自然語言處理
Natural Language Processing
Term Project
LLM – Detect AI Generated Text

國立臺北科技大學 (NTUT)

第 35 組

113598032 張字青

113598043 張育丞

113C53020 劉莉庭

113C53049 黃育承

December 24, 2024

目錄

圖目錄

表目錄

一、 題目介紹

1. **競賽背景**：隨著大型語言模型的廣泛應用，人們擔心它們可能取代或改變人類的工作，特別是在教育領域，可能影響學生技能的發展。學術界擔心 LLM 可能助長抄襲行為，因為它們能生成與人類撰寫的文件極為相似的內容。
2. **競賽目標**：參賽者需建立模型，準確偵測一篇文章是由學生撰寫還是由 LLM 生成。這有助於偵測 LLM 的特徵，推動 LLM 文件檢測技術的發展。
3. **預期目標**：透過文獻[1]的程式碼進行修改，加入文獻[2-4]的程式碼或架構，將其效果提升。
4. **評分標準**：該比賽使用對數損失函數（Logarithmic Loss）作為評估指標。這意味著你的模型需要生成每個文件屬於「AI 生成」和「人類撰寫」的機率分佈，而不是簡單的 0 或 1 分類。

- 損失函數計算

$$LogLoss = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(y^i) + (1-y_i)\log(1-\hat{y}^i))$$

其中：

y_i 是真實標籤（0 或 1）。

$\hat{y}^i$ 是模型預測的機率。

- Efficiency Score 計算

用於評斷提交的結果之依據預測後的機率與實際目標的 ROC 曲線下面積（AUC）進行評估。測試時，將會依每個 ID 進行預測是由 LLM 產生的機率。

$$Efficiency = \frac{AUC}{Benchmark - maxAUC} + \frac{RuntimeSeconds}{32400}$$

二、 資料集剖析

比賽提供了訓練、測試和樣本資料集。每一個檔案內容如下：

2.1. 文件列表

- `train.csv`
  - 包含訓練資料，格式如下：
    - `text`：一段文件。
    - `label`：文件的標籤，**0** 表示人類撰寫，**1** 表示 AI 生成。
- `test.csv`
  - 包含測試資料，只有 `text`，需要參賽者預測它們的標籤。
- `sample_submission.csv`
  - 提供了測試集的預測格式，參賽者需要生成類似的提交文字。

2.2. 資料集描述

競賽資料集包含約 10,000 篇文章，其中部分由學生撰寫，部分由 LLM 生成。所有文章均針對七個不同的作文題目進行撰寫。訓練集與測試集，主要是學生撰寫的文章，僅有少量生成的文章作為案例。

2.3. 評估指標

提交結果將根據預測與實際目標之間的 ROC（Receiver Operating Characteristic） 曲線下面積（Area Under Curve, AUC）進行評估。參賽者需為測試集中的每個 ID 預測該文章由 LLM 生成的機率。

- **評估檔案 <submission.csv>**

```
id, generated
0000aaaa, 0.1
1111bbbb, 0.9
2222cccc, 0.4
```

2.4. 資料特徵的捕捉

- AI 生成的文字可能有特定模式，例如：較高的一致性或缺乏人類特有的錯誤。
- 人類撰寫的文字可能存在錯別字、非正規文法等特徵。

2.5. 過擬合問題

- 訓練模型時需要注意，訓練資料過擬合的問題，可能導致對測試資料的泛化能力差。

2.6. 模型選擇

- 需要選擇適合自然語言處理（NLP）的模型，例如：Transformer-based 模型。

三、 環境簡介與架構說明

3.1. 環境簡介

在題目中，我們使用多種運行環境進行實驗，其中包括必要的 Kaggle 平台，額外也在 Google Colab 和個人 PC 上進行了嘗試。以下將各環境的硬體與軟體配置整理為參考表（見表 1 和表 2），以便於部署需求的確認。在 PC 和 Google Colab 運行時，程式碼可以正常連接網路，因此對於套件的使用無過多限制。然而，在 Kaggle 環境中，由於平台防止作弊的政策，網路連線被限制，導致 Huggingface 模型無法使用連網方式運行。為了解決此問題，我們引用了其他用戶適配的 Kaggle 環境的模型[1-3]，成功完成了部署。

表 1 各式執行環境

No.	Component	PC Specification	Google Colab Specification	Kaggle Specification
1	Running Environment	System: Windows 11 Pro, CPU: Intel Core i9-12900K, RAM: 16GB * 2 (32GB), GPU: NVIDIA RTX 3090	GPU: T4 * 2	GPU: P100
2	Python Version	3.9.18	3.10	3.10.13

表 2 環境之套件版本

No.	Package Name	Kaggle Version	Colab Version	PC Version
1	Pandas	2.2.3	2.2.2	2.3.3
2	PyTorch（GPU）	2.4.0	2.5.1	2.4.0
3	tqdm	4.66.4	4.67.1	4.66.5
4	Transformers	4.46.3	4.47.1	4.47.0
5	Scikit-learn	1.2.2	1.6.0	1.6.0
6	Numpy	1.26.4	1.26.4	1.26.4
7	Matplotlib	3.7.5	3.8.0	3.9.4

3.2. 架構說明

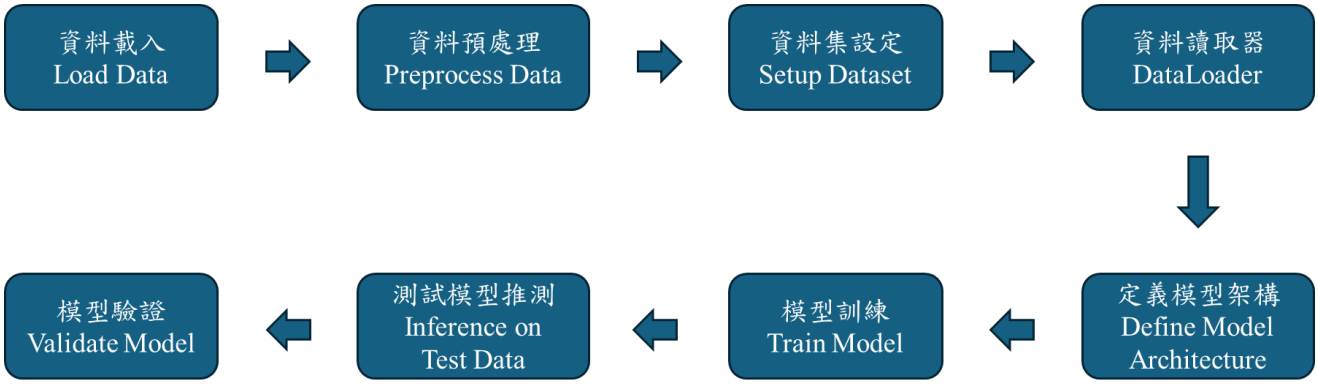

圖 1 系統功能運作流程圖

我們根據完成的程式過程，繪製**錯誤! 找不到參照來源。**系統功能運作流程圖，並整理出以下 8 個步驟：

1. 資料載入 (Load Data)：將原始資料導入系統。
2. 資料預處理 (Preprocess Data)：對資料進行整理、格式轉換及異常值處理。
3. 資料集設定 (Setup Dataset)：將資料分別輸入訓練集和測試集。
4. 資料讀取器 (DataLoader)：使用資料讀取器分批載入資料。
5. 定義模型架構 (Define Model Architecture)：設計 NLP 模型的結構。
6. 模型訓練 (Train Model)：使用訓練集進行模型的參數學習。
7. 測試模型推測 (Inference on Test Data)：在測試集上進行模型推理測試。
8. 模型驗證 (Validate Model)：驗證模型的準確性及泛化能力。

四、 程式實作

```
import pandas as pd
import torch
from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler
from transformers import AutoTokenizer, AutoModel
from torch import nn
from torch.optim import AdamW
from tqdm import tqdm
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import SGDClassifier, LogisticRegression
from sklearn.ensemble import VotingClassifier
import unicodedata
import re
import numpy as np
import random
```

載入這次專案中所有會被使用到的套件。

- **import pandas as pd**

  用於資料處理和分析。pandas 提供了強大的 DataFrame 結構，便於操作清單資料（例如讀取 CSV 檔案、進行資料過濾和清理）。

- **import torch**

  PyTorch 是一個強大的機器學習框架，支持自動微分、模型建構和訓練。

- **from torch.utils.data import Dataset, DataLoader, WeightedRandomSampler**
  - Dataset：用於定義自定義資料集，便於與模型訓練對接。
  - DataLoader：用於將資料集分批處理（mini-batch），並提供多執行序以加速資料載入。
  - WeightedRandomSampler：用於解決資料不平衡問題。
- **from transformers import AutoTokenizer, AutoModel**
  - AutoTokenizer：自動載入適用於預訓練模型的分詞器，將文字轉換為向量。
  - AutoModel：自動載入適用於特定任務的預訓練模型（例如 BERT、GPT 等）。
- **from torch import nn**

  PyTorch 的神經網路模型，提供模型結構（例如線性層、激勵函數）的基礎結構塊。

- **from torch.optim import AdamW**

  AdamW 是一種改進的優化器，適合深度學習模型的訓練，特別是 NLP 領域的預訓練模型。

- **from tqdm import tqdm**

  提供進度條工具，用於跟蹤迭代過程中的進展，增強程式執行的可視化。

- **from sklearn.metrics import roc_auc_score**

  計算 ROC 曲線下面積（AUC），用於二元分類模型的性能評估。

- **from sklearn.model_selection import train_test_split**

  將資料集分為訓練集與測試集，支持隨機拆分並設置比例。

- **from sklearn.feature_extraction.text import TfidfVectorizer**

  用於將文字轉換為 TF-IDF 特徵向量，通常作為傳統機器學習模型的輸入。

- **from sklearn.linear_model import SGDClassifier, LogisticRegression**
  - SGDClassifier：基於隨機梯度下降的分類器，適用於分配不均勻的資料。
  - LogisticRegression：線性分類模型，常用於二分類任務。

- **from sklearn.ensemble import VotingClassifier**

  將多個分類器結合在一起，通過投票決定最終輸出。

- **import unicodedata**

  提供對 Unicode 字元的操作，常用於整理和標準化文字資料。

- **import re**

  正則化模型，用於文字的模式匹配和整理，例如：去除多餘符號或格式化資料。

- **import numpy as np**

  用於數學計算和操作多維資料陣列，常與 PyTorch 和 Scikit-learn 一起使用。

- **import random**

  提供隨機數生成功能，常用於資料打亂或抽樣。

```
# ==================== 資料載入與處理 ====================
DATA_PATH = "/kaggle/input/llm-detect-ai-generated-text"
ADDITIONAL_DATA_PATH = "/kaggle/input/daigt-v2-train-dataset/train_v2_drcat_02.csv"

train_essays = pd.read_csv(f"{DATA_PATH}/train_essays.csv")
train_prompts = pd.read_csv(f"{DATA_PATH}/train_prompts.csv")
test_essays = pd.read_csv(f"{DATA_PATH}/test_essays.csv")
additional_data = pd.read_csv(ADDITIONAL_DATA_PATH)

# 合併資料並重命名標籤
train_data = pd.merge(train_essays, train_prompts, on="prompt_id", how="left")
train_data.rename(columns={"generated": "label"}, inplace=True)

# 加入補充資料並平衡權重
additional_data = additional_data[additional_data["RDizzl3_seven"] == True]
additional_data.rename(columns={"label": "label"}, inplace=True)
train_data = pd.concat([train_data, additional_data], ignore_index=True)
```

讀取這次專案中使用到的資料集，包含訓練集、測試集。

- DATA_PATH: 主資料集的文字路徑，用於讀取專案的訓練和測試資料。
- ADDITIONAL_DATA_PATH: 補充資料集的文字路徑，可能包含來自其他來源的資料，用於擴充訓練資料集。

- 使用 pandas.read_csv 方法讀取 CSV 文字，將資料載入為 DataFrame 格式：
- train_essays: 包含訓練集的文章資料，通常包括文章文字和生成標籤。
- train_prompts: 包含訓練集的文章 題目，可能與作文內容相關聯。
- test_essays: 包含測試集的文章資料，用於驗證模型表現。
- additional_data: 補充資料，提供更多的文件和標籤，可能來自其他來源以增強模型訓練效果。

```
# 資料清理函數
def clean_text(text):
    text = unicodedata.normalize("NFKD", text)
    text = re.sub(r"<.*?>", "", text)
    text = re.sub(r"\[.*?\]", "", text)
    text = text.encode("ascii", "ignore").decode("ascii")
    text = re.sub(r"http\S+", "", text)
    text = re.sub(r"[^\w\s]", "", text)
    text = re.sub(r"\s+", " ", text).strip()
    return text
```

清理文字中的雜訊，為模型提供更乾淨的輸入。

- unicodedata.normalize("NFKD", text)：將文字轉換為規範的 Unicode 格式，便於進一步處理。
- re.sub(r"<.*?>", "", text)：移除 HTML 標籤。
- re.sub(r"\[.*?\]", "", text)：移除中括號及其中的內容。
- text.encode("ascii", "ignore").decode("ascii")：移除非 ASCII 編碼字元，去除特殊符號或異常字元。
- re.sub(r"http\S+", "", text)：移除 URL（以 http 開頭的鏈接部分）。

```
# 資料增強：隨機刪除單字
def augment_text(text, drop_prob=0.1):
    words = text.split()
    augmented = [word for word in words if random.uniform(0, 1) > drop_prob]
    return " ".join(augmented)

train_data["text"] = train_data["text"].map(clean_text).map(lambda x: augment_text(x,
drop_prob=0.1))
test_essays["text"] = test_essays["text"].map(clean_text)
```

- re.sub(r"[^\w\s]", "", text)：移除所有標點符號，保留字母和空格。

- re.sub(r"\s+", " ", text).strip()：移除多餘的空格，並刪除頭尾空白字元。

實現資料增強，通過隨機刪除單字增加訓練資料的多樣性，讓模型更具穩定性。

text.split()：將文字按空格分割成單字列表。

- random.uniform(0, 1) > drop_prob
  - 使用隨機數判斷是否保留單字。
  - 機率由參數 drop_prob 決定，預設為 0.1。

- 刪除單字：只有隨機數大於 drop_prob 時，該單字才會被保留。
- 返回結果：使用 " ".join(augmented) 將剩餘單字拼接回字串。
- 訓練資料處理
  - train_data["text"].map(clean_text)：對訓練資料中的 text 列應用 clean_text 函數，完成文字整理。
  - map(lambda x: augment_text(x, drop_prob=0.1))：在清理後的文字上執行資料增強，模擬更多變的輸入情況。
- 測試資料處理：只執行 clean_text，保持測試資料穩定，避免干擾模型評估。

```
# 分割訓練和驗證集
train_df, val_df = train_test_split(train_data, test_size=0.2, stratify=train_data["label"], random_state=42)
```

將訓練資料分割為訓練集和驗證集，用於模型訓練與評估。

- train_data：原始訓練資料。
- test_size=0.2：20% 的資料用於驗證，其餘 80% 用於訓練。
- stratify=train_data["label"]：根據標籤 label 進行分層抽樣，確保訓練集和驗證集中各類別的比例一致。
- random_state=42：設置亂數種子，確保分割結果可重現。

```
# ====================== Dataset 和 DataLoader ======================
tokenizer = AutoTokenizer.from_pretrained("/kaggle/input/huggingface-bert-variants/bert-base-uncased/bert-base-uncased")
evice = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

- Tokenizer
  - 使用 Hugging Face 的 AutoTokenizer，載入預訓練模型 bert-base-uncased 的 tokenizer。
  - 功能
    - 將文字轉換為模型可接受的格式（如 token IDs 和 attention masks）。
    - 處理包括填充、截斷等，確保輸入長度一致。
    - Device:根據是否有可用的 GPU，自動設置設備為 cuda（GPU）或 CPU。

```
class EssayDataset(Dataset):
    def __init__(self, df, is_test=False):
        self.inputs = tokenizer(
            df["text"].tolist(),
            padding="max_length",
            truncation=True,
            max_length=512,
            return_tensors="pt")
        self.labels = torch.tensor(df["label"].values, dtype=torch.float) if not is_test else
None
        self.is_test = is_test
```

- 初始化
  - df: 輸入資料框，包含文字和標籤。
  - is_test: 指示是否為測試資料。
- 如果是測試資料，則不設置標籤。
  - self.inputs:
- 使用 tokenizer 將文字轉換為模型輸入格式。
  - 參數使用
    - padding="max_length"：將所有文字填充到最大長度（max_length=512）。
    - truncation=True：如果文字長度超過 512，則進行截斷。
    - return_tensors="pt"：將輸出轉換為 PyTorch 張量（Tensors）格式。
  - self.labels:
    - 如果不是測試資料，將標籤轉換為 PyTorch 張量（Tensors）。
    - 標籤類型設為 float，便於用於二分類問題。

```
def __len__(self):
    return len(self.inputs["input_ids"])
def __getitem__(self, idx):
    input_ids = self.inputs["input_ids"][idx]
    attention_mask = self.inputs["attention_mask"][idx]
    if self.is_test:
        return {"input_ids": input_ids, "attention_mask": attention_mask}
    label = self.labels[idx]
    return {"input_ids": input_ids, "attention_mask": attention_mask, "label": label}
```

返回資料集的長度，即樣本數。

- 索引訪問樣本
  - 通過 idx 得到指定索引的資料。
  - 提取 input_ids 和 attention_mask 作為模型輸入。

- 測試資料
  - 如果是測試資料，不提供 label。
- 訓練/驗證資料
  - 返回包含 input_ids、attention_mask 和 label 的字典。

```
# 平衡資料樣本
class_counts = train_df["label"].value_counts()
class_weights = 1.0 / class_counts
```

- 平衡正負樣本數量，減少資料不均對模型的影響。
  - 使用 value_counts 計算每個類別的樣本數。
  - 通過取倒數 (1.0 / class_counts) 計算每個類別的權重。
    - 類別樣本越少，權重越大。

```
sample_weights = train_df["label"].map(class_weights).values
sampler = WeightedRandomSampler(sample_weights, len(sample_weights))
```

- map(class_weights)
  - 根據 class_weights 為每一個樣本分配對應權重。
- WeightedRandomSampler
  - 使用樣本權重進行隨機抽樣。
  - 確保每一個類別的樣本被抽到的機率與權重成正比。

```
train_loader = DataLoader(EssayDataset(train_df), batch_size=16, sampler=sampler)
val_loader = DataLoader(EssayDataset(val_df), batch_size=16, shuffle=False)
test_loader = DataLoader(EssayDataset(test_essays, is_test=True), batch_size=16, shuffle=False)
```

- EssayDataset：將定義的 EssayDataset 類作為資料來源。
- batch_size=16：每批載入 16 條樣本。
- sampler=sampler（僅訓練資料）：使用加權隨機抽樣確保類別平衡。
- shuffle=False：測試和驗證資料不進行隨機打亂，保持資料順序。

```
# ====================== 模型定義 ======================
class CustomBERTModel(nn.Module):
    def __init__(self, pretrained_model="/kaggle/input/huggingface-bert-variants/bert-base-
uncased/bert-base-uncased"):
        super(CustomBERTModel, self).__init__()
        self.bert = AutoModel.from_pretrained(pretrained_model)
        self.dropout = nn.Dropout(0.3)
        self.classifier = nn.Sequential(
            nn.Linear(self.bert.config.hidden_size, 256),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

    def forward(self, input_ids, attention_mask):
        outputs = self.bert(input_ids=input_ids, attention_mask=attention_mask)
        pooled_output = self.dropout(outputs.pooler_output)
        logits = self.classifier(pooled_output)
        return logits

model = CustomBERTModel().to(device)
optimizer = AdamW(model.parameters(), lr=2e-5, weight_decay=1e-4)
criterion = nn.BCEWithLogitsLoss()
```

- **別定義**
  - 繼承自 PyTorch 的 nn.Module，用於建構自定義神經網路模型。
- **初始化**
  - 接收預訓練模型的路徑作為參數（預設使用 bert-base-uncased）。
  - 調用 super() 初始化父類別的功能。
- **self.bert**
  - 載入 Hugging Face 的預訓練 BERT 模型。
  - AutoModel 提供 BERT 的基本架構，不包含特定任務的標頭。
- **Dropout**
  - 設置丟棄率為 30%，用於防止過擬合。
  - 隨機將一部分神經元設為 0，在訓練時增加模型的泛化能力。
- **分類器部分**
  - 分層結構

    i. 第一層

    - 將 BERT 的隱藏層輸出（大小由 self.bert.config.hidden_size 定義）映射到 256 維。
    - 激勵函數：使用 ReLU 非線性激勵。
    - Dropout：丟棄率設為 20%。

ii. 第二層

- 將 256 維輸出映射到 128 維。
- 激勵函數：再次使用 ReLU。

iii. 第三層

- 最終輸出 1 維（因為是二分類問題，輸出單個邏輯值）。

- **AdamW**

  一種改進的 Adam 優化器，加入了權重衰減（weight_decay）以正則化模型。

- **參數**
  - model.parameters()：傳入模型的參數，用於優化。
  - 學習率 (lr=2e-5)：設定為 2e-5，適合微調預訓練模型的初始學習率。
  - 權重衰減 (weight_decay=1e-4)：防止參數過大，提升模型泛化能力。
- **BCEWithLogitsLoss**
  - 二分類交叉熵損失，內部結合了 sigmoid 函數。
  - 適用於輸出未經激勵的邏輯（logits）。

```
# 動態學習率調整
from torch.optim.lr_scheduler import CosineAnnealingLR
scheduler = CosineAnnealingLR(optimizer, T_max=10)
```

- **Optimizer**
  - 將調整的對象設置為之前定義的 optimizer。
  - 該優化器中的學習率會根據餘弦調整策略逐步更新。
- **T_max=10**
  - 指定完整的餘弦週期為 10 epochs，表示學習率在 10 epochs 內從初始值逐步減小至接近零。

```
# ==================== 訓練與驗證 ====================
epochs = 10
for epoch in range(epochs):
    model.train()
    train_loss = 0
    for batch in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["label"].to(device)

        optimizer.zero_grad()
        outputs = model(input_ids, attention_mask).squeeze(-1)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    scheduler.step()
    print(f"Epoch {epoch+1} Loss: {train_loss / len(train_loader):.4f}")
```

- **前向傳播**
  - 將 input_ids 和 attention_mask 傳遞到模型中，得到預測結果 outputs。
  - 使用 .squeeze(-1) 簡化輸出，因為二分類只需要單個輸出值。
- **損失計算**
  - 使用定義的 BCEWithLogitsLoss 計算模型預測結果與標籤之間的誤差。
- **梯度清零 (zero_grad)**
  - 清除上一個 batch 累積的梯度，避免干擾本次更新。
- **反向傳播 (backward)**
  - 計算損失函數對模型參數的梯度。
- **參數更新 (step)**
  - 使用優化器根據計算出的梯度更新模型參數。
  - 將每一個 batch 的損失累加，便於後續計算平均損失。
  - 使用 CosineAnnealingLR 動態調整學習率，逐步減小以促進收斂。
  - 計算每個 Epochs 的平均損失並輸出，用於監控模型的訓練效果。

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from tqdm import tqdm

# ====================== 訓練與驗證 ======================
epochs = 10
for epoch in range(epochs):
    model.train()
    train_loss = 0

    # 訓練
    for batch in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["label"].to(device)

        optimizer.zero_grad()
        outputs = model(input_ids, attention_mask).squeeze(-1)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()
```

- **訓練階段**
  - 訓練過程使用 model.train()，啟用 dropout 等訓練行為。
  - 使用 optimizer.zero_grad() 清除累積的梯度。
    - 前向傳播 (model(input_ids, attention_mask)) 計算輸出結果。
    - 損失函數計算後進行反向傳播 (loss.backward()) 和優化 (optimizer.step())。

```
# 驗證並畫 ROC 圖
model.eval()
all_labels = []
all_preds = []

with torch.no_grad():
    for batch in val_loader:
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        labels = batch["label"].to(device)

        outputs = model(input_ids, attention_mask).squeeze(-1)
        all_labels.extend(labels.cpu().numpy())
        all_preds.extend(outputs.cpu().numpy())

# 計算 FPR, TPR 和 AUC
fpr, tpr, _ = roc_curve(all_labels, all_preds)
roc_auc = auc(fpr, tpr)

# 繪製 ROC 曲線
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f'Receiver Operating Characteristic - Epoch {epoch+1}')
plt.legend(loc="lower right")
plt.show()
```

- **驗證階段**
  - 設定模型為驗證模式，關閉 dropout 等隨機性層。
  - 使用 torch.no_grad() 禁用梯度計算以節省記憶體。
- **收集標籤和預測**

  將每個 batch 的實際標籤 (labels) 和模型預測結果 (outputs) 收集起來，用於後續的評估。
- **ROC 曲線和 AUC 計算**
  - ROC 曲線

    不同閾值下的假正率 (FPR) 和真正率 (TPR)。
  - AUC (Area Under Curve)
    - 計算 ROC 曲線下的面積，用於量化模型的分類性能。
    - AUC 值範圍為 [0, 1]，越接近 1 表示模型性能越好。

- **繪製 ROC 曲線**
  - 使用 Matplotlib 繪製 ROC 曲線。
  - 橘色曲線表示模型的 ROC 曲線，虛線表示隨機猜測的基線。

```
# ==================== 測試階段 ====================
model.eval()
test_logits = []
with torch.no_grad():
    for batch in tqdm(test_loader, desc="Predicting on Test Data"):
        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        outputs = model(input_ids, attention_mask).squeeze(-1)
        test_logits.extend(torch.sigmoid(outputs).cpu().numpy())
```

- **模型設定為推論模式**
  - 設置模型為驗證模式
  - 關閉 dropout 等隨機層，確保模型的行為一致。
  - 增加推論過程的穩定性。
- **禁用梯度計算**
  - 禁用梯度計算，減少記憶體使用，提高計算效率。
  - 減少不必要的梯度圖構建開銷，因為推論過程不需要進行反向傳播。
- **批量推論**
  - 從 test_loader 中批量讀取資料，確保不會因資料量過大而耗盡記憶體。
  - 輸入包括：
    - input_ids: 編碼後的測試文字資料。
    - attention_mask: 注意力遮罩，用於處理填充 (padding)。
    - 使用模型進行前向傳播，獲得每一個樣本的原始預測分數（logits）。
- **收集預測結果**
  - 將模型的輸出通過 sigmoid 函數，將 logits 映射到 [0, 1] 範圍內，表示生成的機率。
  - 將結果移至 CPU，轉換為 NumPy 格式，並追加至 test_logits 列表。

```
# ==================== 傳統特徵工程與集成 ====================
vectorizer = TfidfVectorizer(max_features=30000, ngram_range=(3, 5), sublinear_tf=True)
train_tfidf = vectorizer.fit_transform(train_df["text"])
test_tfidf = vectorizer.transform(test_essays["text"])

ensemble_model = VotingClassifier(
    estimators=[
        ("sgd", SGDClassifier(max_iter=1000, loss="modified_huber")),
        ("logreg", LogisticRegression(solver="liblinear")),
    ],
    voting="soft"
)
ensemble_model.fit(train_tfidf, train_df["label"])
test_predictions = ensemble_model.predict_proba(test_tfidf)[:, 1]
```

- TfidfVectorizer 用來將文字資料轉換為 TF-IDF 特徵。
  - max_features=30000: 限制最大特徵數量為 30,000，防止特徵過多。
  - ngram_range=(3, 5): 使用 3 到 5 節點的 n-gram，這有助於捕捉更高階的語言結構，適用於文章資料中有較長文字模式的情況。
  - sublinear_tf=True: 使用線性轉換，這樣頻率較高的詞語會被壓縮，降低它們對模型的影響。
- fit_transform()：對訓練集文字進行學習，並將其轉換為 TF-IDF 特徵。
- transform()：僅對測試集進行轉換，這樣測試集會使用與訓練集相同的特徵空間。

這會將文字資料轉換為數值矩陣，並且每一行代表一個文字樣本，每一列代表一個特徵（n-gram）在文字中出現的頻率。

- VotingClassifier 是一種集成方法，將多個基礎模型的預測進行集成，最常見的有兩種投票方式：
  - voting="soft": 使用機率進行加權平均，基於每一個模型的預測機率來計算最終的預測結果。這通常能提供比直接投票（基於類別標籤的投票）更穩定的結果。
- 這裡使用了兩個基礎模型：
  - SGDClassifier：使用隨機梯度下降法的分類器，適合處理大規模資料，並且支持多種損失函數（這裡使用 modified_huber 損失）。
  - LogisticRegression：邏輯回歸分類器，使用 liblinear 進行計算。

五、 成品展現

我們基於文獻[1]的程式碼進行復現，並觀察其圖 2 之 ROC 曲線，發現初始效果不佳，在圖 3 其中 Private Score 為 0.621749，Public Score 為 0.614623。經過我們的改進實作，圖 4 的 ROC 曲線顯示出更佳的表現，雖然看似可能過擬合，但從表 3 中的十個 Epochs 的損失函數數值可知，實際並非如此。

在優化後的模型中，我們成功在圖 5 將 Private Score 提升至 0.697016，Public Score 提升至 0.786157，相較於原始結果，Private Score 提升了 0.075267，而 Public Score 則提升了 0.171534，顯示出我們改進後方法的優良性能。這些改進成果表明，我們的調整策略在實現模型性能提升方面具有顯著效果。

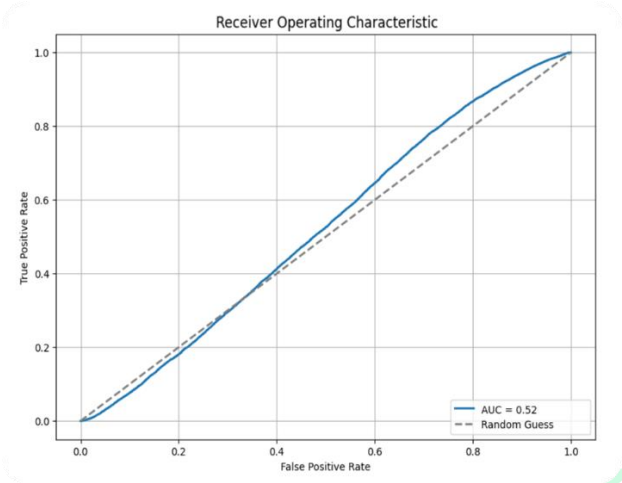

圖 2 原始 ROC 結果

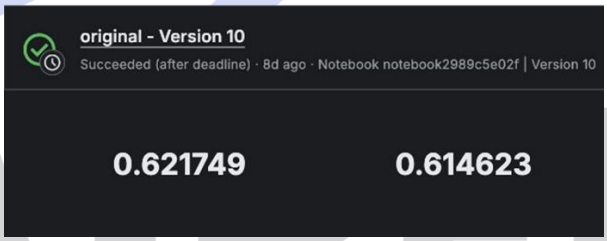

圖 3 原始分數

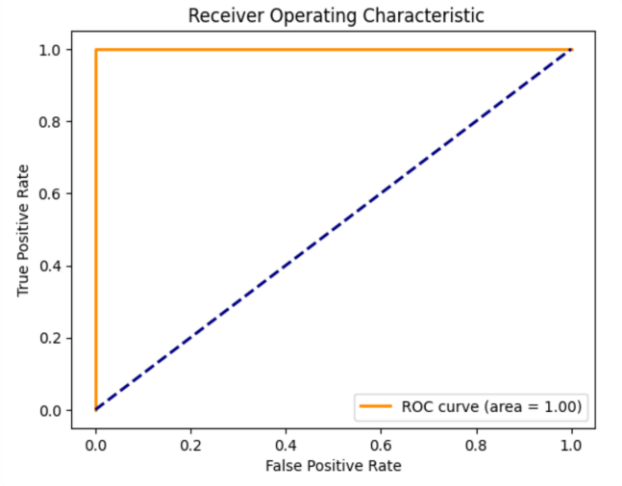

圖 4 調整後 ROC 結果

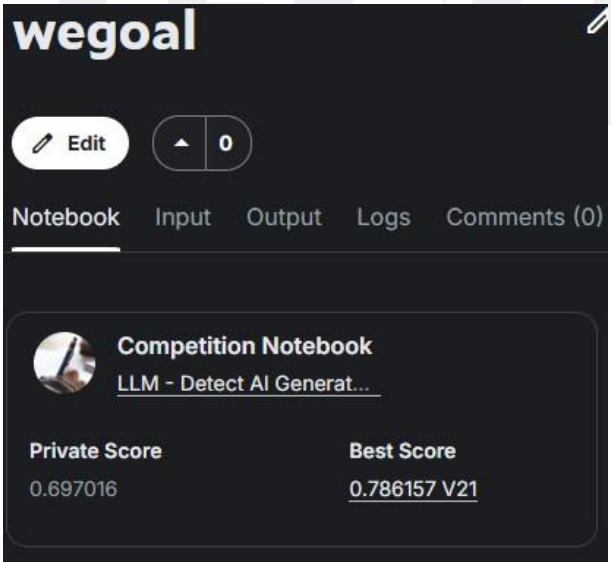

圖 5 調整後分數

表 3 調整後的 10 個 Epochs 之損失函數數值

Epochs	1	2	3	4	5	6	7	8	9	10
Loss Value	0.0966	0.0137	0.0072	0.0084	0.0071	0.0040	0.0043	0.0049	0.0032	0.0001

六、 其他方法

為了嘗試更多創新的解決方案，我們參考了文獻[4]中提出的建議及作法。該作法曾榮獲本次競賽的第七名，顯示其具有一定的參考價值。因此，我們根據其方法進行調整並加以修改。

在實作過程中，我們對資料處理及模型進行了多次測試與更換。然而，由於嘗試多種方法的同時，部分處理方式可能導致模型效果出現微幅下滑。此外，受到作業時間的限制，我們未能進一步優化，故暫時將該部分研究告一段落。

我們的成績如圖 7 所示，雖然最終結果未能超越參考作法圖 6 的分數，因而未列為主要內容，但我們已掌握了部分影響表現的問題，並正嘗試透過多元方向的修改與優化來解決這些挑戰，為後續改進奠定了基礎。

Model	Dataset	Public LB	Private LB
DeBERTa-v3-base-512	only persuade	0.869	0.875
DeBERTa-v3-base-512	persuade & slimpajama	0.921	0.920
DeBERTa-v3-base-1024	persuade & slimpajama	0.910	0.901
DeBERTa-v3-large-512	persuade & slimpajama	0.942	0.965
DeBERTa-v3-large-1024	persuade & slimpajama	0.922	0.957
DeBERTa-v3-large-512 + TF-IDF	persuade & slimpajama	0.942	0.965

圖 6 原始成績[4]

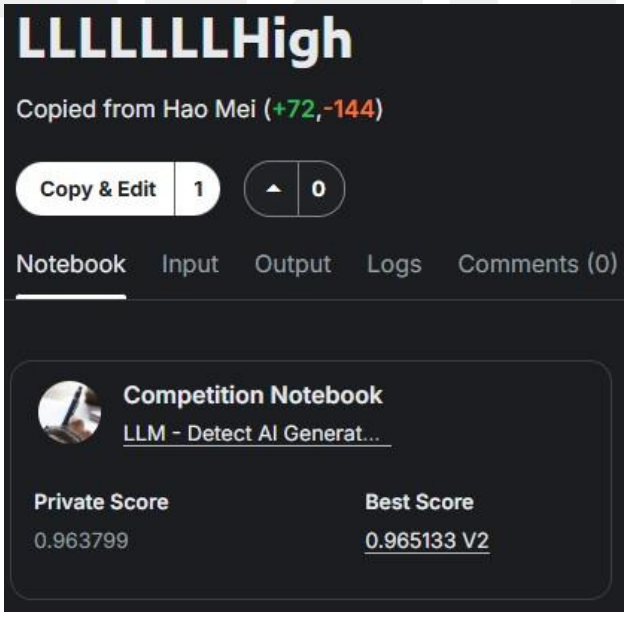

圖 7 調整成績

七、 總結

本研究基於文獻[1]的基礎程式碼進行調適，在初期提案過程中發現程式碼可能存在「先看到答案」的潛在疑慮。經過多次測試與驗證，我們透過修改數據引用及處理方式（例如平衡資料分布等方法），成功排除了該疑慮。

經此調整後，不僅有效解決了原始程式碼中的問題，還顯著提升了模型的效率與成果表現。這些改進為本研究的結果提供了更高的可靠性與說服力，並展示了調適策略在處理實際應用問題中的價值。

八、 未來展望

任何實驗都會遇上許多瓶頸，或許現在無法突破但我們試圖解決，以下為我們期盼進行達成的目標。

8.1. 資源問題

在資源方面，我們目前採用的是公開的 **bert-base-uncased** 模型[5]，但由於該模型的限制，無法更深入分析語意。此外，我們曾嘗試切換至 **microsoft/deberta-v3-large** 模型，但受到硬體資源不足的限制，無法順利運行。因此，資源瓶頸是我們無法進一步提升分數的主要原因之一。

在程式碼方面，我們基於文獻[1]為基底追加文獻[2]及文獻[3]的架構進行修改，儘管嘗試增加其他資料集，分數未出現顯著提升，顯示出當前程式碼架構存在一定的性能瓶頸。然而，透過多次優化與調整，我們已將原始分數成功提升了 0.171534。雖然尚未達到理想的高度，但這證明了我們當初的假設方向是正確的，且具備進一步探索與優化的潛力。

未來，我們希望能克服資源與程式碼架構的限制，進一步嘗試更多先進模型和技術，探索能夠實現更高效能的方法，為研究的深化提供更堅實的基礎。

8.2. 程式問題

目前，由於文獻[1]中所使用模型的限制，其最佳分數只能停留在 0.78 的準確率。基於此限制，我們預期可以嘗試混合其他模型的結果，以期突破目前的瓶頸。

在本次專案中，我們於「六、其他方法」部分展示了一種嘗試的新方法（參考圖 7）。儘管目前該方法的結果尚未超越原始結果（參考圖 6），但初步實驗表明，其具有潛力，可以作為提升準確率的補充手段。

未來，我們計劃進一步研究如何有效整合該方法與原始模型，探索多模型混合策略，以提升整體的準確率，進一步突破現有的分數上限。這也為後續的改進與優化奠定了良好的基礎。

九、 結案心得

9.1. 張字青

這次專案是我首次使用 Kaggle，對我來說是一個嶄新的挑戰。起初花了不少時間熟悉平台功能，包括建立 notebook、配置環境和提交結果等過程，但隨著進度推進，逐漸掌握了操作技巧。

我們小組的選題是 LLM - Detect AI Generated Text，這是一個具時代性和挑戰性的題目。特別是 Kaggle 比賽對斷網執行的要求，對於依賴雲端資源的流程是一大障礙。為克服這個限制，我們提前下載所需模型與資料集，配置出完全離線運行的環境，同時確保所有套件都能支持離線運作，這過程中增強了我們在資源受限環境下開發的能力。

此外，這次挑戰也促使我深入學習大語言模型（LLM）的運作原理與應用。我們探討了 AI 生成文字與人類撰寫文字的差異，並嘗試利用詞嵌入、序列建模（如 BiLSTM、Transformer），以及微調預訓練模型（如 BERT）來解決問題。

整體來說，這次專案讓我熟練了 Kaggle 的操作，深入理解了 AI 文件辨識技術，也學會如何在斷網限制下高效開發。儘管過程中充滿挑戰，但在技術能力與問題解決上，我獲得了極大的成長，對未來相關工作帶來很大助益。

9.2. 張育丞

這次的題目相較於課堂作業，難度明顯提升許多。在應用課程所學的技巧時，我們將其延伸至 Kaggle 平台的實際選題中，挑戰了一個更具實際價值的問題。為了應對這次的挑戰，我們採取了一種相對務實的策略，聚焦於找尋多篇表現優異與相對普通的方法進行整合與改良，而非完全自行設計一套全新的解決方案。起初我們認為，採用相對簡單的方法能快速達成目標，但在實踐中發現，即使是參考既有方法，也需要深入的分析與適配，才能真正實現模型效能的提升與問題的解決。

這次的過程中，我們學會了如何高效檢索和篩選高質量的解決方案，同時在整合不同方法的過程中，不斷驗證並調整細節，使其能更好地適應我們的特定任務需求。這段經歷讓我們深刻體會到，學習與借鑒他人經驗固然重要，但能否在此基礎上進行創新與突破，才是技術提升的真正關鍵。這也讓我們更加理解資料分析與模型優化的複雜性，並為後續的學習與實踐奠定了堅實的基礎。

9.3. 劉莉庭

在這次 term project 中，我的目標是設計一個結合多種技術的混合模型，來提升對 AI 生成文本的檢測準確率。在初期設計過程中，我採用了基於預訓練模型（如 DeBERTa）的深度學習模型，並結合了 TF-IDF 特徵與邏輯回歸的傳統機器學習方法。然而，在編寫與執行程式碼的過程中，遇到了一些挑戰，比如資料標籤單一導致邏輯回歸模型無法正常訓練，這讓我重新思考模型架構的設計。

經過多次嘗試與調整，我使用第一名的解法，從中學習了幾個關鍵的優化策略，包括數據多樣化生成、強化模型對抗典型文本檢測系統攻擊的能力，以及利用資料增強技術來處理文本中的細微變化。這些方法啟發我進一步改良了模型，特別是增強了對生成文本的偵測能力。

此外，這次 project 也讓我深刻體會到資料處理的重要性。我將數據分成不同的版本（原始版本與強化預處理版本），並假設這種策略能讓模型學習更深層次的特徵，抵禦隱藏測試集中可能出現的隨機字元攻擊。

整體過程讓我更加理解機器學習與深度學習模型的相輔相成，也體會到創新思考在解決複雜問題中的價值。未來，我希望能將這些經驗應用到更廣泛的 NLP 領域中，推動更準確與穩健的模型發展。

9.4. 黃育承

在這次的功課我覺得非常有趣，我們透過整合多個資料集並結合深度學習與傳統機器學習方法。並透過 BERT 模型結合 TF-IDF 特徵，充分發揮深度學習的語意理解能力與傳統特徵工程的補充效果；並提升了原始的程式碼分數 16 分，這點我覺得非常驚喜！

我們透過數據清理與增強、模型優化（如隨機取樣）、以及模型融合策略是關鍵提升分數的因素，也充分利用外部數據提升了模型的泛化能力。也希望未來可進一步探索自動調參與更高效的模型融合方式，提升結果表現並減少訓練時間。

整體而言，這次的專案讓我對 NLP 任務的數據處理、模型設計與融合策略有了更深刻的感受！

十、 參考文獻

[1] Kaggle - LLM Detect: Text Cluster [中文]， https://www.kaggle.com/code/finlay/llm-detect-text-cluster。

[2] Kaggle – LLM Detect AI Generated (Bert)，https://www.kaggle.com/code/sunshine888888/llm-detect-ai-generated-bert。

[3] Kaggle - Detect AI Generated Text Using BLSTM & Distilbert，https://www.kaggle.com/code/shahbodsobhkhiz/detect-ai-generated-text-using-blstm-distilbert。

[4] Kaggle = AI Generated Text [7th Place Solution] Generate Data with Non-Instruction-Tuned Models，https://www.kaggle.com/competitions/llm-detect-ai-generated-text/discussion/470643。

[5] Kaggle – Haggingface BERT Variants，https://www.kaggle.com/datasets/sauravmaheshkar/huggingface-bert-variants。