Machine Learning Engineer - P5

Note technique :

Catégorisation automatique des questions du site openclassroom

Créé le : 10/07/2024

Par Pégeot Guillaume

Table des matières

Table des figures

Introduction

Stack overflow est le site du réseau Stack Exchange Network pour toute question relative aux problèmes de programmations. Le site sert de plateforme pour que les utilisateur puissent poser des questions et aussi répondre à celle des autres utilisateurs de manière collaborative.

Le site est une véritable référence pour toute personne travaillant dans le monde l'informatique avec ses 23 millions d'utilisateurs enregistrés, 24 millions de question et 35 millions de réponses (chiffres de mars 2024)

Il faut savoir que les questions sont classées en fonction de leurs tags qui sont définis par les utilisateurs lors de la création de leur question. C'est ici que nous intervenons afin de proposer des tags en fonction de l'énoncé de la question

1 Préparation des données

1.1 Données d'entrainement

1.1.1 Récupération des données pour l'entrainement

Pour entrainer notre modèle nous avons fait appel au site data.stackexchange.com afin d'effectuer des requetes sql, qui nous on permis de récupérer :

- Id.
- Titre.
- Contenu de la question (BODY).
- Score.
- Tags.
- Date de création.
- Nombre de vues.
- Nombre de réponses.
- Nombre de commentaires.
- Nombre de favoris.

Il est a noter que les requètes retournent au maximum 50 000 entrées.

Nous avons pris la décision d'entrainer notre modèle sur les individus ayant le meilleur score et au moins 4 tags sur la période allant du 01/01/2021 au 01/01/2024.

1.1.2 Point d'attention

Parmis les données récupérés, seul le Titre et les Tags nous intéressent pour notre modèle actuel. Il est cependant intéressant de regarder la distribution des questions dans le temps.

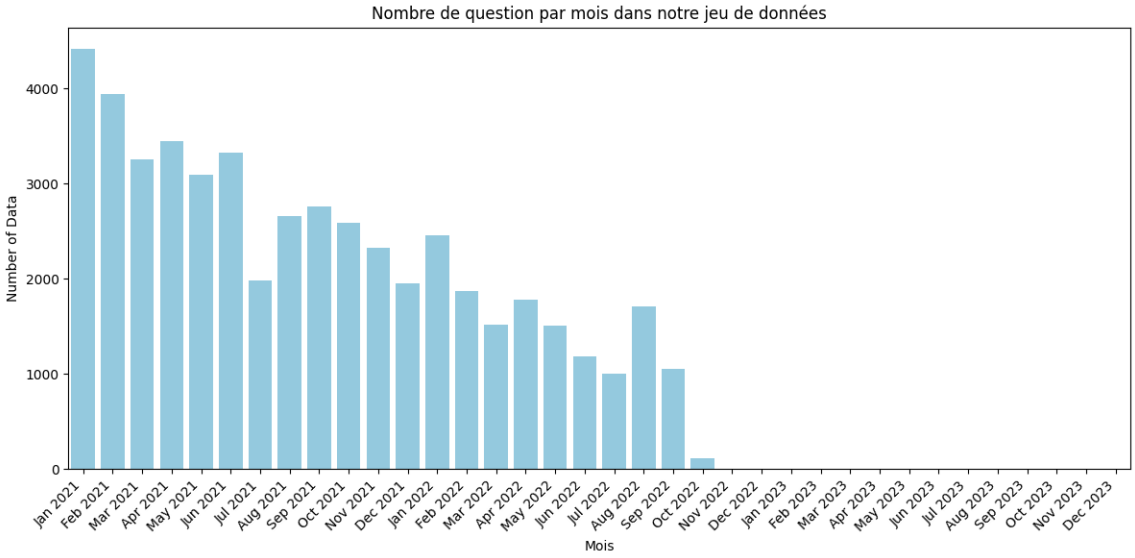

FIGURE 1.1 – Répartition des questions dans le temps pour le jeu d'entrainement

On remarque une grande diminition du nombre de question dans le temps, avec un très faible nombre de question sur l'année 2023. Cela s'explique par le choix de trier les question en fonction de leur score, en effet le score va dépendre du nombre de votes positifs que celle-ci a reçu, favorisant alors les questions les plus anciennes. Ce biais à des chances de pénaliser notre modèle, car il pénalise les documents récent et donc risque de passer à coter des problèmes et des tags émergents.

1.1.3 Traitement des données en vue de l'entrainement

Pour l'entrainement seul les titres et les tags seront conservés.

1.1.3.1 Traitement des tags

Les valeurs obtenu dans le champs tags sont de la forme suivante :

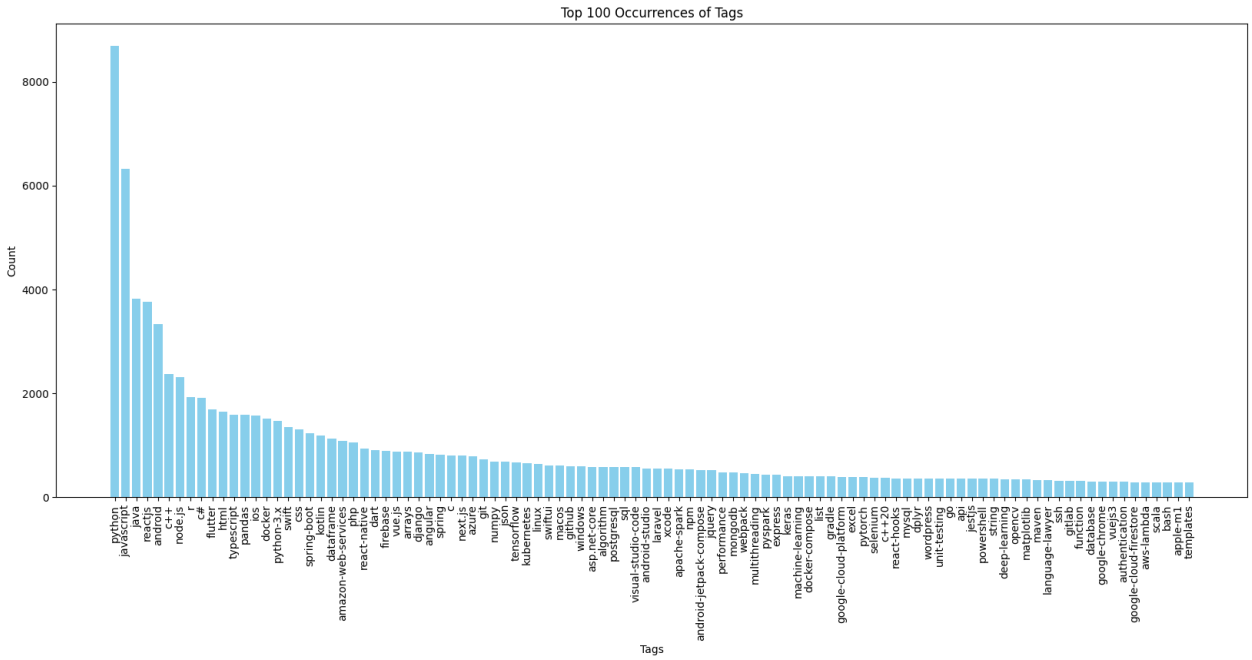

FIGURE 1.2 – Tags les plus représenté lors de l'entrainement

`<string><scala><collections><format>`

Il faudra retirer les chevrons et passer les tags sous forme de liste.

On constatera que les tags les plus courrant concernent des langage de programmation qui sont très courrant en 2024.
A titre indicatif, voici les représentation des 10 tag les plus présents dans notre jeu de donnée ainsi que leur représentation sur le site.

	Tag	jeu de donnée	site le 10/07/2023
1	*python*	8 682	2 204 091
2	*javascript*	6 326	2 533 078
3	*java*	3 820	1 920 082
4	*reactjs*	3 760	479 875
5	*android*	3 325	1 419 170
6	$c++$	2 364	809 766
7	*node.js*	2 308	473 540
8	*r*	1 922	508 111
9	*c#*	1 910	1 619 844
10	*flutter*	1 683	178 728

1.1.3.2 Traitement des titres

Les titres se présentent sous forme de string, nous allons donc leur appliquer :

1. Split la liste sur le charactère espace.
2. Passer en lower l'ensemble des individus issue du split
3. Retirer les individus dans les stopwords de la librairie nltk
4. Appliquer un stemmer sur les individus restant
5. Joindre les individus

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer

# get the stopwords corpus
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))

# get the stemmer used
stemmer = PorterStemmer()

def preprocess_text(text):
    """
    Preprocess text to use it for our trained model
    => tokenize => to_lower => apply stopwords => apply stemmer
   => join

    Parameters:
    text (str): text to process.

    Returns:
    str: Text that can be send to the model.
    """

    tokens = text.split()
    tokens = [stemmer.stem(token) for token in tokens if token.
   lower() not in stop_words]
    return ' '.join(tokens)
```

FIGURE 1.3 – stopword & lemmation

1.1.3.3 Cas du *Body/Post* (non utile pour le moment)

Lors de notre analyse l'utilisation du body pour déterminer les tags a été envisagé. Il faudrait alors retiré les reliquats de balises html présente dans le champ *Body* puis le rajouter

au *Titre* dans une nouvelle variable que l'on appelera *Post*. Il faudra alors appliquer au *Post* le même traitement que celui effectué au *Titre*.

L'utilisation du *Body* n'a pas été plus approfondie, suite à des soucis d'ordre logistique, la machine à disposition mettant plus de 2 heures uniquement pour process la partie de préparation des données.

2 Modèle BERT

Avant toute chose il faut savoir pourquoi l'utilisation de BERT a été choisi.

2.1 Limitations matériel

Pour des raisons logistique, et pour des raisons de délai l'entrainement s'est vu restraindre sur deux point :

1. Limiter le nombre de tags à estimer aux 10 tags les plus fréquent.
2. Réduire la quantité de titres utilisé pour l'entrainement.

2.2 Paramétrage

Pour faire fonctionner notre modèle Bert nous aurons besoin de deux dictionnaire :

1. tag2id pour traduire les tag en id
2. id2tag pour décoder les id sous forme de tag

On utilisera un *BertTokenizer*[1] que l'on applique à nos intitulés de questions. La valeur de len(unique_tags) étant ici de 10^2.

1. De base *'bert-base-uncased'* a été utilisé. Il est possible de prendre d'autres paramètres initiaux pour réduire le temps nécessaire à l'entrainement
2. cf décision prise en 2.1 Limitations matériel

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

model = AutoModelForSequenceClassification.from_pretrained(
    model_path,
    num_labels=len(unique_tags),
    id2label=id2tag,
    label2id=tag2id,
    problem_type="multi_label_classification"
)

encoded_texts = tokenizer(texts, truncation=True, padding=True,
   max_length=512, return_tensors='pt')
```

FIGURE 2.1 – Définition du modèle et du tokenizer

La fonction *compute_metrics* est lutilisé pour calculer les différentes métriques d'évaluation. On a choisi une valeur seuil de 0.5, tout logits ayant une valeur supérieur sera traduite par une évaluation a 1 du tag.

```
def compute_metrics(pred):
    labels = pred.label_ids
    logits = pred.predictions

    threshold = 0.5
    preds = (logits > threshold).astype(float)

    precision, recall, f1, _ = precision_recall_fscore_support(
   labels, preds, average='micro', zero_division=0)
    acc = accuracy_score(labels, preds)

    return {
        'accuracy': acc,
        'f1': f1,
        'precision': precision,
        'recall': recall,
    }
```

Le choix de *BCEWithLogitsLoss* est couramment utilisée pour des tâches de classification binaire où chaque classe est traitée de manière indépendante.
On choisira de travailler sur 10 epochs. A chaque fin d'époque on évalura leur performance et on le sauvegardera toutes les epochs[3] dans le dossier renseigné dans *output_dir*. L'époch avec la meilleur accuracy sera celle chargé à la fin de l'entrainement.

3. Attention, la sauvegarde des modèles prend de la place !

On utilisera une *DataCollatorWithPadding*[4] pour assurer que nos inputs auront la bonne taille. En cas d'input avec une taille inférieur, notre data collector injectera des valeurs pour correspondra à nos attentes en vu du training.

```
    loss_fn = torch.nn.BCEWithLogitsLoss()

    training_args = TrainingArguments(
        output_dir="./model_epochs",
        learning_rate=2e-5,
        per_device_train_batch_size=16,
        per_device_eval_batch_size=16,
        num_train_epochs=10,
        weight_decay=0.01,
        evaluation_strategy="epoch",
        save_strategy="epoch",
        metric_for_best_model="accuracy",
        load_best_model_at_end=True,
    )
    data_collator = DataCollatorWithPadding(tokenizer)
```

FIGURE 2.2 – Définition des paramètres du trainer

Si l'on désire augmenter le nombre d'epoch pour notre entrainement, il faudrait songer à modifier notre *save_strategy*, afin d'effectuer une sauvegarde toute les X epochs afin d'alléger la place prise par notre entrainement ainsi que son temps de calcul.

```
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=test_dataset,
        tokenizer=tokenizer,
        data_collator=data_collator,
        compute_metrics=compute_metrics,
    )
```

FIGURE 2.3 – Définition du trainer avec les paramètres précédement définis

4. Padding $\Longleftrightarrow$ *Rembourrage*

2.3 Performances

Le modèle a été entrainé sur 10 epochs.

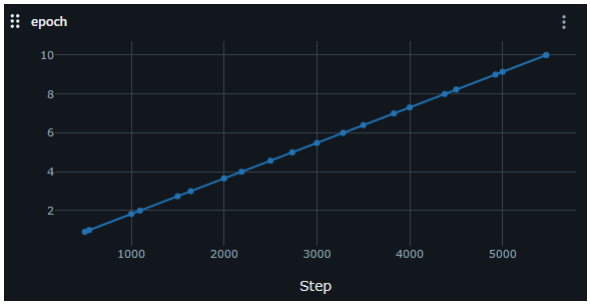

FIGURE 2.4 – Performance observé avec ml flow

Hormis pour *python* & *javascript*, la répartition des tags dans notre jeu de donnée est assez équilibré rendant précision pertinante.

$$Accuracy = \frac{Nombre\ de\ predictions\ correctes}{Nombre\ total\ de\ predictions}$$

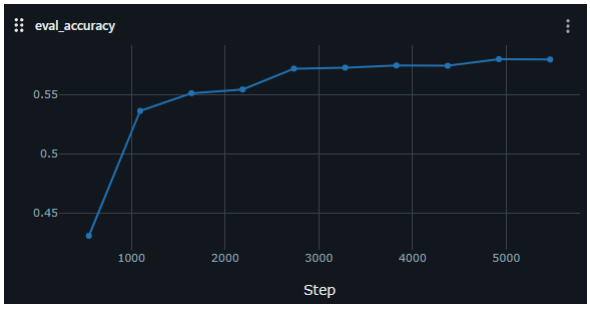

FIGURE 2.5 – Accuracy observé avec ml flow

Le Recall permet d'évaluer la capacité de notre modèle à identifier les instances positives. On arrive donc a prédire 30% des tags, de manière pertinante

$$Recall = \frac{Vrais\ positifs}{Vrais\ prositifs + Faux\ negatif}$$

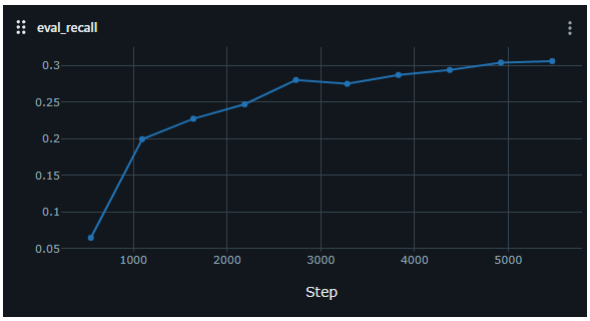

FIGURE 2.6 – Recall observé avec ml flow

Le F1-score est la moyenne harmonique de la précision et du rappel. Ici nous avons une F1-score semblable à notre Recall.

$$F1 - score = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

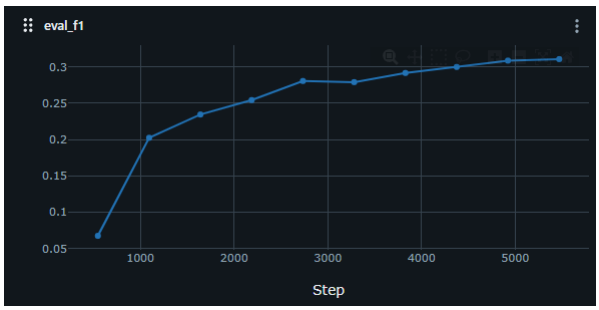

FIGURE 2.7 – F1 score observé avec ml flow

3 Mise en place de l'api

3.1 API

L'API a été développé sur la base /Azure-Samples/msdocs-python-flask-webapp-quickstar, qui est une base par Azure pour avoir une api python utilisant fastapi à déployer sur leur site.

Nous avons légerement modifié l'API hello world pour qu'elle effectue une prédiction pour un titre donné plutôt que de juste dire bonjour au nom de la personne saisie.

Nous avons donc rajouté le code vu dans la figure 1.3 ainsi que de charger le modèle que l'ont aura au préalable sauvegardé dans le dossier ./model[5]

```
model_path = './model'

model = BertForSequenceClassification.from_pretrained(model_path)
tokenizer_config_file = os.path.join(model_path, "
    tokenizer_config.json")
with open(tokenizer_config_file, "r") as f:
    tokenizer_config_data = f.read()
    print("Content of tokenizer_config.json:")
    print(tokenizer_config_data)

tokenizer = BertTokenizer.from_pretrained(model_path)
print(tokenizer) # check if the tokenizer if initialized

id2label = model.config.id2label
```

FIGURE 3.1 – Définition du modèle et du tokenizer

On passera au modèle des titres qui auront donc traité avec stopword et lemmation et on récupérera une liste de tags. Il sera aussi possible de définir un seuil[6] qui définir la certutide minimale des tags en sortie. Plus la valeur est proche de 1, plus le modèle est sur de la

5. Penser a utiliser git ltfs pour les fichier volumineux

présence du tag.

```python
def predict(texts, threshold=0.2):
    # Appliquer le pr traitement
    processed_texts = [preprocess_text(text) for text in texts]

    # Tokenizer les entr es
    print(processed_texts)
    inputs = tokenizer(processed_texts, padding=True, truncation=True, return_tensors="pt")
    with torch.no_grad():
        outputs = model(**inputs)

    # Appliquer la fonction sigmo de pour obtenir les probabilit s
    probabilities = torch.sigmoid(outputs.logits).cpu().numpy()

    # Convertir les probabilit s en labels
    predictions = []
    for probs in probabilities:
        labels = [id2label[idx] for idx, prob in enumerate(probs) if prob > threshold]
        predictions.append(labels)
    return predictions
```

FIGURE 3.2 – Prédire les tags avec l'API

Comme on s'est basé sur une base possédant un affichage web, on a défini 2 cas d'usage pour la détermination des tags en fonction d'un titre. Soit on passe par le navigateur via

```python
@app.get("/get_tags", response_class=HTMLResponse)
async def get_tags(request: Request, title: str = Query(...)):
    predictions = predict([title])
    return templates.TemplateResponse('hello.html', {"request": request, "predictions": predictions[0], 'title': title})
```

FIGURE 3.3 – call web pour avoir les tag via navigation

Ou bien par le call directement auprès de l'API sans passer par aucune interface

6. thresold $\in [0, 1]$

```python
@app.post("/get_tags_api")
async def get_tags_api_endpoint(request: TitleRequest):
    title = request.title
    threshold = request.threshold
    tags = get_tags_api(title, threshold)
    return {"tags": tags}
```

FIGURE 3.4 – appel API sans interface

3.2 Tests unitaire

Des tests unitaire ont été mis en place. Ils sont effectué à chaque commit sur la branche principale via github action.

	Test	effectue
1	index	la racine répond ?
2	preprocess_text	test de la fonction qui effectue stopwords + lemmation
3	get_tags_api_endpoint	appel réussi via API
4	get_tags_api_endpoint_invalid	appel raté car titre vide via
5	get_tags_api_function	test de la fonction de prédiction appelé par l'api
6	get_tags_api_endpoint_valid_threshold	appel api réussi avec un threshold
7	get_tags_api_endpoint_invalid_threshold_below	appel api threshold<0 est interdit
8	get_tags_api_endpoint_invalid_threshold_above	appel api threshold>1 est interdit
9	hello	hello.html est le point d'entré web
10	get_tags	/get_tags ?title répond avec un title
11	get_tags_empty_title	/get_tags ?title répond sans un title

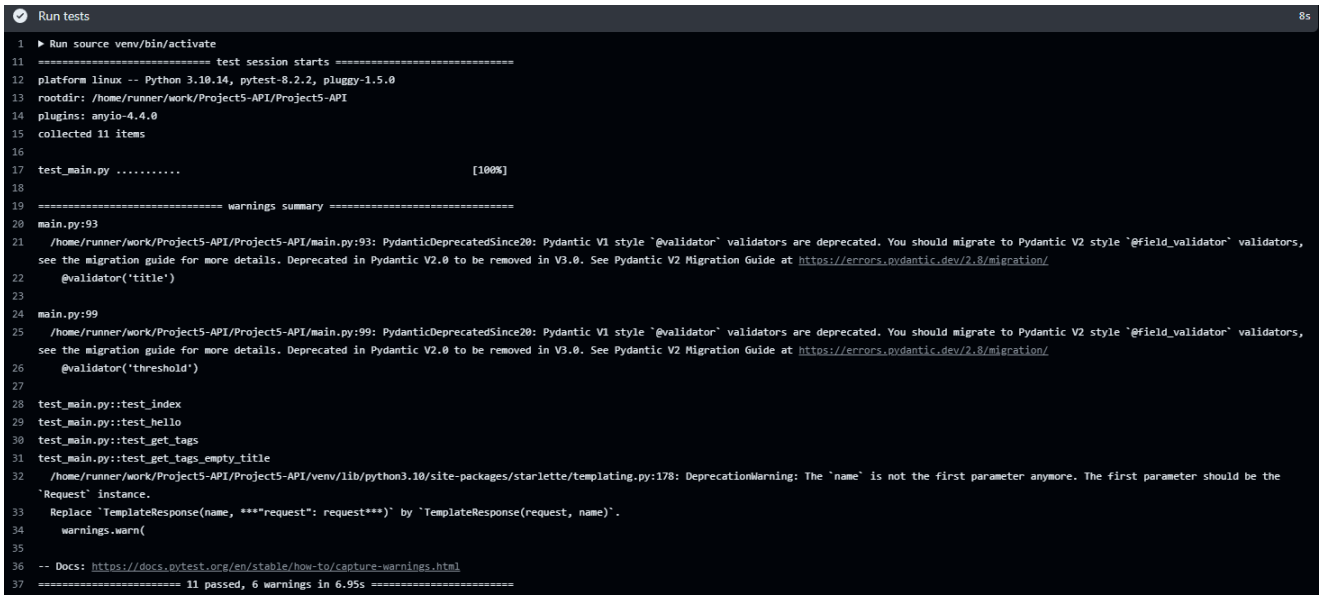

FIGURE 3.5 – Tests unitaire via github action

On remarquera la présence de warning qu'il faudra, à l'avenir, retirer.

3.3 Déploiement

Un deploiement automatisé via github action sur la plateform Azure avait été mis en chantier mais suite à des erreurs de manipulation, la quantité de data disponible pour l'upload via git a été saturé.

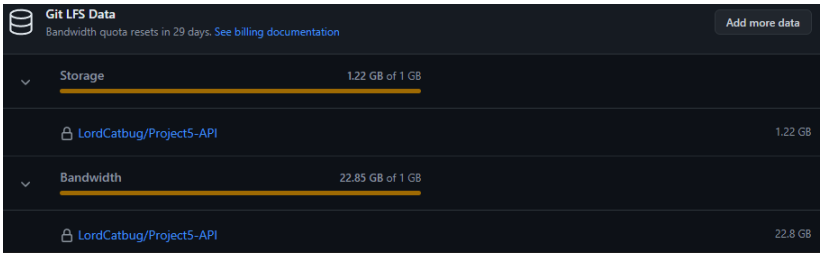

FIGURE 3.6 – Statuts des data upload via github

Le déploiement s'est donc fait via une encapsulation docker ainsi que via un push via le client azure AD.

Le ressource groupe *rg-project5* a été créé contenant :

- Container instances : *registeryfastapi*.

- Container registry : *registeryfastapi* : .

```
#creer l'image pour docker
docker build -t myimage .

#run l'image locally
docker run -d --name mycontainer 80:80 myimage

# cf figure3.7 pour avoir les informations
docker login registeryfastapi.azurecr.io -u registeryfastapi -p "
    password"

# changer le X de buildX par le num ro de build/version
docker build -t registeryfastapi.azurecr.io/rlfastapi:buildX
docker push registeryfastapi.azurecr.io/rlfastapi:buildX
```

FIGURE 3.7 – Commande terminal pour build et commit l'image docker

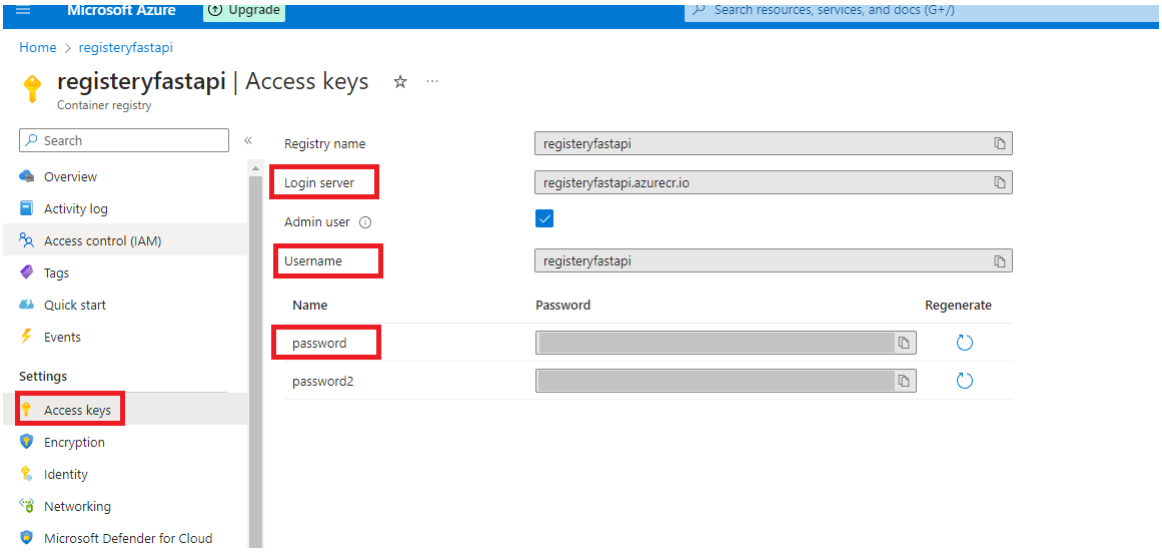

FIGURE 3.8 – Récupérer le password pour upload docker image

Conclusion

Nous avons entrainé un modèle et avons rendu possible l'appel à celui-ci via des appels Api auprès d'Azure.
Les résultats du modèle ne sont pas parfait mais son encourageant. Une des pistes envisageable pour améliorer les performances serait d'intégrer les *Body* afin d'avoir une description plus précise du problème. Cependant, cela induira un temps de pré-traitement et d'entrainement drastiquement plus long.
Nos résultats ne sont compris que dans une liste de 10 tags, il serait bon de réentrainer un modèle avec une liste de tag plus conséquante.
Il est aussi a noter que nous utilisons Azure avec un compte d'essai, ce qui nous limite dans le type de machine dont nous avons accès, se répercutant sur le temps de traitement de nos requètes