MLOps specifications document:

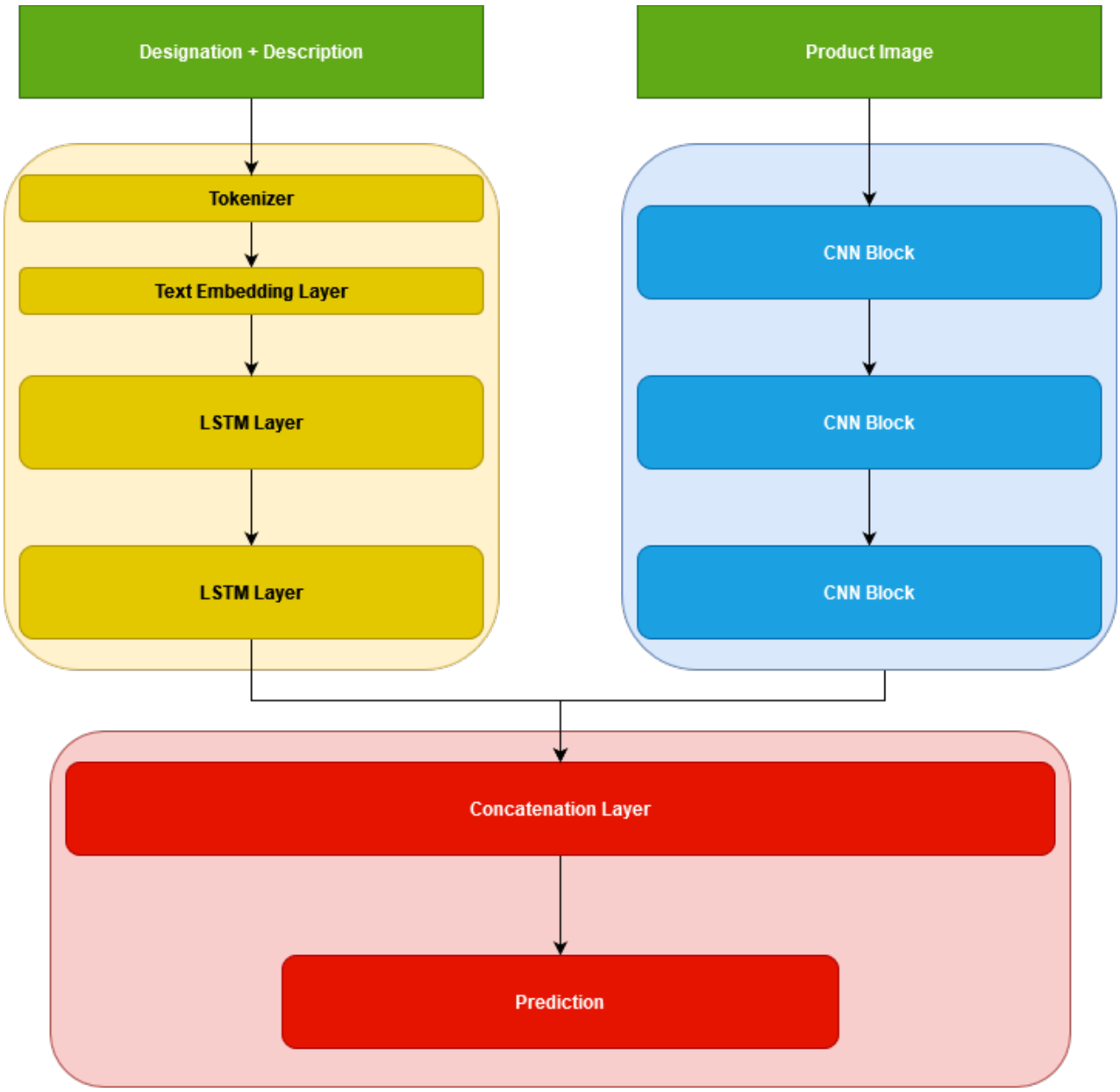
1.Context and Objectives

The app aims to classify e-commerce products of Rakuten France into their product types (for the project here we only use 27 classes not all of them). To be more specific, we want to develop an API (which might be hosted by a cloud server) based on deep learning models which can predict the product type based on an image or a text description or both. This API is called during the listing of a product by a seller. It can then be used to give sellers on the Rakuten website product type recommendations such that they list their product in the correct category. This can either be a single recommendation or multiple recommendations, i.e. we give the seller some options based on the likelihood to belong to a certain product type. In total, the API eases the listing process of a product and, hence, making the usage of the Rakuten website more accesible and easy.
It will be the responsibiliy of Rakuten to re-train and up-keep the API such that it is always working properly giving menaingful recommendations since it will be directly integrated in to their website (or accessed via their website). But it isn't necessary that the API is directly hosted on the Rakuten cloud servers.
As there are many similar product types (new books, second-hand books, comic books etc.), it can be quite challenging to predict the correct product type. Thus, it might be benefitial to choose the approach in which multiple options are given to the seller. This should result in a high likelihood that the right category is among them. Moreover, the information are provided by humans to the model so there will be human errors like using wrong pictures or text (HTML tags can also appear in the text due to automation or copy-pasting). All in all, we expect more difficulties in the model part than in the API part of the problem. Another important factor will be execution time of the model (for re-training and evaluation).
There are also some choices about the architecture of the API. For the authentication process we can use the Rakuten user database as every seller should be registered there. Furthermore, we can automatically add new products to a product database on which the model gets re-trained periodically. Here we have to find a solution to how the images are stored in a database, i.e. they can be stored as files or directly as numerical arrays/tensors.
The minimum launch includes a functioning API which can predict the category of a product based on an image or text or both to an user in a “fake“ Rakuten database (this will be just a manually populated database with normal users and admins). It should also include a montoring and re-training pipeline which implies that we have a product database with test and image data.

2.Model

Due to the possibility of providing only text or text and images we have trained three different models. In particular, these models are a Recurrent Neural Network (RNN) for the text classification, a Convolutional Neural Network (CNN) for the image classification and a fusion model which is a combination of the previous models. All of these models are evaluated according to their accuracy because that is the most relevant metric in our context.
The RNN is based on a standard Long Short-Term Memory (LSTM) architecture, whereas the CNN consists of multiple blocks of convolutional layers combined with max-pooling layers. The former achieved an accuracy of ~76% but the latter only realised an accuracy of ~48%. For the fusion model we took the architectures of our RNN and CNN and removed the classification head such that the two models could be combined with a new predictive part. In the end we managed to improve our accuracy to ~81% on our local machines. Depending on the specific preprocessing this number can vary, but this should reflect the potential of our model architecture.

Depending of the input of a user, i.e. just text or text and image, the different models are called.
There are some more metrics which might be import for our application. First, we should have a short prediction time as this should be happening live. Moreover, the model should be robust which requires some monitoring of the performance of the model and possibly some re-training. But, since human users add new data to the existing data, there can be errors which means that we should also check our database from time to time. Finally, training time shouldn't be an important factor as this can be run in the background or during times of low website traffic, e.g. during the night.
In the end our model will be containerized and only be spun up when we re-train our model on a weekly cycle. The newest version of the model will be save in an AWS S3 bucket, but more on that later.
Below you can see a depiction of our model architecture. On the left there is the part which is processing the text input, whereas the image part is shown on the right side.



3.Database

Databases are ubiquitous in any machine learning project. However, usually the databases are not direct part of the project and are hosted elsewhere. As we still wanted to practice the interactions with databases, we decided to create and implement a database with simple tables in our project.
These include a mapper table which maps each category code to its category name. This table is mainly used by our /predict endpoint of our API such that it can directly give the name of the category. Moreover, there is a user table in which we register all users which have access to the API. This table can further be used for the authentication and authorization of users who want to access the different endpoints. In our databse we also save the logs of users who are using our /predict endpoint. Lastly, the way, we have included our Airflow instance, also requires Airflow tables which can be conviently included in the existing database.
For the future it would also make sense to create a database which holds all the product data, i.e. the designation, the description, the image and the product code of each product. Then, we could use this database for re-training the model instead.

4.API

The core part of this project is the app or API providing the model to the users or customers. Currently, we only have a /prediction endpoint which uses the fusion model. This means a registered user can upload a designation, a description and an image and our fusion model will predict the product category including the likelihood. This request will also be logged. In the future this endpoint could be expanded to give the top n predictions with their respective probabilities. Furthermore, we could also automatically save each product, which gets predicted, into a database. Once there is a product database, it make sense to also include an endpoint which allows to add or delete new data.
The other big part of the API is the user handling. This includes the standard endpoints, like /login or /register. Since we already save logs of each user accessing the API, it should be possible for each user to request his or her own logs. This is another functionality which should be included in the future. On the flip side the logs are very rudimentary right now which means we should further extend the logging feature such that they become more meaningful.

5.Testing

Testing is an integral part of any machine learning project at any scale, so we have implemented a simple CI/CD pipeline based on GitHub Actions. GitHub Actions is a very convenient tool as it is directly build into GitHub. By only providing a yml-file with instructions you set off an automatic testing pipeline on a specific trigger. In our case the primary trigger is a “push“ to the main branch of the GitHub repository. All of this happens in a seperate environment which gets cleaned up after all the tests have been run.
Our pipeline has two parts or so called “jobs“. The first one consists of a python syntax check and a set of unit tests with PyTest. These unit tests check the functionality of the user handling endpoints and /predict endpoint. In the latter we only check that we get a correct existing category (i.e. a category which is part of the mapper table in the database) and that the resulting probability is valid, i.e. between 0 and 1. We aren't checking that we get the correct category because this depends on the model performance which changes over time. Also the model performance usually gets monitored elsewhere. In the future all of these tests could be refined further and we could add more unit tests such that all the functions including sub-functions are properly tested.

The second job consists in a simple Docker test. This test checks that all the container run correctly. This means all the images are created and the containers are run. After that the test concludes. This test can also be extended in multiple way. First, there is the possibilty to directly push the Docker images to the DockerHub after the containers are run successfully. Secondly, the interactions between the containers aren't tested currently. These tests were only done locally which means this part could also be further automated.

6.Implementation scheme

The implementation scheme can be neatly summarized in the picture below. There you can see all the interactions between the different parts of the application. The big boxes also represent the way we have isolated or containerized our app.