Marktguru Senior Data & GenAI / ML Engineer

Lazar Gugleta MSc

Agenda

01. Introduction

02. Architecture

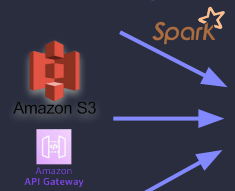
03. Prototype

04. Conclusions & Future work

Architecture overview – ML ETL pipeline

Lazar Gugleta MSc

Data ingestion



INGESTING RAW IMAGES INTO THE PIPELINE

- Start first Prefect flow and task
- Fetch last 24 hours worth of images
- Load the data using PySpark into memory for metadata
- The images directly stream from S3 to DataLoader for processing

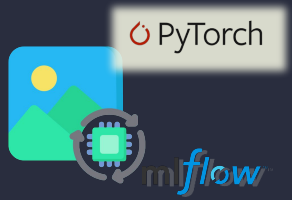
Consider:
- distributed processing
- speed and size of loading connection
- incremental loading if required

Preprocessing

PREPARING IMAGE FOR MODEL INFERENCE

- resizing of images to make model processing efficient
- normalize the images

Enrichment



USE ML/GEN AI TO IMPROVE DATA EXTRACTED FROM THE IMAGE

- make predictions using ML models self-hosted or API (depending on scale)
- classify product in image
- provide description of the image
- extract individual products/detect ingredients

Storage

APPROPRIATE FORMAT AND DATA TO STORE

Define formats for storing the feature embeddings, model inference results and other metadata from before.
Using S3 Bucket and Delta Lake for such data.

Downstream usage

FIND USAGES FOR THIS DATA

- Improve the app with better recommendations, search functionality, visual matching search to find household products in current deals
- Price tracking and history show to user (reference to keepa for amazon)
- Use processed data as a separate product or introduce new pipelines on top of it

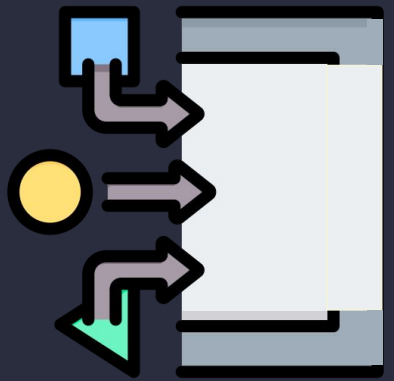
PREFECT

Prometheus

To comprehend the full extent of this example we introduce two steps before the pipeline:
1. User sends a photo through interface over API (can be self-hosted or aws api gateway for easier scaling and no maintenance)
2. Store raw data into the S3 bucket using a serverless function along with simple validity checks and storing them
With proper format/path (e.g. images/year/month/day/image_id)
3. Every step of the pipeline can be separated into Prefect flows and every responsibility can be a separate task belonging to that flow to keep it modular
4. Monitoring using prometheus and mlflow, more on that later
5. To make this prototype production ready we would have to put this prefect orchestration setup into a kubernetes cluster so that each instance can handle it's own batch for distributed processing

Data ingestion

- If we consider a full extent of this real life example user must upload this image first using the interface, behind it is the secure API set up to send the image to the store such as S3 bucket
- By using the serverless function we can receive the file, validate it for simple rules like acceptable type and size (if there is a need it can be extended with more rules for advanced checking)
- Important is to consider security issues and possible attacks through user input here. Rate limiting and checking first few bytes are good first steps
- Raw images along with generated unique id are stored in AWS S3 Bucket in a structured path format (can be with year-month-day folder or separated by year/month/day)
- Here the triggered daily prefect orchestration starts the first flow and task of finding out which were the images uploaded during the last day by querying the storage
- PySpark is used to keep track of the metadata, lists of images and paths and is good for distributed processing
- Next prefect task is to fetch these images as files from S3 and load them into memory using smart_open and get them ready as tensors with DataLoader class in pytorch
- Images are ready for preprocessing flow



Tools/Frameworks/Services:
S3 bucket, PySpark, Prefect, Pytorch

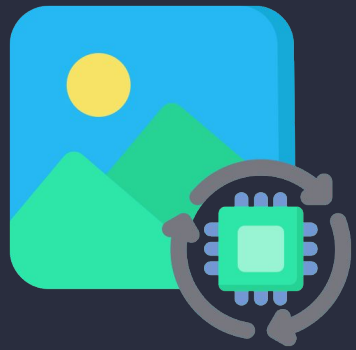
Preprocessing

- Batches of images are passed to the next flow of prefect, which is preprocessing using pytorch vision
  Preprocessing steps:
1. Image resizing to size of 224x224, making it easier for model to process with enough information for good inference
2. Picture normalization by averaging the pixel values for RGB ranges to adjust for the model and by reducing values hence models perform better
- Images are now ready to be enriched in the next flow
- Depending on the specific data, this step can be extended for further preprocessings

Tools/Frameworks/Services:
Pytorch, PySpark, AWS resources, Prefect

Enrichment

- We come to playground of ML where we can extract additional information about the images by using self-hosted models / calling an API to get classifications of images, recognizing objects
- Approach here can be very efficient to make inference on self-hosted models using SageMaker like ResNet-50 and collect: classification labels, embeddings, accuracy scores, description of the objects/scene (other models are also possible like clip by openai and vit by google)
- We obtain these results in the PySpark dataframe and send it directly to next flow of storage
- Using self-hosted models can save money in the long run (as the pipeline scales accommodating a lot of images) but using an API can provide faster ramp up of the pipeline
- Let's break it down for processing 5k images per day:
- **Self-hosted**: T4GPU processing ~30 min to for 5k images
  ~€0.15 - €0.50 per 5k images -> 15€ monthly
- **API Calls**: With batch discount and low resolution processing uses 200 tokens:
  $0.0022125 per image hence -> 300€ monthly
- API Calls can be a good starting point but for scaling the self-hosted models with more flexibility and benefit of being able to train them on specific data is a way to go



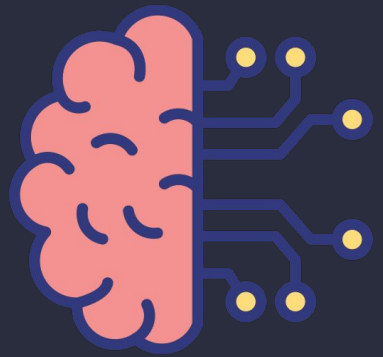
Storage

- For model inference results and feature embeddings we use Delta Lake on S3 (schema enforcement and acid transactions), which is capable of storing such data efficiently and everything goes directly from PySpark along with the initial metadata we already have
- Store processed image separately in S3 bucket
- Another benefit of this approach is immediate preparation for downstream usage meaning that embeddings, classification labels etc. can be used by other pipelines/models and ML applications
- Could be improved by using Feast as feature store for even faster processing

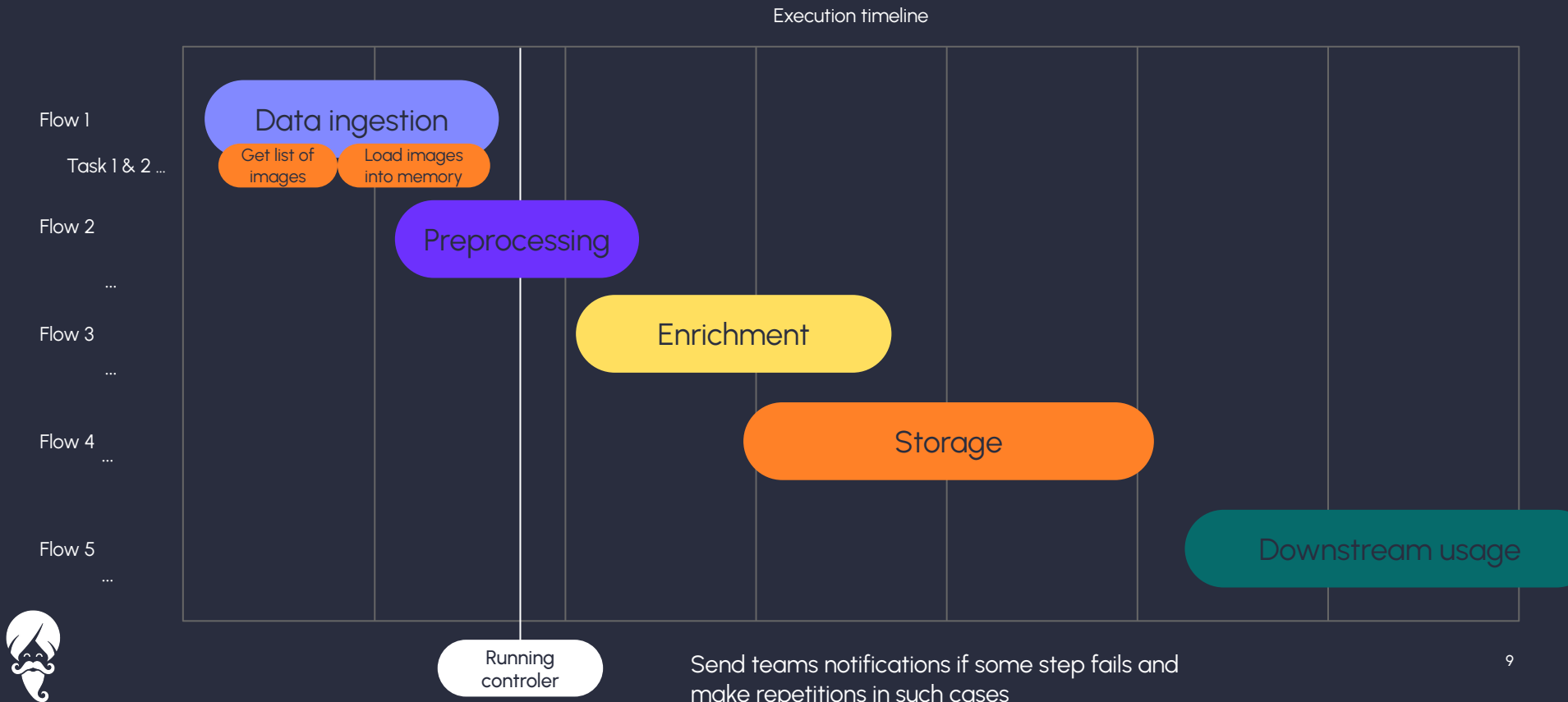
Tools/Frameworks/Services:
S3 Bucket, Delta Lake (possible Feast to use), PySpark

Downstream usage

- After extracting information from images we can build further ML applications
- One example would be Business Intelligence application with visualization dashboards in Tableau showing possible insights into data that is being processed to have impact on the business
  These dashboards can show popular products and from there the trends compared to the price would also be interesting
- Another application of the processed data can be semantic search by using enriched metadata embeddings extracted from the images
- Users can go into detailed search of such products
- We can also improve app flow by enabling visual match search making it less friction to find specific product and current deals with the best price
- Weather it is internal or external use of this data, we can produce value in many places and that is the whole idea behind this -> general application for business use



Prefect overview example



Send teams notifications if some step fails and make repetitions in such cases

Databricks setup ETL pipeline

Create new ETL pipeline in the workspace and enable serverless functions

Data ingestion

Databricks Volumes offers storage along with SQL capability to fetch and store data.

Automatically trigger scheduled workflow by using Databricks scheduler.

List all images from previous day using Spark and load them.

Preprocessing

Spark cluster is for preprocessing batch of images.
Easy integration of notebooks directly provided by Databricks to execute code.

Enrichment

By using Spark ML we keep the environment in databricks and have access to powerful pre-trained models using Databricks model serving and deployment.

Storage

Write results to a Delta Lake using Spark and store processed images in Volumes.

Downstream usage

Use other types of pipelines to get insights from data directly in Databricks or Spark ML

Databricks setup  – ETL pipeline

Pros:

- Databricks offers all capabilities for ML pipelines in one platform, whereas Prefect only orchestrates jobs and requires other services such as AWS object stores, kubernetes setup and more engineering/maintenance effort
- Good for processing huge amount of data at scale using Spark in a distributed manner
- If already part of the existing environment and we can build on top of it
- Has other features included like monitoring, pipeline templates

Cons:

- Prefect kubernetes setup has more flexibility into configuration of usage of clusters and computational power
- Vendor lock in and cost control

Monitoring

- Going with Prometheus to observe underlying infrastructure like prefect kubernetes cluster and to be combined with mlflow tracking for model underlying metrics
- Such monitoring allows insight into unreliable outputs and can help us determine root problems
- Prefect has it's dashboard and connecting to different services such as microsoft teams to send notifications is straightforward
- Acting early and including other techniques to prevent problems can save costs
- Databricks has all kinds of monitoring already built in (data lake, unity catalog and query usage etc.)

Combine ML/Gen AI with Software engineering

CLEAN CODE

Software engineering is a key practice that makes code scale and work for a long time.
In order to achieve clean, modular and readable code we must respect rules like DRY principle.
Making functions have one concern and break them down makes the code modular and decoupled.
Dependency management is crucial step into making code clean.

SCALE

Software engineering is not only writing code but considering architecture of the system, inputs and outputs, build system, tooling for all etc.
When starting such a project we have to consider current as well as foreseeable future requirements.

TESTING

Testing is also part of trunk based development in which there are small increments done to move fast and test properly.
Making code modular makes it easier to test.
Writing clear requirements also makes this process much easier when there are clear intentions.

Scaling, cost-control and reliability

SCALING

Storage: S3 Buckets are auto scalable to "infinity"
Orchestration: in kubernetes setting we can scale the pods for large batches and instantiate them as required without problems (if prefect cloud is used this is done automatically)
Tasks in prefect are modular making them decoupled in a clean architecture
Model inference: API does not scale as well as the self-hosted models because of cost
Serverless components also scale without issue on AWS
Databricks: scales out of the box for every step of the pipeline and requires less overhead for maintenance

COST-CONTROL

Storage: S3 Buckets data can be moved to glacier after certain amount of time to save costs
Not costly in the first place but efficient storage of such data
Delta Lake and features embeddings storage is also efficient for the cost topic
Orchestration: If prefect is deployed on kubernetes we pay for raw AWS resources such as vms and can be a bit cheaper for pure usage but requires engineering effort and maintenance
Model inference: API good to start with but self-hosted model is much more affordable as the user base grows
Databricks: a bit more costly compared to prefect setup because of all the features it offers and less insight into computational control but reduces time and effort of maintenance

RELIABILITY

Storage: Reliable as it is a pay to use service, AWS promises high level of availability
Orchestration: Flows and tasks and equipped with retries and data keeping mechanisms as well as acid transactions for out data
If setup correctly can provide flexibility and reliability in prefect kubernetes setup but requires engineering efforts and knowledge of cloud services
Model inference: Needs monitoring of results to keep it stable
Databricks: Reliable in all steps of the pipeline as it is a pay to use service, less maintenance and space for error

Next steps

01.

Make a prototype pipeline with short ramp up time but not sacrificing the code quality and always thinking about standard practices and costs.

02.

Observe how it acts in production, monitor resources and results you get. Iterate and optimize steps of the pipeline accordingly.

03.

Build upon this pipeline other ML apps and produce value for the business.

Downstream usage coding example - Part 2

For this example I chose to do a image classification task using Polars, Pytorch, to show one example of ML Applications for a classification task

- Input: Load the dataset from huggingface using API and Polars integration (very easy to use and blazing fast as it uses lazy evaluation directly on hugging face file system and has other polars benefits)

Preprocessing

- Unnest image column to find image data as binary and paths in string. Only keep the binary data for the image and leave path column out for model processing
- If this was a full example we could keep the path for later storage
- Label column can be converted to int8 to save memory (does not make much difference in this example but in large scale processing checking the data type and converting can save a ton of memory)
- Downsample the dataset to 5 classes and split 750 images in the training dataset and 250 for validation of each class respectively
- Normalizing the images and converting them to tensor is the essential part of preparing images for training and validation (makes it faster to process)

I ran the notebook using kaggle because of the stronger GPU and faster processing times.

Downstream usage coding example – Part 2

Model training

- I chose the pretrained resnet-50 as ideally fast and precise for this task Essentially we fine tune this model on food-101 dataset
- The model initialization is done by choosing a optimizer and loss functions
- Adapting to the task at hand which is classification a common choice is adam optimizer and cross entropy loss (this adjusts model parameters during training to minimize loss)
- We set multiple variables during training of such a model: Number of epochs, loss, learning rate, weight decay and batch size
- During training we monitor loss to see the trends and in this case it is decreasing steadily meaning model is learning efficiently during each epoch (per training set pass) (from 0.8100 (Epoch 1) to 0.0303 (Epoch 20))
- Learning rate I chose a standard 0.001 as the model had no issue of convergence speed at 20 total epochs
- Before introducing weight decay model was showing great results, but after it was introduced it showed even more stable results indicating it prevents overfitting of the model

```
Epoch 1, Loss: 0.8100
Epoch 2, Loss: 0.5248
Epoch 3, Loss: 0.3953
Epoch 4, Loss: 0.3394
Epoch 5, Loss: 0.2375
Epoch 6, Loss: 0.3080
Epoch 7, Loss: 0.2057
Epoch 8, Loss: 0.1434
Epoch 9, Loss: 0.1440
Epoch 10, Loss: 0.1168
Epoch 11, Loss: 0.1314
Epoch 12, Loss: 0.0773
Epoch 13, Loss: 0.1164
Epoch 14, Loss: 0.1383
Epoch 15, Loss: 0.0821
Epoch 16, Loss: 0.0696
Epoch 17, Loss: 0.0884
Epoch 18, Loss: 0.0809
Epoch 19, Loss: 0.0537
Epoch 20, Loss: 0.0303
```

Downstream usage coding example – Part 2

Results

- Final accuracy on the validation dataset got me 87.68%
- In the spirit of showcasing an example, this accuracy is sufficient
- When talking of improving we can take many steps to further improve this (discussed later)

Evaluation

- Accuracy to show how good the model performs on the validation set in combination with recall and f1 score

Test Accuracy: 0.8768

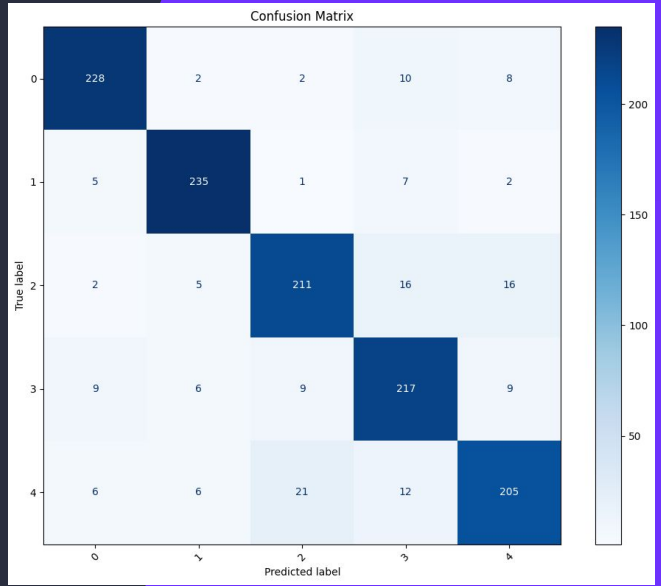
	precision	recall	f1-score	support
0	0.91	0.91	0.91	250
1	0.93	0.94	0.93	250
2	0.86	0.84	0.85	250
3	0.83	0.87	0.85	250
4	0.85	0.82	0.84	250
accuracy			0.88	1250
macro avg	0.88	0.88	0.88	1250
weighted avg	0.88	0.88	0.88	1250

Downstream usage coding example – Part 2

Unreliable outputs/hallucinations

- RAG approach where after finding base ingredients of the identified dish in the image we enrich the data by providing the ingredients of it to limit the scope
- Class imbalance: I looked at the confusion matrix to find out which classes are showing less accuracy than others
- Food-101 dataset mentions there is noise in the training dataset so this could be analyzed and improved during preprocessing to have cleaner separation of classes in the classification
- Possible approach is to feed images with unreliable results into a different model (or make api call) or include human examination for improved results down the line (find such results by confidence thresholding)
- I chose 20 epochs after many experiments seeing the model overfits after 20 epochs at this learning rate
- This can be found out by hypertuning methods of making combinations between learning rate, batch size, weight decay and number of epochs
- After performing this step I found better parameters (lr: 0.0001, wd: 0.0001, bs: 64) and achieved accuracy of **96.16%** *(end of the cell before last in the notebook, in the last one I accidentally left two epochs instead of 20)*
- All F1-scores are between 0.94-0.99 stating that it is a stable balance of precision and recall

No class is significantly underperforming, quite opposite they are performing very good
Loss is steadily decreasing and I find no signs of overfitting, although it is low in the end



Downstream usage coding example – Part 2

Conclusion & Next steps

- This is a small example to showcase the possibility of powerful models doing ML tasks
- Showing modularity and clean code
- A lot of these functions can be further extended with parametrization but for this example there was no need
- Wanted to show some of the best practices but even this can be further improved
  1. Introducing optional and mandatory parameters
  2. Observing the abstractions and putting them together cleanly etc.
  3 .Good error handling with clear statements of what went wrong
  4. Repetition of steps (prefect has that)

Thank you