Questy - Translate and Quality Estimation

Introduction

Notre projet **Translate and Quality Estimation** a pour objectif de développer une application web permettant de traduire un texte d'une langue source à une langue cible tout en évaluant automatiquement la qualité de cette traduction. Cette évaluation repose sur un score de fiabilité compris entre 0 et 1, calculé grâce à des modèles de **Quality Estimation (QE)**. Notre application est conçue pour être intuitive, performante et extensible, en intégrant des technologies modernes comme **FastAPI** vu en cours, des modèles de traduction neuronale et des outils d'évaluation basés sur des architectures de type Transformer.

Objectifs et Enjeux

Objectifs Fonctionnels

1. **Traduction Automatique** : Permettre aux utilisateurs de traduire un texte en choisissant parmi plusieurs modèles de traduction.
2. **Évaluation de la Qualité** : Fournir un score de qualité pour chaque traduction, sans nécessiter de référence humaine.
3. **Interface Utilisateur** : Offrir une interface web simple et ergonomique pour interagir avec le système.
4. **Export des Résultats** : Permettre aux utilisateurs de sauvegarder les traductions et les scores dans des fichiers pour un usage ultérieur.

Objectifs Techniques

1. Intégrer des modèles de traduction et d'évaluation performants comme **Marian-NMT**, **Google Translate**, **DeepL** et **TransQuest**.
2. Utiliser **FastAPI** pour le backend afin de garantir des performances élevées et une gestion efficace des requêtes.
3. Assurer la compatibilité avec des technologies modernes comme Docker pour le déploiement.

Enjeux

- **Qualité des Traductions** : Garantir des traductions précises et adaptées au contexte.
- **Fiabilité de l'Évaluation** : Proposer un score de qualité pertinent et cohérent avec la réalité.
- **Accessibilité** : Rendre l'application accessible à un large public, y compris les utilisateurs non techniques ; mais aussi avoir une application responsive et avec un affichage dynamique et adaptatif.
- **Extensibilité** : Prévoir la possibilité d'ajouter de nouveaux modèles ou fonctionnalités à l'avenir.

Architecture et Technologies

Architecture Générale

Notre application repose sur une architecture modulaire comprenant :

1. **Frontend** : Une interface utilisateur développée avec HTML, CSS et Jinja2 pour le rendu dynamique des templates.
2. **Backend** : Un serveur FastAPI gérant les requêtes, les traductions et les évaluations.
3. **Modèles de Traduction** : Intégration de plusieurs modèles via des bibliothèques comme **HappyTransformer** et **DeepL API**.
4. **Évaluation de la Qualité** : Utilisation de **TransQuest** pour calculer un score de fiabilité.
5. **Stockage** : Sauvegarde des résultats dans des fichiers texte ou JSON.

Technologies Utilisées

- **Langage** : Python
- **Framework Backend** : FastAPI
- **Modèles de Traduction** : Marian-NMT, Google Translate, DeepL
- **Évaluation de la Qualité** : SiameseTransQuestModel
- **Frontend** : HTML, CSS, Bootstrap
- **Conteneurisation** : Docker
- **Gestion des dépendances** : requirements.txt

Fonctionnalités

1. Traduction Automatique

L'utilisateur peut choisir parmi plusieurs modèles de traduction :

- **Marian-NMT** : Modèle open-source performant pour des paires de langues spécifiques.
- **Google Translate** : Modèle propriétaire offrant une large couverture linguistique.
- **DeepL** : Connu pour la qualité de ses traductions, notamment en langues européennes.

Processus

1. L'utilisateur sélectionne la langue source et la langue cible.
2. Il saisit le texte à traduire dans un formulaire.
3. Le backend traite la requête et renvoie la traduction.

2. Évaluation de la Qualité

L'évaluation est réalisée à l'aide du modèle **SiameseTransQuestModel**, qui prédit un score de qualité basé sur la similarité entre le texte source et le texte traduit.

Méthodologie

- Le modèle Siamese utilise des représentations vectorielles des phrases pour calculer un score de similarité.
- Le score est compris entre 0 (traduction de mauvaise qualité) et 1 (traduction parfaite).

3. Interface Utilisateur

Notre interface web est conçue pour être intuitive et responsive. Elle permet :

- De sélectionner un modèle de traduction.
- De choisir les langues source et cible.
- D'afficher les résultats sous forme de texte traduit et de barre de progression pour le score (avec pourcentage).

4. Export des Résultats

Les traductions et les scores sont sauvegardés dans des fichiers texte dans le répertoire Output.

Développement, Analyse Technique et Implémentation

Backend

Le backend est implémenté dans les fichiers suivants :

- main.py : Script principal pour exécuter la traduction et l'évaluation en ligne de commande.
- web.py : Gestion des routes et des interactions utilisateur via FastAPI.

1. main.py : Script principal

Ce fichier est le point d'entrée pour exécuter la traduction et l'évaluation de la qualité via la ligne de commande.

Explications

- **Chargement du modèle de Quality Estimation (QE)** :

```
model_QE = SiameseTransQuestModel("TransQuest/siamesetransquest-da-multilingual")
model_QE.model.to(device)
```

  - Le modèle **SiameseTransQuest** est chargé pour évaluer la qualité des traductions. Il est transféré sur le GPU si disponible, sinon il utilise le CPU.
  - Le temps de chargement est mesuré pour évaluer les performances.

- **Arguments de la ligne de commande** :

```
parser = argparse.ArgumentParser(description="Traduction et évaluation de la qualité.")
parser.add_argument("-src", "--source", type=str, required=True, help="Langue source")
```

```
parser.add_argument("-tgt", "--target", type=str, required=True, help="Langue cible")
parser.add_argument("-ipt", "--input", type=str, required=True, help="Texte à traduire")
```

- Les arguments permettent de spécifier la langue source, la langue cible et le texte à traduire.
- Cela rend le script flexible pour différents cas d'utilisation.

- **Traduction et évaluation** :

```
output_txt, timer_t = translate(model_LANG, input_txt)
score_QE = score(input_txt, output_txt)
```

- La fonction `translate` effectue la traduction en utilisant le modèle Marian-NMT.
- La fonction `score` calcule le score de qualité en comparant le texte source et le texte traduit.

- **Sauvegarde des résultats** :

```
save("./Output/translate.txt", output_txt)
save("./Output/score.txt", score_QE)
```

- Les résultats (texte traduit et score) sont sauvegardés dans des fichiers pour un usage ultérieur.

2. web.py : Backend FastAPI

Ce fichier gère les routes et les interactions utilisateur via FastAPI.

Explications

- **Configuration de l'application** :

```
app = FastAPI()
templates = Jinja2Templates(directory="templates")
app.mount("/static", StaticFiles(directory="static"), name="static")
```

- Une instance FastAPI est créée pour gérer les requêtes.
- Les templates HTML sont chargés depuis le répertoire templates.
- Les fichiers statiques (CSS, images) sont servis depuis le répertoire static.

- **Route pour la page d'accueil** :

```
@app.get("/")
async def root(request: Request):
    context = {"request": request, "models": MODEL_DATABASE, "languages": LANG_DATABASE}
    return templates.TemplateResponse("index.html", context)
```

- Cette route renvoie la page d'accueil (`index.html`) avec les modèles et langues disponibles.
- Les données sont injectées dans le template via le contexte.

- **Route pour soumettre une traduction** :

```
@app.post("/submit/")
async def get_parameters(request: Request, model: str = Form(...),lang_src: str = Form(...),
                         lang_tgt: str = Form(...), text_src: str = Form(...)):
    ...
    context = {"request": request, "models": MODEL_DATABASE, "model_selected": model,
            "languages": LANG_DATABASE, "lang_src": lang_src, "lang_tgt": lang_tgt,
            "text_src": text_src, "text_tgt": text_tgt, "score": int(score_QE)}
    return templates.TemplateResponse("index.html", context)
```

- Les paramètres du formulaire (modèle, langues, texte source) sont récupérés.
- En fonction du modèle sélectionné (DeepL, Google Translate, Marian-NMT), la traduction et l'évaluation sont effectuées.
- Les résultats sont renvoyés à la page d'accueil pour affichage.

Points Clés

- Gestion des requêtes HTTP avec FastAPI.

- Intégration des modèles de traduction et d'évaluation.
- Sauvegarde des résultats dans des fichiers texte.

Frontend

Notre fichier index.html contient le formulaire et l'affichage des résultats. On utilise ici Bootstrap pour un design moderne et responsive qui s'adapte également aux diverses manipulations de l'utilisateur pour plus d'ergonomie (si l'utilisateur modifie la taille de sa fenêtre, les dimensions de l'affichage de notre application s'adapte en conséquence).

1. index.html : Interface utilisateur

Ce fichier définit l'interface utilisateur pour interagir avec l'application.

Explications

- **Formulaire de traduction** :

```
<form action="/submit/" method="post">
    <select id="model" name="model" class="form-select" required>
        {% for model in models.keys() %}
            <option value="{{ model }}" {{ "selected"
                    if model == model_selected else ""}}>{{ model }}</option>
        {% endfor %}
    </select>
    ...
</form>
```

  - L'utilisateur peut sélectionner un modèle de traduction, une langue source, une langue cible et saisir un texte à traduire.
  - Les données sont envoyées au backend via la méthode POST.

- **Affichage des résultats** :

```
<textarea id="text_tgt" name="text_tgt" class="form-control"
          rows="5">{{ text_tgt if text_tgt }}</textarea>
<div class="progress" role="progressbar" aria-valuenow="{{ score }}"
     aria-valuemin="0" aria-valuemax="100">
    <div class="progress-bar" style="width: {{ score }}%">{{ score }}</div>
</div>
```

  - Le texte traduit est affiché dans une zone de texte.
  - Le score de qualité est représenté sous forme de barre de progression.

Points Clés

- Notre formulaire est interactif pour la saisie des paramètres.
- On a un affichage dynamique des résultats grâce à Jinja2.
- Une utilisation de Bootstrap pour une mise en page claire et esthétique.

Modèles, Données et Exemple d'Execution

- Les modèles de traduction sont intégrés via des bibliothèques comme **HappyTransformer** et **DeepL API**.
- Les langues disponibles sont chargées depuis le fichier JSON lang_tags.json.
- A voir aussi, un exemple d'exécution pas à pas des modèles avec execution_steps.ipynb execution_steps.ipynb.

1. execution_steps.ipynb : Notebook d'exécution

Ce fichier contient des exemples d'utilisation des modèles, étapes par étapes, de traduction et d'évaluation.

Explications

- **Traduction avec HappyTransformer** :

```
happy_tt = HappyTextToText("MARIAN", model_LANG)
outputs = happy_tt.generate_text(inputs, args=settings).text
```

  - Le modèle Marian-NMT est utilisé pour traduire un texte.

- Les paramètres comme `top_k` et `temperature` contrôlent la génération du texte.

- **Traduction avec Google Translate** :

```
outputs = GoogleTranslator(source='fr', target='de').translate(text=inputs)
```

  - La bibliothèque `deep_translator` est utilisée pour accéder à l'API Google Translate.

- **Évaluation avec TransQuest** :

```
model = SiameseTransQuestModel("TransQuest/siamesetransquest-da-multilingual")
pred = model.predict([[inputs_txt, outputs_txt]])
```

  - Le modèle SiameseTransQuest prédit un score de qualité pour une paire de phrases (texte source et texte traduit).

Résultats

Exemple de Traduction

- **Texte source** : “Bonsoir, quelle belle journée n'est-ce pas ?”
- **Traduction** : “Good night, what a nice day is it?”
- **Score de qualité** : 0.8315

Interface Utilisateur

Notre interface permet une interaction fluide et affiche les résultats sous forme de texte traduit et de barre de progression pour le score.

Déploiement

Local

Pour exécuter notre application localement :

1. Installer les dépendances avec `pip install -r requirements.txt`.
2. Lancer le serveur avec `uvicorn web:app --reload`.

Docker

Notre application peut être déployée dans un conteneur Docker. Le fichier docker-compose.yml facilite le déploiement en exposant le service sur le port 8005.

Explications

- **Dockerfile** :

```
FROM python:3.10-slim
WORKDIR /app
COPY . /app
RUN pip install -r requirements.txt
CMD ["uvicorn", "web:app", "--host", "0.0.0.0", "--port", "8000"]
```

  - L'image Docker est basée sur Python 3.10.
  - Les dépendances sont installées via requirements.txt.
  - L'application est lancée avec Uvicorn.

- **docker-compose.yml** :

```
services:
  web:
    build: .
    ports:
      - "8005:8000"
    volumes:
```

```
      - .:/app
```

- Le service expose l'application sur le port 8005.
- Le volume permet de synchroniser les fichiers locaux avec le conteneur.

Conclusion

Notre projet combine des technologies modernes pour offrir une solution complète de traduction et d'évaluation de qualité. Il est extensible et peut intégrer de nouveaux modèles ou fonctionnalités, comme la détection automatique de la langue source ou l'export des résultats (ou historique de recherche) en JSON.

Annexes

Structure des Fichiers

- (Exemple d'utilisation : execution_steps.ipynb)
- **Backend** : main.py, web.py
- **Frontend** : index.html
- **Données** : lang_tags.json
- **Déploiement** : requirements.txt, Dockerfile, docker-compose.yml

Références

- FastAPI Documentation
- TransQuest GitHub Repository
- HappyTransformer Documentation