_model.components.data_ingestion import DataIngestion
from Mura_model import logger

STAGE_NAME = "Data Ingestion stage"

class DataIngestionTrainingPipeline:
    def __init__(self):
        pass

    def main(self):
        config = ConfigurationManager()
        data_ingestion_config = config.get_data_ingestion_config()
        data_ingestion = DataIngestion(config=data_ingestion_config)
        data_ingestion.get_data_file()
        data_ingestion.extract_zip_file()

if __name__ == '__main__':
    try:
        logger.info(f">>>>>> stage {STAGE_NAME} started <<<<<<")
        obj = DataIngestionTrainingPipeline()
        obj.main()
        logger.info(f">>>>>> stage {STAGE_NAME} completed <<<<<<\n\nx==========x")
    except Exception as e:
        logger.exception(e)
        raise
```

Step 7: Upgrade main.py

main.py is the execution script for all processes defined. if we need to run a process, we need to run m ain.py and we will get our result. To run only the data ingestion part

In main.py

```
from  Mura_model import logger
from Mura_model.pipeline.stage_01_data_ingestion import DataIngestionTrainingPipeline

STAGE_NAME = "Data Ingestion stage"
try:
    logger.info(f">>>>>> stage {STAGE_NAME} started <<<<<<")
    obj = DataIngestionTrainingPipeline()
    obj.main()
    logger.info(f">>>>>> stage {STAGE_NAME} completed <<<<<<]\n\n[x==========x")
except Exception as e:
    logger.exception(e)
```

```
        raise e
```

To run main.py

In cmd

```
    python main.py
```

Output looks like

```
    [2024-08-12 19:45:58,838: INFO: main: >>>>>> stage Data Ingestion stage started <<<<<<]
    [2024-08-12 19:45:58,840: INFO: common: yaml file: config\config.yaml loaded successful
    [2024-08-12 19:45:58,840: INFO: common: yaml file: params.yaml loaded successfully]
    [2024-08-12 19:45:58,840: INFO: common: created directory at: artifacts]
    [2024-08-12 19:45:58,840: INFO: common: created directory at: artifacts/data_ingestion]
    [2024-08-12 19:46:29,494: INFO: main: >>>>>> stage Data Ingestion stage completed <<<<<

    [x==========x]
```

1 Deploying model in AWS Cloud

We need to make docker file in-order to convert the source code into docker image.

We need to Create a iam