Model Deep Learning Pour detection de Log error

Introduction

Ce document a pour objectif de fournir une explication détaillée du code que nous avons écrit pour prétraiter les données de logs, les labelliser et entraîner un modèle de deep learning basé sur des réseaux de neurones. Chaque section du code sera expliquée en détail afin de faciliter la compréhension et la collaboration au sein de notre équipe.

I. Prétraitement des données

La première étape de notre pipeline consiste à identifier et extraire les lignes de logs contenant des erreurs. Pour ce faire, nous avons développé une fonction qui recherche des mots-clés spécifiques indiquant une erreur, puis nous avons

une autre fonction pour parcourir les fichiers de logs et écrire ces erreurs dans un fichier de sortie.

```
import os

def is_error_line(line):
    # Définir les occurrences qui indiquent une erreur
    error_keywords = [
        'level=error',
        'Network is unreachable',
        'Bad argument',
        'Another app is currently holding the xtables loc
k',
        'dropping malformed DNS',
        'Resource temporarily unavailable',
        'Usage:'
    ]
    # Vérifier si la ligne contient l'une des occurrences
    return any(keyword in line for keyword in error_keyword
s)

def find_errors_in_logs(directory, output_file):
    with open(output_file, 'w', encoding='utf-8') as error_
file:
        for filename in os.listdir(directory):
            if filename.endswith('.txt'):  # Filtrer les fi
chiers de log
                filepath = os.path.join(directory, filenam
e)
                with open(filepath, 'r', encoding='utf-8')
as file:
                    log_lines = file.readlines()
                for line in log_lines:
                    if is_error_line(line):
                        error_file.write(line)

# Exemple d'utilisation
log_directory = 'label1'  # Dossier contenant les fichiers
```

```
de logs avec label 1
output_file = 'error.txt'  # Fichier de sortie pour les log
s avec erreurs

find_errors_in_logs(log_directory, output_file)
```

1. **Définition de la fonction `is_error_line`**:
   - Cette fonction prend en entrée une ligne de log et vérifie si elle contient l'un des mots-clés d'erreur définis dans la liste `error_keywords`.
   - La fonction retourne `True` si l'un des mots-clés est présent dans la ligne, sinon elle retourne `False`.
2. **Définition de la fonction `find_errors_in_logs`**:
   - Cette fonction prend en entrée un répertoire contenant des fichiers de logs et un fichier de sortie.
   - Elle parcourt chaque fichier de log dans le répertoire et lit les lignes une par une.
   - Si une ligne contient un mot-clé d'erreur (vérifié en utilisant la fonction `is_error_line`), elle est écrite dans le fichier de sortie.
3. **Exemple d'utilisation**:
   - Nous spécifions le répertoire contenant les fichiers de logs (`log_directory`) et le fichier de sortie pour les erreurs (`output_file`).
   - Nous appelons la fonction `find_errors_in_logs` pour extraire les lignes d'erreur et les écrire dans `error.txt`.

Cette première étape permet d'extraire efficacement les lignes de logs contenant des erreurs, facilitant ainsi le traitement et l'analyse ultérieurs

II. Traitement et Labellisation des Données

Dans cette section, nous allons traiter et labelliser les données de logs pour préparer l'entraînement du modèle. L'objectif est de transformer les logs bruts en un format structuré et étiqueté, ce qui facilitera l'analyse et l'entraînement du modèle de deep learning.

Analyse des Lignes de Logs

Cette partie du code se concentre sur l'analyse de chaque ligne de log pour en extraire des informations pertinentes telles que la date, le service, le PID (Process ID) et le message.

```
def parse_log_line(line):
    # Définir plusieurs patterns pour différents formats de
logs
    patterns = [
        re.compile(r'(?P<date>\w{3} \d{2} \d{2}:\d{2}:\d
{2}) (?P<host>\S+) (?P<service>\S+)\[(?P<pid>\d+)\]: (?P<me
ssage>.+)'),
        re.compile(r'(?P<date>\d{4}-\d{2}-\d{2}T\d{2}:\d
{2}:\d{2}\+\d{4}) (?P<host>\S+) (?P<service>\S+)\[(?P<pid>
\d+)\]: (?P<message>.+)'),
        re.compile(r'(?P<date>\w{3} \d{2} \d{2}:\d{2}:\d
{2}) (?P<host>\S+) (?P<service>\S+): (?P<message>.+)'),
        re.compile(r'\[(?P<date>[\d\.]+)\] (?P<message>.
+)'),
        re.compile(r'(?P<host>\S+) (?P<service>\S+)\[(?P<pi
d>\d+)\]: (?P<message>.+)'),
        re.compile(r'(?P<date>\d{4}-\d{2}-\d{2}T\d{2}:\d
{2}:\d{2}\+\d{4}) (?P<host>\S+) (?P<service>\S+): (?P<messa
ge>.+)'),
        re.compile(r'(?P<date>\d{4}-\d{2}-\d{2}T\d{2}:\d
{2}:\d{2}\.\d+Z) level=(?P<level>\w+) msg="(?P<message>.
+)"'),
        re.compile(r'(?P<host>\S+) (?P<service>\S+)\[(?P<pi
d>\d+)\]: (?P<message>.+)')
    ]

    for pattern in patterns:
        match = pattern.match(line)
        if match:
            log_dict = match.groupdict()

            # Nettoyage spécifique pour certains services
            if 'service' in log_dict:
```

```
                if log_dict['service'].startswith('balena-e
ngine-daemon'):
                    message = log_dict['message']
                    message_cleaned = re.sub(r'time="[^"]+"
', '', message)
                    message_cleaned = re.sub(r'namespace=[^
]+ ', '', message_cleaned)
                    log_dict['message'] = message_cleaned

                elif log_dict['service'].startswith('python
3'):
                    message = log_dict['message']
                    message_cleaned = re.sub(r'\[pid \d+\]
', '', message)
                    log_dict['message'] = message_cleaned

                elif log_dict['service'].startswith('chill
i'):
                    message = log_dict['message']
                    message_cleaned = re.sub(r'chilli\[\d+
\]: ', '', message)
                    log_dict['message'] = message_cleaned

            return log_dict

    return None
```

- **Patterns d'expressions régulières** : Utilisés pour analyser différents formats de logs et extraire des informations comme la date, l'hôte, le service, le PID et le message.
- **Nettoyage spécifique** : Pour certains services comme `balena-engine-daemon`, `python3`, et `chilli`, des nettoyages supplémentaires sont effectués pour supprimer les informations non pertinentes.

Sauvegarde des Logs dans un Fichier CSV

Cette section du code convertit les logs en un format CSV afin de faciliter leur utilisation ultérieure.

```
def save_logs_to_csv(logs, output_file):
    # Convertir la liste de dictionnaires en DataFrame
    df = pd.DataFrame(logs)
    # Sauvegarder le DataFrame dans un fichier CSV
    df.to_csv(output_file, index=False)
```

- **Conversion en DataFrame** : Les logs sont convertis en DataFrame à l'aide de pandas pour une manipulation et une sauvegarde faciles.
- **Sauvegarde en CSV** : Le DataFrame est ensuite sauvegardé dans un fichier CSV, ce qui permet une analyse ultérieure des données.

Traitement et Étiquetage des Fichiers de Logs

Cette partie lit tous les fichiers de logs dans un répertoire, analyse chaque ligne, et étiquette les logs en fonction d'un label fourni.

```
def process_log_files(directory, label):
    logs = []
    for filename in os.listdir(directory):
        if filename.endswith('.txt'):  # Filtrer les fichie
rs de log
            filepath = os.path.join(directory, filename)
            with open(filepath, 'r', encoding='utf-8') as f
ile:
                log_lines = file.readlines()
            parsed_logs = [parse_log_line(line) for line in
log_lines if parse_log_line(line) is not None]
            for log in parsed_logs:
                log['label'] = label
            logs.extend(parsed_logs)
    return logs
```

- **Filtrage des fichiers** : Seuls les fichiers avec une extension `.txt` sont traités.
- **Étiquetage des logs** : Chaque log analysé se voit attribuer un label (0 ou 1) en fonction du répertoire d'origine.

Utilisation

Voici comment utiliser les fonctions définies pour traiter et étiqueter les logs, puis les sauvegarder dans un fichier CSV.

```
log_directory_1 = 'label1'  # Dossier contenant les fichier
s de logs à étiqueter avec 1
log_directory_0 = 'label0'  # Dossier contenant les fichier
s de logs à étiqueter avec 0

logs_label_1 = process_log_files(log_directory_1, 1)
logs_label_0 = process_log_files(log_directory_0, 0)

all_logs = logs_label_1 + logs_label_0
save_logs_to_csv(all_logs, 'all_logs.csv')
```

- **Combinaison des logs étiquetés** : Les logs des deux répertoires sont combinés en une seule liste.
- **Sauvegarde finale** : La liste combinée est sauvegardée dans un fichier CSV pour une utilisation ultérieure.

III. Préparation et Suréchantillonnage des Données

Dans cette partie, nous allons transformer les messages de logs en vecteurs de caractéristiques, appliquer une technique de suréchantillonnage pour équilibrer les données, et sauvegarder le tout dans un fichier CSV. Cela permet d'assurer que notre modèle de machine learning ait suffisamment de données équilibrées pour chaque classe.

Chargement et Vectorisation des Données

Nous commençons par charger les données de logs depuis un fichier CSV, puis nous convertissons les messages de logs en vecteurs de caractéristiques numériques.

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer

# Charger les données
df = pd.read_csv('all_logs.csv')

# Préparation des données
vectorizer = TfidfVectorizer(max_features=1000)
X = vectorizer.fit_transform(df['message']).toarray()
y = df['label']
```

- **Chargement des données** : Les logs sont chargés depuis un fichier CSV en utilisant pandas.
- **Vectorisation des messages** : La classe `TfidfVectorizer` de sklearn est utilisée pour convertir les messages en vecteurs de caractéristiques. Nous limitons le nombre de caractéristiques à 1000 pour des raisons de performance.

Application du Suréchantillonnage avec SMOTE

Pour équilibrer les classes dans nos données, nous appliquons le suréchantillonnage en utilisant SMOTE (Synthetic Minority Over-sampling Technique).

```
from imblearn.over_sampling import SMOTE

# Appliquer SMOTE pour suréchantillonner les données
smote = SMOTE(sampling_strategy='auto', random_state=42)
X_res, y_res = smote.fit_resample(X, y)
```

- **SMOTE** : Cette technique génère des exemples synthétiques pour la classe minoritaire afin d'équilibrer la distribution des classes.
- **Suréchantillonnage** : `fit_resample` est utilisé pour appliquer SMOTE sur nos données vectorisées.

Conversion et Sauvegarde des Données Suréchantillonnées

Nous convertissons les tableaux numpy résultants en DataFrames pandas et les combinons pour former un seul DataFrame, qui est ensuite sauvegardé dans un fichier CSV.

```
# Convertir les tableaux numpy en DataFrames
X_res_df = pd.DataFrame(X_res, columns=vectorizer.get_featu
re_names_out())
y_res_df = pd.DataFrame(y_res, columns=['label'])

# Recombiner les données suréchantillonnées en DataFrame
df_res = pd.concat([X_res_df, y_res_df], axis=1)

# Sauvegarder les données suréchantillonnées dans un nouvea
u fichier CSV
df_res.to_csv('logs_resampled.csv', index=False)
```

- **Conversion en DataFrames** : Les tableaux numpy issus de SMOTE sont convertis en DataFrames pour une manipulation plus facile.
- **Combinaison des DataFrames** : Les vecteurs de caractéristiques et les labels sont combinés en un seul DataFrame.
- **Sauvegarde en CSV** : Le DataFrame final est sauvegardé dans un nouveau fichier CSV, prêt à être utilisé pour l'entraînement du modèle de machine learning.

IV. Entraînement et Évaluation du Modèle de Réseau de Neurones

Dans cette dernière partie, nous allons entraîner un modèle de réseau de neurones en utilisant les données suréchantillonnées. Nous allons également évaluer les performances de ce modèle pour s'assurer qu'il est efficace dans la classification des logs.

Chargement et Préparation des Données

Nous commençons par charger les données suréchantillonnées à partir du fichier CSV, puis nous les préparons pour l'entraînement du modèle.

```
import pandas as pd
from sklearn.model_selection import train_test_split

# Charger les données suréchantillonnées
df_res = pd.read_csv('logs_resampled.csv')

# Préparation des données
X = df_res.drop('label', axis=1).values
y = df_res['label'].values
```

- **Chargement des données** : Les données suréchantillonnées sont chargées depuis le fichier `logs_resampled.csv`.
- **Préparation des données** : Les messages de logs vectorisés sont séparés des labels pour former les ensembles de caractéristiques (`X`) et de labels (`y`).

Division des Données en Ensembles d'Entraînement, de Validation et de Test

Nous divisons les données en ensembles d'entraînement, de validation, et de test pour pouvoir évaluer la performance du modèle de manière indépendante.

```
# Diviser les données en ensembles d'entraînement et de tes
t
X_train_val, X_test, y_train_val, y_test = train_test_split
(X, y, test_size=0.1, random_state=42)

X_train, X_val, y_train, y_val = train_test_split(X_train_v
al, y_train_val, test_size=0.2, random_state=42)
```

- **Division des données** : Les données sont divisées en ensembles d'entraînement et de test dans un premier temps, puis l'ensemble d'entraînement est divisé en sous-ensembles d'entraînement et de validation.

Construction du Modèle de Réseau de Neurones

Nous construisons un modèle de réseau de neurones à l'aide de la bibliothèque TensorFlow/Keras.

```
from tensorflow.keras import layers, models

# Construire le modèle de réseau de neurones
model = models.Sequential()
model.add(layers.Dense(128, activation='relu', input_shape=
(X_train.shape[1],)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

Détails Techniques :

- **Modèle séquentiel** : Le modèle est construit de manière séquentielle en ajoutant des couches une par une.
- **Couches du modèle** : Le modèle comporte des couches denses (fully connected) avec des activations ReLU et des couches de dropout pour éviter le surapprentissage. La dernière couche utilise une activation sigmoïde pour produire une sortie binaire.

Compilation et Entraînement du Modèle

Nous compilons le modèle avec un optimiseur et une fonction de perte, puis nous l'entraînons sur les données d'entraînement.

```
# Compiler le modèle
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Entraîner le modèle
history = model.fit(X_train, y_train, epochs=10, batch_size
=32, validation_data=(X_val, y_val))
```

Détails Techniques :

- **Compilation** : Le modèle est compilé avec l'optimiseur Adam, la perte de cross-entropie binaire et une métrique d'exactitude.
- **Entraînement** : Le modèle est entraîné pendant 10 époques avec une taille de lot de 32, en utilisant les données de validation pour ajuster les hyperparamètres.

Évaluation du Modèle

Nous évaluons les performances du modèle sur l'ensemble de test.

```
from sklearn.metrics import classification_report, confusio
n_matrix

# Évaluer les performances du modèle
loss, accuracy = model.evaluate(X_test, y_test)
print(f'Loss: {loss}')
print(f'Accuracy: {accuracy}')
```

Détails Techniques :

- **Évaluation** : Le modèle est évalué sur l'ensemble de test pour mesurer la perte et l'exactitude.
- **Résultats** : Les résultats montrent une perte de `0.0058` et une exactitude de `99.90%`, indiquant que le modèle est très performant sur les données de test.

Conclusion

Avec ces étapes, nous avons construit, entraîné et évalué un modèle de réseau de neurones capable de classifier les logs avec une grande précision. Ce document devrait fournir une compréhension claire et détaillée de chaque étape du processus, permettant à d'autres membres de l'équipe de reproduire, maintenir et améliorer le modèle en fonction des besoins futurs.