Guide d'utilisation de l'outil de benchmark de NER

Ce guide est destiné à toute personne souhaitant utiliser directement l'outil de benchmark de NER ou reprendre le code pour ajouter des fonctionnalités ou des librairies du projet.

I. Présentation du projet :

A. Description :

Le projet consiste en la création d'une application web afin de permettre à des utilisateurs de comparer des librairies de NER pour le français et de tester leurs performances afin de pouvoir choisir la solution la plus adaptée à leur problématique.
Pour cela, l'utilisateur pourra importer différents jeux de données sous un format d'annotation précis (cf: outil d'annotation: https://github.com/cognitivefactory/annotationtool-dev). A chaque importation de fichiers d'entraînement, ces données sont ajoutées à une banque de données locale. L'utilisateur peut ensuite entraîner des modèles issus de plusieurs librairies en sélectionnant ces librairies et des options d'entraînement parmi une liste de paramètres disponibles. Enfin, une visualisation des résultats est proposée afin de pouvoir comparer les performances des modèles.

B. Technologie utilisée :

Le projet consiste en une application

- Le front est réalisé en HTML, CSS et javascript/JQuery.
- Pour le back, le framework est Flask, utilisant le langage Python.
- La communication entre les deux s'effectue grâce au template web Jinja qui permet d'envoyer des variables de Flask au HTML.
- Le dashboard est réalisé avec le framework Dash.

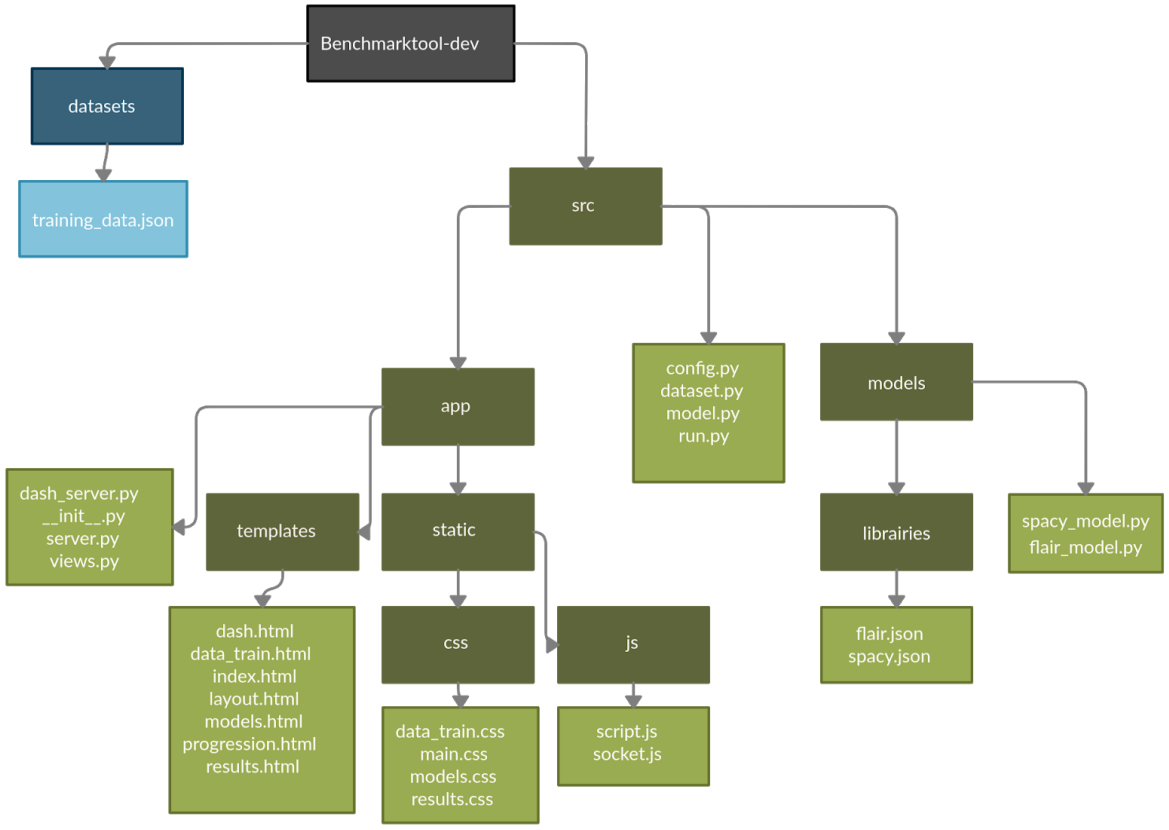
II. Prise en main du projet :

A. lancement du code :

Le code est disponible à l'adresse https://github.com/cognitivefactory/benchmarktool-dev, vous pouvez fork la branche master afin de récupérer la version actuelle du projet.
Vous pouvez vous déplacer dans le dossier benchmarktool-dev et lancer le script *settings* qui installera les librairies Python nécessaires au bon fonctionnement du projet. Assurez-vous d'avoir "*virtualenv*" d'installé sur votre machine afin de permettre la bonne création de l'environnement virtuel.

Si l'installation s'est bien déroulée, vous pourrez lancer le script "*run_script*" afin de lancer le programme. Enfin vous pourrez vous rendre à l'adresse http://localhost:5000/ sur votre navigateur pour accéder à l'application.

B. Architecture du code :



Dossier "Benchmarktool-dev" :

Racine du projet.

Dossier "datasets" :

Contient les fichiers de métadonnées au format JSON.

Après l'entraînement d'un modèle, un fichier de métadonnées est créé s'il n'existe pas déjà dans ce dossier.

Dossier "src" :

Comporte à la racine les fichiers Python de configuration et des classes *Model* (model.py) et *Dataset* (dataset.py).

- dossier "models" :
  - un fichier Python pour chaque librairie, avec la définition de la classe associée, qui hérite de la classe *Model*.

- un sous-dossier “librairies” qui contient un fichier .json par librairie disponible.

Ces fichiers sont utilisés pour générer automatiquement le contenu HTML et gérer le lancement de l'entraînement et du test d’un modèle, et ce de manière générique sans distinguer explicitement la librairie sélectionnée.

- dossier “app” :
  - les fichiers propres à l’initialisation et la configuration des frameworks *Flask* et *Dash*.
  - un sous-dossier “template” qui contient les pages *.html* ainsi qu’un fichier *layout.html*

layout.html permet de définir un template pour chaque page *HTML* en utilisant *Jinja2*. De cette façon, dans ce fichier, on appelle les feuilles *.css* et les scripts *.js*, on crée un menu qui est le même sur chaque page et on définit l’emplacement pour le contenu de chaque page. Cela permet de générer du code *HTML* de façon automatique, et ce, sans redondance.

De cette manière, dans les autres pages HTML, il suffit d’ajouter le contenu propre à la page à l’intérieur du bloc “content” défini dans *layout.html*.

- un sous-dossier “static” : contient le code *CSS* et les scripts *.js* en *Javascript* et *JQuery* dans des sous-dossier “css” et “js”.

C. Ajout d'une librairie :

Si vous souhaitez ajouter une librairie à la liste des librairies disponibles, il faudra dans un premier temps :

- installer les packages nécessaires en exécutant le fichier "./settings")
- mettre à jour le fichier requirements.txt. en supprimant le fichier puis enexécutant *"pip freeze > requirements.txt"*

Il vous faudra ensuite créer les différents fichiers:

- <maLibrairie>.json que vous placerez dans le dossier librairies

Ce fichier doit être structuré de la manière suivante:
nom : le nom de la librairie
options : dictionnaire regroupant les paramètres pouvant être changé dans le formulaire pour l'entraînement des modèles. On retrouve des paramètres tels que le nom du modèle, le nombre d'itération ainsi que des paramètres spécifiques à la librairie.
Chaque paramètre correspond à une liste. Le premier champ correspond au type (int, string) ou aux valeurs qui peuvent être prises. Le deuxième champ correspond à un message d'information qui sera affiché en tant qu'aide pour l'utilisateur. Le dernier champ correspond à la valeur par défaut.

```
{
    "nom" : "spacy",
    "options" : {
        "model_name" : ["string", "nom donné pour
        sauvegarder le modèle", "spacyModel"],
        "nb_iter" : ["int", "nombre d'itérations, par
        défaut 10", "10"]
    },
    "technologie" : "Réseau de neurones convolutif",
    "auteur"  : "Matthew Honnibal",
    "version" : "2.3.2",
    "date" : "28/10/2020"
}
```

Fichier d'exemple : spacy.json

technologie : architecture utilisée par la librairie (ex : CNN, transformer etc…)
auteur : personne ou groupe de personnes ayant créé la librairie ou étant en charge de leur développement.
version : version de la librairie utilisée dans l'application
date : date de la version

- <maLibrairie>Model.py

Ce fichier doit être structuré de la manière suivante :
Le fichier doit importer le module Model.
Le constructeur de la classe <maLibrairie>Model doit appeler la classe Model dont il hérite.
La classe doit impérativement contenir les méthodes suivantes :

train : permet de lancer l'entraînement d'un modèle. Ne prend pas d'arguments car tous les paramètres utilisés sont définis lors de la création d'une instance de la classe.
Après l'entraînement, cette méthode doit mettre à jour la variable *is_ready*. La valeur doit passer à 1 qui signifie que l'entraînement est fini mais que le modèle n'a pas encore été testé.
test : prend en argument un *Dataset* correspondant aux jeu de données de test.
Doit retourner un dictionnaire contenant les scores du modèles :
{
model_name : le nom du modèle
precision : précision
recall : rappel
f-score : f-score
score_by_label : dictionnaire dont chaque clé correspond à un label et la valeur est un dictionnaire avec la précision, le rappel et le f-score pour le label associé.
losses : tableau contenant la valeur de la fonction de perte à chaque itération
}
convert_format : prend en argument un Dataset afin de convertir les données du format d'annotation standard en un format utilisable par la librairie. Renvoie le Dataset modifié en conséquence.

D'autres fonctions spécifiques à la librairie peuvent être ajoutées afin de faciliter son utilisation.

III. Futur du projet :

A. Fonctionnalités à ajouter :

- Page de visualisation de l'état des modèles
- Explicabilité des modèles pour mieux comprendre leur performance
- Entraînement sur un serveur distant
- Nouvelles métriques de performance d'un modèle
- Plus de paramètres d'entraînement d'un modèle

B. Optimisations/Correctifs :

- Vérifier si les données importées comportent du multi-tagging ou non (un mot donné peut être associé à plusieurs labels différents). Cette vérification est indispensable car certaines librairies ne supportent pas le multi-tagging (par exemple Flair). Il faut rajouter une étape de vérification au moment de l'importation des fichiers pour détecter ou non le multi-tagging et ainsi ajuster la liste des librairies proposées en cas de dataset contenant du multi-tagging.
- Lors de l'utilisation de Flair, l'étape de test est très longue tandis que l'outil reste figé avec une popup ouverte pour indiquer que le traitement est en cours.
    - Cause du problème : L'envoi du fichier de test pour lancer le calcul de la performance dans le back se fait en Ajax dans script.js. Le code Ajax attend une réponse du back pour pouvoir fermer la popup. Or cette réponse n'est

envoyée qu'à la fin de tout le traitement des données test (importation, vérification, calcul des performances), ce qui peut être assez chronophage.
    - Solution : Dans le front, il faudrait envoyer le fichier de test avec socket.io et non pas en Ajax. De cette façon, dans le back, à la fin de l'importation et de la vérification du fichier test, on pourrait envoyer une réponse positive au Javascript, ce qui permettrait de fermer la popup.
- Charte graphique : il serait souhaitable de retravailler le design de l'application, en ajoutant notamment du css dans le dashboard. Cela contribuerait à donner un aspect plus "poli" à notre outil afin de donner envie à des utilisateurs de l'utiliser. Pour aller dans la même idée, trouver un nom pour l'application est une solution simple mais qui contribuerait à regrouper une communauté autour du projet.