

ML training cycle for production

Thanos Tagaris

Premise

- We are working in a manufacturing company
- Management wants to better organize the warehouse inventory.
- One requirement for this is creating a service for **demand forecasting**.
- We are called to **design** and **implement** this service

High-level design

- A **ML model for forecasting** will be the central component of this service.
- The model will be deployed as a **REST API**, which can be used by any other service needed.
- The model needs to be **retrained** periodically to also include new items, pick up on new trends, etc.
- A proper design of this service will also take into account that in the **future** we might want to **experiment** with different models, hyperparameters, preprocessing techniques, etc.

Requirements (informally)

1. Model

- target forecasting accuracy on past data

2. Service

- model deployed as a REST API
- latency, throughput, resources, etc.

3. Pipeline

- data: gathering, preprocessing, validation
- model: training, validation, deployment

4. Experimentation

- framework for experiment tracking and model registry
- framework for offline experimentation (i.e. on past data)
- framework for online experimentation (i.e. A/B testing)

Experimental framework

- If you've ever experimented with ML models you've experienced that after some point it's extremely hard to keep track of the experiments...
 - what models/hparams have been tested
 - which datasets they were evaluated on
 - what metrics were used
 - if you find out a bug, which experiments are invalidated?
 - all these magnify if you resume a project after a significant amount of time
- To solve these problems we need a proper experimental framework to keep track of everything.

Experimental framework



- Frameworks like mlflow help with this.
- With little code overhead, you get
 - full experiment tracking (model, hparams, dataset, metrics, etc.)
 - a UI to compare experiments
 - a model registry with versioning (i.e. you can see what experiment produced each model, their respective metrics, etc.)
- Some frameworks even offer dataset versioning (though this is hard for larger datasets)

mlflow UI

[Github](#) [Docs](#)

Listing Price Prediction

Experiment ID: 0


Artifact Location: /Users/matei/mlflow/demo/mlruns/0

Search Runs:

Filter Params:

Filter Metrics:

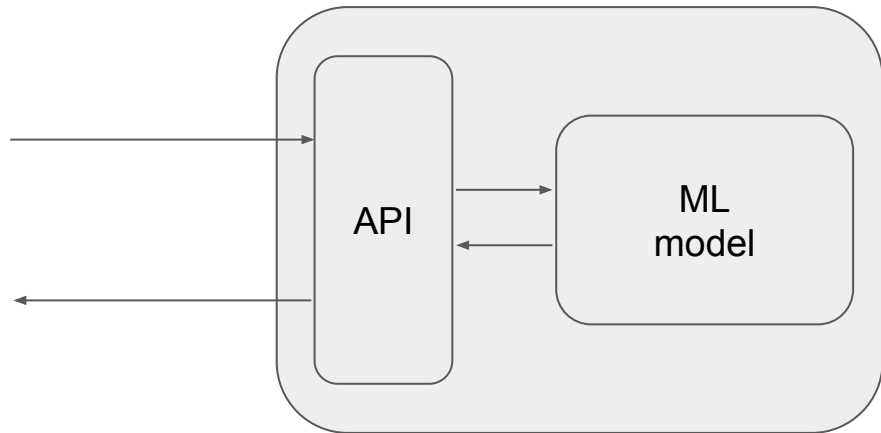
4 matching runs

	Time	User	Source	Version	Parameters		Metrics		
					alpha	l1_ratio	MAE	R2	RMSE
<input type="checkbox"/>	17:37	matei	linear.py	3a1995	0.5	0.2	84.27	0.277	158.1
<input type="checkbox"/>	17:37	matei	linear.py	3a1995	0.2	0.5	84.08	0.264	159.6
<input type="checkbox"/>	17:37	matei	linear.py	3a1995	0.5	0.5	84.12	0.272	158.6
<input type="checkbox"/>	17:37	matei	linear.py	3a1995	0	0	84.49	0.249	161.2

Service

- Consists of a REST API (e.g. implemented through FastAPI), a trained ML model and optionally caching.
- The API listens to a certain port and specific endpoints
- When a request arrives from a user, the model will be called to produce a prediction, which will be returned to the user.
- KPIs:
 - **latency**: how long it takes for the service to respond to a request
 - **throughput**: number of requests the service can serve in a given amount of time
 - **resources**: how many CPU/memory/storage does the service take up
 - **availability**: what percentage of the time is the service operational



Retraining: conception

- What is the need of retraining? Can't we train a model once and let it serve forever?
- In certain cases it may be possible, where the data might not change through time.
- However in most cases you should, due to **new items** and **data drift**.
- In our example
 - we might have new inventory items that have not been seen before
 - older items that the model has been trained on might have been discontinued
 - the demand of each item might change through time, so the model's performance will deteriorate through time
- To account for this we will need to periodically retrain the model on the newest data available.

Retraining: implementation

There are two main steps:

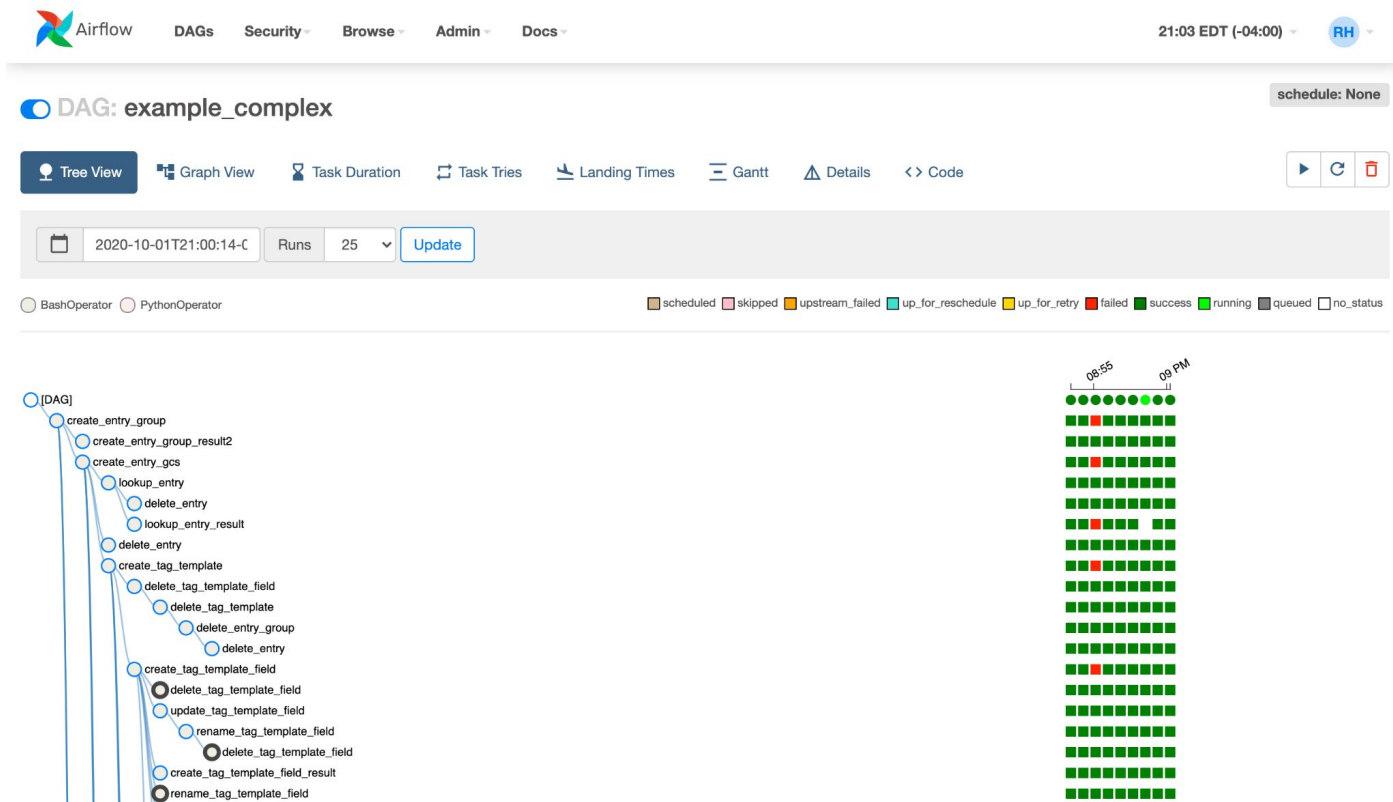
- **A data pipeline**

- **Ingestion:** gather the data from their respective sources
- **Processing:** join the different data sources, resolve inconsistencies, fill missing values, extract features, etc.
- **Validation:** make sure the data satisfies some quality standards

- **A model training pipeline**

- **Training:** train the model on the newest data
- **Validation:** evaluate the model to make sure it satisfies some standards (i.e. validation metrics)
- **Deployment:** replace the previous model with the new one without downtime

Airflow

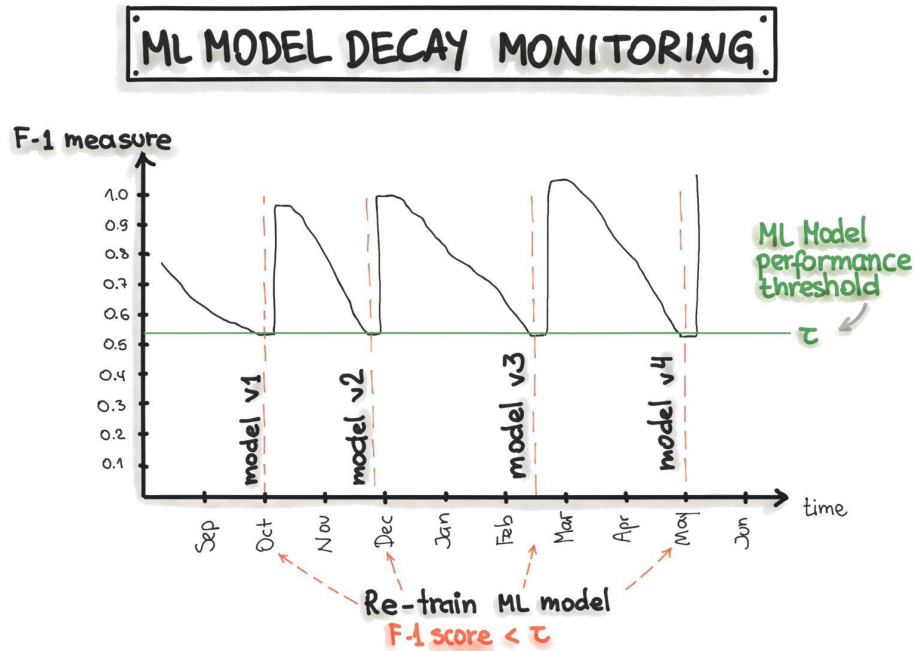


Data/Model Validation

- Since the retraining process is automated, there is a chance that something will go wrong.
- To ensure that it won't, we usually place several validation checks.
- In a **data** level we might want to make sure
 - we have enough data
 - there is adequate representation in several segments we are interested in
 - data is not imbalanced (much more than usual)
 - data has not drifted significantly
- In a **model** level we might want to make sure
 - the validation metrics (offline) of our model exceed some thresholds
 - or they haven't deteriorated significantly from previous trainings

Questions about retraining

- How often to retrain?
 - depends on how rapidly the model's performance deteriorates
- What if we don't have enough data?
 - we don't need to replace the old data with the new ones, we can train on a union of older and newer data



Model evaluation

There are two types of model evaluation

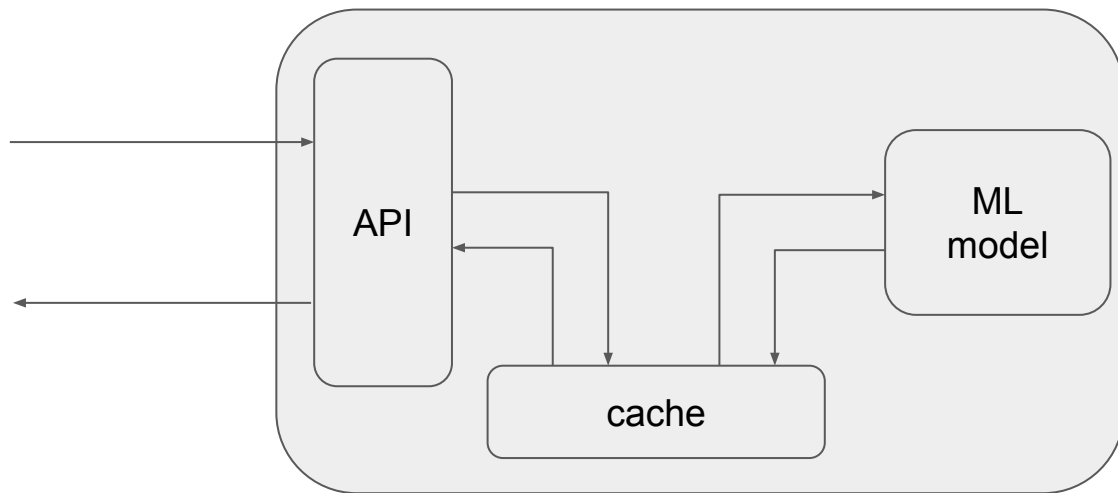
- Offline
 - what we're most familiar with
 - use a validation set to evaluate our model on past data
- Online
 - evaluate a deployed model on real-time data
 - this is much more direct as we can evaluate models on real data and not only on ML metrics but also business KPIs
 - can compare models with A/B tests

Serving with large models

- Large models inherently have slower inference speeds
 - ... and as a consequence smaller throughput
 - ... so if we need to serve many requests we'll need more and more resources
- There are a few techniques we can use to make our service serve faster, without much impact in its performance.

Performance: caching

- If a lot of requests are arriving with the same exact data, we can cache the model's response instead of asking it to make a new inference



Performance: quantization

- Quantization is a technique to **reduce the precision** of the weights of the model to make it **require less memory** and **serve faster**
- e.g.
 - Models are unusually trained with 32-bit float weights. A sample weight: 0.47330648
 - If we reduce the weights to 16 bits the memory requirements of the model will drop significantly. The previous weight 0.4734
 - This reduction shouldn't have significant impact to the model's capabilities
- In some cases we go as far as 8-bit quantization

Performance: adapters

- This technique is more commonly found in LLMs where we have huge model sizes.
- In this context we have a large *base* model (i.e. a foundational model) that then might be fine-tuned for several purposes (the most common: instructions).
- Instead of training the whole model, instead train a small portion of the weights.
- If we require multiple trainings of this model for different contexts, instead of many large LLMs, we will have 1 large model and many smaller adapters which are used to *adapt* the model to the different contexts.

Performance: other techniques

- **Knowledge distillation:**

- *distill* the knowledge of a larger model into a smaller one
- e.g. DistillBERT is 40% smaller and 60% faster than BERT while retaining 97% of its natural language capabilities!

- **Weight pruning:**

- prune some of the weights of the model that are found to have little impact in its decisions
- can significantly reduce the size of a network (even up to 90%) and also improve latency

- **Weight clustering:**

- group individual weight values within a layer into different clusters, through the use of a clustering algorithm
- the weights within each group are all represented by a single value