End to End Text Summarisation

1. Introduction & GitHub Repository Setup
2. Project Template Creation
3. Project Setup & Requirements Installation
4. Logging, Utils & Exception Module
5. Project Workflow
6. All Components Notebook Experiment
7. All Components Modular Code Implementation
8. Training Pipeline
9. Prediction Pipeline & User App Creation
10. Final CI/CD Deployment on AWS

Text Summariser

1. **Def -** It is a NLP based System that converts Long Text to short , meaningful summary without losing the main meaning of the text.
2. **Types** -
   1. Extractive Summarisation : Model select important sentences from the original text , Example: selecting key lines directly from the article , Tools: TextRank, Sumy, Spacy, etc.
   2. Abstractive Summarisation : Model understands the text and then rephrases it in its own words (like a human) , Tools: Transformers (e.g., BART, T5, Pegasus).
3. **Working :** I/P text to Model.
   1. Model Tokenises it (into words/sentences)
   2. NN processes it to understand context
   3. Model generates shorter sentences capturing key meanings
   4. O/P = Summary
4. In my **project** I am using HuggingFace Transformer for making abstract summariser which will do real-time text summarise using FastAPI.

Transformers

1. Unlike old modes (RNN , LSTMs) Transformer read the entire sentence at once.
2. They use self attention to understand which words are related to which.
3. Transformers preservers context and relationships (semantic meaning) btw words even they are far apart .
4. That's why they r perfect for : summarisation, translation, question answering, semantic analysis, chatbots.
5. **Importance** : they understand the full meaning of the text using self attention, n they also preserve the semantic relationships btw objects…which makes it perfect for text summarisation.
6. **Working** :
   1. I/P Text -> Tokens (words broken into small tokens)
   2. Self attention layer -> Finds which words r imp to each other.
   3. Encoder-Decoder Layer ->
      1. Encoder understands the text meaning.
      2. Decoder generates a shorter, meaningful version (summary)

7. In my project :
    1. Transformer performs the actual summarisation.
    2. Flask app sends text to transformer model -> gets the summary back
    3. MLOps handles data, training, deployment, and monitoring around the model.

Libs :

1. **Torch :**
    1. It is the main package of PyTorch, a DL framework made by Facebook
    2. It helps in building and training NNs - just like tensor flow does for google.
    3. Works with tensor (multi-dimensional arrays like Numpy, but with GPU support)
    4. Supports automatic differentiation (autograd) -> calculates gradients automatically for back propagation.
    5. Used for training DL models (CNN, RNN, Transformers, etc)
2. **PyTorch :**
    1. An open source ML framework based on torch
    2. It provides high level APIs for building and training NNs
    3. In my project used behind transformers models - eg AutoModelForSeq2SeqLM is built on PyTorch.
3. **tqdm :**
    1. Means “progress bar”
    2. It shows progress bar when running loops.
    3. Helpful to track model training or data processing progress (when generating summaries).
4. **ntlk :**
    1. Natural Language Toolkit, py lib for NLP
    2. Helps text processing like : Tokenization, Stopword removal. Stemming/ lemmatization.
    3. Used to tokenize text into sentence before training & summarisation.
5. **Pandas :**
    1. A lib for data analysis & manipulation
    2. Provides DataFrame, like excel tables in py
    3. Great for CSV, Excel, JSON handling
    4. Used to shoe ROUGE metric scores in a DataFrame.
6. **NumPy :**
    1. A lib for numerical computing in py
    2. Provides N-dimensinoal arrays (ndarray)
    3. Performs mathematical operations super fast (vectorised)
    4. Backbone for many libs like pandas , scikit-learn
7. **Matplotlib :**
    1. A py data visualization library.
    2. For creating graphs n charts - line charts, histograms, etc
8. **Seaborn :**
    1. A statistical data visualisation library built on top of matplotlib
    2. It makes chats prettier and adds statistical insights like heatmaps, pairplots, etc

Exceptions :

1. **Exceptions** are errors that stop normal flow of program - like when something unexpected happens.
2. **Built-in (Box) Exceptions** are default exceptions provided by python. Eg : ZeroDivisionError -> Divide by zero.
3. C**ustom Exceptions** these are own custom error class, for specific use case, these controls what happens when specific project errors occur.

.yaml

1. Yet Another Markup Language
2. Used for Config files & structured data
3. Syntax : Simple key- value pairs
4. Advantage : Human-Friendly, easy to edit
5. In my project used for : Paths, parameters, and pipeline setup
6. .yaml files are like the "control center" of your project — they store all important configurations (like model paths, parameters, data folders) in one easy-to-read place.

ConfigBox

It is a special class from the python-box library that allows to access dictionary values using dot notation (.) instead of square brackets ([ ])

Data Ingestion

1. Data Ingestion is the process of collecting data from external sources (like APIs, databases, files, sensors, or URLs) and loading it into a centralized storage or pipeline (like a local folder, data lake, or database) for further processing.
2. In my project : dataset(summariser-data.zip from GitHub) ; store it locally in (artifacts/data_ingestion/data.zip)
3. Stores the raw data in a controlled folder
4. Types :
    1. Batch
    2. Real time (Streaming)
    3. Lambda.Hybrid
5. I built a modular data ingestion component with configuration management, logging, and idempotent downloads to ensure reproducibility and automation in the ML workflow.
6. Pipeline flow :
    1. Source (URl / API / DB)
    2. Data Ingestion (download + extract)
    3. Data Validation (check for missing, duplicates)
    4. Data Transformation (Tokenization, cleaning)
    5. Data Training
7. I built a modular data ingestion component with configuration management, logging, and idempotent downloads to ensure reproducibility and automation in the ML workflow

1. **Modular Data Ingestion Component** : a separate, reusable part of project that only handles data downloading and extraction
2. **Configuration Management** : managing all imp project settings (like file paths, URLs parameters) .. so that I can change values easily without touching code
3. Logging : whenever code downloads, extracts, or fails — it writes messages to a log file (e.g., running_logs.log) ; It helps in debugging and understanding what happened during a run.
4. **Idempotent** : running the same code multiple times gives the same result, without side effects
5. **Reproducibility** : Ability to reproduce (repeat) the same results again with the same configuration and code ; same O/P.
6. **Automation** : Making the entire ML workflow run automatically — no manual steps ; Each stage (ingestion → validation → training → evaluation) runs one after another through pipelines.

Data Validation
1. Data Validation checks whether the required files - train, test, and validation - exist after the data ingestion stage
2. Basically making sure that the dataset is present and not broken before training the model. See Step - 9
3. In the data validation stage , I created a configuration-driven pipeline that ensures all necessary dataset files (train, test, validation) are present after ingestion. It checks for missing files and logs the validation result. This helps prevent downstream model training failures and ensures data reliability

Data Transformation
1. It means converting raw textual data into a format that the ML model can understand
2. Step 10

Model Training
1. Model Training Means feeding the preprocessed (tokenized) data into a ML model
2. Step 11

Model Evaluation
1. It means measuring how good ur trained model performs on unseen/test data.
2. It helps to know whether model is generating accurate & meaningful summaries

FastAPI
1. FastAPI is a modern, high speed web framework used to build APIs with Python
2. It lets users send input (like text) -> model processes it -> returns the result instantly
3. Used for real time ML model deployment
4. Faster than flask (because its built on ASGI instead of WSGI)….refers to web frameworks like FastAPI or Quart, which leverage the Asynchronous Server Gateway

Interface (ASGI) for potentially higher performance compared to traditional Web Server Gateway Interface (WSGI) based frameworks like Flask.
5. high performance python framework for creating APIs
6. uvicorn -> web server that run FastAPI apps

Docker

1. It is a containerisation platform, it lets us to package entire project(code + dependencies + OS libs) into one portable unit called a container
2. Inside docker setup :
   1. Dockerfile : file telling docker about how to build app image step by step
   2. Docker Image : it is like a blueprint of app
   3. Docker Container : docker takes that image and launches a container
   4. Port Mapping : -p 8080:8080 means inside docker, FastAPI runs on port 8080, docker exposes that port to local host machine
3. Why Docker is crucial :
   1. Dependency conflicts : same environment always
   2. Works everywhere
   3. One command deploy
   4. Cloud deployment : supported by AWS, Azure, GCP, etc
   5. Reproducibility : model + code + dependencies are preserved

Steps in making project

Step 1 -

1. Instead of making manually folders , I will use template.py file.
2. This script is an auto initialiser it automatically creates the complete folder structure and empty files for my MLOps project in one go.
3. Used 3 libs for it
   1. os : to handle file n folder
   2. path : to manage file paths
   3. logging : to print message

Step 2 -

Requirement.txt file

Step 3 -

1. In setup.py file, firstly imported setuptools (helps create n distribute py packages)
2. This file shows the installation blueprint of the project defines how textSummarizer is structured, who made, how to install it, for professionalism.

Step 4 -

1. Created a file in src/textSummarizer/logging/__init__.py , this file creates n configures a customer logger (a tool that records what's happening in my program - useful for debugging & tracking execution)
2. When this file runs this happens :
   1. Import libs :
      1. os (for file & directory handling) ;

    2. sys (for sys operations like stdout) ;
    3. logging (pi's built in logging system)
2. Def log msg format , includes :
    1. asctime -> time of log
    2. levelname -> type of message (INFO , ERROR, etc)
    3. module -> which file generated it
    4. message -> actual log msg
3. To store logs safely a logs folder is created.
4. Create log file path
5. Configure logging system, Configures handlers -> where logs will go :
    1. FileHandler(log_filepath) → save in file
    2. StreamHandler(sys.stdout) → print on console
6. Create a logger object
7. When the main file runs it imports logger setup, it automatically logging gets configured (created folder + file) then msg is printed on terminal & saved inside logs/running_logs.log. *idhr sab save ho rha hai date tym msg jo print hua tha terminal m.

Step 5 -

1. Now inside utils folder created a common.py file , it is a small helper toolkit
2. It helps the text summarizer to load configs, set up folders, and log everything properly - all in a clean, reusable, and professional way.
3. This file contains utility (helper) functions which r used again n again , basically it makes my project more modular, reusable, and cleaner.
4.
    1. read_yaml() -> reads yaml file and returns config as ConfigBox
    2. create_directories() -> creates multiple folders safely
    3. get_size() -> returns size of a file.

Step 6 -

1. Use of ensure_annotations shown in trails.ipynb file.
    1. Ensure -> a library -> used for runtime validation of function type
    2. ensure_annotations -> a decorator from ensure -> checks function arguments and return types match ur type hints
    3. ensure_annotations automatically verifies function's I/P n O/P types at runtime - helping to catch mistakes early.
2. In my project used this to ensure that :
    1. path_to_taml is actually a path
    2. Function returens a ConfigBox
    3. To make code is safer, cleaner, and more professional.

Step 7 -

1. Now the Text_Summarization.ipynb file, in this I loaded a pertained Pegasus summarisation model, prepare the Samsum dialogue dataset, fine-tune Pegasus (tokenize -> collate -> trainer -> train), evaluate generated summaries with ROUGE, save the model/tokenizer, and run inference with a pipeline.
2. Step by step :
    1. **Environment & installs**

    1. Check gpu -> to speed up training, install req.ed libs (transformers, datasets. Evaluate/rouge, accelerate) -> these provide models, datasets, tokenisers and evaluations matrices.
    2. Nvidia-smi : to confirm gpu availability & free memory.
2. **Import & device**
    1. Import : transformers, datasets, torch, tqdm, nltk, pandas, etc
3. **Load pretrained model & tokenizer**
    1. Tokenizer : converts raw text into tokens (no.s) that model can understand. And during O/P it also converts tokens back into text (decoding).
    2. AutoTokenizer is a universal class from hugging face that automatically loads the correct tokeniser for chosen model… I won't need to manually specify the tokeniser type (like pegasus Tokeniser or bartTokenizer).
    3. AutoModelForSeq2SeqLM : AutoModel -> automatically selects the correct model architecture and Seq2SeqLM -> Sequence to sequence language model (means when one sequence is given(I/P Text) the model O/Ps another sequence (summary)).
    4. Loaded a pretrained google/pehasus-cnn model fine tuned for summarisation tasks.
    5. Pretrained weights give strong starting point; faster & better than training from scratch
    6. Pegasus is pretrained (already learned how to summarise text by training on thousands of news articles) with summarisation-style objectives (gap-sentence generation) and performs well at abstractive summarisation.
4. **Download & load dataset**
    1. Loaded Samsum (dialogue -> summary) dataset stored locally.
    2. Dialogues require specialised training for conversation summarization.
    3. Samsum dataset is human-generated summaries for chat dialogues - used to train/validate dialogue summarisers.
5. **Inspect dataset**
    1. Print split lengths, column names, and example dialogue/summary.
    2. To understand data shape and content before preprocessing
    3. To confirm format, sizes, missing values, and design preprocessing accordingly.
6. **Tokenization/Preprocessing**
    1. Prepared(raw data -> model readable format) dataset for Pegasus Model ,
       I/P text -> dialogue & target text -> summary.
    2. Converted raw text(I/P dialogue & summary both with max length =1024 $ 124 respectively) to numeric token ids , then returning combined encodings , that model consumes.
    3. Model trains on token ids; truncation prevents out-of-memory.
7. **Training + evaluation setup**
    1. DataCollatorForSeq2Seq :
        1. Data collector prepares batches of data for training n ensures both I/P (dialogue) n O/P (summary) are tokenised & properly aligned
        2. It bundles (batches), pads (sequences dynamically to same length), and format data into mini-batches for training.
        3. DataCollatorForSeq2Seq is like a helper that organises data before feeding into the model

2. TrainingArguments : this defines how training happens - learning schedule, evaluation frequency, saving etc
3. Trainer :
    1. it is a hugging face's built-in training engine - it handles all the boring ML work (Forward pass, backward pass, optimisation, evaluation)
    2. It connects everything : Model , tokenizer , data , training settings , data collator.
    3. It automates training loop .. to remove the manual need to code epochs , loss etc
4. Start training :
    1. Feed batches into model - the dataset is spliced into **51 mini-batches** (training steps).
    2. Computes **loss** - the model calculates how far its generated summary is from actual - **3.046**
    3. Updates model weights - backpropagation adjusts weight to reduce loss
    4. Evaluate performance - after every few steps, metrics like loss and accuracy are checked
    5. Goal - make the model generate summaries that sound more human-like and accurate.
5. After training , pegasus can summarise dialogues in a way that closely mimics human understanding of context and meaning.
6. Pegasus model is fine-tuned for 1 epoch , it ran for 157 seconds and updated its weights 51 times , the final training loss = ~3.0
7. What I can do :
    1. Train for more epochs ,
    2. test the model ,
    3. Save & load the fine - tuned model for inference in my app or pipeline.

8. **Evaluation Phase**
    1. Splitting dataset into smaller bathes , batch_size=16 -> ~62 small chunks
    2. calulate_metric_on_test_ds() : main evolution function
        1. Firstly separated I/P text & summary
        2. Then tokenised with max_length = 1024
        3. Now pegasus generates its own summary for the dialogue : summaries = model.generate(…) with length_penalty=0.8 -> avoids generating overly long summaries , num_beams=8 -> uses beam search , max_length=128 -. Max summary length
        4. Convert numerical tokens to readable text & remove special tokens
        5. Using the ROUGE metrics model's summaries & human-written summaries are compared.
    3. Rouge Evaluation :
        1. ROUGE-1 : measures overlap of single word
        2. ROUGE-2 : overlap of 2-word pairs (bigrams)
        3. ROUGE-L : measures longest common subsequence (structure similarity)
        4. ROUGE-Lsum : overall senetence-level similarity
        5. The higher the ROUGE score (closer to 1) -> the better model summaries like human.
    4. Displayed ROUGE scores in pd DataFrame , saved the fine-tuned wts & tokenised vocabulary
    5. In short :

        1. Split test data in small chunks for evaluation
        2. Each dialogue is tokenised & summarised by pegasus
        3. ROUGE metics compare model-generated summaries with human-written ones.
        4. Finally save model & tokenizer for later use.
    9. **Prediction :**
        1. Defined genration parameters : 0.8 , 8, 128
        2. Picked a test sample [0]
        3. Created a summarization pipeline (automatically handles tokenisation, model inference, and decoding)
        4. Generated & displayed :
            1. Original dialogue
            2. Humman summary (refernce)
            3. Model's generated summary

Step-8

1. Workflow overview :
    1. Update config.yaml (file that stores settings, parameters, configurations necessary for the project's execution and behavior)
    2. Update params.yaml (a standard way to store and manage a model's hyperparameters and other configuration settings in a human-readable format.)
    3. Update entity (data classes)
    4. Update configuration manager in src/config (reads YAML and prepares dirs)
    5. Update components (data_ingestion implementation)
    6. Update pipeline (stage scripts)
    7. Update main,py
    8. Update app.py
2. One by one :
    1. config.yaml : human-editable configuration for data ingestion paths & sources URL ; reproducibility + separation of concerns ; can change data location or URL without modifying Py
    2. params.yaml : hyperparameters / runtime settings
    3. entity : typed container ; makes function signature explicit & easier to test
    4. ConfigurationManager : reads YAML files into ConfigBox objs & creates req.ed artefact directories ; centralised factory that prepares all config-driven objects for pipeline stages ; returns data class instances, ensure exit before use.
    5. components/data_ingestion.py : concrete logic to download dataset from a remote URL and extract it locally.
    6. pipeline/stage_01_data_ingestion.py : orchestrator for the data ingestion stage.
    7. main.py : top level script that runs pipeline stages and logs start/completion ; a single entry point to run full pipeline or selected stages .
3. What happened :
    1. ConfigurationManager() read config.yaml and params.yaml via read_yaml() - log shows yaml file loaded successfully
    2. create_directories([self.config.artifacts_root]) created artifacts/ - log: created directory at: artifacts.
    3. get_data_ingestion_config() created artifacts/data_ingestion - log: created directory

        4. DataIngestion.download_file() checked artifacts/data_ingestion/data/zip - file did not exist -> it downloaded using urllib.request.urlretrieve. Log printed filename and HTTP headers (content-Length etc)
        5. extract_zip_file() created unzip dir and extracted data.zip into artifacts/ data_ingestion.
        6. Proces finished
4. Purpose & benefits :
    1. config, entities, components, pipeline, and runner are separate
    2. Reproducibility: configuration files (YAML) capture the exact dataset URL and paths
    3. Idempotency: check-if-exists avoids redownloading; safe to re-run pipeline
    4. Traceability & Debugging: logging every step (created folders, download headers, sizes) makes debugging and audits easy
    5. MLOps-friendly: each stage is independent and can be plugged into CI/CD or scheduled jobs.
    6. Type safety & clarity: dataclasses (DataIngestionConfig) and ConfigBox allow easy, readable access to config values.
5. Some questions :
    1. Why ConfigurationManager instead of hard-coding paths? : to keep code environment-agnostic, allow quick config changes, and support reproducibility and multi-environment runs
    2. Why dataclass for DataIngestionCongif? : as databclass provide typed, immutable containers with a clear contract - easier to test and prevents accidental modification.
    3. HTTP headers for confirm file size, server response, and for debugging
    4. Separate pipeline stages into independent classes for modularity - run, test, or parallelise stages separately; easier to add monitoring/rollback
6. Data Ingestion Stage — Overview
    1. config.yaml: defines data source (URL) and artifact directories.
    2. params.yaml: training and runtime hyperparameters.
    3. DataIngestionConfig (dataclass): typed container for ingestion settings.
    4. ConfigurationManager: reads YAMLs, returns dataclass objects, creates artifact directories.
    5. DataIngestion component:
        1. - download_file(): idempotent download using urllib; logs HTTP headers.
        2. - extract_zip_file(): extracts downloaded zip into unzip_dir
    6. Pipeline stage (stage_01_data_ingestion): orchestrator that calls the component using config from ConfigurationManager.
    7. main.py: top-level runner that runs stage with structured logging and exception handling.
    8. Design benefits: separation of concerns, reproducibility, idempotency, modular MLOps-friendly stages, and audit-able logs.

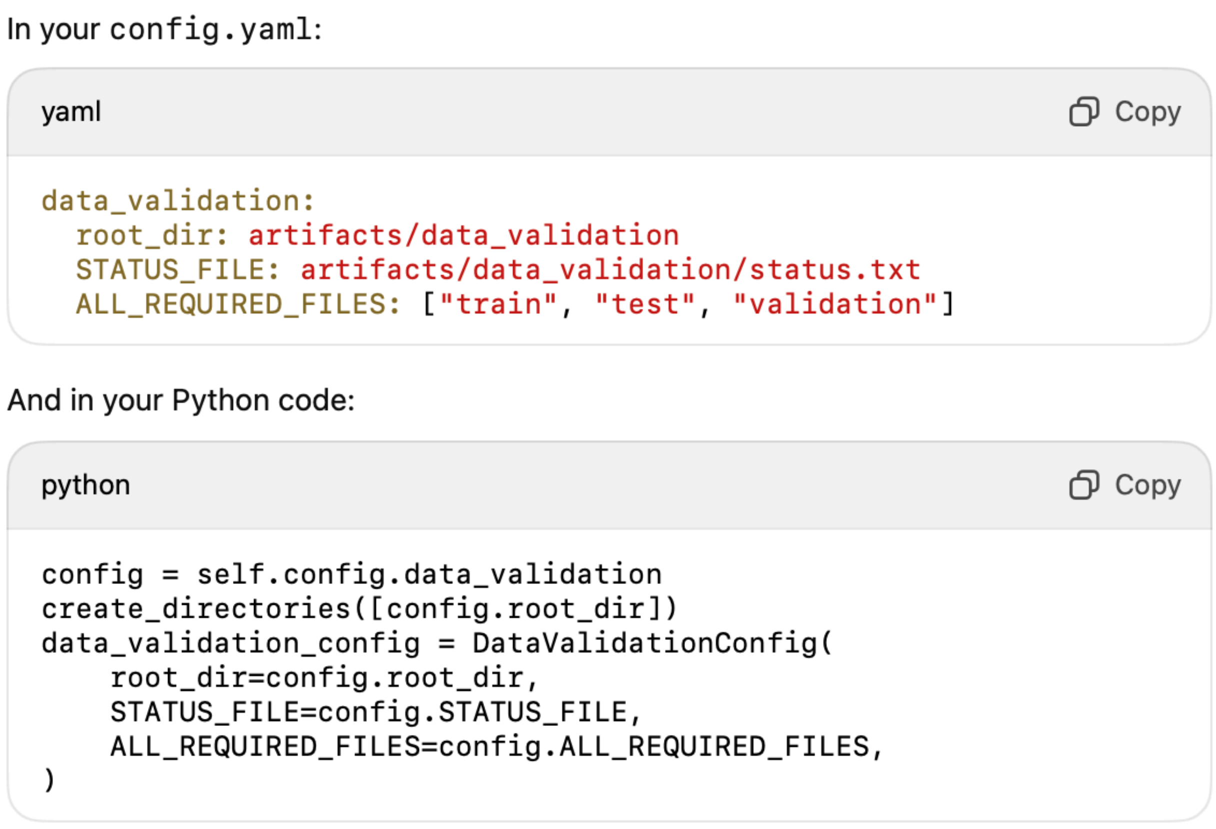
Step - 9

Data Validation

1. Goal : to make sure all the necessary files (train, test, and validation) are present after data ingestion before the model starts training
2. In the data validation stage , I created a configuration-driven pipeline that ensures all necessary dataset files (train, test, validation) are present after ingestion. It checks for

missing files and logs the validation result. This helps prevent downstream model training failures and ensures data reliability

3. Configuration Driven pipeline : means that ur code does not hardcode paths or parameters - instead everything (like paths, URLS, filenames, etc) comes from config files such as config.yaml , params.yaml ; to make ML pipeline flexible &b reusable
4. See pic :



In your `config.yaml`:

```yaml
data_validation:
  root_dir: artifacts/data_validation
  STATUS_FILE: artifacts/data_validation/status.txt
  ALL_REQUIRED_FILES: ["train", "test", "validation"]
```

And in your Python code:

```python
config = self.config.data_validation
create_directories([config.root_dir])
data_validation_config = DataValidationConfig(
    root_dir=config.root_dir,
    STATUS_FILE=config.STATUS_FILE,
    ALL_REQUIRED_FILES=config.ALL_REQUIRED_FILES,
)
```

5. Logging the validation Result : means recording what's happening in the pipeline (success, failure, missing files, etc)
    1. Logging via the logger module :
        imported : from textSummarizer.logging import logger
        used : logger.info(f">>>>>> stage {STAGE_NAME} started <<<<<<")
        and : logger.exception(e)
        this automatically writes info into log file about : when the stage started or completed, if there were errors/exceptions ; This helps you trace what happened at every stage in your pipeline
    2. Writing Validation Status to a Status File :
        code : with open(self.config.STATUS_FILE, 'w') as f:
                f.write(f"Validation status: {validation_status}")
        Validation status : T or F

Step - 10

1. Loading raw dataset (Samsun - dialogues and their summaries)
2. Tokenising input and target texts
3. Converting them into numerical tensors (input_ids, attention_mask, and labels)
4. Saving the processed dataset for model training
5. Steps :

    1. Tokenizer initialisation
    2. Convert egs to Features
    3. Mapping the Transformation
    4. Saving Transformed Data

6. Used hugging face AutoTokenizer to convert text into input_ids, attention_mask, and labels
7. Numeric I/Ps ; pre-trained model's vocabulary ; preprocessing tym saved ;
8. Challenges : variable length (truncation or padding) ; mismatch (btw tokeniser & model) ; out of vocabulary ; maintaining alignment btw I/Ps and O/Ps

Step - 11

1. Model Training : feeding the preprocessed (tokenized) data into a ML model
2. A Transformer (Seq2Seq) model (used)
3. Steps :
    1. ModelTrainerConfig (for hyperparameters aur paths)
    2. ConfigurationManager (automatically reads all settings : config.yaml and params.yaml then wraps in ModelTrainerConfig object)
    3. ModelTrainer Class :
        1. Load Model & Tokenizer
        2. Prepare Data (data transformation stage)
        3. Data Collator (batching and padding handle)
        4. Training Arguments (num_train_epocs , batch_size , warmup_setps , weight_decay , gradient_accumulation_steps , eval_steps/logging_steps)
        5. Trainer Setup (training loops, evaluation , logging , model saving)
        6. Stat trainng (predicting summaries from dialogues , gradients to update wts)
        7. Save model & tokeniser

4. The model trainer stage is responsible for fine tuning a pre trained transformer mode (pegasus) on the samsum dataset , used hugging face's trainer API , the training pipeline loads the tokenised data (prepared In the data transformation stages), initialises the model and tokeniser , set the training hyper parameters , adn trained the mode;
5. Trainer handles the full training loop - Forward pass, loss calculation, back propagation, optimiser updates, and evaluation

Step - 12

1. Model evaluation : see above def
2. Evaluate how good my trained summarisation model (pegasus fine tuned on samsum dataset) performs - i.e. how close the machine generated summaries are to the human written summaries.
3. Steps :
    1. Configuration Setup (in config.yaml) , this makes pipeline modular and reusable , instead of hardcoding parts , I used a config system , so all stages follow the same organised structure
    2. Loading model , tokenizer & data
    3. Generating predictions in batches : since datasets are large so will process them in small batches to avoid memory overflow , num_beams = 8 to ensure the best possible summaries , length_penalty=0.8 → discourages overly long summaries , max_length=128 → limits output length.
    4. Computing ROUGE Metrics :

        1. ROUGE (Recall Oriented Understudy for Gisting Evaluation) is a set of metrics used to evaluate the quality of machine-generated summaries
        2. It compares overlap btw : words or sequences in generated summary vs human summary.
        3. Each metric produces : Precision , Recall , F1 -> harmonic mean of precision and recall
        4. Types :
            1. ROUGE-1 : unigram overlap , measures many individuals words match - 0.43-0.47 mine
            2. ROUGE-2 : bigram overlap , measures fluency / phrasing - 0.20-0.2 in project
            3. ROUGE-L : LCS , measures overall structure similarity - 0.36-0.38
            4. ROUGE-Lsum : Variant for summaries , adapted for sentence-level summaries
    5. Saving Evaluation Results
    6. ModelEvaluation Pipeline , so that the pipeline can be called directly from main.py
4. ROUGE scores are high , it means model's summaries are close to human written ones showing good performance.

Step - 13

1. Created 2 main part : prediction.py -> handles text summarisation prediction , app.py -> exposes model as a web API using FastAPI (with endpoints for training and prediction)
2. Prediciton.py :
    1. This file creates a pipeline to summarise new user text using the fine-tuned Pegasus model.
    2. Working :
        1. ConfigurationManager() -> loads tokenzier for encoding user text
        2. AutoTokenizer -> loads tokenizer -> loads tokenizer for encoding user text
        3. Pipeline (summarization) -> loads pegasus model for inference
        4. Predic() function -> takes text input , passes it through the summarisation model , returns the generated summary
        5. Did it : so that after training , I can use the model for real - time summarization without re-trainig it.
3. App.py :
    1. It creates a web based API where I can : train the model , predict summaries
    2. Working :
        1. FastAPI() -> framework to create web APIs in Python
        2. Unicorn -> lightweight ASGI server that runs the app
        3. @app.get("/") -> redirects to Swagger UI (docs)
        4. @app.post("/predict") -> calls predictionpipeline to summarise text
4. Why :
    1. To make text summarisation model accessible via an API
    2. This helps integrable AI models with front end apps, web dashboards , or mobile apps
    3. It demonstrates deployment and integration skills.
5. I created a FastAPI-based deployment pipeline where users can either retrain the model or send new text to get AI-generated summaries in real-time. The backend

uses the Hugging Face Pegasus model loaded through a configuration-driven prediction pipeline.

Step - 14

1. Project CI/CD Deployment on AWS : a Continuous Integration and Continuous Deployment (CI/CD) pipeline that automatically deploys text summarisation project on AWS cloud whenever I push new code to GitHub.
2. In simple words - automated the process of building, testing, and deploying project to the cloud using Docker, AWS ECR, EC2, and GitHub Actions.
3. Steps :
    1. Created IAM (Identity and Access Management) User in AWS , this user has limited permissions to access EC2(virtual machine) and ECR (container registry)
    2. Created an ECR(Elastic Container Registry) Repository : it is like a storage space for Docker images on AWS
    3. Created and Configured EC2 Instance : EC2 is AWS's virtual machine where app runs
    4. Dockerized the project : wrote a Dockerfile that defines how to package project into a docker container , docker ensures project runs the same everywhere - no dependency or setup issues.
    5. Setup Github actions workflow : this automates the CI/CD process when push to GitHub , main.yaml stages :
        1. Continous Integration (CI) : checks code and builds the Docker image
        2. Continuous Delivery (CD) : pushes the built Docker image to ECR
        3. Continuous Deployment (CD) : pulls the image from ECR and runs it on the EC2 instance
    6. Added Github Secrets : added confidential values (like AWS credentials) : AWS_ACCESS_KEY_ID , AWS_SECRET_ACCESS_KEY , AWS_REGION , ECR_REPOSITORY , ECR_LOGIN_URI .
    7. Connected EC2 as GitHub Runner : turned EC2 instance into a self-hostel GitHub Runner . So Github actions can directly run deployment commands inside EC2 machine
    8. Triggered CI/CD by pushing code
4. Summary :
    1. Dockerized text summarisation app
    2. Pushed docker image to AWS ECR
    3. Deployed app to AWS EC2
    4. Set up a full CI/CD production-ready for real-world deployment

Step - 15

1. Made an image on docker of my project
2. Pushed it onto docker hub
3. FastAPI : code :
    1. Import libs, prediction logic
    2. Obj (app = FastAPI()) , now when I visit http://loaclhost:8080/ , it redirects them to /docs- which is the FastAPI Swagger UI (an interactive web interface to test APIs).
    3. /train -> to trigger model training
    4. def predcit(text : str) -> use predictionPipeline to summarise text and return summarised text (summary)

4. Docker starts container with app -> unicorn runs FastAPI server inside that container -> the API becomes available at http://localhost:8080 -> FastAPI automatically crates documentation at /docs -> now I can train my model , predict summary
5. In short :
    1. Model : PredictionPipeline -> loads trained model + tokeniser
    2. Backend : FastAPI -> Creates web API for train & predict
    3. Server : Unicorn -> runs app localhost : 8080
    4. Container : Docker -> packages everything for deployment

What to say at interview start :

1. I built an end to end MLOps project : NLP Text Summarisation with MLOps.
2. It fine tunes a hugging Face Pegasus transformer on SAMSum dialogue dataset (abstractive summarisation), Evaluates it with ROUGE, and exposes realtime inference via a FastAPI.
3. The Whole pipeline - data ingestion , data validation , transformation , training , evaluation , prediction - is configuration- driven, logged, containerised with docker, and CI/CD ready.
4. I also automated builds and image publishing so the model can be deployed anywhere

Systematic Explanation

1. **Goal** : Convert long dialogues/ text to short, meaningful summaries without losing intent - using abstractive summarization (model generates new text).
2. **Core model** : Hugging Face Pegasus (a Seq2Seq transformer pre-trained for summarisation).
3. **Pipeline/MLOps focus** : make the ML work reproducible, configurable, logged, and deployable. Each stage is modular so it can be tested, rerun, or automated.

Thankuuuuuuuuuu…..
Deep , deep841,….