**Section 5**

Sezione che spiega a modo come dovrebbe essere formattato un progetto in produzione, cerca di seguire le naming convention.

Usa pytest per I test (i file devono cominciare con test\_, e lanciando pytest te li lancia tutti. In un file setta le fixtures da passare in input ai vari test.

Puoi usare libreria pydantic per validare I tipi delle variabili.

Cose invece come flake8, isort, mypy ecc.. fanno automaticamente il check per te di cagate tipo linting e altro (spazi, formattazione ecc..)

Usa tox (installalo e configura tox.ini) per lanciare velocemente I tuoi vari test e testare diversi settings. Comando:

tox run -e test\_package (nome del pacchetto da lanciare)

Per buildare un package python, crea il setup.py, il MANIFEST.in e il file .toml, poi lancia:

python -m build

**Section 6**

tox run -e run

E’ settato per fare un python main.py che lancia un fast api server. Di solito faresti:

uvicorn main:app –reload

Da lì se fai /docs hai autocamaticamente accesso allo swagger che ti documenta le varie api.

Fat api è veloce come django, asincrono, può produrre documentazione delle api in automatico e gestire I tipi di dati che gli passi (gli passi I vari schemi che validano I dati, magari con anche esempi, e hai il tuo swagger già popolato.

Quando fai /predict, usa il modello che hai buildato prima (mettilo nei requirements.txt, poi importerai tutte le varie funzioni che hai creato prima)

**Section 7**

Devi mettere un folder .circleci nella root del progetto con dentro un file di config.yml. Lì dentro puoi specificare dei job che saranno quelli che il tuo processo di CI eseguirà quando lanci una build. Ad esempio installi tox e tutto e lanci tutti I tuoi vari test sul modello con tox run. Nel file di config nei workflow specifichi poi che cosa lanciare e quando triggerarli (solo quando c’è una push sul branch di master o demo, solo con tag github ecc..). Ovviamente in circleci devi specificare le varie variabili di ambiente per connetterti ai vari ambienti (kaggle, railway ecc..).

Stessa cosa per deploy del modello, installi la cli e tutto e poi fai railway up –detach

Questo comando deve essere lanciato nel folder dove hai il Procfile (che fa partire il web server con uvicorn, stile flask o django) e I requirements.txt (che installano tutte le dipendenze stile docker container, devi chiamarlo esattamente così).

Da notare che nei requirements c’è il tuo tid-regression-model (funziona perchè il tipo ha rilasciato il pacchetto pubblicamente e o vedi nel pypi). Normalmente invece dovresti scaricarlo da un private index server (ad es. Gemfury, AWS bucket, a generic web server ecc..). Per farlo nei requirements.txt aggiungi una riga con --extra-index-url=${PIP\_EXTRA\_INDEX\_URL}. In questo modo se il pacchetto non viene trovato in pypi verrà scaricato da gemfury. Questa variabile di ambiente andrà messa dentro circleci.

Quindi tu fai modifiche sul branch di demo, pushi, vai su github e ti apri una pull request su master. A questo punto vedrai che partiranno I vari job di circleci (perchè nel file hai specificato che su quei branch partissero tutti I vari test della tua applicazione e anche il deploy su railway.

Il job di circleci invece che riguarda la pubblicazione di un nuovo modello su gemfury scatta solo se crei un tag (su github code 🡪 tags 🡪 create new release). In particolare tox -e publish\_model verrà triggerato. Ti servirà la variabile di ambiente GEMFURY\_URL, che dovrai andare a settare sempre su circleci.

Quando fai la build dell’applicazione in locale con Docker, dovrai passare --build-arg PIP\_EXTRA\_INDEX\_URL=<gemfury\_token> .Il token verrà intercettato e settato come variabile di ambiente nel Dockerfile (ARG PIP\_EXTRA\_INDEX\_URL) in modo che nei requirements.txt abbia questo extra url. Quando fai la build passando da crircleci (perchè hai pushato sul branch demo), il comando che verrà eseguito sarà railway up –detach (che avrà bisogno del procfile e dei requirements). Qui per non scoppiare quella variabile dovrà essere settata di ambiente in railway.

Un vantaggio è che se railway vede un dockerfile parte da solo a creare una docker image, molto comodo

**Section 9**

Il codice è un pò sminchiato, quindi a volte per seguire il corso vai di git checkout <hash\_specific\_commit> e fai le prove direttamente su quel codice.

Se vuoi testare i differential tests devi fare un checkout sullo specifico commit. Nel test che lanci (guarda lo yaml di circleci) aggiungi not differential oppure un’altra stringa per includerli. Tutti I vari test marcati con @pytest.mark.differential verranno o meno eseguiti.

Questi test praticamente servono per comparare il set delle predizione vecchie (lancerai un test che installa nei requirements un modello con una versione vecchia e uno script che carica tutta la baracca e salve le predizioni in un csv, seguito da un altro script che fa la stessa roba col nuovo modello e poi le confronta, e con math.isclose ti accerti che siano più o meno vicine (che I valori ad esempio non siano più distanti del 5%).

Vengono utili in situazioni in cui: 1) ti sbagli e deploy un modello sbagliato, 2) salti uno step del preprocessing, 3) prendi I dati sbagliati di training ecc..

**Section 10**

Aws ECS (Elastic Container Service) è pensato apposta per gestire container.

In pratica tu definisci un task (una specie di dockerfile), specificando il numero di container, risorse per container ecc.. poi ci sarà un service incaricato di eseguire quei task.

Di solito usi Kubernetes come container orchestration engine (self managed, puoi configurarlo come vuoi, ma il setup è complesso); l’alternativa è usare ECS (fully-managed by AWS, più semplice).

Go to the management console, IAM, to setup a user with the right permissions to IAM, EC2 containers and to the container registry. Be sure to create and save user admin access key, needed for auth (for example when using the aws cli). NEVER commit them to your repo (you can spawn EC2 instances with it!).

Le nostre immagini verranno caricate nel ECR (Elastic Container Registry), lo crei sempre dalla management console.

Ti buildi sempre l’immagine Docker in locale e lo tagghi con l’ID che vedi nel ECR che hai creato (l’extra index ti serve per installarti il modello custom che tu hai nel tuo registry privato su gemfury):

docker build --build-arg PIP\_EXTRA\_INDEX\_URL=%PIP\_EXTRA\_INDEX\_URL% -t udemy-ml-api:latest .

docker tag udemy-ml-api:latest %AWS\_ACCOUNT\_ID%.dkr.ecr.us-east-1.amazonaws.com/udemy-ml-api:latest

At this point you will have a local built image with that tag.

aws ecr get-login –no-include-email –region us-east-1

docker push %AWS\_ACCOUNT\_ID%.dkr.ecr.us-east-1.amazonaws.com/udemy-ml-api:latest

A questo punto hai la tua immagine docker nel container registry, e puoi usare in un ECS cluster (durante la creazione, vai su custom e specifica la tua image URI). Qui setterai anche la porta su cui dovrà essere in ascolto uvicorn (5000). Aggiungi anche la variabile di ambiente della chiave di Gemfury nel cluster (altrimenti non la trova nell’immagine).

Quando il cluster sarà stato creato, selezionando service e poi task, nei logs puoi vedere che uvicorn sta andando, e sotto se posti il public IP accederai alla tua API del ML.

Quindi hai qualche passaggio in più per l’IAAS (installa aws cli, setta permessi nell’IAM, setta ECR, pusha la tua immagine lì, setta ECS e farlo partire). Questo però ti permette più scalabilità, controllo, e costi inferiori.

Nuovo scenario ora fai modifiche alla tua immagine, buildi con docker in locale e pushi questa tua nuova immagine nel ECR. A questo punto (trovi le tue informazioni nel cluster ECS che hai creato sopra):

aws ecs update-service --cluster <name> --service <name> --task-definition <name> force-new-deployment

il tuo service verrà aggiornato con la nuova immagine con il tag latest.

Se vuoi automatizzare questo task (guarda nel checkout della repo), dovrai aggungere un job in circleci dove metti I comandi per installare l’aws cli, si logghi, buildi e pushi l’immagine, e fare l’update del service.

Per funzionare avrai sempre bisogno del AWS\_ACCESS\_KEY\_ID, AWS\_ACCOUNT\_ID, AWS\_DEFAULT\_REGION, AWS\_SECRET\_ACCESS\_KEY

Nel makefile lui ha specificato meglio I comandi da lanciare e I parametri da passare

**Section 11**

Ti costruisci il dataset come “path immagine” e classe da predire. Di solito dovresti sempre bilanciare il numero di esempi per target. Poi stampi alcune immagini di classi diverse per renderti conto delle varie classi che hai, e anche di esempi della stessa classe per vedere se/come cambiano.

Ovviamente non puoi predire una stringa, farai un ohe e diventerà una multiclass classification.

Another important thing: images need to be scale to the same exact size! (you use opencv for this)

Per la CNN, il dataset ha bisogno di 4 dimensioni: len(df), image\_height, image\_width, 3 (because is a color image). Quindi metti le tue immagini in questo dataset dopo aver fatto il resize.

Per capire come sta classificando, vai di confusion matrix.

For the code:

it is always better to put all the configs of your models ecc.. in a conf file, in order to have a centralized place where to look at.

Another file is for saving the model architecture. One function defines the layer and compiles the model, another function uses a wrapper (KerasClassifier) in order to make the model suitable for keras pipeline

In a preprocessor file, you put all the steps required, by creating classes with the methods fit and transform.

In pipeline file, you just define a Pipeline with the two steps required (load and transform the dataset, and create the model).

In a data\_management file, you load the dataset and put it in the desired format (with the two columns), you put the functions for dividing in train and test sets, and save / load pipelines.

In the train file, you will just load data,divide in train / test, encode the target, and leverage the pipeline to train.

To fix the seed for neural networks:

1) set seed for numpy (also for keras backend).

2) Force tensorflow to use a single thread.

3) In the code set PYTHONHASHSEED=0.

4) Set cuDNN as deterministic

random.seed(seed\_value)

np.random.seed(seed\_value)

tf.set\_random\_seed(seed\_value)

# Configure a new global `tensorflow` session

from keras import backend as K

session\_conf = tf.ConfigProto(intra\_op\_parallelism\_threads=1, inter\_op\_parallelism\_threads=1)

sess = tf.Session(graph=tf.get\_default\_graph(), config=session\_conf)

K.set\_session(sess)

For fixed cudnn:

THEANO\_FLAGS="dnn.conv.algo\_bwd\_filter=deterministic,dnn.conv.algo\_bwd\_data=deterministic" python yourKerasPythonScript.py

In packages/ml\_api/api/controller guarda come ha esposta una nuova api per questo servizio di dl