Heron test Kunal Kapoor

How to use written in the newReadme file

High level flow of system

Features

- In memory SQL lite DB to mimic a DB. With this style of information where you store meta information in a DB and the file itself in a file system (I chose s3) a non relational would make more sense but I found getting SQL Lite up and running easier
  - 2 tables
    - 1 for Customers and one for Files
    - Customer tables hold customer meta info and files hold file meta information
    - the actual file gets uploaded to s3
- github action to run tests on push to branch – normally this would run on push to a branch with an open pr to develop but I was not going to get bogged down on PR's
- Bag of words scikit learn model which predicts file type via text inside
- Factory class for text extraction. This base factory class provides a template I extend for multiple files types. Provides a maintainable and extendable way to add new file types to extract text.
- HTTP based service which has 3 endpoints
  - one to upload files
  - one to classify the uploaded file type
  - one to view the file in the db

User Flow

- the user will upload a file type to the /upload_file endpoint along with a customer id
  - this will write to an sql lite in memory db
- Then the user uses the classify_file endpoint to classify the file for a given customer_id and file_name
  - this will add the file classification value to the file object in the DB
- Then the user can view the file item via the view_file endpoint and see the classification added to the DB.
- Examples of these endpoints used can be found in the notebook in the notebooks folder and in the newReadme file

Asusmptions

- every customer you add a file for exists
- I imagine a user would log into some system before uploading a file so they would already possess a valid customer ID
- Legal right to store files. Since these are financial documents I assumed we have permission to store these files but if not you would simply have a worker that periodically runs and deletes all files from s3 or simply not even save the files to begin with
- Assume we want other things from the file so by having the meta information in a DB with the path to the file we can extract other information from the file and save to the files DB

DB Schema

- customers tables stores customer meta information
  - id

    - name
- files table stores file meta information
    - id
    - filename
    - S3path
    - customerId: foreign key from the customers table
    - fileClassification: default null but is filled with a value post classification

Scale

- Using Fast API you can make async endpoint easily enough
- The upload file endpoint in prod would be writing to a queue and then a worker can pick up this job and upload to the DB and s3 to avoid hammering the DB if we got a burst of 1000s of files at once.
- The view file endpoint is purely IO for reading a file so this could be fully asyncd and multithreaded so we could have 1000s of hits to this endpoint concurrently as long as your thread count is limited to not overload the DB or you use db pooling connections to limit your db connections.
- The upload file and view file endpoints are simply put helper functions here (in prod these would break the service boundary of this service so would be placed in a separate API altogether.
- The classify file endpoint will be CPU based since a lot of its work is extracting text and classifying so this is pointless to try to multithread. For this you need parallelism so with your web server (gunicorn, uvicorn) set up multiple workers so that specific web server instance can handle n parallel requests.
- The classify file endpoint would be deployed as a container into some highly scalable service which can handle parallelism. Depending on cold start issues we can use AWS lambda container based deployments which is cheap and managed but if cold start is an issue a container orchestration system such as ECS using an auto scaler or Kubernetes would be ideal.
- I felt like there was no point in doing a deployment to AWS ECS for this task as I would have just got bogged down and wasted time purely for an accessible endpoint.
- In terms of classifier scalability, you would need to add more data from different file types to the training set and then retrain to extend it to other file types.

Deployment

- For simple deployment I would use github actions for continuous testing and then also for CD.
- We could have an ECS task running and when a push to master happen we trigger the CICD workflow to update the task running.
- In prod though, eployed as a container into some highly scalable service which can handle parallelism. Depending on cold start issues we can use AWS lambda container based deployments which is cheap and managed but if cold start is an issue a container orchestration system such as ECS using an auto scaler or Kubernetes would be ideal.
- Since we use a containerised system for this problem deployment is quite easy

Code Features

- Maintainability
    - use of functions, comments, doc strings, validation classes, type hints, tests should allow someone to be able to work on this system and maintain it.
    - Extendability
        - use of an ORM allows us to extend this service to a different DB and easily integrate with more tables using data classes

        - use of factory base class for text extraction gives us a simple pattern to add text extraction capability for more and more file types.

Classifier

- bag of words model via scikit learn
- we remove stop words, punctuation from text
- get unigrams, bigrams and trigrams from the text
- Using tf-idf we look for document frequency rather than pure count
- train a classifier on our given labels to predict text
- very basic method of text classification. Some much more modern neural network based approach would probably be much better at scale.
- code can be found in the notebook in the notebooks folder

Testing

- I tested the text extraction and test classification
    - with a variety of files and then variety of text
- I test the app as well to see how it behaves under different circumstances
    - request to classify non existent file in the DB
    - request to classify a file not in s3 yet
    - successful request for a file classification
    - I used mocking here to mock db and s3 actions
    - In prod I would set up some local testing db and somehow have a local mock s3 but I got bogged down on this and did not feel it worth it for the task.

What I would do differently in prod setting

- not upload env file to github, this was for simplicity sake, in prod setting this would be stored in s3 , or as separate env vars in a parameter store like aws parameter store/secret store and injected or pulled down at run time
- would use credentials/IAM role to manage s3 reading and writing, the bucket I use for this test is public so I did not need credentials to use it
- more modern technique to classify the file using up to date ML techniques.
- Split the CRUD endpoints into their own api for CRUD work.
- Any writes to the db be done using a queue as a buffer to ensure no spamming of DB connections
- Authentication on the endpoints.
- Increased validation on the endpoints
- deployment of the model api via a container orchestrator service which can scale up or down depending on load.
- I would also probably extract on file upload and place this alongside the raw file as a txt file in s3 so when classifying you don’t need to extract information and save time in classifying. You can also catch any errors before a file is uploaded so your customers don’t get a half working system but instead get a guard railed system which will alert you to erroneous situations.