A

Capstone Project Report

On

Chest Disease Classification from Chest CT Scan Image

Submitted during 6th semester in partial fulfillment of the requirements for the award of degree of

Bachelor of Technology

In

Computer Engineering

by

Jatin Sareen (21001003059)

Under supervision of

Dr. Deepika

Department of Computer Engineering
FACULTY OF INFORMATICS & COMPUTING

J.C. Bose University of Science & Technology, YMCA
Faridabad – 121006

May 2024

CANDIDATE'S DECLARATION

I hereby certify that the work being presented in this Project titled **"Chest Disease Classification from Chest CT Scan Image**" in partial fulfillment of the requirements for the degree of Bachelor of Technology in Computer Engineering and being submitted to "J.C. Bose University of Science and Technology, YMCA, Faridabad", is an authentic record of my own work carried out under the supervision of **Dr. Deepika**.

The work contained in this project has not been submitted to any other University or Institute for the award of any degree or diploma by me.

(Jatin Sareen)

Student's Signature

(Dr. Deepika)

Supervisor Name and Signature

TABLE OF CONTENTS

Candidate's Declaration

Chapter 1: Introduction

1.1 Brief of the Project

1.2 Technologies Used

1.3 System Design

1.4 Data Description

Chapter 2: Project Walkthrough

2.1 Project Structure details

2.2 Data Ingestion Pipeline

2.3 Prepare Base Model Pipeline

2.3 Model Training Pipeline

2.4 Model Evaluation Pipeline

Chapter 3: Proposed Work

3.1 Prediction Pipeline

3.2 DVC Pipeline

3.3. Docker Implementation

3.4 User InterFace

Chapter 4: Result Analysis and Future scope

4.1 Jupter Notebok Observations

4.2 Prediction Outputs and Future Scopes

References

Chest Disease Classification from Chest CT Scan Image

1) INTRODUCTION

Chest CT scan images hold invaluable diagnostic potential in identifying various pulmonary conditions, including malignant tumors. Our project aims to streamline the classification process of these images into four distinct classes: 'adenocarcinoma', 'large cell carcinoma', 'normal', and 'squamous cell carcinoma'. Through the utilization of advanced techniques such as Deep Learning and Data Version Control (DVC), coupled with the robust ResNet50 architecture. I have developed an end-to-end pipeline that ensures efficient and accurate classification of chest CT scan images.

In addition to these methodologies, our project also integrates MLflow, a powerful machine learning lifecycle management tool. MLflow facilitates experiment tracking, model management, and reproducibility, allowing us to monitor and compare multiple models, hyperparameters, and experiments seamlessly. Through the synergy of DVC for data versioning and MLflow for model management, our project ensures transparency, reproducibility, and continuous improvement in the development and deployment of machine learning models.

2) Problem Statement

Despite the diagnostic potential of chest CT scans, the manual interpretation of these images is time-consuming, prone to error, and heavily reliant on the expertise of radiologists. There is a growing need for automated systems capable of accurately classifying chest CT scan images into distinct disease categories, thereby facilitating efficient diagnosis and treatment decision-making.

3) Tool Used

The project utilizes several key technologies to streamline the classification process of chest CT scan images. Python serves as the primary programming language, while Flask facilitates the development of the web application. Deep learning algorithms, particularly the ResNet50 architecture, are employed for image classification tasks. Data Version Control (DVC) ensures efficient management of data versions, while MLFlow provides tools for experiment tracking and model management. Infrastructure components include DockerHub for containerization, DagsHub for data pipeline management, and GitHub for version control. This comprehensive tech stack enables the seamless integration of advanced techniques, ensuring transparency, reproducibility, and continuous improvement in the development and deployment of machine learning models for chest disease classification. Visual Studio Code is used as python IDE for all modular coding and custom APIs creation. And storing all code files for publically available we will use GitHub.

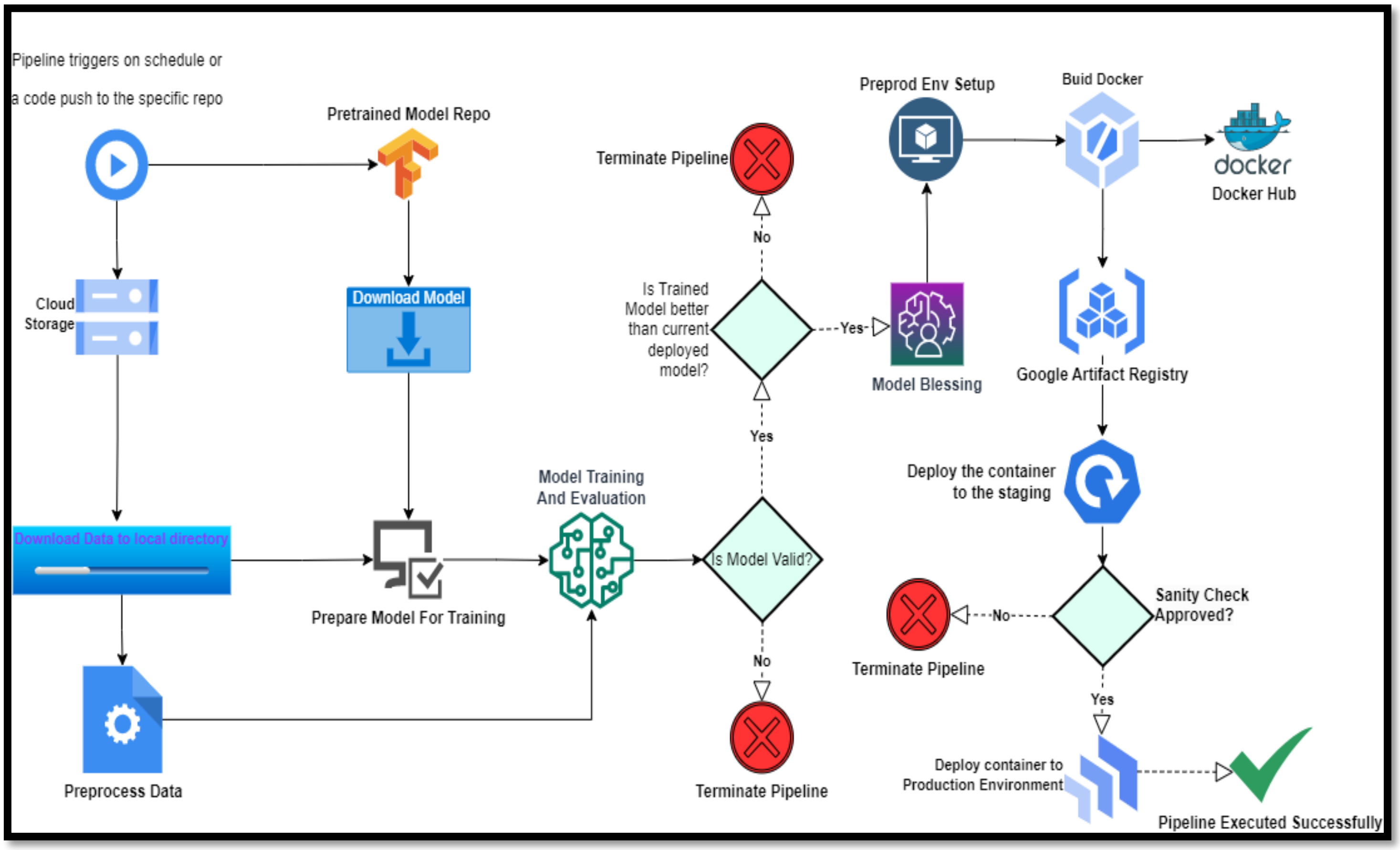
➢ TECH STACK USED

- Python
- Flask
- Deep Learning Algorithms
- Data Version Control (DVC)
- MLFlow

➢ Infrastructure:

- DockerHub
- DagsHub
- GitHub

4) System Design


Pipeline triggers on schedule or
a code push to the specific repo
Pretrained Model Repo
Cloud Storage
Download Model
Download Data to local directory
Prepare Model For Training
Preprocess Data
Model Training And Evaluation
Is Model Valid?
No
Terminate Pipeline
Yes
Is Trained Model better than current deployed model?
No
Terminate Pipeline
Yes
Model Blessing
Preprod Env Setup
Buid Docker
docker
Docker Hub
Google Artifact Registry
Deploy the container to the staging
Sanity Check Approved?
No
Terminate Pipeline
Yes
Deploy container to Production Environment
Pipeline Executed Successfully

5) Data Desciption

The dataset utilized in the Chest Disease Classification project comprises chest CT scan images sourced from **Kaggle**. These images are essential for identifying various pulmonary conditions, including adenocarcinoma, large cell carcinoma, squamous cell carcinoma, and normal lung tissue. The dataset consists of images in JPG or PNG format, rather than DICOM (Digital Imaging and Communications in Medicine) format, to suit the model's requirements.

The dataset is organized into three main folders within the 'Data' directory: 'test', 'train', and 'valid'. The 'test' folder contains images reserved for evaluating the trained model's performance. The 'train' folder comprises images utilized for training the model, representing 70% of the dataset. Similarly, the 'valid' folder contains images used for validation, constituting 10% of the dataset.

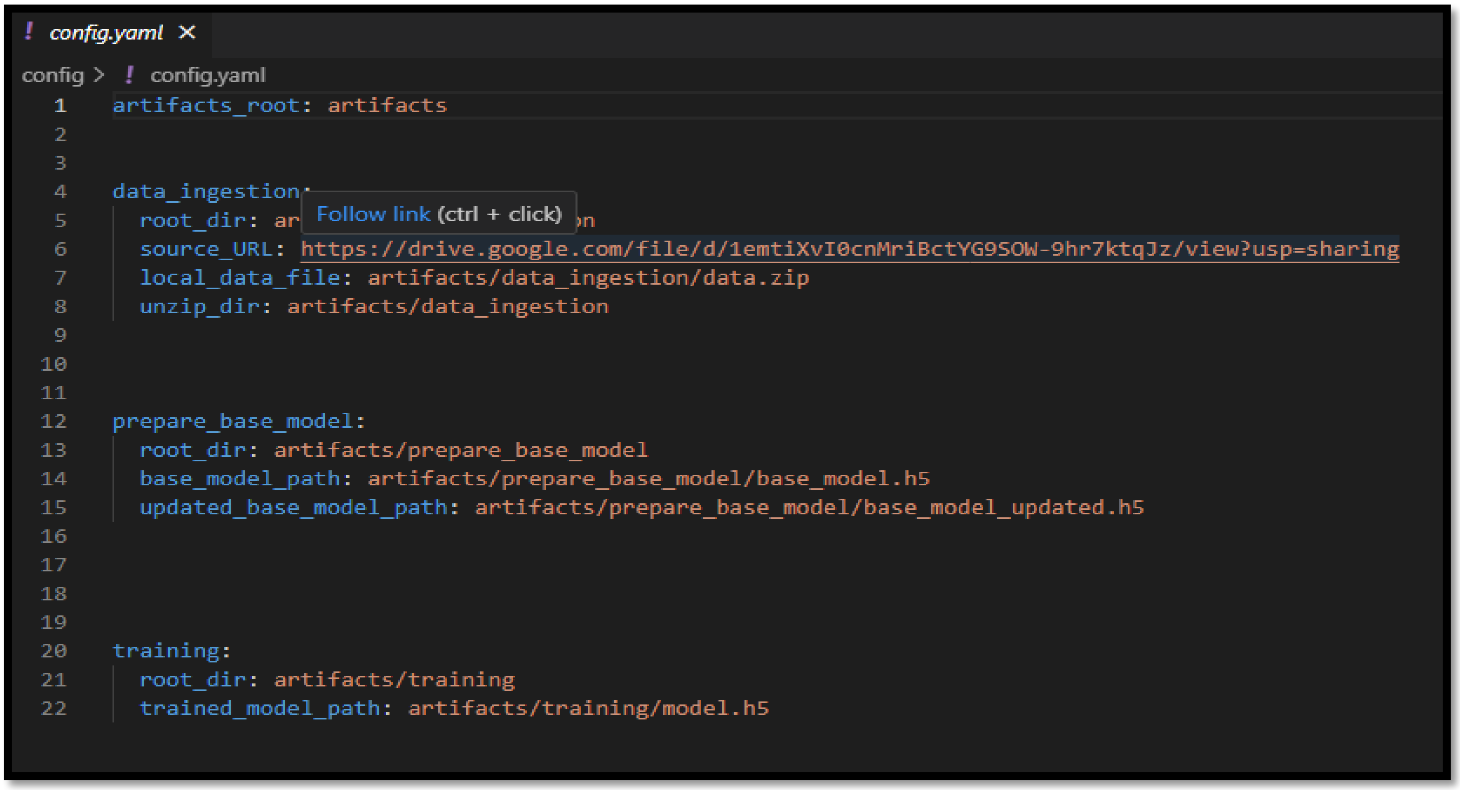
In total, the dataset consists of 1000 images, with a total size of approximately 124 MB. Specifically, the training set contains 700 images, the testing set contains 200 images, and the validation set contains 100 images. This partitioning ensures a balanced distribution of data for training, testing, and validation, thereby facilitating robust model development and evaluation.

Kaggle Dataset Link: https://www.kaggle.com/datasets/mohamedhanyyy/chest-ctscan-images/data

Now let's see the Chest Disease Classification from Chest CT Scan Image Project folder structure……..

6) <u>Project Structure Details</u>

(A) <u>Config.yaml file</u>



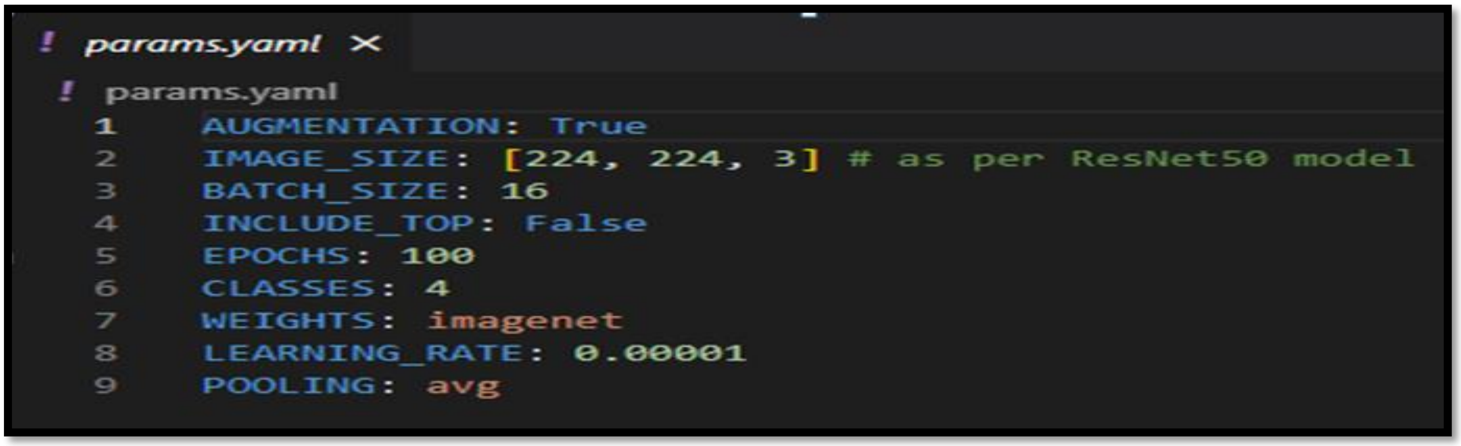
```
artifacts_root: artifacts


data_ingestion:
  root_dir: ar Follow link (ctrl + click) on
  source_URL: https://drive.google.com/file/d/1emtiXvI0cnMriBctYG9SOW-9hr7ktqJz/view?usp=sharing
  local_data_file: artifacts/data_ingestion/data.zip
  unzip_dir: artifacts/data_ingestion



prepare_base_model:
  root_dir: artifacts/prepare_base_model
  base_model_path: artifacts/prepare_base_model/base_model.h5
  updated_base_model_path: artifacts/prepare_base_model/base_model_updated.h5




training:
  root_dir: artifacts/training
  trained_model_path: artifacts/training/model.h5
```

The config.yaml file serves as a central configuration file for managing various components and directories within the Chest Disease Classification project. It defines the directory structure and paths for different stages of the project pipeline, facilitating organization and accessibility of artifacts and resources.

The artifacts_root parameter specifies the root directory where all project artifacts will be stored. This includes data, models, and any other outputs generated during the project lifecycle.

Similarly, the prepare_base_model and training sections define directories and paths for preparing the base model and storing trained model artifacts, respectively. This structured configuration file ensures consistency and ease of management across different stages of the project pipeline, streamlining development and facilitating reproducibility.

(B) <u>Params.yaml file</u>



```
AUGMENTATION: True
IMAGE_SIZE: [224, 224, 3] # as per ResNet50 model
BATCH_SIZE: 16
INCLUDE_TOP: False
EPOCHS: 100
CLASSES: 4
WEIGHTS: imagenet
LEARNING_RATE: 0.00001
POOLING: avg
```

The params.yaml file contains essential parameters and configurations utilized during the training phase of the Chest Disease Classification project. These parameters define various aspects of the training process, including data augmentation, image size, batch size, number of epochs, number of classes, choice of pre-trained weights, learning rate, and pooling strategy.

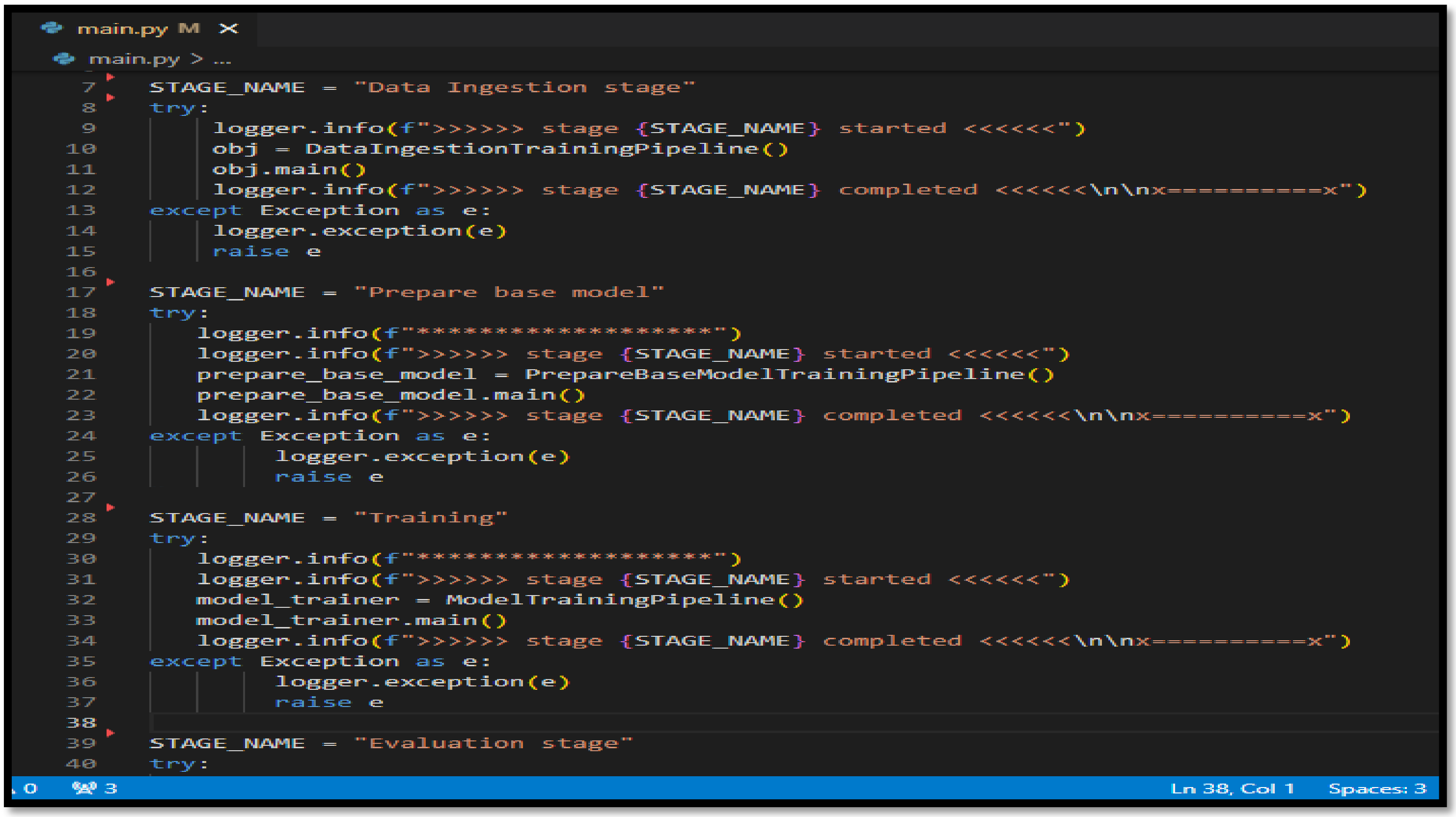
For instance, the AUGMENTATION parameter specifies whether data augmentation techniques should be applied during training, enhancing model robustness and generalization. The IMAGE_SIZE parameter defines the dimensions of input images, aligned with the requirements of the ResNet50 model architecture. BATCH_SIZE determines the number of samples processed per training iteration, while EPOCHS specifies the number of training epochs for model convergence. Additionally, CLASSES denotes the number of classes in the classification task, guiding model output configuration. The WEIGHTS parameter specifies the choice of pre-trained weights, with options such as 'imagenet' for using weights trained on the ImageNet dataset. LEARNING_RATE defines the rate at which the model adjusts its parameters during training, impacting training speed and convergence. Finally, POOLING determines the pooling strategy employed in the model architecture, with options such as 'avg' for average pooling.

(C) <u>main.py</u>

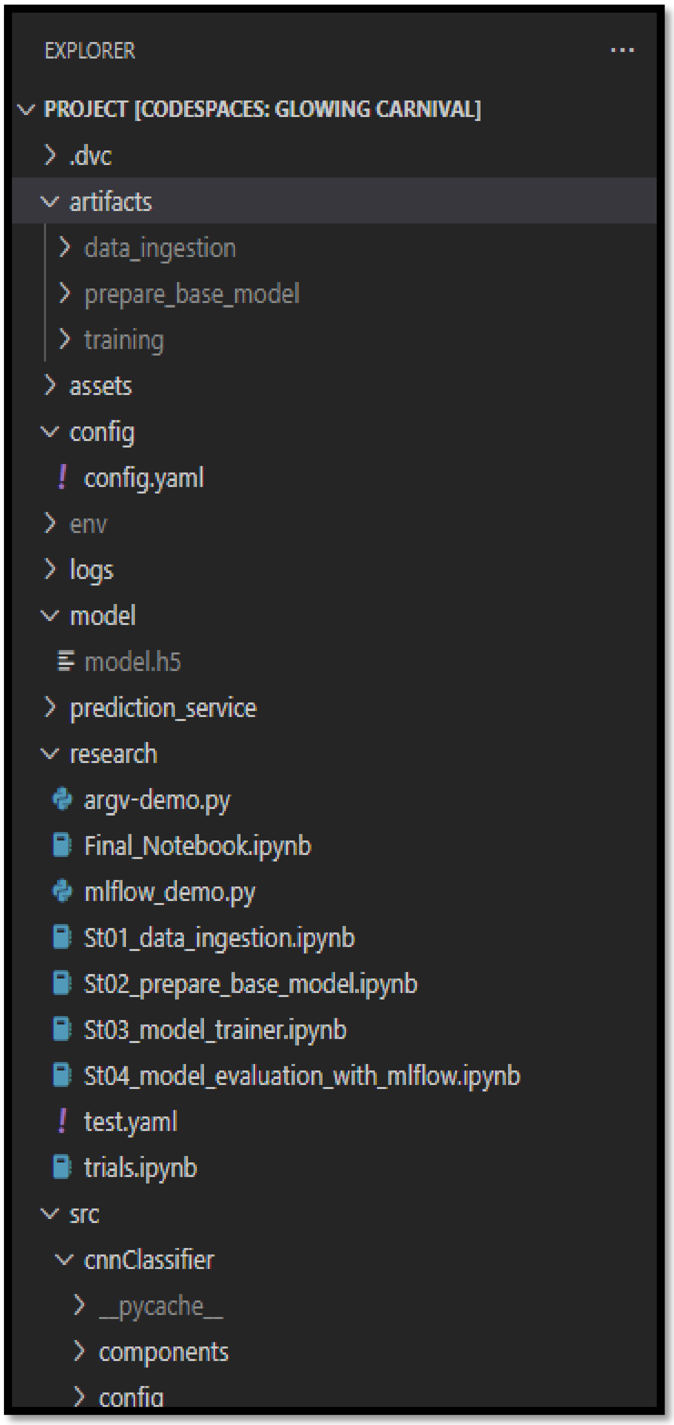
The main.py file orchestrates the execution of the different stages in the Chest Disease Classification project pipeline. It imports modules corresponding to each stage, including data ingestion, base model preparation, model training, and evaluation.

The file begins by initializing the data ingestion stage, followed by the preparation of the base model, model training, and finally, model evaluation. At each stage, the main() method of the corresponding pipeline class is invoked to execute the stage-specific tasks.

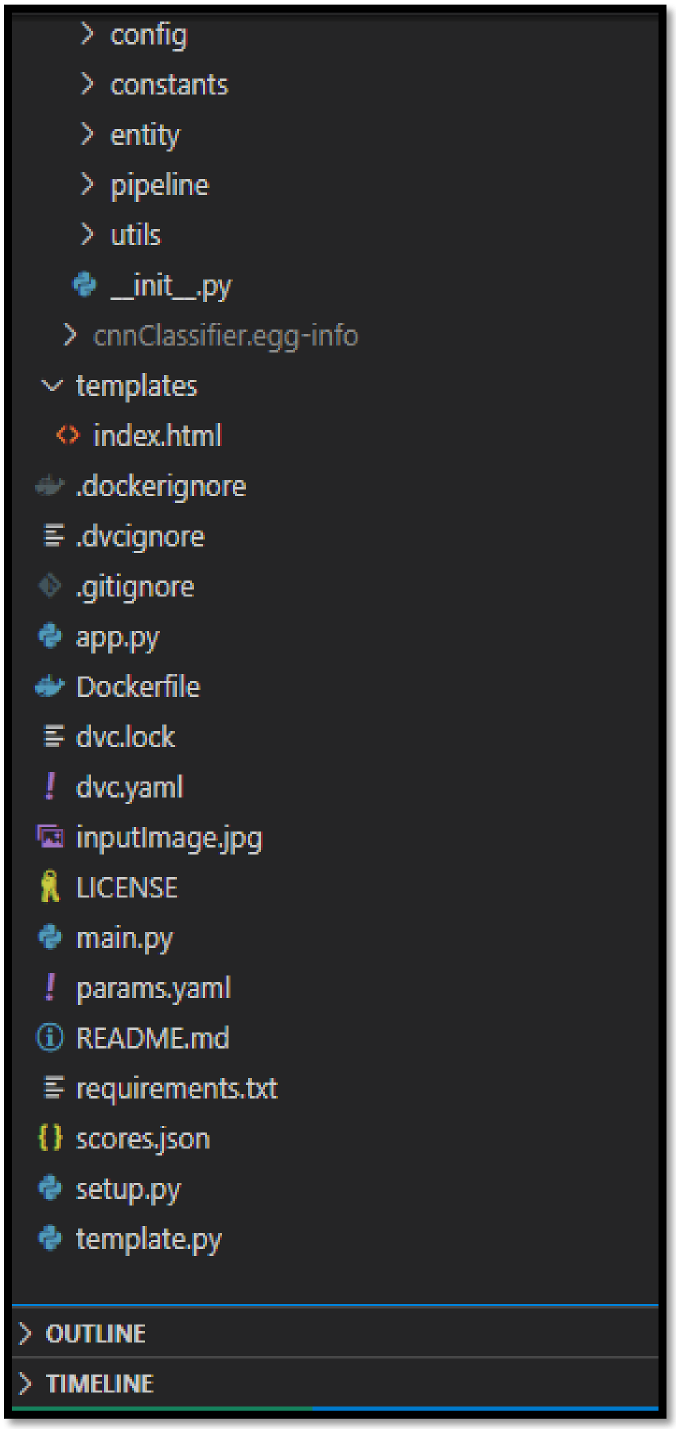
This modular and organized approach ensures that the project pipeline progresses smoothly from data ingestion to model evaluation, with comprehensive error handling to address any unexpected issues that may arise during execution.


```python
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\nx==========x")
except Exception as e:
    logger.exception(e)
    raise e

STAGE_NAME = "Prepare base model"
try:
   logger.info(f"*******************")
   logger.info(f">>>>>> stage {STAGE_NAME} started <<<<<<")
   prepare_base_model = PrepareBaseModelTrainingPipeline()
   prepare_base_model.main()
   logger.info(f">>>>>> stage {STAGE_NAME} completed <<<<<<\n\nx==========x")
except Exception as e:
        logger.exception(e)
        raise e

STAGE_NAME = "Training"
try:
   logger.info(f"*******************")
   logger.info(f">>>>>> stage {STAGE_NAME} started <<<<<<")
   model_trainer = ModelTrainingPipeline()
   model_trainer.main()
   logger.info(f">>>>>> stage {STAGE_NAME} completed <<<<<<\n\nx==========x")
except Exception as e:
        logger.exception(e)
        raise e

STAGE_NAME = "Evaluation stage"
try:
```

This is the Project Folder Structure….



```
EXPLORER ...
PROJECT [CODESPACES: GLOWING CARNIVAL]
  > .dvc
  v artifacts
    > data_ingestion
    > prepare_base_model
    > training
  > assets
  v config
    config.yaml
  > env
  > logs
  v model
    model.h5
  > prediction_service
  v research
    argv-demo.py
    Final_Notebook.ipynb
    mlflow_demo.py
    St01_data_ingestion.ipynb
    St02_prepare_base_model.ipynb
    St03_model_trainer.ipynb
    St04_model_evaluation_with_mlflow.ipynb
    test.yaml
    trials.ipynb
  v src
    v cnnClassifier
      > __pycache__
      > components
      > config
```



```
      > config
      > constants
      > entity
      > pipeline
      > utils
        __init__.py
    > cnnClassifier.egg-info
  v templates
    index.html
  .dockerignore
  .dvcignore
  .gitignore
  app.py
  Dockerfile
  dvc.lock
  dvc.yaml
  inputImage.jpg
  LICENSE
  main.py
  params.yaml
  README.md
  requirements.txt
  scores.json
  setup.py
  template.py
> OUTLINE
> TIMELINE
```

- **requirements.txt** file consists of all the packages that you need to deploy the app in the cloud.
- **app.py** is the entry point of our application, where the flask server starts.
- **research folder** contains all jupyter notebooks related to try and testing codes for different components of pipeline.

Now let's start with DVC Pipeline and its' Components………

7) Data Ingestion

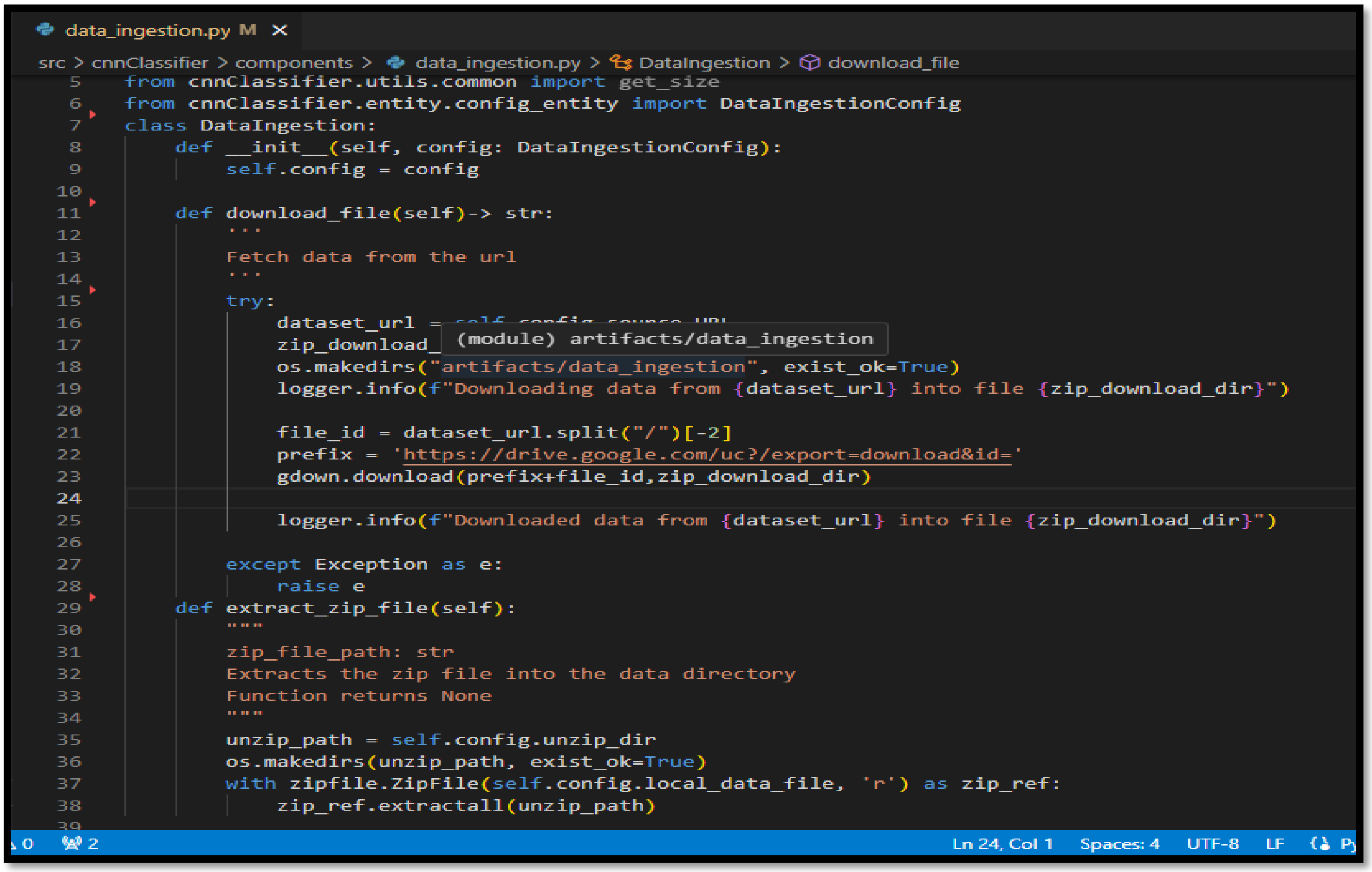
The data ingestion part of the Chest Disease Classification project involves fetching and preparing the dataset for subsequent processing. This process is divided into two main components: **data_ingestion.py** and **stage01_data_ingestion.py**.

In data_ingestion.py, the **DataIngestion** class encapsulates methods for downloading the dataset from a specified Google Drive URL and extracting it into the designated directory. The **download_file()** method fetches the dataset from the provided Google Drive URL using the **gdown** library and saves it to the local filesystem. The **extract_zip_file**() method then extracts the downloaded zip file into the designated directory, readying the data for further processing.

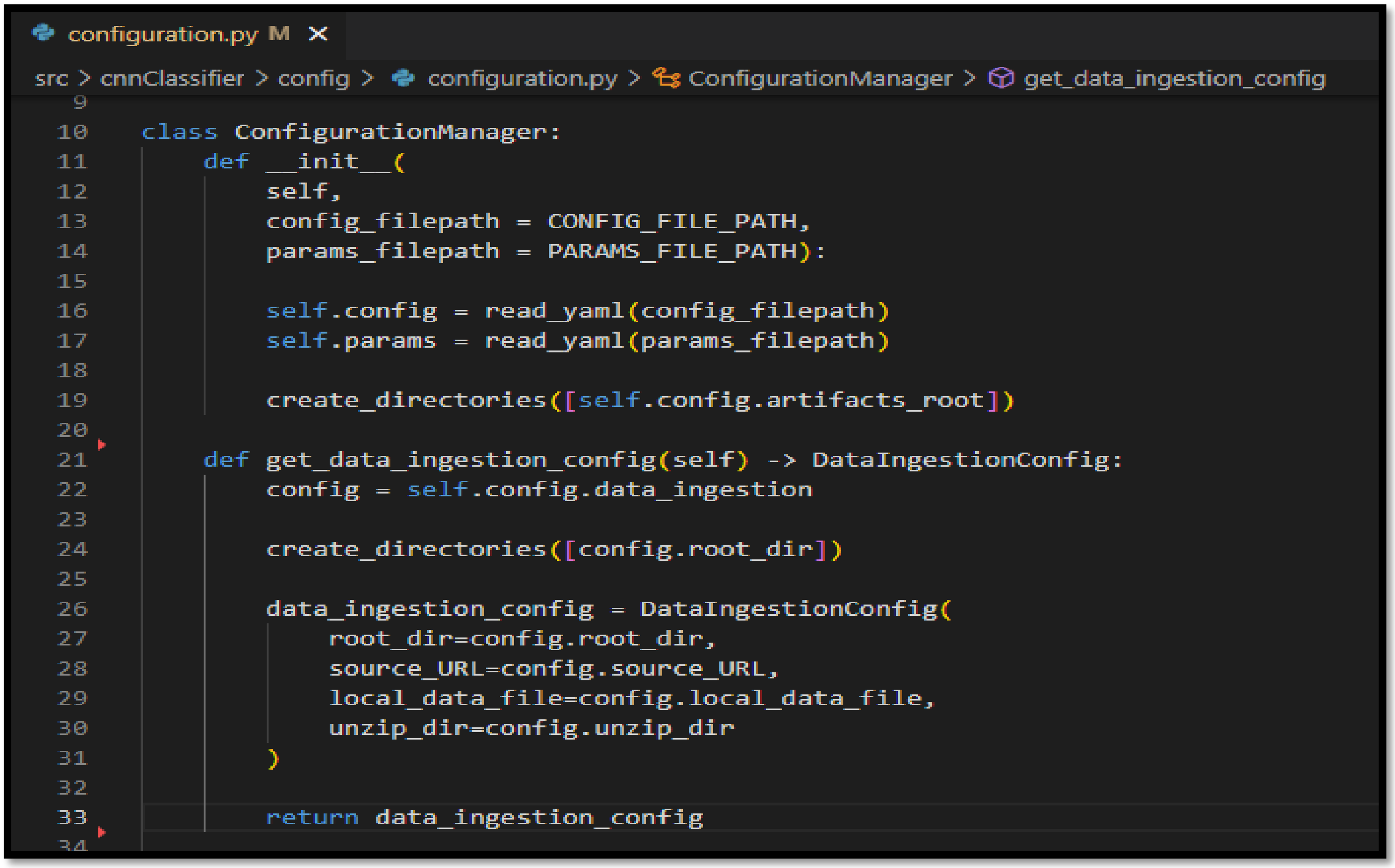
In stage01_data_ingestion.py, the **DataIngestionTrainingPipeline** class orchestrates the data ingestion process. It initializes the configuration manager to retrieve the data ingestion configuration, instantiates the **DataIngestion** class with the retrieved configuration, and executes the download and extraction methods sequentially.

The **ConfigurationManager** class acts as the cornerstone for organizing and accessing configuration settings within the Chest Disease Classification project. By centralizing configuration management, it streamlines the retrieval of essential parameters for different project components. Upon initialization, it reads YAML files containing configuration details, ensuring the existence of necessary directories for storing project **artifacts**. The **get_data_ingestion_config** method specifically retrieves data ingestion configuration settings, facilitating the setup of data retrieval and extraction processes.

Overall, the data ingestion part ensures the acquisition and preparation of the dataset, laying the foundation for subsequent stages in the project pipeline, such as model training and evaluation.


data_ingestion.py M ×
src > cnnClassifier > components > data_ingestion.py > DataIngestion > download_file
(module) artifacts/data_ingestion
Ln 24, Col 1   Spaces: 4   UTF-8   LF

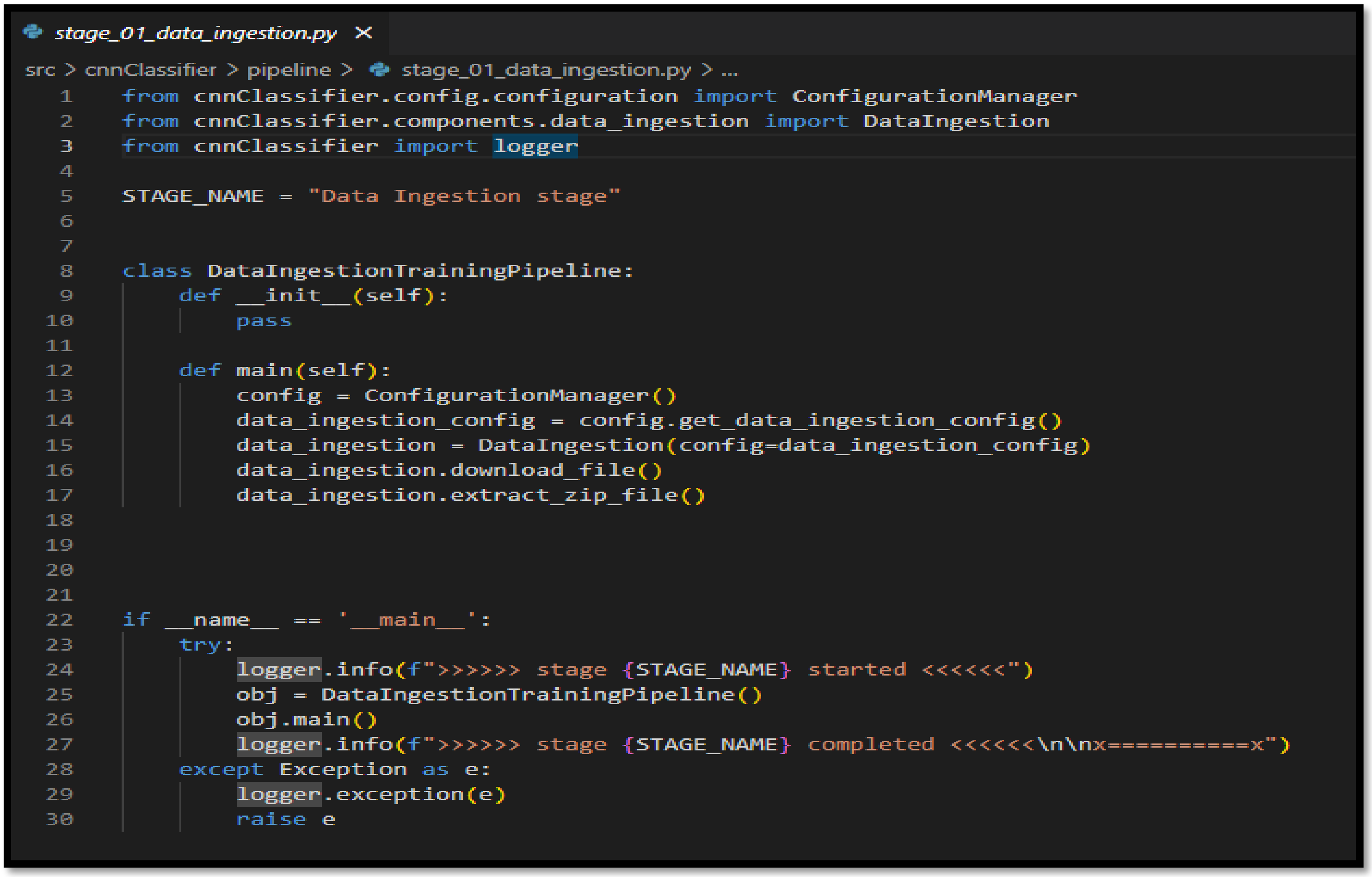
```python
from cnnClassifier.utils.common import get_size
from cnnClassifier.entity.config_entity import DataIngestionConfig
class DataIngestion:
    def __init__(self, config: DataIngestionConfig):
        self.config = config

    def download_file(self)-> str:
        '''
        Fetch data from the url
        '''
        try:
            dataset_url = self.config.source_URL
            zip_download_
            os.makedirs("artifacts/data_ingestion", exist_ok=True)
            logger.info(f"Downloading data from {dataset_url} into file {zip_download_dir}")

            file_id = dataset_url.split("/")[-2]
            prefix = 'https://drive.google.com/uc?/export=download&id='
            gdown.download(prefix+file_id,zip_download_dir)

            logger.info(f"Downloaded data from {dataset_url} into file {zip_download_dir}")

        except Exception as e:
            raise e
    def extract_zip_file(self):
        """
        zip_file_path: str
        Extracts the zip file into the data directory
        Function returns None
        """
        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)
```



configuration.py M ×

src > cnnClassifier > config > configuration.py > ConfigurationManager > get_data_ingestion_config

```python
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_URL=config.source_URL,
            local_data_file=config.local_data_file,
            unzip_dir=config.unzip_dir
        )

        return data_ingestion_config
```



stage_01_data_ingestion.py ×

src > cnnClassifier > pipeline > stage_01_data_ingestion.py > ...

```python
from cnnClassifier.config.configuration import ConfigurationManager
from cnnClassifier.components.data_ingestion import DataIngestion
from cnnClassifier 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.download_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 e
```

8) Prepare Base Model

The "**Prepare Base Model**" pipeline is a crucial component in the development of a convolutional neural network (CNN) classifier for chest disease classification from chest CT scan images. This pipeline is responsible for acquiring a pre-trained base model, typically a **ResNet50** architecture, and then customizing it to suit the specific requirements of the classification task.

The "**Prepare base model**" pipeline is a pivotal phase within the Chest Disease Classification project, designed to configure and tailor the foundational architecture of the neural network model. This pipeline, managed by the **PrepareBaseModelTrainingPipeline** class and instantiated in **stage02_prepare_base_model.py**, serves as a critical bridge between the project's initial setup and subsequent training phases.

At its core, the pipeline leverages the capabilities of **TensorFlow**, a leading deep learning framework, to instantiate a base model architecture. Specifically, the ResNet50 architecture is utilized due to its proven effectiveness in image classification tasks. The **PrepareBaseModel** class orchestrates this process, considering various configuration parameters provided through the project's configuration files.

Upon initialization, the pipeline retrieves essential configuration settings via the **ConfigurationManager**, ensuring consistency and adaptability across different project components. These settings include parameters such as image size, pre-trained weights, inclusion of top layers, and pooling strategies, all of which significantly influence the model's architecture and performance.

Following the instantiation of the base model, the pipeline further customizes it to align with the project's specific requirements and objectives. This customization involves adding additional layers, adjusting layer trainability, setting learning rates, and compiling the model with appropriate loss functions and evaluation metrics. Notably, the pipeline offers flexibility in customizing the base model's architecture, allowing for experimentation and optimization based on the unique characteristics of the dataset and classification task.

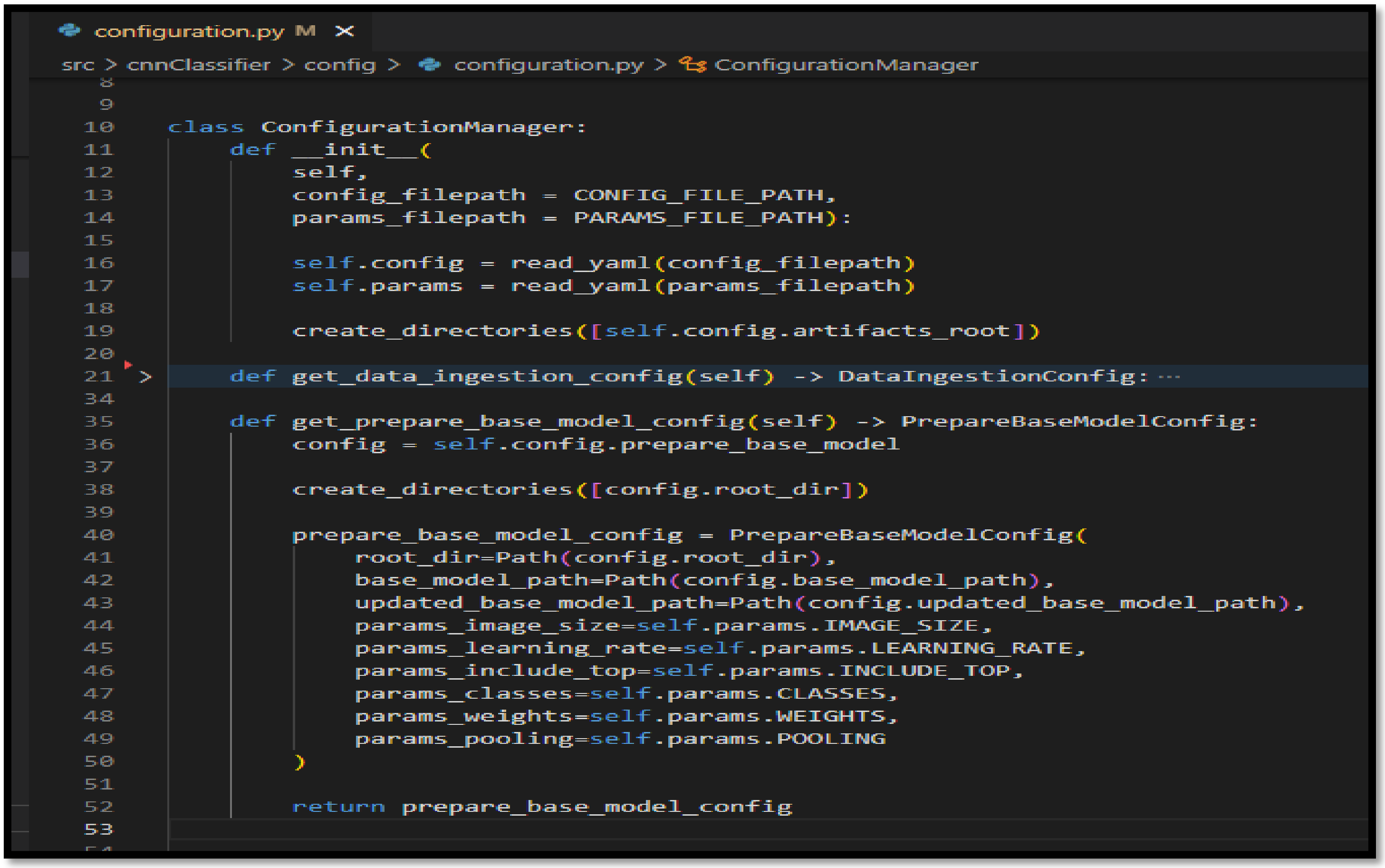
Once the base model is prepared and configured to meet the project's needs, it is saved to the designated path, ensuring its availability for subsequent stages of the project pipeline, such as model training and evaluation.

The **ConfigurationManager** class plays a pivotal role in managing and providing access to various configuration settings within the Chest Disease Classification project. It facilitates the retrieval of both general project configurations and specific configurations tailored to different stages of the project pipeline.

In particular, the **get_prepare_base_model_config** method within the **ConfigurationManager** class focuses on retrieving configuration settings specifically related to preparing the base model. This method accesses the section **prepare_base_model** within the project's configuration file and extracts relevant parameters such as the root directory for storing model **artifacts**, paths for the base model and its updated version, as well as parameters influencing the model architecture such as image size, learning rate, number of classes, weights initialization, and pooling strategy.

The retrieved configuration settings are encapsulated within a **PrepareBaseModelConfig** object, ensuring structured and organized access to these parameters throughout the base model preparation process. Additionally, the method ensures the creation of necessary directories specified in the configuration, fostering an environment conducive to seamless model preparation and subsequent training tasks.

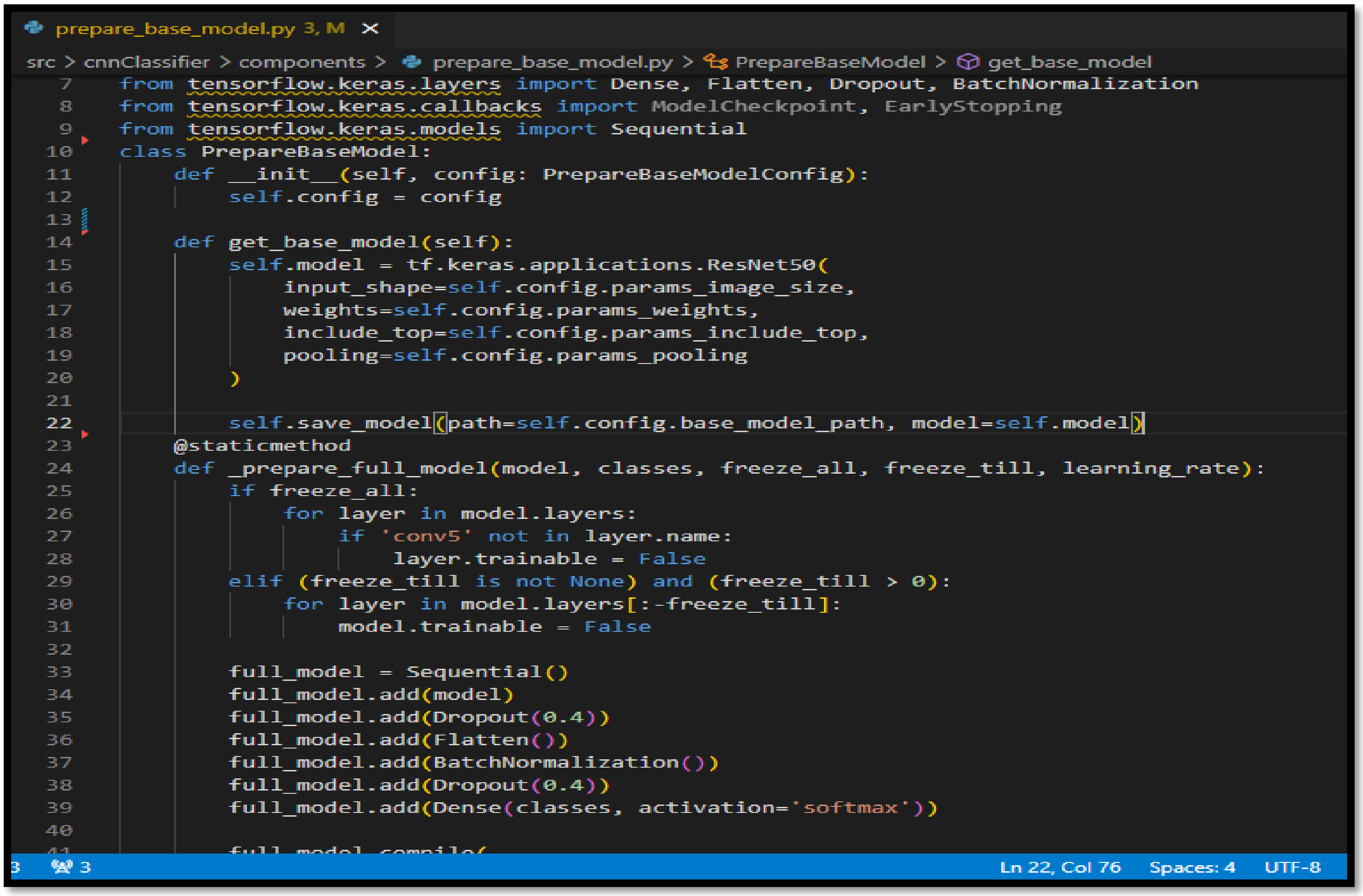
Overall, the "**Prepare Base Model**" pipeline streamlines the process of acquiring, customizing, and saving a pre-trained base model, setting the stage for subsequent stages of the classification pipeline, such as fine-tuning and evaluation. It ensures consistency and efficiency in model preparation, laying a solid foundation for the development of an accurate and robust chest disease classification system.



configuration.py M ×

src > cnnClassifier > config > configuration.py > ConfigurationManager

```python

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: ···

    def get_prepare_base_model_config(self) -> PrepareBaseModelConfig:
        config = self.config.prepare_base_model

        create_directories([config.root_dir])

        prepare_base_model_config = PrepareBaseModelConfig(
            root_dir=Path(config.root_dir),
            base_model_path=Path(config.base_model_path),
            updated_base_model_path=Path(config.updated_base_model_path),
            params_image_size=self.params.IMAGE_SIZE,
            params_learning_rate=self.params.LEARNING_RATE,
            params_include_top=self.params.INCLUDE_TOP,
            params_classes=self.params.CLASSES,
            params_weights=self.params.WEIGHTS,
            params_pooling=self.params.POOLING
        )

        return prepare_base_model_config

```

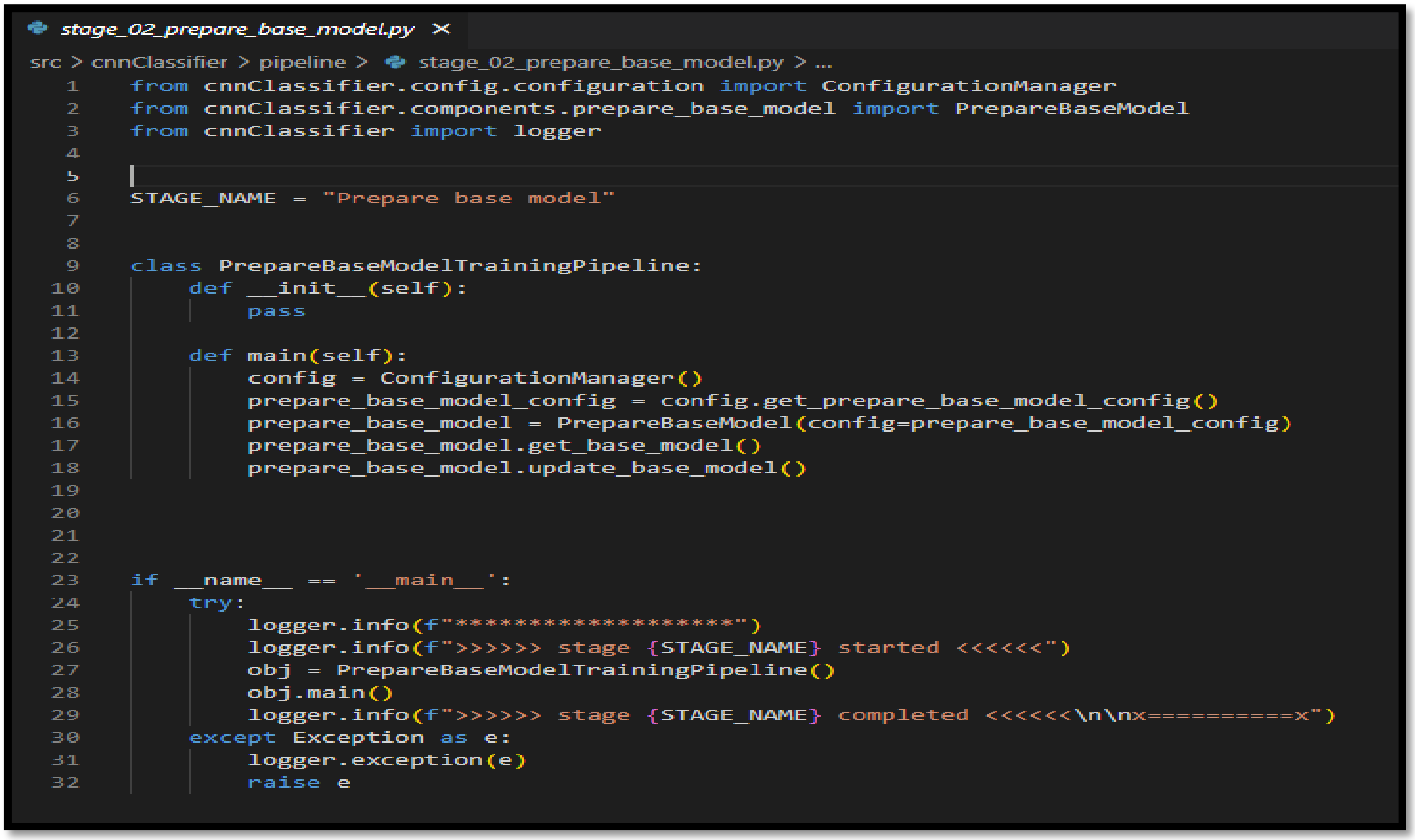


prepare_base_model.py 3, M ×

src > cnnClassifier > components > prepare_base_model.py > PrepareBaseModel > get_base_model

```python
from tensorflow.keras.layers import Dense, Flatten, Dropout, BatchNormalization
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.models import Sequential
class PrepareBaseModel:
    def __init__(self, config: PrepareBaseModelConfig):
        self.config = config

    def get_base_model(self):
        self.model = tf.keras.applications.ResNet50(
            input_shape=self.config.params_image_size,
            weights=self.config.params_weights,
            include_top=self.config.params_include_top,
            pooling=self.config.params_pooling
        )

        self.save_model(path=self.config.base_model_path, model=self.model)
    @staticmethod
    def _prepare_full_model(model, classes, freeze_all, freeze_till, learning_rate):
        if freeze_all:
            for layer in model.layers:
                if 'conv5' not in layer.name:
                    layer.trainable = False
        elif (freeze_till is not None) and (freeze_till > 0):
            for layer in model.layers[:-freeze_till]:
                model.trainable = False

        full_model = Sequential()
        full_model.add(model)
        full_model.add(Dropout(0.4))
        full_model.add(Flatten())
        full_model.add(BatchNormalization())
        full_model.add(Dropout(0.4))
        full_model.add(Dense(classes, activation='softmax'))

```

3 3 Ln 22, Col 76 Spaces: 4 UTF-8


stage_02_prepare_base_model.py
src > cnnClassifier > pipeline > stage_02_prepare_base_model.py > ...

```python
from cnnClassifier.config.configuration import ConfigurationManager
from cnnClassifier.components.prepare_base_model import PrepareBaseModel
from cnnClassifier import logger


STAGE_NAME = "Prepare base model"


class PrepareBaseModelTrainingPipeline:
    def __init__(self):
        pass

    def main(self):
        config = ConfigurationManager()
        prepare_base_model_config = config.get_prepare_base_model_config()
        prepare_base_model = PrepareBaseModel(config=prepare_base_model_config)
        prepare_base_model.get_base_model()
        prepare_base_model.update_base_model()




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

9) Model Trainer

The "**Model Trainer**" pipeline, orchestrated by the **ModelTrainingPipeline** class and defined in stage03_model_trainer.py, is a pivotal stage within the Chest Disease Classification project. This pipeline oversees the training of the neural network model using the prepared base model architecture, fine-tuning it to accurately classify chest CT scan images into distinct categories.

At its core, the pipeline relies on **TensorFlow**, a powerful deep learning framework, to facilitate model training. The Training class, responsible for managing the training process, initializes by retrieving essential configuration settings via the **ConfigurationManager**. These settings encompass various parameters crucial for training, including the paths to training and validation datasets, batch size, augmentation settings, and image dimensions.

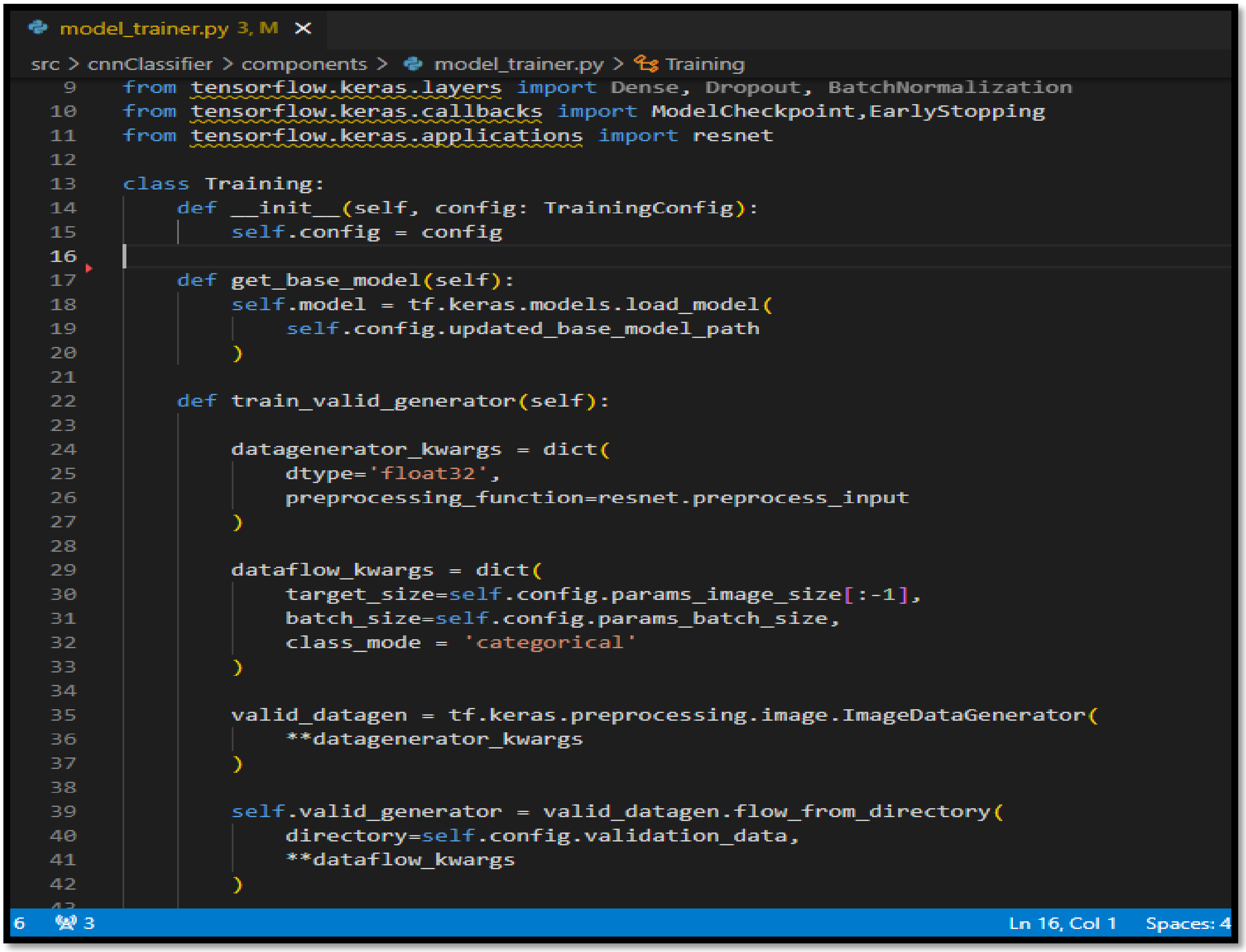
Once initialized, the pipeline proceeds to instantiate the base model architecture previously prepared in the "**Prepare Base Model**" stage. Leveraging the updated base model, the **train_valid_generator** method sets up data generators for both training and validation datasets, ensuring efficient data flow and preprocessing. Depending on configuration settings, data augmentation techniques such as rotation, flipping, and shifting may be applied to augment the training dataset, enhancing model generalization.

The pipeline begins by initializing the necessary configurations, including hyperparameters such as the number of epochs, batch size, and image size, along with file paths for training and validation data. The pipeline prepares the training and validation data using data generators provided by TensorFlow's **ImageDataGenerator** class. Data augmentation techniques such as rotation, flipping, and shifting may be applied to augment the training dataset, thereby enhancing the model's ability to generalize.

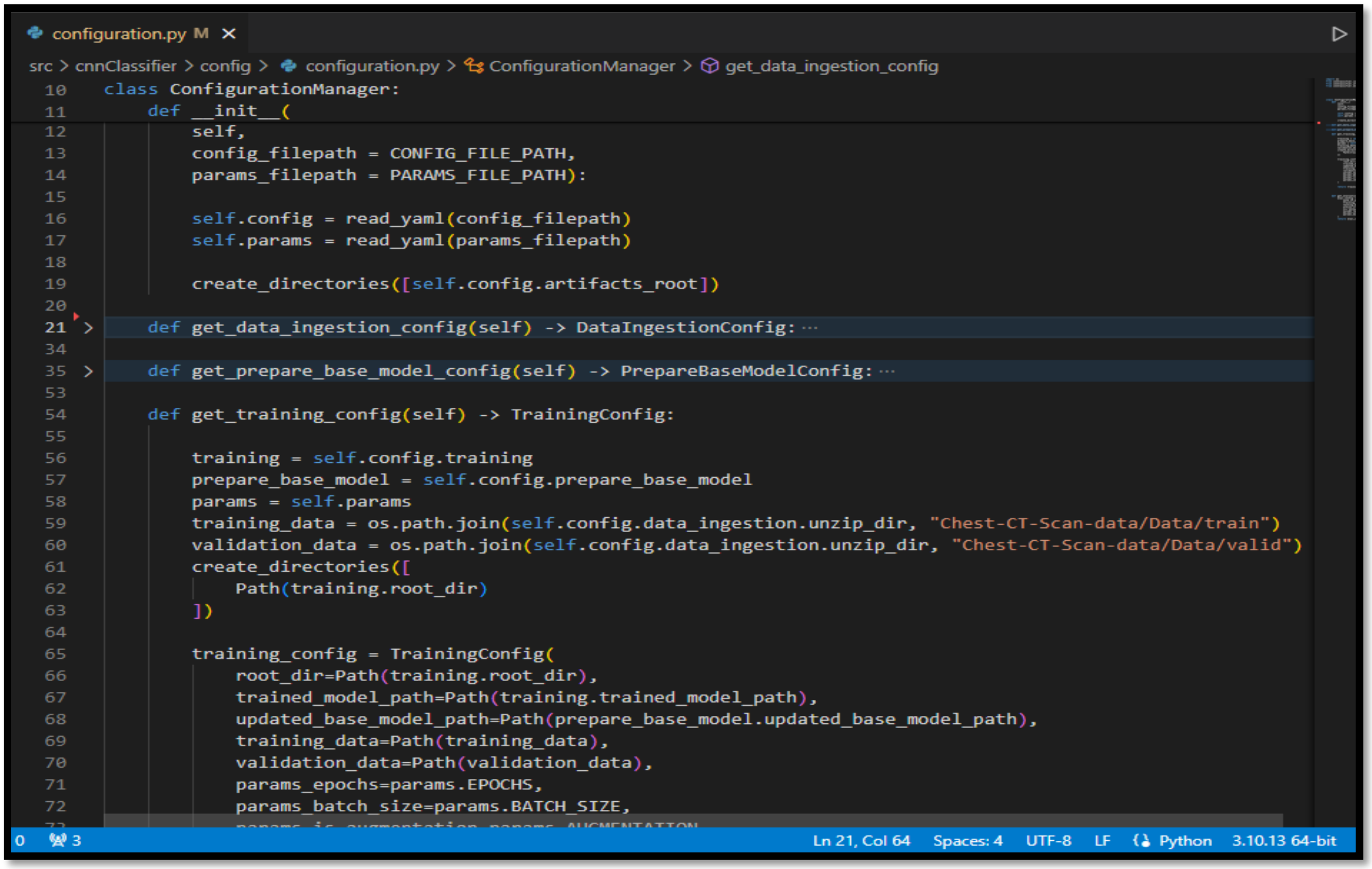
Subsequently, model training commences utilizing the train method, which orchestrates the training process with extensive monitoring and checkpoints. Key **callbacks** such as **ModelCheckpoint** and **EarlyStopping** are employed to save the best-performing model weights and prevent overfitting. The model undergoes training over multiple epochs, iteratively optimizing its parameters to minimize loss and maximize accuracy on the validation dataset.

The **get_training_config** method in the **ConfigurationManager** class is pivotal for acquiring configuration settings specific to the model training phase in the Chest Disease Classification project. It retrieves essential parameters such as paths to training and validation datasets, the location of the updated base model, and specifications like epochs, batch size, and image dimensions. These settings are then structured into a **TrainingConfig** object, ensuring organized access throughout the training process and enabling seamless integration with other project components.

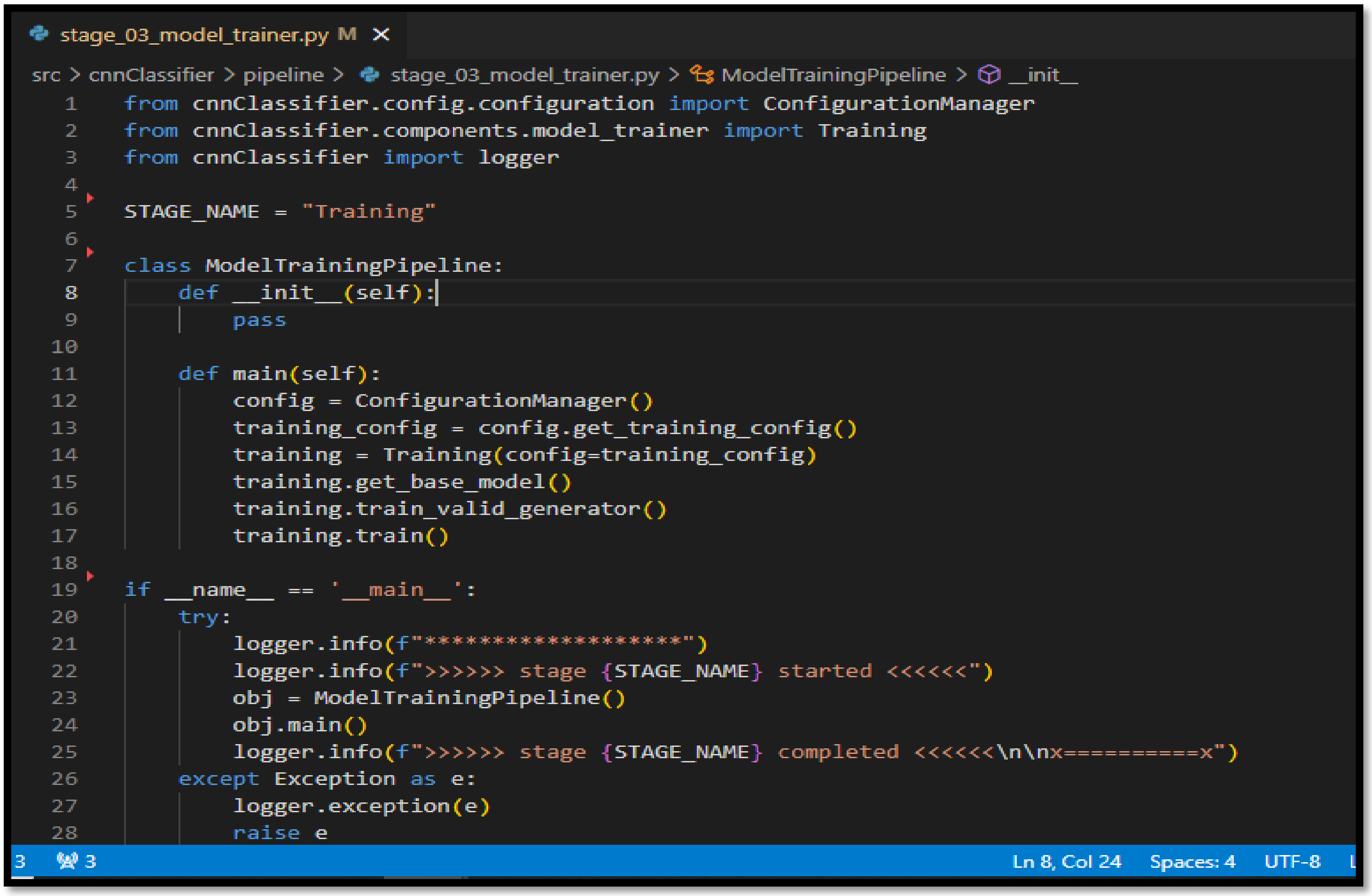
Upon completion of training, the trained model is saved to the designated path, ensuring its preservation for subsequent evaluation and deployment stages. By effectively training the model on the provided dataset, the "**Model Trainer**" pipeline lays the groundwork for the development of a robust and accurate classifier capable of identifying chest diseases from CT scan images. By automating key tasks such as data preparation, model training, and result logging, this pipeline accelerates the development cycle and enables researchers and practitioners to focus on refining the model's performance and addressing real-world challenges in healthcare diagnostics.



```
model_trainer.py 3, M ×
src > cnnClassifier > components > model_trainer.py > Training
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.callbacks import ModelCheckpoint,EarlyStopping
from tensorflow.keras.applications import resnet

class Training:
    def __init__(self, config: TrainingConfig):
        self.config = config

    def get_base_model(self):
        self.model = tf.keras.models.load_model(
            self.config.updated_base_model_path
        )

    def train_valid_generator(self):

        datagenerator_kwargs = dict(
            dtype='float32',
            preprocessing_function=resnet.preprocess_input
        )

        dataflow_kwargs = dict(
            target_size=self.config.params_image_size[:-1],
            batch_size=self.config.params_batch_size,
            class_mode = 'categorical'
        )

        valid_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
            **datagenerator_kwargs
        )

        self.valid_generator = valid_datagen.flow_from_directory(
            directory=self.config.validation_data,
            **dataflow_kwargs
        )
```


configuration.py M
src > cnnClassifier > config > configuration.py > ConfigurationManager > get_data_ingestion_config
Ln 21, Col 64 Spaces: 4 UTF-8 LF Python 3.10.13 64-bit

```python
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: ···

    def get_prepare_base_model_config(self) -> PrepareBaseModelConfig: ···

    def get_training_config(self) -> TrainingConfig:

        training = self.config.training
        prepare_base_model = self.config.prepare_base_model
        params = self.params
        training_data = os.path.join(self.config.data_ingestion.unzip_dir, "Chest-CT-Scan-data/Data/train")
        validation_data = os.path.join(self.config.data_ingestion.unzip_dir, "Chest-CT-Scan-data/Data/valid")
        create_directories([
            Path(training.root_dir)
        ])

        training_config = TrainingConfig(
            root_dir=Path(training.root_dir),
            trained_model_path=Path(training.trained_model_path),
            updated_base_model_path=Path(prepare_base_model.updated_base_model_path),
            training_data=Path(training_data),
            validation_data=Path(validation_data),
            params_epochs=params.EPOCHS,
            params_batch_size=params.BATCH_SIZE,
            params_is_augmentation=params.AUGMENTATION
```


stage_03_model_trainer.py M
src > cnnClassifier > pipeline > stage_03_model_trainer.py > ModelTrainingPipeline > __init__
Ln 8, Col 24 Spaces: 4 UTF-8

```python
from cnnClassifier.config.configuration import ConfigurationManager
from cnnClassifier.components.model_trainer import Training
from cnnClassifier import logger

STAGE_NAME = "Training"

class ModelTrainingPipeline:
    def __init__(self):
        pass

    def main(self):
        config = ConfigurationManager()
        training_config = config.get_training_config()
        training = Training(config=training_config)
        training.get_base_model()
        training.train_valid_generator()
        training.train()

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

10) Model Evaluation

The “**Model Evaluation**” pipeline represents the culmination of efforts in the Chest Disease Classification project, where the efficacy of the trained deep learning model is rigorously assessed to ensure its reliability in real-world scenarios. At the heart of this pipeline lies the Evaluation class, meticulously designed to orchestrate the intricate evaluation process with precision and accuracy.

Upon initialization, the Evaluation class seamlessly loads the trained model from the specified path, leveraging TensorFlow's powerful model loading capabilities. Subsequently, it meticulously sets up the test data generator, a crucial step in preparing the unseen data for evaluation. Through meticulous preprocessing and configuration, the data generator ensures that the test data aligns seamlessly with the model's input requirements, facilitating robust evaluation.

Once the groundwork is laid, the evaluation process commences, with the model being put to the test against the unseen test data. Leveraging TensorFlow's comprehensive evaluation functionalities, the model's performance is meticulously scrutinized across various metrics, including loss and accuracy. This comprehensive evaluation provides valuable insights into the model's strengths and weaknesses, shedding light on areas for improvement and optimization.

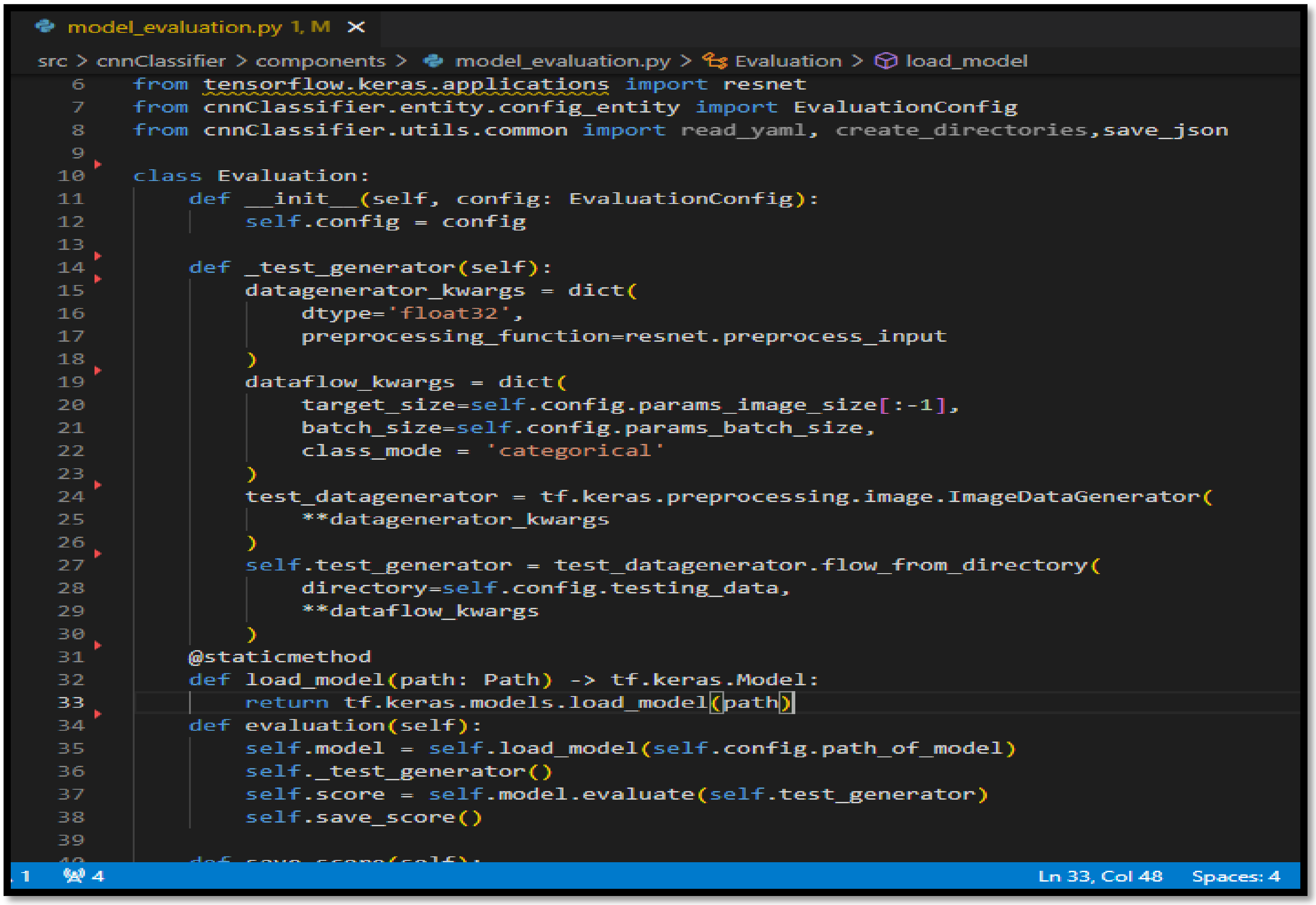
Next, the pipeline evaluates the model's performance using the evaluate method provided by TensorFlow. This method computes key metrics such as loss and accuracy by comparing the model's predictions against ground truth labels from the test dataset. By quantifying the model's performance in terms of these metrics, the pipeline provides valuable insights into its predictive capabilities and generalization to unseen data. As the evaluation unfolds, the Evaluation class diligently saves the evaluation scores into a structured **JSON file**, ensuring that crucial performance metrics are securely archived for future reference and analysis. This meticulous record-keeping not only facilitates post-hoc analysis but also serves as a benchmark for comparison in subsequent experiments and iterations.

In parallel, the pipeline seamlessly integrates with **MLflow**, a versatile platform for tracking and managing machine learning experiments. Leveraging MLflow's robust capabilities, the pipeline logs crucial evaluation metrics, including loss and accuracy, into a centralized repository, enabling stakeholders to monitor model performance in real-time and make data-driven decisions.

Moreover, the pipeline goes a step further by potentially registering the trained model in a model registry within MLflow. By systematically cataloging model versions and their associated metadata, the registry streamlines model management and deployment processes, ensuring reproducibility and scalability across diverse environments.

At the helm of the evaluation pipeline is the **EvaluationPipeline** class, serving as the conductor orchestrating the entire evaluation symphony. Through seamless coordination and meticulous execution, the pipeline navigates through each evaluation stage with finesse, ensuring that every step contributes to the overarching goal of validating the model's efficacy and robustness.

In essence, the model evaluation pipeline represents the culmination of rigorous experimentation and validation, where the trained model's performance is scrutinized, evaluated, and documented with meticulous care. Through comprehensive evaluation and metric logging, the pipeline empowers stakeholders to make informed decisions, driving continuous improvement and innovation in the quest for superior model performance and reliability. By automating key tasks such as data preparation, model evaluation, and result logging, this pipeline empowers researchers and practitioners to iteratively improve the model and make informed decisions regarding its deployment and optimization.

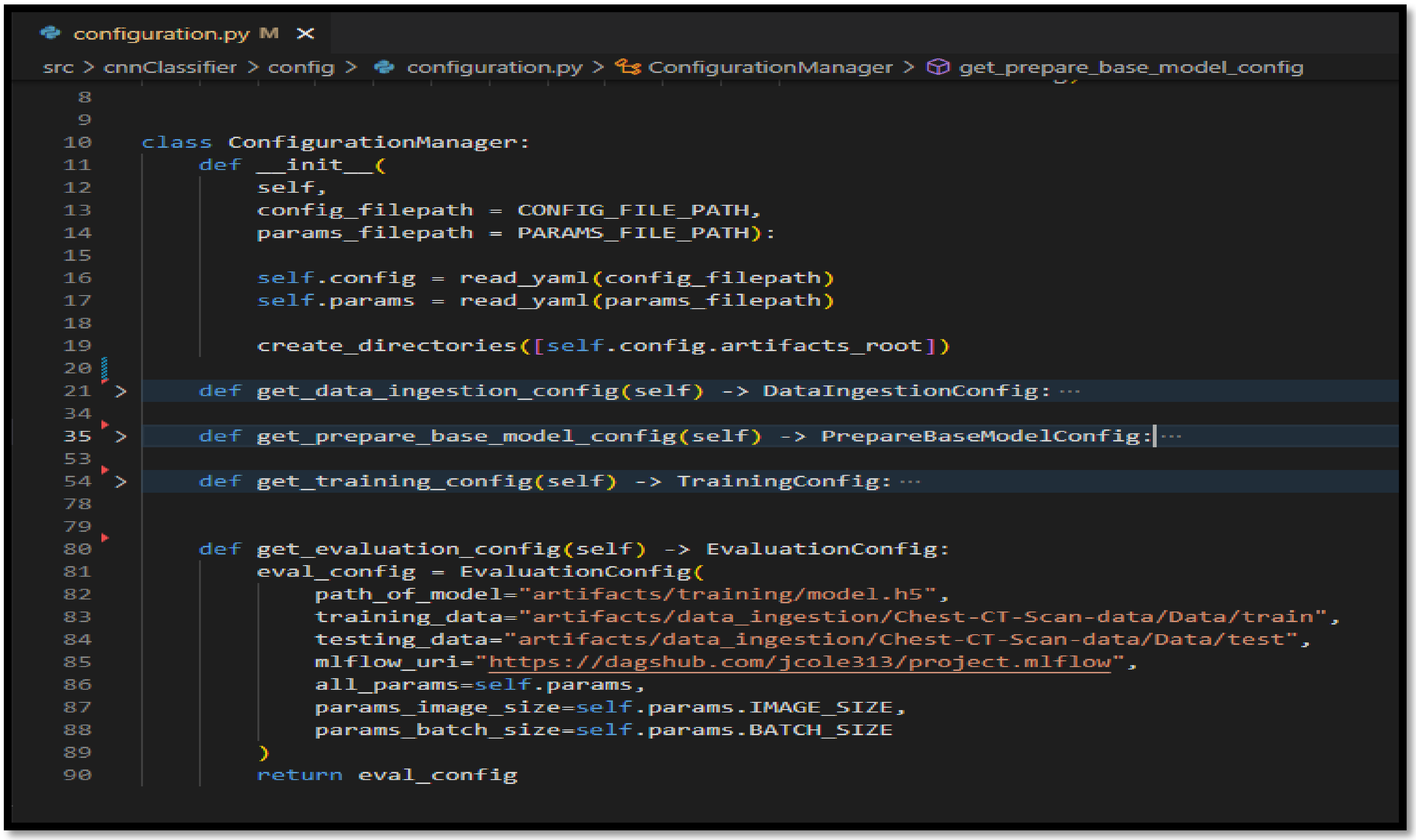


model_evaluation.py 1, M ×

src > cnnClassifier > components > model_evaluation.py > Evaluation > load_model

```python
from tensorflow.keras.applications import resnet
from cnnClassifier.entity.config_entity import EvaluationConfig
from cnnClassifier.utils.common import read_yaml, create_directories,save_json

class Evaluation:
    def __init__(self, config: EvaluationConfig):
        self.config = config

    def _test_generator(self):
        datagenerator_kwargs = dict(
            dtype='float32',
            preprocessing_function=resnet.preprocess_input
        )
        dataflow_kwargs = dict(
            target_size=self.config.params_image_size[:-1],
            batch_size=self.config.params_batch_size,
            class_mode = 'categorical'
        )
        test_datagenerator = tf.keras.preprocessing.image.ImageDataGenerator(
            **datagenerator_kwargs
        )
        self.test_generator = test_datagenerator.flow_from_directory(
            directory=self.config.testing_data,
            **dataflow_kwargs
        )
    @staticmethod
    def load_model(path: Path) -> tf.keras.Model:
        return tf.keras.models.load_model(path)
    def evaluation(self):
        self.model = self.load_model(self.config.path_of_model)
        self._test_generator()
        self.score = self.model.evaluate(self.test_generator)
        self.save_score()

```

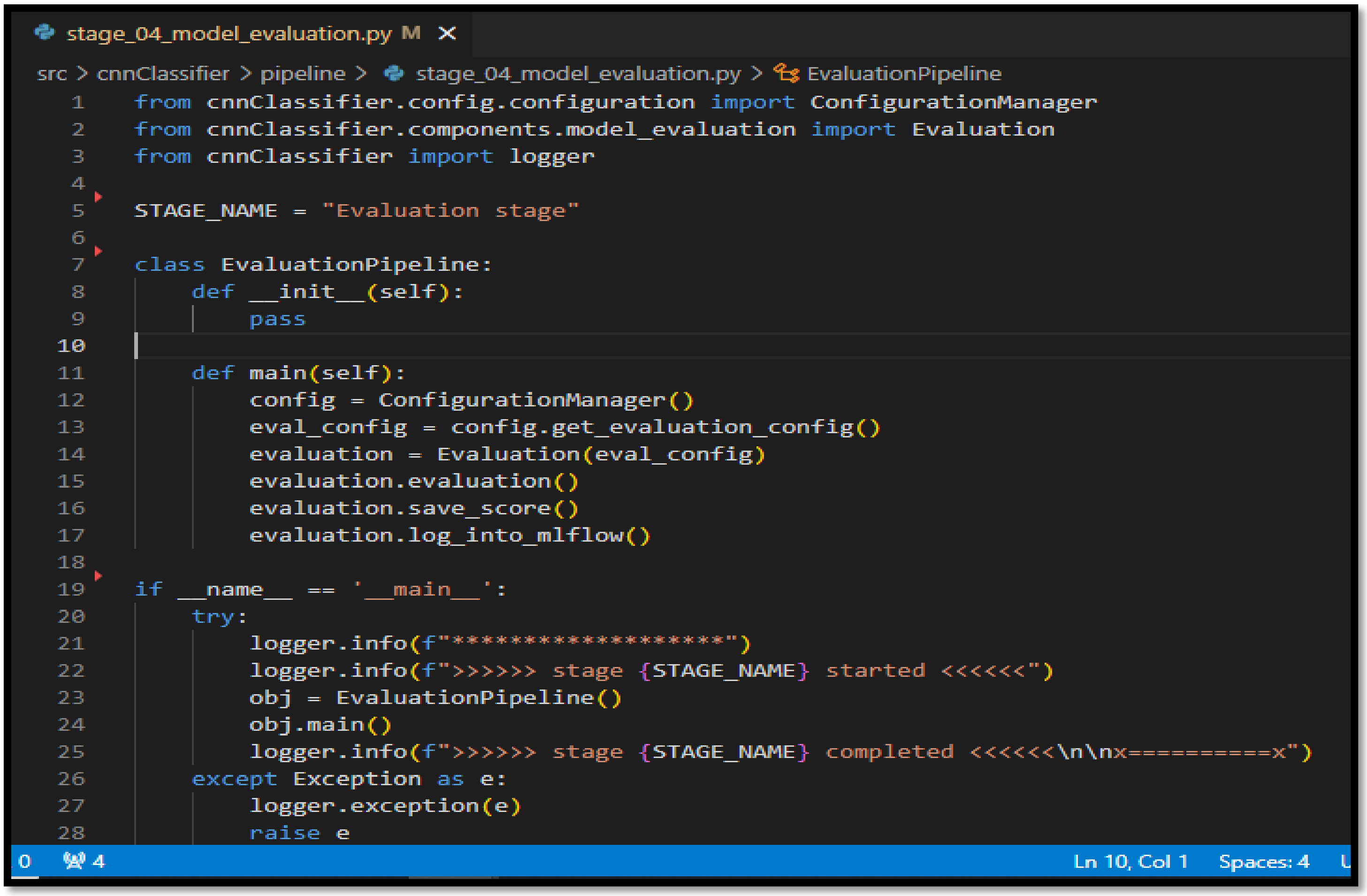
Ln 33, Col 48   Spaces: 4



configuration.py M ×

src > cnnClassifier > config > configuration.py > ConfigurationManager > get_prepare_base_model_config

```python


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: ...

    def get_prepare_base_model_config(self) -> PrepareBaseModelConfig: ...

    def get_training_config(self) -> TrainingConfig: ...


    def get_evaluation_config(self) -> EvaluationConfig:
        eval_config = EvaluationConfig(
            path_of_model="artifacts/training/model.h5",
            training_data="artifacts/data_ingestion/Chest-CT-Scan-data/Data/train",
            testing_data="artifacts/data_ingestion/Chest-CT-Scan-data/Data/test",
            mlflow_uri="https://dagshub.com/jcole313/project.mlflow",
            all_params=self.params,
            params_image_size=self.params.IMAGE_SIZE,
            params_batch_size=self.params.BATCH_SIZE
        )
        return eval_config
```


stage_04_model_evaluation.py M ✕

src > cnnClassifier > pipeline > stage_04_model_evaluation.py > EvaluationPipeline

```python
from cnnClassifier.config.configuration import ConfigurationManager
from cnnClassifier.components.model_evaluation import Evaluation
from cnnClassifier import logger

STAGE_NAME = "Evaluation stage"

class EvaluationPipeline:
    def __init__(self):
        pass

    def main(self):
        config = ConfigurationManager()
        eval_config = config.get_evaluation_config()
        evaluation = Evaluation(eval_config)
        evaluation.evaluation()
        evaluation.save_score()
        evaluation.log_into_mlflow()

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

Ln 10, Col 1 Spaces: 4

11) <u>Prediction Pipeline</u>

The Prediction pipeline is a critical component of the Chest Disease Classification project, designed to provide real-time predictions for chest X-ray images using a pre-trained deep learning model. At its core is the PredictionPipeline class, meticulously engineered to facilitate seamless inference and classification of chest X-ray images with precision and efficiency.

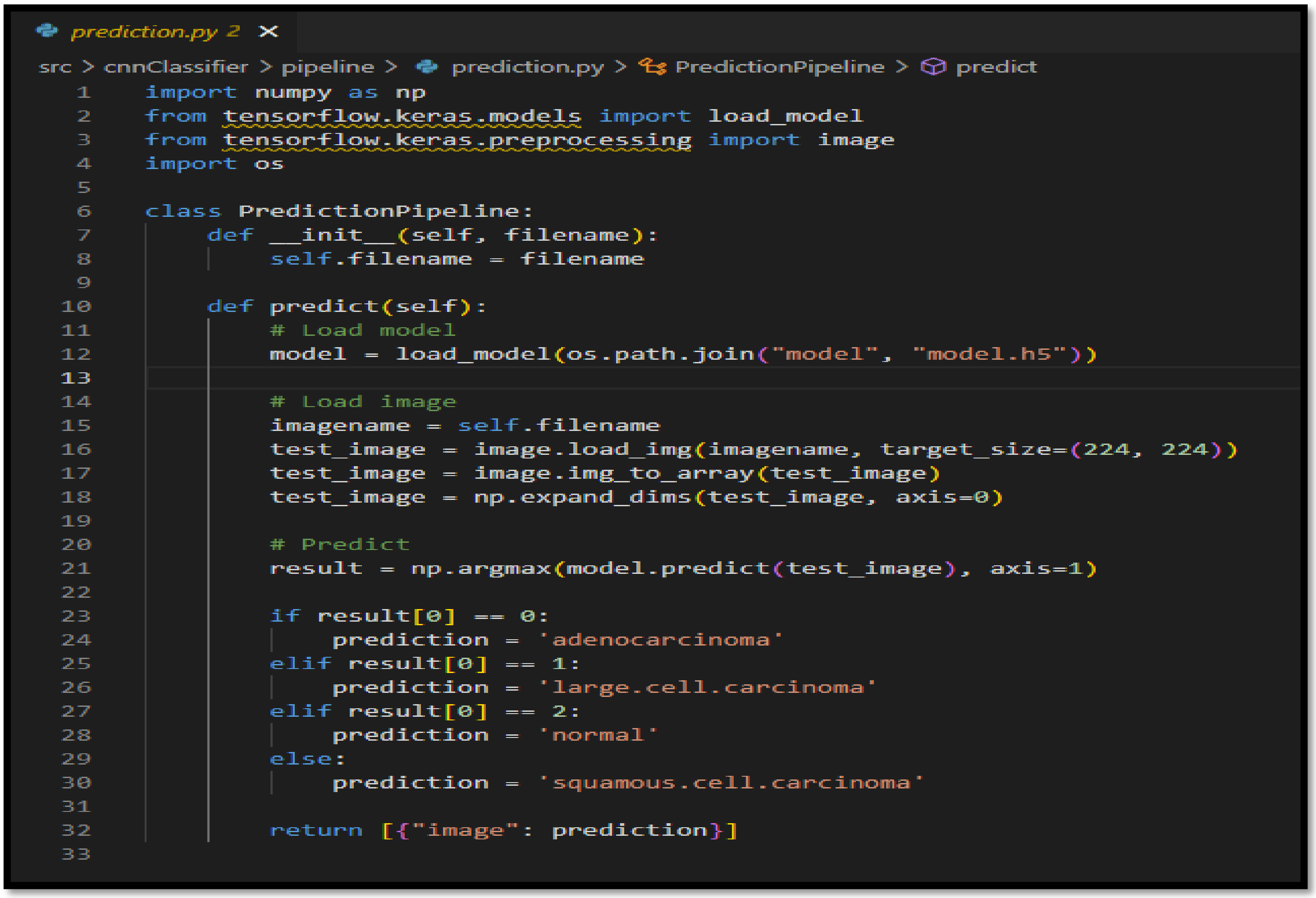
Upon initialization, the PredictionPipeline class accepts the filename of the chest X-ray image to be predicted. Leveraging TensorFlow's powerful model loading capabilities, the pipeline loads the pre-trained deep learning model from the designated directory. This model, typically a convolutional neural network (CNN), has been trained on a diverse dataset of chest X-ray images and has learned to recognize patterns indicative of different chest diseases.

Once the model is loaded, the pipeline proceeds to load the specified image using TensorFlow's image preprocessing utilities. The image is resized to match the input dimensions expected by the model, ensuring compatibility and consistency during prediction. Subsequently, the image is converted into an array format and expanded to accommodate the model's batch processing requirements.

With the image prepared and the model primed for inference, the pipeline executes the prediction process. Leveraging the model's predictive capabilities, the pipeline predicts the class label corresponding to the input chest X-ray image. The prediction is typically performed using the predict method of the loaded model, which returns the probability distribution over the possible classes.

Post-prediction, the pipeline maps the predicted class label to a human-readable format, providing meaningful interpretations of the prediction results. This mapping ensures that the prediction output is not only accurate but also easily interpretable by end-users and stakeholders. In this specific case, the predicted class labels correspond to different types of chest diseases, including adenocarcinoma, large cell carcinoma, normal, and squamous cell carcinoma.Finally, the prediction result is encapsulated within a structured format, typically a JSON object, and returned to the calling function or application. This structured output facilitates seamless integration with downstream applications, enabling further analysis, visualization, and decision-making based on the prediction results.

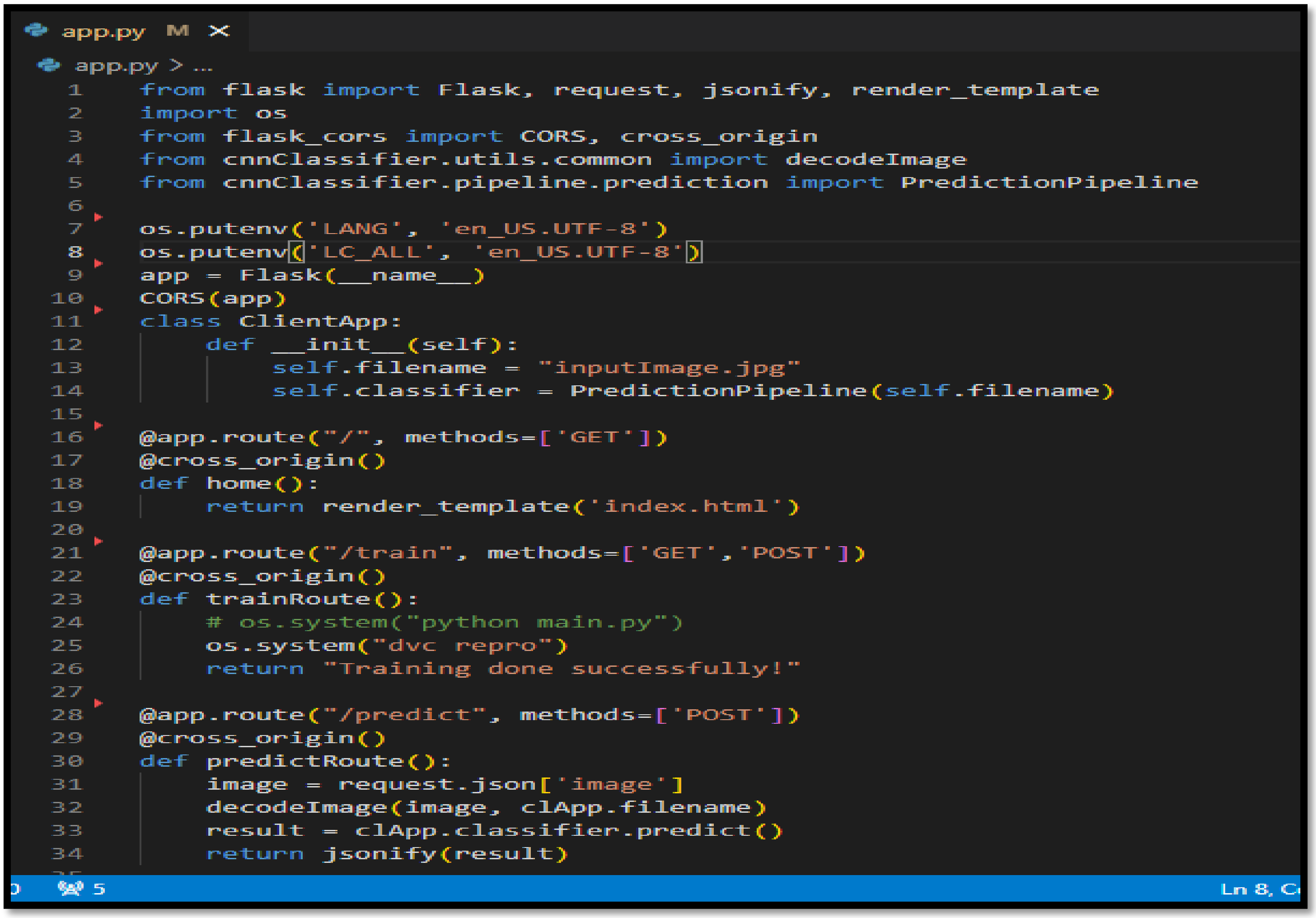
In summary, the Prediction pipeline plays a pivotal role in the Chest Disease Classification project, enabling real-time inference and classification of chest X-ray images with accuracy and efficiency. Through meticulous preprocessing, inference, and result interpretation, the pipeline empowers stakeholders to derive actionable insights from chest X-ray images, facilitating early detection and intervention in chest diseases.


prediction.py 2
src > cnnClassifier > pipeline > prediction.py > PredictionPipeline > predict

```python
import numpy as np
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import os

class PredictionPipeline:
    def __init__(self, filename):
        self.filename = filename

    def predict(self):
        # Load model
        model = load_model(os.path.join("model", "model.h5"))

        # Load image
        imagename = self.filename
        test_image = image.load_img(imagename, target_size=(224, 224))
        test_image = image.img_to_array(test_image)
        test_image = np.expand_dims(test_image, axis=0)

        # Predict
        result = np.argmax(model.predict(test_image), axis=1)

        if result[0] == 0:
            prediction = 'adenocarcinoma'
        elif result[0] == 1:
            prediction = 'large.cell.carcinoma'
        elif result[0] == 2:
            prediction = 'normal'
        else:
            prediction = 'squamous.cell.carcinoma'

        return [{"image": prediction}]

```

12) <u>app.py</u>

The app.py file serves as the core component of the Flask-based web application, facilitating interactions between the user interface and the underlying prediction pipeline. It initializes a Flask application, configures CORS (Cross-Origin Resource Sharing) for handling requests from different origins, and defines routes for various functionalities. These routes include handling the home page, triggering model training, and performing image predictions. Through these routes, users can interact with the application, upload images for inference, and receive prediction results in a JSON format. Additionally, the application ensures seamless integration with the prediction pipeline, enabling efficient deployment and utilization of machine learning models for real-time predictions.



```python
from flask import Flask, request, jsonify, render_template
import os
from flask_cors import CORS, cross_origin
from cnnClassifier.utils.common import decodeImage
from cnnClassifier.pipeline.prediction import PredictionPipeline

os.putenv('LANG', 'en_US.UTF-8')
os.putenv('LC_ALL', 'en_US.UTF-8')
app = Flask(__name__)
CORS(app)
class ClientApp:
    def __init__(self):
        self.filename = "inputImage.jpg"
        self.classifier = PredictionPipeline(self.filename)

@app.route("/", methods=['GET'])
@cross_origin()
def home():
    return render_template('index.html')

@app.route("/train", methods=['GET','POST'])
@cross_origin()
def trainRoute():
    # os.system("python main.py")
    os.system("dvc repro")
    return "Training done successfully!"

@app.route("/predict", methods=['POST'])
@cross_origin()
def predictRoute():
    image = request.json['image']
    decodeImage(image, clApp.filename)
    result = clApp.classifier.predict()
    return jsonify(result)
```

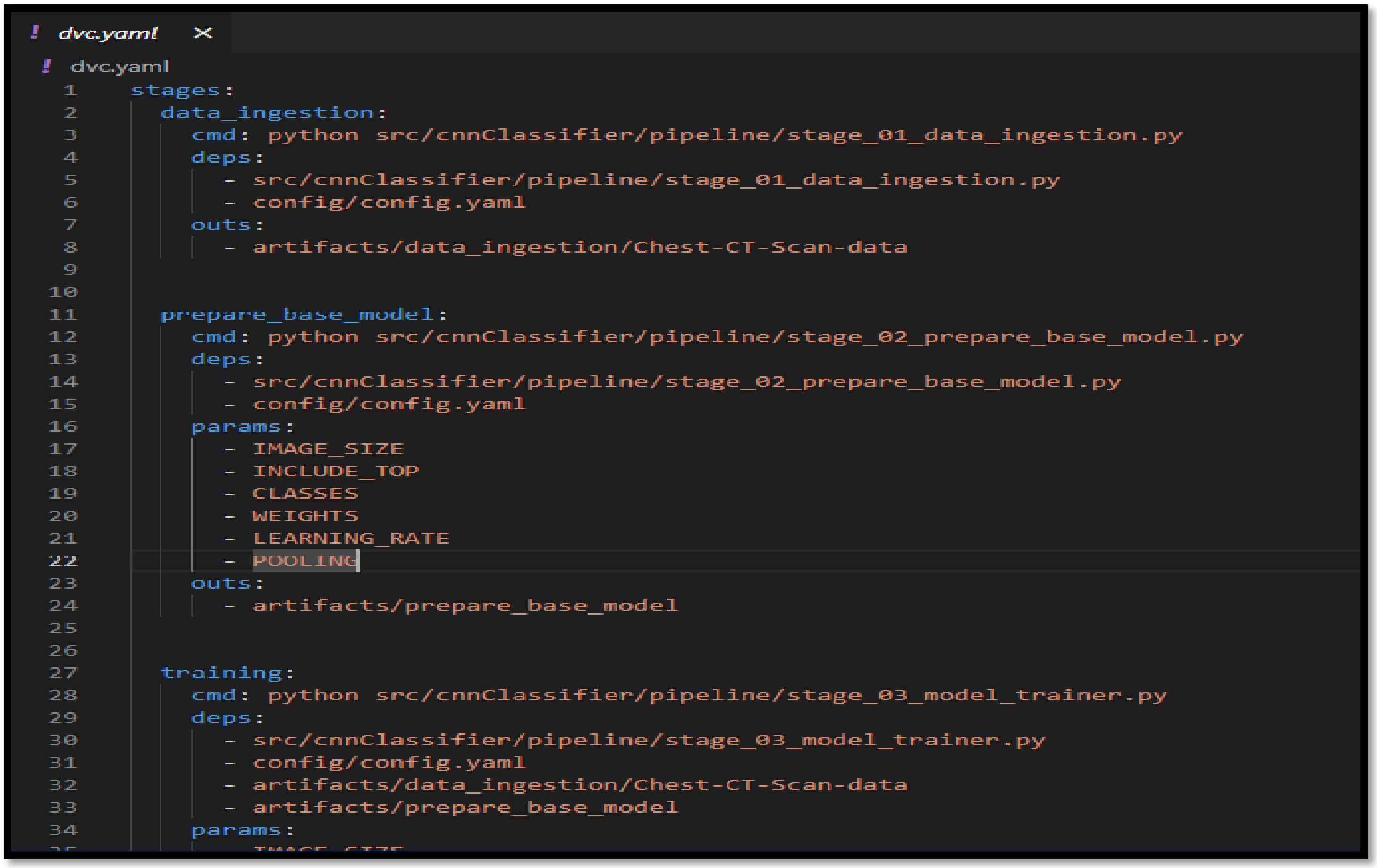
13) <u>DVC.yaml</u>

The DVC (Data Version Control) pipeline orchestrates the entire workflow of the Chest Disease Classification project, managing data ingestion, model preparation, training, and evaluation seamlessly. Each stage within the pipeline is meticulously designed to handle specific tasks while maintaining traceability, reproducibility, and efficiency.

The pipeline begins with the data ingestion stage, where raw chest CT scan images are processed and organized for subsequent stages. This stage ensures that the data is properly formatted and available for model training and evaluation.Next, the prepare_base_model stage focuses on preparing the foundational components required for building the classification model. This includes configuring key parameters such as image size, classes, and learning rates, as well as loading a pre-trained ResNet50 model with specified weights. By standardizing the model architecture and settings, this stage lays the groundwork for consistent and reliable model training.

The training stage employs the prepared base model to train the classification model using the preprocessed chest CT scan images. It fine-tunes the model's weights through iterative epochs, optimizing its performance based on specified parameters such as batch size, augmentation, and training duration. Throughout the training process, the model's progress is continuously monitored, and checkpoints are saved to ensure that the best-performing model is retained for evaluation. Finally, the evaluation stage assesses the trained model's performance using unseen test data. It computes key metrics such as loss and accuracy to gauge the model's predictive capabilities and generalization to real-world scenarios. The evaluation results are logged, allowing for easy tracking and comparison of model performance over time.

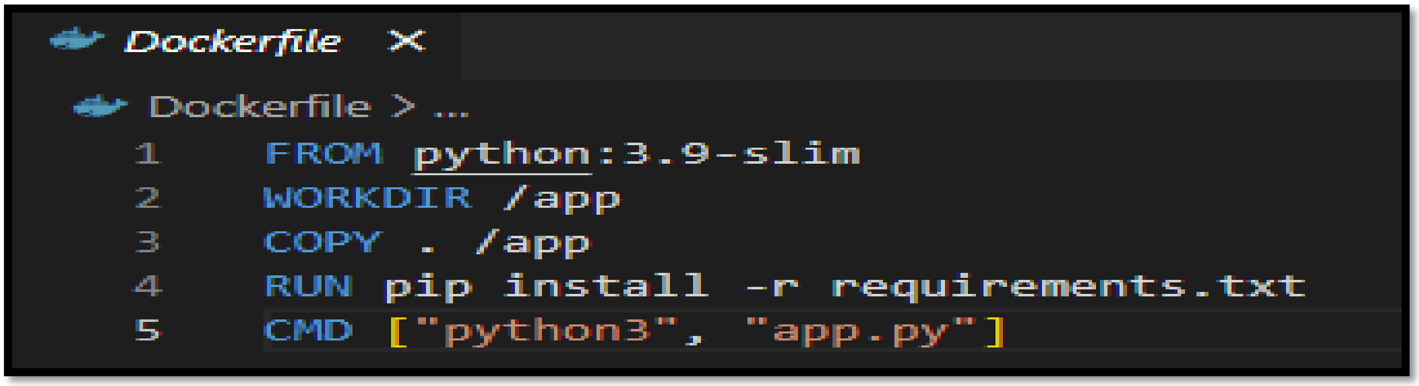
Overall, the DVC pipeline streamlines the project workflow, enabling researchers to focus on refining the model for deployment and optimization.



```yaml
stages:
  data_ingestion:
    cmd: python src/cnnClassifier/pipeline/stage_01_data_ingestion.py
    deps:
      - src/cnnClassifier/pipeline/stage_01_data_ingestion.py
      - config/config.yaml
    outs:
      - artifacts/data_ingestion/Chest-CT-Scan-data


  prepare_base_model:
    cmd: python src/cnnClassifier/pipeline/stage_02_prepare_base_model.py
    deps:
      - src/cnnClassifier/pipeline/stage_02_prepare_base_model.py
      - config/config.yaml
    params:
      - IMAGE_SIZE
      - INCLUDE_TOP
      - CLASSES
      - WEIGHTS
      - LEARNING_RATE
      - POOLING
    outs:
      - artifacts/prepare_base_model


  training:
    cmd: python src/cnnClassifier/pipeline/stage_03_model_trainer.py
    deps:
      - src/cnnClassifier/pipeline/stage_03_model_trainer.py
      - config/config.yaml
      - artifacts/data_ingestion/Chest-CT-Scan-data
      - artifacts/prepare_base_model
    params:
```

14) Dockerfile

The provided **Dockerfile** encapsulates the setup for containerizing a Python application, specifically tailored for a project involving a CNN classifier. It starts by selecting a slim Python 3.9 base image, optimizing the container's size and resource footprint. The working directory is then set to /app, where all subsequent operations take place. Next, the file system is populated with the project's contents using the COPY instruction, ensuring that all necessary files are available within the container. The subsequent RUN command installs the Python dependencies listed in the requirements.txt file, facilitating seamless execution of the application. Finally, the **Dockerfile** specifies the command to be executed when the container is launched, which in this case is python3 app.py, indicating that the app.py script should be run using Python 3 as the entry point. This **Dockerfile** streamlines the process of deploying the CNN classifier project within a containerized environment, promoting consistency, portability, and scalability across different deployment environments.



```dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["python3", "app.py"]
```

15) User Interface:

The provided HTML file index.html constitutes a user interface for a Chest Cancer Classification application. It leverages modern web technologies like **HTML5**, **CSS3**, and **JavaScript** to create an interactive and visually appealing experience for users. The interface is designed to facilitate two primary functionalities: uploading an image for prediction and displaying the prediction results.

The layout is divided into sections, with a prominent header displaying the application's title. The main section features an image upload component, where users can either upload an image from their device or capture one using their device's camera. Upon selecting or capturing an image, it is displayed within a designated area for preview.Below the image upload component, there's a set of buttons for initiating the prediction process and submitting the uploaded image to the server for analysis. Additionally, there's an input field for specifying the URL of the prediction endpoint, providing flexibility for users to interact with different APIs.

The prediction results section is visually separated from the image upload area, comprising two sub-sections: one for displaying the prediction outcome in a structured JSON format and another for presenting any accompanying visual output, such as an image representing the predicted class.

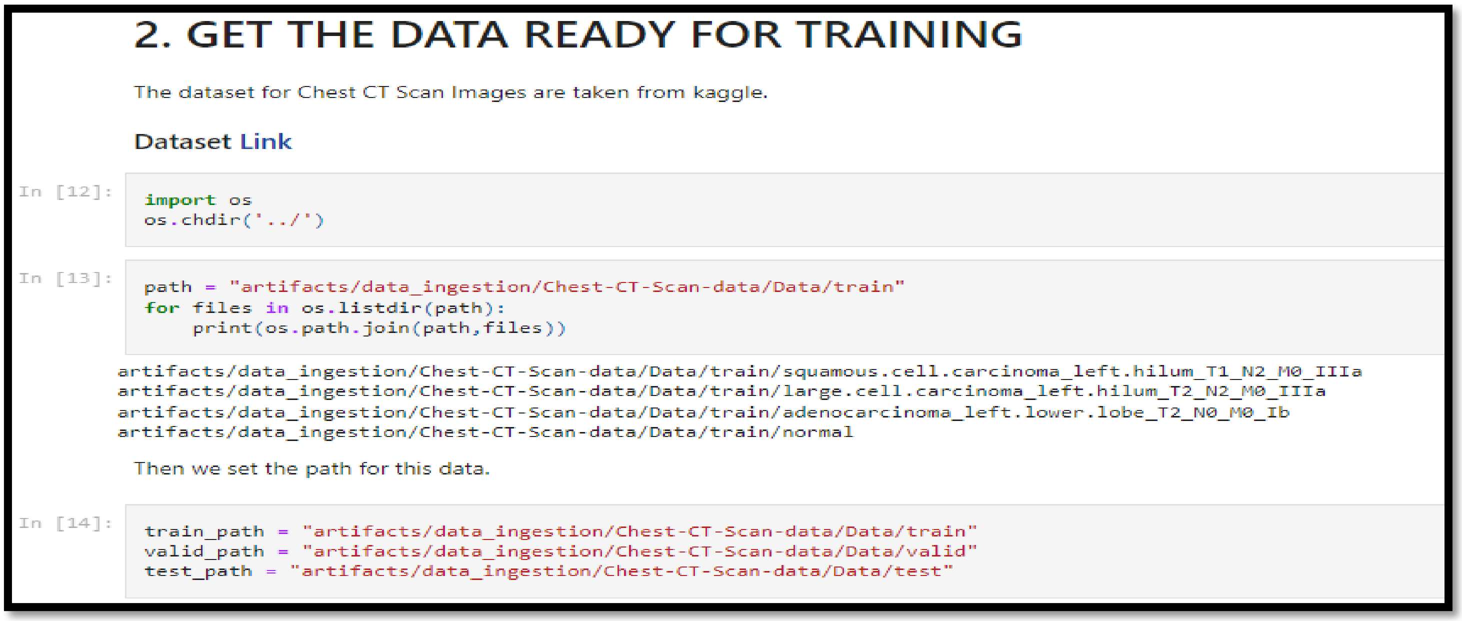
The interface is complemented with responsive design principles, ensuring optimal usability across various devices and screen sizes. It also includes a footer section containing links to additional resources and information about the project, enhancing user engagement and transparency. Overall, the UI offers an intuitive and seamless experience for users interacting with the Chest Cancer Classification application.

16) OBSERVATION OF DATA: -

Jupyter NoteBook OBSERVATIONS -:

- In this step we get the dataset of CT Scan Images stored in Google Drive and get the train_path,

test_path, valid_path of the dataset of their respective folders.



2. GET THE DATA READY FOR TRAINING

The dataset for Chest CT Scan Images are taken from kaggle.

Dataset Link

In [12]:
```
import os
os.chdir('../')
```

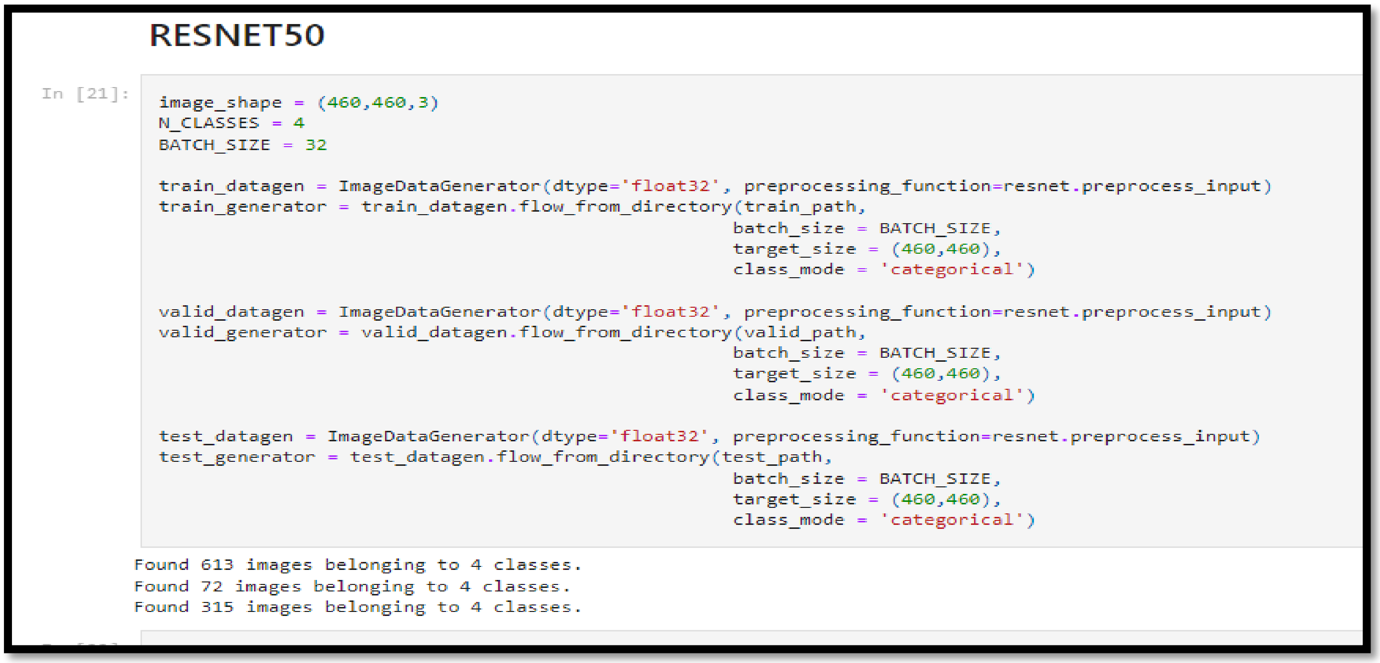
In [13]:
```
path = "artifacts/data_ingestion/Chest-CT-Scan-data/Data/train"
for files in os.listdir(path):
    print(os.path.join(path,files))
```

```
artifacts/data_ingestion/Chest-CT-Scan-data/Data/train/squamous.cell.carcinoma_left.hilum_T1_N2_M0_IIIa
artifacts/data_ingestion/Chest-CT-Scan-data/Data/train/large.cell.carcinoma_left.hilum_T2_N2_M0_IIIa
artifacts/data_ingestion/Chest-CT-Scan-data/Data/train/adenocarcinoma_left.lower.lobe_T2_N0_M0_Ib
artifacts/data_ingestion/Chest-CT-Scan-data/Data/train/normal
```

Then we set the path for this data.

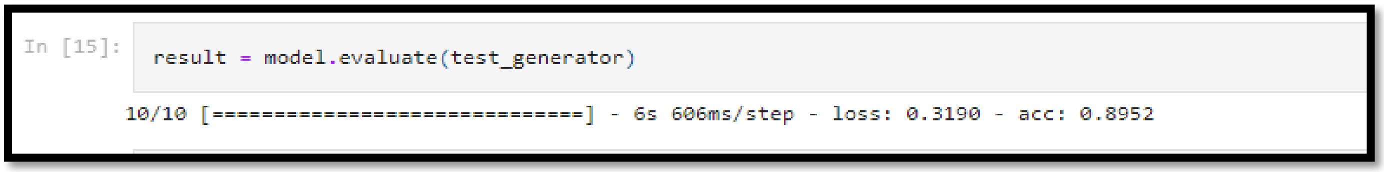
In [14]:
```
train_path = "artifacts/data_ingestion/Chest-CT-Scan-data/Data/train"
valid_path = "artifacts/data_ingestion/Chest-CT-Scan-data/Data/valid"
test_path = "artifacts/data_ingestion/Chest-CT-Scan-data/Data/test"
```

In Next step, we start with training our train dataset with **ResNet50** deep Learning CNN model with different hyperparameters and using **evaluate** method, we calculate the test accuracy of our model using test dataset, which comes out around 89.52%.



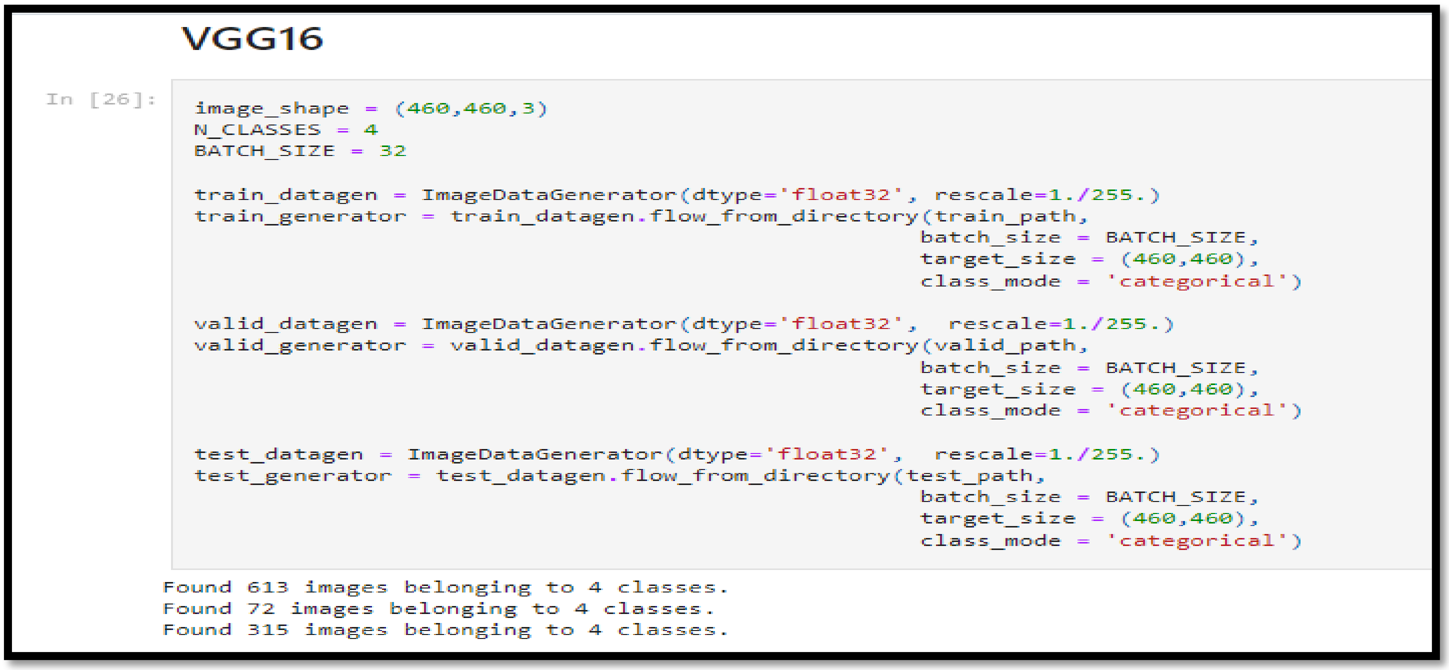
RESNET50

```
In [21]:
image_shape = (460,460,3)
N_CLASSES = 4
BATCH_SIZE = 32

train_datagen = ImageDataGenerator(dtype='float32', preprocessing_function=resnet.preprocess_input)
train_generator = train_datagen.flow_from_directory(train_path,
                                                     batch_size = BATCH_SIZE,
                                                     target_size = (460,460),
                                                     class_mode = 'categorical')

valid_datagen = ImageDataGenerator(dtype='float32', preprocessing_function=resnet.preprocess_input)
valid_generator = valid_datagen.flow_from_directory(valid_path,
                                                     batch_size = BATCH_SIZE,
                                                     target_size = (460,460),
                                                     class_mode = 'categorical')

test_datagen = ImageDataGenerator(dtype='float32', preprocessing_function=resnet.preprocess_input)
test_generator = test_datagen.flow_from_directory(test_path,
                                                   batch_size = BATCH_SIZE,
                                                   target_size = (460,460),
                                                   class_mode = 'categorical')

Found 613 images belonging to 4 classes.
Found 72 images belonging to 4 classes.
Found 315 images belonging to 4 classes.
```



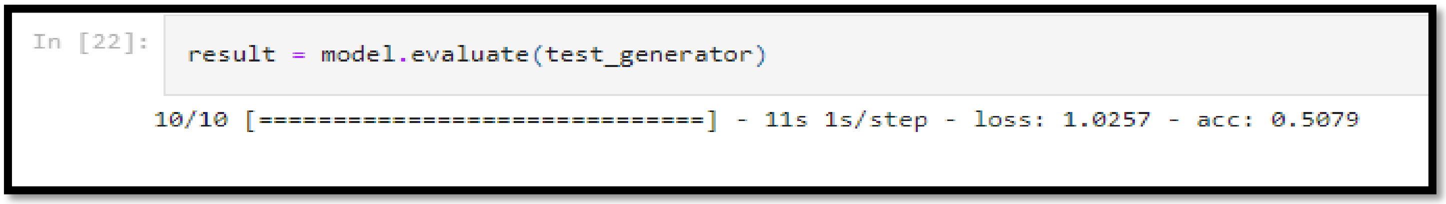
```
In [15]:
result = model.evaluate(test_generator)

10/10 [==============================] - 6s 606ms/step - loss: 0.3190 - acc: 0.8952
```

In Next step, we start with training our train dataset with **VGG16** deep Learning CNN model with different hyperparameters and using **evaluate** method, we calculate the test accuracy of our model using test dataset, which comes out around 50.79%.



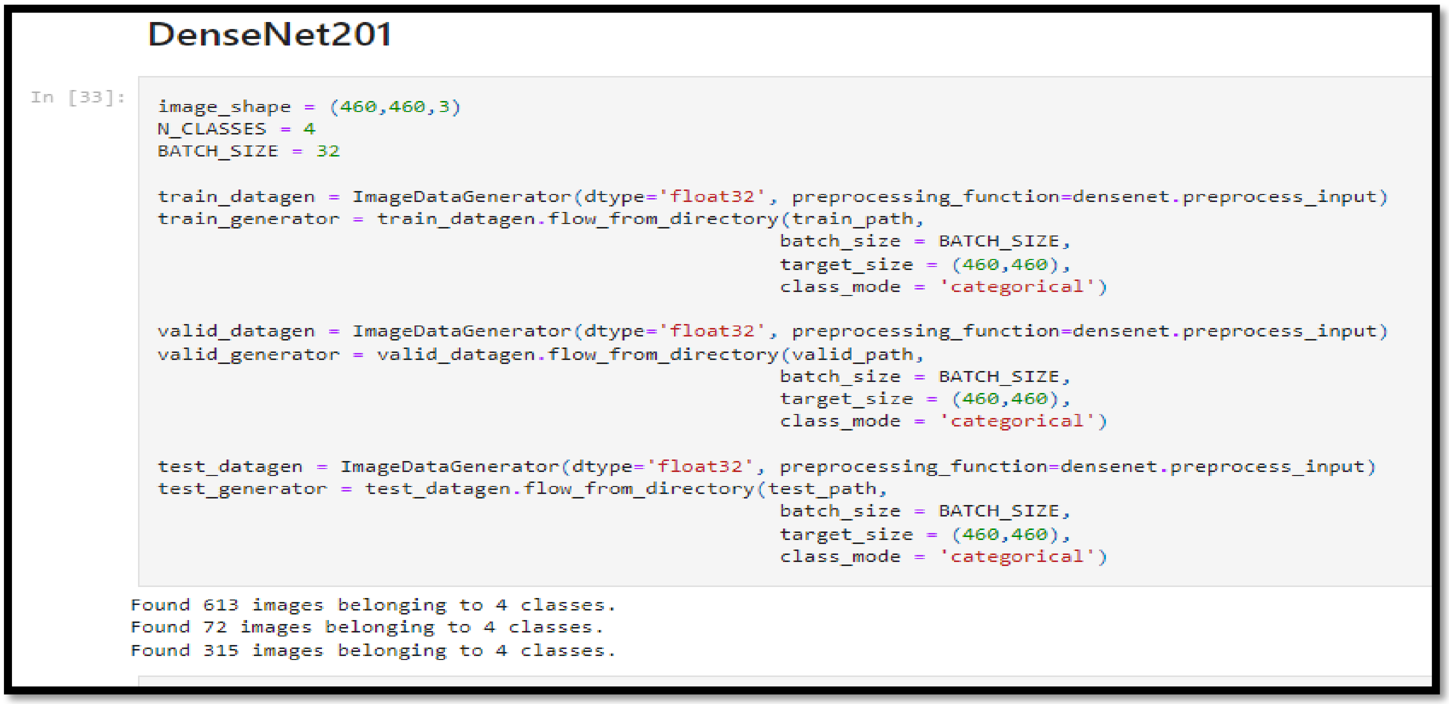
VGG16

```
In [26]:
image_shape = (460,460,3)
N_CLASSES = 4
BATCH_SIZE = 32

train_datagen = ImageDataGenerator(dtype='float32', rescale=1./255.)
train_generator = train_datagen.flow_from_directory(train_path,
                                                     batch_size = BATCH_SIZE,
                                                     target_size = (460,460),
                                                     class_mode = 'categorical')

valid_datagen = ImageDataGenerator(dtype='float32',  rescale=1./255.)
valid_generator = valid_datagen.flow_from_directory(valid_path,
                                                     batch_size = BATCH_SIZE,
                                                     target_size = (460,460),
                                                     class_mode = 'categorical')

test_datagen = ImageDataGenerator(dtype='float32',  rescale=1./255.)
test_generator = test_datagen.flow_from_directory(test_path,
                                                   batch_size = BATCH_SIZE,
                                                   target_size = (460,460),
                                                   class_mode = 'categorical')

Found 613 images belonging to 4 classes.
Found 72 images belonging to 4 classes.
Found 315 images belonging to 4 classes.
```



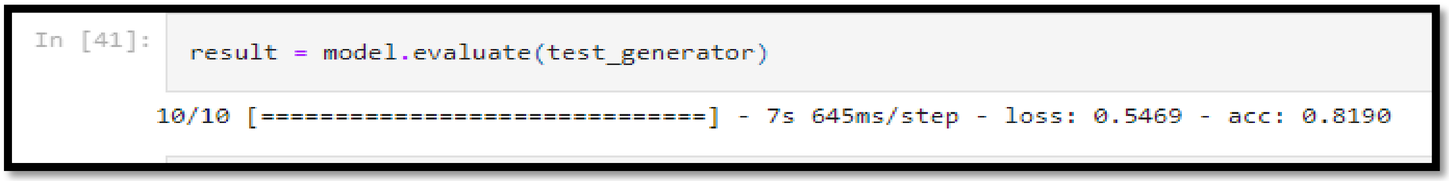
```
In [22]:
result = model.evaluate(test_generator)

10/10 [==============================] - 11s 1s/step - loss: 1.0257 - acc: 0.5079
```

In this step, we start with training our train dataset with **DenseNet201** deep Learning CNN model with different hyperparameters and using **evaluate** method, we calculate the test accuracy of our model using test dataset, which comes out around 81.90%.



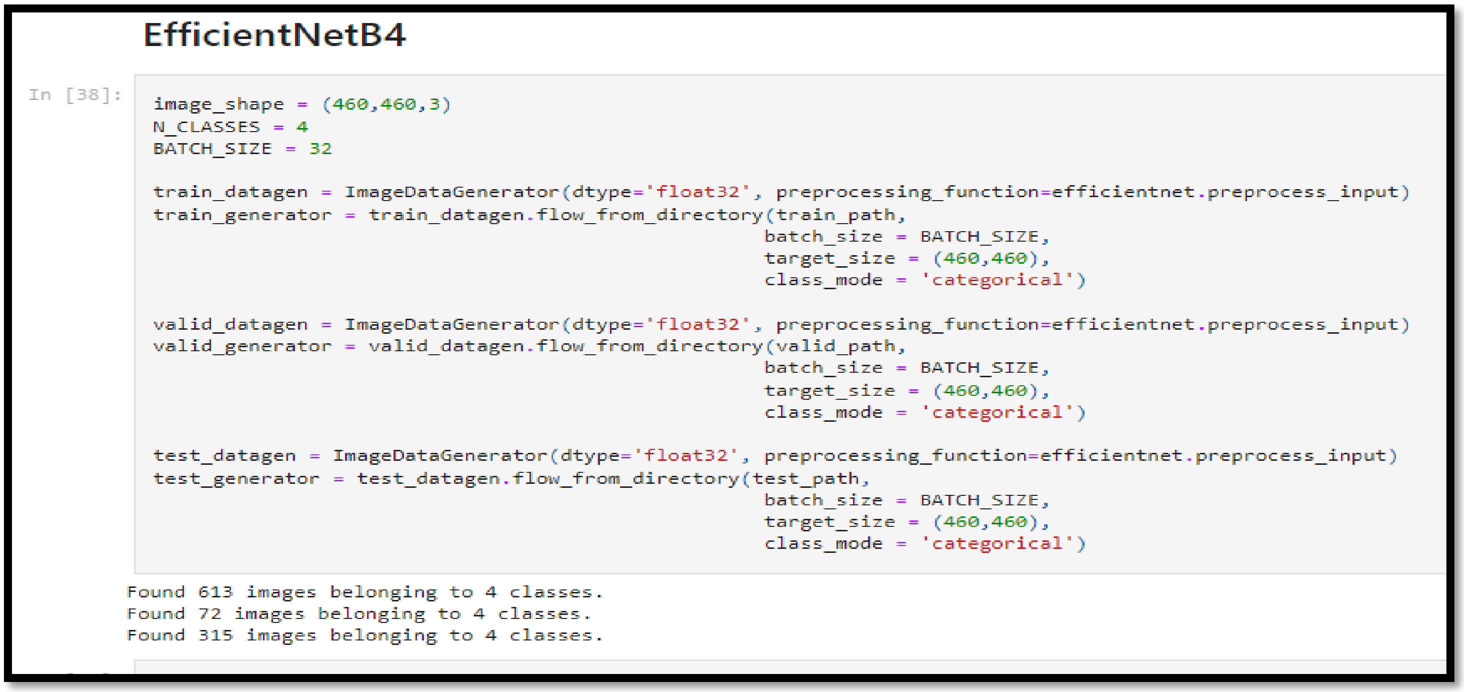
DenseNet201

```
In [33]: image_shape = (460,460,3)
         N_CLASSES = 4
         BATCH_SIZE = 32

         train_datagen = ImageDataGenerator(dtype='float32', preprocessing_function=densenet.preprocess_input)
         train_generator = train_datagen.flow_from_directory(train_path,
                                                           batch_size = BATCH_SIZE,
                                                           target_size = (460,460),
                                                           class_mode = 'categorical')

         valid_datagen = ImageDataGenerator(dtype='float32', preprocessing_function=densenet.preprocess_input)
         valid_generator = valid_datagen.flow_from_directory(valid_path,
                                                           batch_size = BATCH_SIZE,
                                                           target_size = (460,460),
                                                           class_mode = 'categorical')

         test_datagen = ImageDataGenerator(dtype='float32', preprocessing_function=densenet.preprocess_input)
         test_generator = test_datagen.flow_from_directory(test_path,
                                                           batch_size = BATCH_SIZE,
                                                           target_size = (460,460),
                                                           class_mode = 'categorical')

         Found 613 images belonging to 4 classes.
         Found 72 images belonging to 4 classes.
         Found 315 images belonging to 4 classes.
```



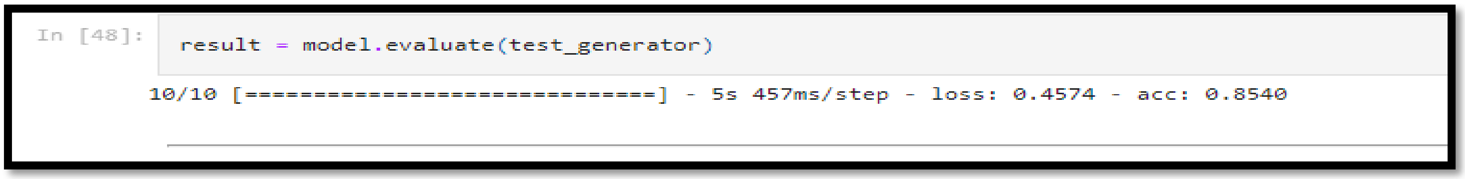
```
In [41]: result = model.evaluate(test_generator)

         10/10 [==============================] - 7s 645ms/step - loss: 0.5469 - acc: 0.8190
```

In Next step, we start with training our train dataset with **EfficientNetB4** deep Learning CNN model with different hyperparameters and using **evaluate** method, we calculate the test accuracy of our model using test dataset, which comes out around 85.40%.



EfficientNetB4

```
In [38]: image_shape = (460,460,3)
         N_CLASSES = 4
         BATCH_SIZE = 32

         train_datagen = ImageDataGenerator(dtype='float32', preprocessing_function=efficientnet.preprocess_input)
         train_generator = train_datagen.flow_from_directory(train_path,
                                                           batch_size = BATCH_SIZE,
                                                           target_size = (460,460),
                                                           class_mode = 'categorical')

         valid_datagen = ImageDataGenerator(dtype='float32', preprocessing_function=efficientnet.preprocess_input)
         valid_generator = valid_datagen.flow_from_directory(valid_path,
                                                           batch_size = BATCH_SIZE,
                                                           target_size = (460,460),
                                                           class_mode = 'categorical')

         test_datagen = ImageDataGenerator(dtype='float32', preprocessing_function=efficientnet.preprocess_input)
         test_generator = test_datagen.flow_from_directory(test_path,
                                                           batch_size = BATCH_SIZE,
                                                           target_size = (460,460),
                                                           class_mode = 'categorical')

         Found 613 images belonging to 4 classes.
         Found 72 images belonging to 4 classes.
         Found 315 images belonging to 4 classes.
```



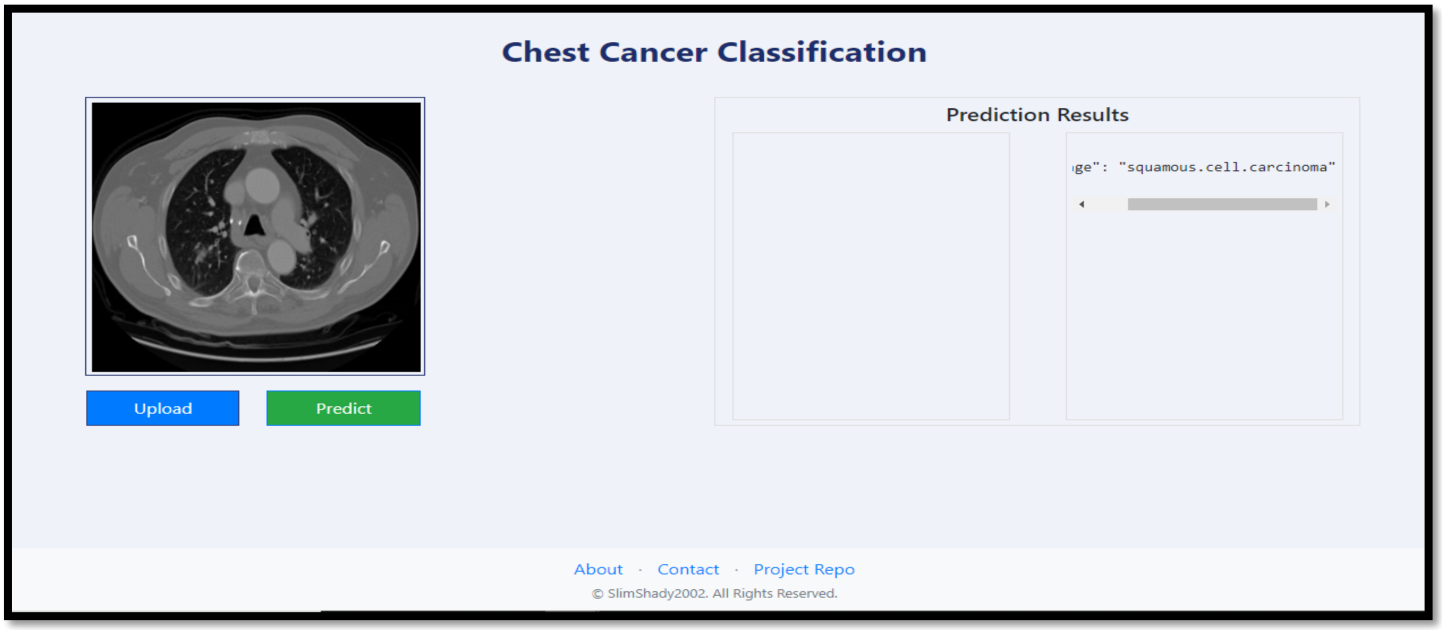
```
In [48]:  result = model.evaluate(test_generator)

          10/10 [==============================] - 5s 457ms/step - loss: 0.4574 - acc: 0.8540
```

The Most three promised models are:

- ResNet50 89.52% accuracy on test data and 31.9% Loss
- DenseNet201 81.9% accuracy on test data and 54.69% Loss
- EfficientNetB4 85.4% on test data and 45.74% Loss

From above we can conclude that ResNet50 will be the best model to implement in our DVC Pipeline end to end project as we are getting very high accuracy of 89.52% and very less loss percentage of 31.9%.

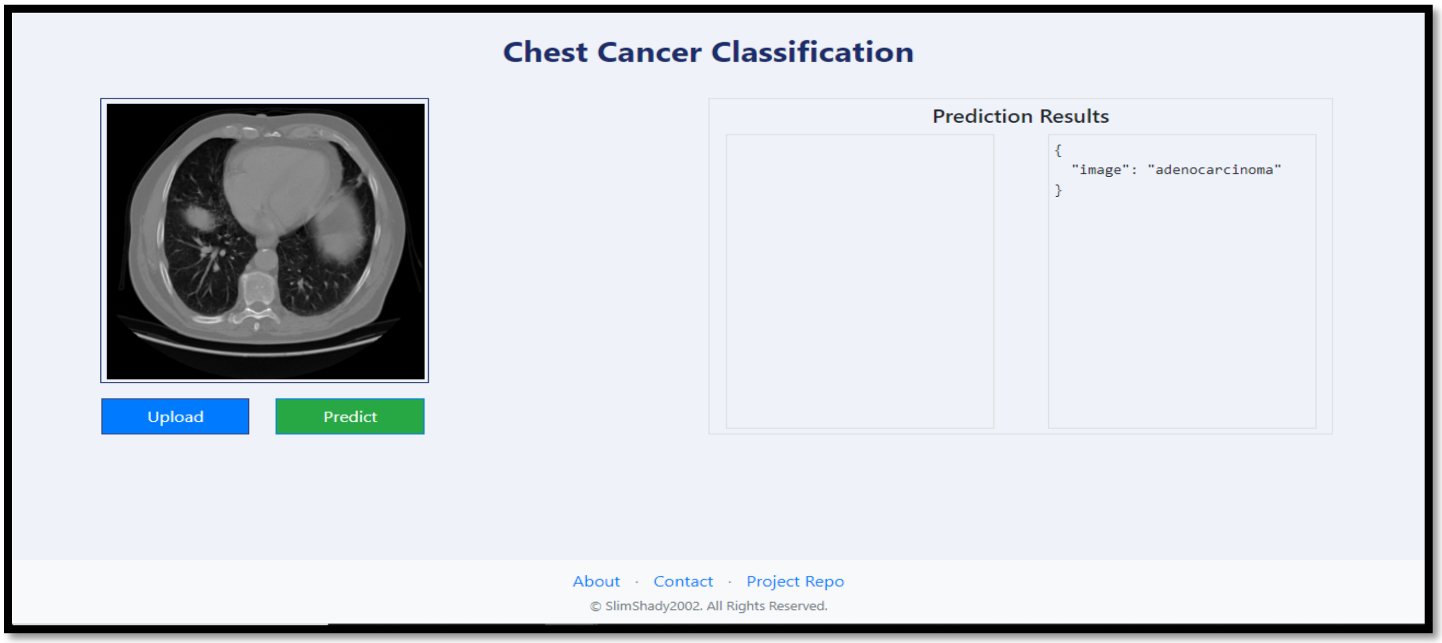
Prediction Outputs



Chest Cancer Classification

Prediction Results
Upload
Predict
About · Contact · Project Repo
© SlimShady2002. All Rights Reserved.

```
ge": "squamous.cell.carcinoma"
```



Chest Cancer Classification

Prediction Results
Upload
Predict
About · Contact · Project Repo
© SlimShady2002. All Rights Reserved.

```
{
  "image": "adenocarcinoma"
}
```

CONCLUSION

1) This project is production ready for similar use cases and will provide the automated and orchestrated pipeline using DVC(Data Version Control).

2) MLFlow helped us to keep a track of Parameters and metrics with respect to different set of parameters and helped to fine tune our model.

3) ResNet50 model classify Chest CT scan images upto **89.52%** respective classes and hence reduces the chance of incorrect predictions of diseases.

FUTURE WORK

1) Explore techniques for fine-tuning the model architecture and hyperparameters to optimize performance further. This may involve experimenting with different network architectures, adjusting learning rates, or incorporating advanced regularization techniques.

2) Implement more sophisticated data augmentation techniques to artificially increase the diversity and size of the dataset. This can help mitigate issues related to data scarcity and improve the model's ability to generalize to unseen data.

3) Explore ensemble learning approaches to combine predictions from multiple models trained with different architectures or subsets of the dataset. Ensemble methods can often yield superior performance by leveraging diverse model strengths and mitigating individual model weaknesses.

4) Investigate techniques for enhancing the interpretability and explainability of the model's predictions, particularly in the context of medical diagnostics. This can help clinicians better understand and trust the model's decisions, fostering its clinical adoption.

SOURCE CODE →

https://github.com/jatin-12-2002/Chest-Disease-Classification-Final-Version

REFERENCES→

- https://keras.io/api/applications/
- https://www.kaggle.com/code/prthmgoyl/ensemblemodel-ctscan
- https://www.tensorflow.org/
- https://mlflow.org/
- https://dagshub.com/jatin-12-2002/Chest-Disease-Classification-Final-Version