Rapport d'Analyse des Modifications Apportées à un Notebook d'Entraînement

Ce rapport détaille les différences clés entre une version originale d'un notebook Jupyter d'entraînement et une version adaptée. Les modifications ont été apportées principalement pour corriger des erreurs importantes et améliorer l'autonomie du processus d'entraînement.

1. Changements Clés Identifiés

La version adaptée intègre des modifications substantielles, principalement axées sur l'autonomie de la configuration et l'ajout d'un pipeline complet de prétraitement des données directement dans le notebook. Voici les points majeurs:

1.1. Gestion de la Configuration

- **Version Originale:** S'appuie sur un fichier de configuration externe (par exemple, `config.py`) pour définir les paramètres d'entraînement et les chemins de données. Elle utilise également un outil de suivi d'expériences (`wandb`) pour le logging et le suivi des métriques.
- **Version Adaptée:** La configuration est désormais **définie manuellement et directement** au début du notebook via un dictionnaire. Cela rend le notebook autonome en termes de paramètres. Les chemins des données sont définis de manière relative. L'outil de suivi d'expériences a été remplacé par une configuration de logging Python standard (`logging`). Des répertoires nécessaires (`./models`, `./logs`) sont explicitement créés avec `os.makedirs(..., exist_ok=True)`.

1.2. Préparation et Chargement des Données (Ajout Majeur)

C'est la différence la plus significative. La version adaptée ajoute un bloc de code complet pour le prétraitement des données, absent de la version originale. Cet ajout a été crucial pour résoudre des problèmes liés au formatage des données.

- **Version Originale:** Suppose que les fichiers de données sont déjà au format attendu par le modèle TPLinker et sont simplement chargés. Il n'y a pas de logique intégrée pour transformer des données brutes en ce format.

- **Version Adaptée:** Contient un **pipeline de prétraitement des données JSON brutes** pour les adapter au format spécifique de TPLinker. Ce processus est détaillé ci-dessous :
    - **Chargement et Tokenisation Initiale :** Le notebook charge le fichier JSON brut (par exemple, `nyt_dataset.json`). Pour chaque échantillon de données, le texte est tokenisé à l'aide d'un tokenizer Hugging Face (`AutoTokenizer.from_pretrained('bert-base-cased')`). Cette étape génère non seulement les tokens, mais aussi les `offset_mapping`, qui sont des paires `(start_char, end_char)` pour chaque token, indiquant sa position dans le texte original. Ces `offset_mapping` sont cruciaux pour la conversion des spans de caractères en spans de tokens.
    - **Fonctions Utilitaires pour le Mapping :**
        - `char_to_token(offsets, char_index)` : Cette fonction prend les `offset_mapping` générés par le tokenizer et un index de caractère. Elle parcourt les offsets pour trouver à quel token (identifié par son index) correspond l'index de caractère donné. Cela permet de traduire les positions d'entités ou de relations définies par des indices de caractères en indices de tokens, format requis par TPLinker.
        - `find_key(possibles, keys)` : Cette fonction est utilisée pour identifier de manière flexible les noms de champs pertinents (comme `text`, `raw_subjects`, `predicates`, `subj_char_span_starts`, etc.) dans les données JSON d'entrée, car les schémas de données peuvent varier.
    - **Génération de `relation_list` et `entity_list` :** Pour chaque échantillon, le code parcourt les sujets, objets et prédicats bruts. Il utilise `char_to_token` pour convertir les `char_span` (début et fin en caractères) des sujets et objets en `tok_span` (début et fin en tokens). Ces `tok_span` sont essentiels pour que le modèle TPLinker puisse identifier les entités et leurs relations au niveau des tokens. Une attention particulière est portée à la gestion des cas où les mappings de caractères/tokens peuvent échouer (par exemple, si une entité ne correspond pas parfaitement aux limites des tokens), ce qui était une source d'erreurs importantes dans la version originale. Les `relation_list` et `entity_list` sont construites avec ces informations normalisées.
    - **Création Dynamique de `rel2id.json` :** Au lieu de s'appuyer sur un fichier `rel2id.json` préexistant, la version adaptée collecte tous les prédicats uniques (`predicate`) trouvés dans les `relation_list` générées à partir de l'ensemble des données. Elle crée ensuite un mappage `rel2id` (relation vers ID numérique) et le sauvegarde dans un nouveau fichier `rel2id.json`. Cela garantit que le mappage des relations est cohérent avec les données réellement utilisées.

- **Fractionnement et Sauvegarde des Données :** Une fois tous les échantillons prétraités et normalisés, l'ensemble des données est mélangé (`random.shuffle`) pour assurer une distribution aléatoire des échantillons. Il est ensuite divisé en ensembles d'entraînement et de validation (`train_data` , `valid_data`) en fonction d'un `train_ratio` (par exemple, 80% pour l'entraînement, 20% pour la validation). Enfin, ces données prétraitées sont sauvegardées dans un fichier JSON (par exemple, `nyt_dataset.json`), potentiellement écrasant le fichier d'entrée original avec la version formatée pour TPLinker.

1.3. Gestion du Logging

- **Version Originale:** Utilise un outil de logging avancé (`wandb`) ou un logger personnalisé.
- **Version Adaptée:** Simplifie le logging en utilisant le module standard `logging` de Python. Les logs sont écrits à la fois dans un fichier et sur la console, offrant une approche plus directe et moins dépendante d'outils externes pour le suivi de base.

1.4. Dépendances et Imports

- **Version Originale:** Importe des bibliothèques spécifiques à l'outil de suivi d'expériences et à l'encodeur BiLSTM (`wandb` , `config` , `glove`).
- **Version Adaptée:** Supprime les imports de ces bibliothèques et ajoute celles nécessaires au nouveau pipeline de prétraitement des données (`random` , `collections.defaultdict`).