# Méthodologie d’entrainement des modèles d’encodage (Sentence Transformers) :

## Les Librairies utilisées :

* sentence\_transformers
* torch
* datasets
* sklean
* transformers

## Etapes d’entrainement des modèles :

* Finetuning des modèles sur des données d’entrainement (paires d’adresses « EDI – destinataires »)
* Tester les modèles sur des données de test qu’ils n’ont jamais vu auparavant.

## Processus d’entrainement :

### Finetuning des modèles : le finetuning des modèles est une étape cruciale dans l’entrainement et elle a pour but de trouver les hyper-paramètres optimaux des modèles.

### Les hyper-paramètres à optimiser :

* + - Batch\_size : c’est la taille de l’échantillon pris à chaque fois que le modèle rétro-propage sa mise à jour des poids.
    - Learning\_rate : c’est le delta ou le taux de mise à jour des poids du modèle pendant la rétro-propagation
    - Epoch : c’est le nombre d’itérations pour lesquelles, le modèle s’entraine sur tout le dataset du train

### Les métriques d’évaluation pour le finetuning : puisque les modèles seront entrainés dans le but de faire, par la suite, de la récupération d’information (IR), l’évaluation du finetuning sera effectuée via les métriques IR adaptées:

* + - **MAP@k** : **La précision moyenne (AP)** est la moyenne des précisions obtenues à chaque rang où un document pertinent apparaît pour une requête donnée. Le **MAP** est ensuite la moyenne des AP sur un ensemble de requêtes.

**But :** Évaluer globalement la précision du système de récupération d'information sur toutes les requêtes, en mettant l'accent sur les documents pertinents en haut de la liste.

* + - **MRR@k** : La moyenne des inverses des rangs des premiers documents pertinents pour un ensemble de requêtes.

**But :** Mesurer la position du premier document pertinent dans les résultats de recherche, valorisant les systèmes qui placent les documents pertinents plus haut.

* + - **NDCG@k** : La **DCG normalisée** est le **DCG** d'une liste de résultats divisé par le **DCG** **idéal**, tenant compte de la pertinence et de la position des documents.

**But :** Comparer l'efficacité des systèmes de récupération en tenant compte à la fois de la pertinence des résultats et de leur position, normalisée pour différentes requêtes.

### La stratégie d’entrainement :

* + - **Le finetuning sera effectué via la cross-validation** : c’est une méthode d’entrainement qui a pour but d’entrainer et valider le modèle sur différentes divisions de dataset (train data – val data) au nombre de fois qui égale au nombre de divisions.
    - **La validation des modèles** : la validation se fera à travers une fonction d’évaluation qui aura comme tâche de calculer les métriques d’évaluation voulues sur la val data.

### La structuration des inputs pour les modèles : Le type de la fonction de perte choisie pour les modèles, définit comment l’input doit être structuré :

* + - **Type de fonction de perte utilisée** : La **MultipleNegativeRankingLoss** est une fonction de perte qui est préférée lorsqu’on souhaite entrainer des sentences transformer pour la tâche de récupération d’information.
    - **La structure des inputs** : la perte requière qu’on structure le dataset sous forme de paires de texte (adresse EDI, adresse destinataire).

### Les étapes du finetuning :

* + - Chargement du dataset préparé au préalable pour l’entrainement.
    - Chargement des données préparées pour la création des dictionnaires d’évaluation.
    - Définitions des listes d’hyper-paramètres dans lesquels, nous devrons trouver la combinaison optimale.
    - Chargement du modèle de base sur lequel, le sentence transformers sera finetuné (le modèle de base est un Transformers également).
    - Définitions des divisions du dataset (les kfolds ou k divisions).
    - La lecture du train data dans un **data\_loader** qui s’en chargera de mettre les données en format d’entrainement.
    - Initialisation du modèle sentences transformers.
    - Initialisation de l’évaluateur du modèle qui s’en chargera de calculer les métriques sur le val data.
    - Entrainement du modèle en bouclant sur les différentes combinaisons d’hyper-paramètres puis sur les différents kfolds.
    - Chargement des scores d’évaluation et calculer la moyenne à travers les kfolds pour valider la stabilité du modèle et choisir les meilleurs hyper-paramètres.

### Réentrainement finale :

### Entrainement des modèles sur tout le dataset sans division : une fois les hyper-paramètres optimaux validés, nous allons réentrainer les modèles sur tout le dataset utilisé lors du finetuning en prenant en compte les hyper-paramètres choisis.

## Test finale des modèles :

### Evaluer les modèles finaux sur le dataset de test : calculs des scores d’évaluation sur un dataset de test jamais vu auparavant.

### Validation des modèles : comparaison entre les scores moyens préenregistrés pendant le finetuning et les scores finaux de test.

## Code d’entrainement des modèles d’encodage :

### Installation des librairies utilisées :

# !pip install sentence\_transformers

# !pip install torch

# !pip install pandarallel

# !pip install datasets

# !pip install transformers[torch]

### Importation des packages d’entrainement et d’autres packages auxiliaires

import pandas as pd

import scipy.stats as stats

import pandarallel

from pandarallel import pandarallel

import numpy as np

from tqdm.auto import tqdm

import random

from random import shuffle

import multiprocessing

import pickle

import os

import shutil

import time

from collections import Counter

pandarallel.initialize(progress\_bar=True, nb\_workers=8)

# os.environ["TOKENIZERS\_PARALLELISM"] = "false"

from sentence\_transformers import SentenceTransformer, losses, evaluation, models

from sentence\_transformers import datasets, util, InputExample

from torch.utils.data import DataLoader

from datasets import Dataset

from sklearn.model\_selection import KFold

import logging

from transformers import AdamW

import torch

from torch import optim

from CustomInformationRetrievalEvaluator import CustomInformationRetrievalEvaluator

from sentence\_transformers.evaluation.SimilarityFunction import SimilarityFunction

from sentence\_transformers.util import cos\_sim, dot\_score

### Chargement des données

# Données de train et de test

data\_df = pd.read\_csv('data\_cv/data\_df.csv', sep=';')

test\_df = pd.read\_csv('data\_cv/test\_df.csv', sep=';')

# Données complémentaires pour la création des dictionnaires d’évaluations

with open('data\_cv/unique\_queries.pickle', 'rb') as f:

unique\_queries= pickle.load(f)

with open('data\_cv/query\_groups.pickle', 'rb') as f:

query\_groups= pickle.load(f)

### Finetunig

### Fonctions complémentaires pour la création des dictionnaires d’évaluations

# fonction complémentaire pour récupérer les alias par destinataire

def get\_alias\_or\_dest(d):

try:

return dest\_groups.get\_group(d)['alias'].tolist()

except KeyError: # Assuming KeyError is the likely exception

return [d]

# fonction qui met en liaison la requête et les alias ou destinataire associés

def get\_id\_of\_target(lst, reverse\_val\_corpus):

id\_list = []

for l in lst:

try:

id\_list.append(reverse\_val\_corpus[l])

except:

pass

return id\_lis

### code de finetuning

# Re-initialize pandarallel with no progress bars

pandarallel.initialize(progress\_bar=False, nb\_workers=8)

# Determine the device (cuda)

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# Ensure all tensors are on the same device during evaluation

def move\_to\_device(batch):

batch = [tensor.to(device) for tensor in batch]

return batch

# define hyper parameters

batch\_sizes = [64, 32, 16]

l\_rate = 3e-5

num\_epochs = 1

# loop through hyper-parameters

for batch in batch\_sizes:

# start cross validation

for seed in [49, 1, 42]:

# define kfolds since we're doing cross validation

kf = KFold(n\_splits=5, shuffle=True, random\_state=seed)

# group aliases by destinataire for evaluation dictionaries

dest\_groups = merge\_dest\_alias.groupby('dest')

k\_fold = 1

# unique\_queries loaded earlier will allows us to split every time data\_df into train and validation data

# loop through folds

for train\_index, val\_index in kf.split(unique\_queries):

# split into train and val

train = [unique\_queries[i] for i in train\_index]

val = [unique\_queries[i] for i in val\_index]

# use query\_groups to create information tuples for every query

train\_data = [(pos[4], pos[6], pos[8], pos[1], pos[2], pos[3],

pos[5], pos[7], pos[9], pos[10], pos[11], pos[12], query, pos[0]) for query in train for pos in query\_groups[query]]

val\_data = [(pos[4], pos[6], pos[8], pos[1], pos[2], pos[3],

pos[5], pos[7], pos[9], pos[10], pos[11], pos[12], query, pos[0]) for query in val for pos in query\_groups[query]]

# shuffle training data

shuffle(train\_data)

shuffle(val\_data)

## Convert sentence pairs to InputExample objects

train\_examples = []

val\_examples = []

# indexes -2 and -1 are for query and positive example respectively

for i, example in enumerate(train\_data):

train\_examples.append(InputExample(texts=[example[-2].lower(), example[-1].lower()]))

for i, example in enumerate(val\_data):

val\_examples.append(InputExample(texts=[example[-2].lower(), example[-1].lower()]))

# this part here is the creation of evaluation dictionaries

whole\_queries = [liv[-2] for liv in val\_data]

whole\_dest = [liv[-1] for liv in val\_data]

val\_queries = {str(k) : v for k, v in enumerate(whole\_queries)}

val\_corpus = list(set(list\_dest) - set(whole\_queries))

val\_corpus = {str(k): v for k, v in enumerate(val\_corpus)}

reverse\_val\_corpus = {v: k for k, v in val\_corpus.items()}

target\_docs = [get\_alias\_or\_dest(d) for d in whole\_dest[:]]

target\_docs = [lst + [d] if lst[0] != d else [d] for d, lst in zip(whole\_dest[:], target\_docs)]

target\_docs = [list(set(lst) - set([q])) for q, lst in zip(whole\_queries[:], target\_docs)]

target\_docs = {str(q): list(set(get\_id\_of\_target(v, reverse\_val\_corpus))) for (q, \_), v in zip(enumerate(whole\_queries[:]), target\_docs)}

# load transformer model

word\_embedding\_model = models.Transformer('LLMs/distilbert-base-multilingual-cased-finetuned-ban/checkpoint-18090',

tokenizer\_name\_or\_path = 'LLMs/distilbert-base-multilingual-cased-finetuned-ban/checkpoint-18090')

## use a pool function over the token embeddings of the transformer

pooling\_model = models.Pooling(word\_embedding\_model.get\_word\_embedding\_dimension())

# sentence model intiialisation

model = SentenceTransformer(modules=[word\_embedding\_model, pooling\_model])

model.to(device)

# Set logging level to INFO

logging.basicConfig(level=logging.INFO

## Define data loaders and loss function

train\_dataloader = datasets.NoDuplicatesDataLoader(train\_examples, batch\_size=batch)

train\_loss = losses.MultipleNegativesRankingLoss(model=model)

# define number of warmup steps

warmup\_steps = int(len(train\_dataloader) \* num\_epochs \* 0.1) #10% of train data

# define optimizer

optimizer = optim.AdamW(model.parameters())

# pass to device the dataloader

train\_dataloader.collate\_fn = move\_to\_device

# intialize evaluator

evaluator = CustomInformationRetrievalEvaluator(

queries=val\_queries,

corpus=val\_corpus,

relevant\_docs=target\_docs,

show\_progress\_bar=False,

map\_at\_k=[10],

accuracy\_at\_k = [k for k in range(1, 11)],

precision\_recall\_at\_k = [k for k in range(1, 11)],

score\_functions = {'cos\_sim': cos\_sim},

name = 'ir\_validation' )

print(f"training sentence transformer: batch={batch}, learning\_rate={l\_rate}, seed={seed}, fold\_number={1}...")

# start finetuning

start = time.time()

model.fit(train\_objectives=[(train\_dataloader, train\_loss)],

evaluator=evaluator,

epochs=num\_epochs,

steps\_per\_epoch=len(train\_dataloader),

warmup\_steps=warmup\_steps,

# evaluation\_steps=1000,

scheduler = 'WarmupLinear',

optimizer\_class= optimizer.\_\_class\_\_,

optimizer\_params={'lr': l\_rate, 'weight\_decay': 0.01},

use\_amp = True,

save\_best\_model= True,

show\_progress\_bar= True,

output\_path =f"Bi\_Encoders\_cv/distilbert\_models/MNR\_GeoSDistilbert\_batch\_{batch}\_lr\_{l\_rate}\_seed\_{seed}\_fold\_{k\_fold}",

checkpoint\_save\_total\_limit = 1)

end = time.time()

training\_time = end-start

print(f"time of training for epoch {1} : {(training\_time)/60} minutes...")

k\_fold += 1