問題一：稀疏檢索(TF-IDF vs. BM25)

在稀疏檢索方法中，比較 **TF-IDF** 和 **BM25** 的檢索性能。在此次作業中，哪種方法表現更好？並分析造成差異的可能原因（例如：詞頻處理、文件長度正規化）。

在此次作業的實驗中，經過多輪優化的 **TF-IDF 表現顯著優於 BM25**
在自建的本地驗證集上，基準 TF-IDF 的 Recall@10 分數為 0.7400，而 BM25 僅為 0.6720
即使在對 BM25 的 k1 和 b 參數進行多輪調整後，其最高分也只達到 0.6780，仍未超過 TF-IDF
最終經由對 TF-IDF 進行次線性詞頻縮放 (Sublinear TF Scaling)和查詢擴展 (Query Expansion)兩項優化，使其本地驗證分數達到了 0.7860，在 Kaggle 上取得了 **0.73600** 的成績，成功超越了 Strong Baseline

造成此差異（TF-IDF 優於 BM25）的原因與理論預期相反，分析主要有以下幾點：

1. **文件長度相對一致** BM25 的核心優勢之一是其精細的「文件長度正規化」，它在處理長度差異懸殊的文件時特別有效
   然而，本次作業的程式碼片段長度可能相對平均，這使得 BM25 的該項優勢無法發揮，甚至可能因不當的懲罰而降低了性能

2. **查詢詞與文件詞頻特性** BM25 的另一個優勢是詞頻飽和度機制，即一個詞在文件中出現次數的邊際效益會遞減
   但在本次實驗中發現對 TF-IDF 採用類似的次線性詞頻縮放 (1 + log(tf))策略後，性能得到了巨大提升
   這表明，雖然詞頻飽和度的思想是正確的，但 TF-IDF 配合簡單的對數縮放，其組合效果在本資料集上恰好優於 BM25 更複雜的正規化公式

3. **參數敏感度**
   BM25 的表現高度依賴 k1 和 b 兩個參數，儘管進行了調整但可能仍未找到全局最優解
   而 TF-IDF 沒有這麼敏感的超參數（特別是經過對數縮放後），使其表現更為穩健

總結來說，雖然 BM25 在理論上更先進，但在本次特定的資料集和任務上，一個經過優化（特別是詞頻處理）的 TF-IDF 模型展現了更強的實用性和性能

補充實驗細節：

優化過程的關鍵數據如下：

- **基準性能**: 在本地驗證集上，TF-IDF (0.7400) 明顯優於使用預設參數的 BM25 (0.6680)。
- **BM25 參數調優**: 嘗試了多組 k1 和 b 的組合，最佳成績僅為 0.6780 (k1=2.0, b=0.9)，依然落後於 TF-IDF。

BM25 參數調整

為了嘗試提升BM25的性能，有嘗試對 `k1` 和 `b` 參數進行了調整。以下是實驗結果：

k1	b	Recall@10
1.2	0.6	0.6580
1.2	0.75	0.6660
1.2	0.9	0.6620
1.5	0.6	0.6640
1.5	0.75	0.6680
1.5	0.9	0.6680
2.0	0.6	0.6700
2.0	0.75	0.6740
2.0	0.9	0.6780

- **N-gram 實驗**: 嘗試加入 bigrams 和 trigrams 後，TF-IDF 和 BM25 的性能均未提升，甚至略有下降，表明單詞 (unigrams) 是此數據集最有效的特徵。
- **查詢擴展**: 使用詞形還原 (Lemmatization) 進行查詢擴展，使 TF-IDF 分數從 0.7400 提升至 0.7480，證明了其有效性。

  - **Unigrams (無查詢擴展):**
    - TF-IDF Recall@10: 0.7400
    - BM25 Recall@10: 0.6680
  - **Unigrams (有查詢擴展):**
    - TF-IDF Recall@10: 0.7480
    - BM25 Recall@10: 0.6720

- **次線性 TF 縮放**: 這是最重要的優化。將 TF-IDF 的詞頻計算改為 1 + log(tf) 後，結合查詢擴展，本地分數從 0.7480 大幅提升至 **0.7860**。這個模型最終在 Kaggle 上取得了 0.73600 的分數。

模型	原始 TF Recall@10	次線性 TF Recall@10	提升
TF-IDF (基礎)	0.7400	**0.7660**	+2.6%
TF-IDF (含查詢擴充)	0.7480	**0.7860**	+3.8%

問題二：密集檢索(預訓練 vs. 微調)

在密集檢索方法中，比較直接使用預訓練模型與使用訓練資料進行微調後的性能。哪種方法表現更好？並解釋造成差異的可能原因。

使用訓練資料進行微調後的模型，其性能遠遠優於直接使用的預訓練模型。

多次的實驗清晰地證明了這一點
在實驗初期，直接使用預訓練的 CodeBERT 在本地驗證集上的 Recall@10 僅有 0.2600
然而，在更換為性能更強的 microsoft/unixcoder-base 模型，並採用了**困難負樣本挖掘 (Hard Negative Mining)策略進行微調後，模型在 Kaggle 上的分數達上升到了 0.87200**

造成這種巨大差異的原因如下：

1. **任務適應性 (Task-Specific Adaptation)** 預訓練模型（如 Unixcoder）雖然從海量程式碼中學到了通用的語法和語義結構，但它並不理解我們這個特定的檢索任務
   它不知道要如何將一句自然語言查詢，精準地映射到解決該問題的程式碼片段上
   微調的核心目的，就是讓模型去學習這個特定的映射關係

2. **優化向量空間** 在微調中使用了三元組損失(Triplet Loss)
   這個損失函數的目標非常明確：在向量空間中，將「查詢向量」與「正確的程式碼向量（正樣本）」的距離拉近，同時將其與「錯誤的程式碼向量（負樣本）」的距離推遠
   這使得最終生成的**向量空間**是為檢索此語料庫量身打造的，極大地提升了區分相似程式碼的能力

3. **高質量的訓練信號** 為了提高模型表現，不僅進行了微調，還採用了困難負樣本挖掘策略
   相比於隨機找一個負樣本，使用 TF-IDF 預先找出那些與查詢在相似度高但實際是錯誤的程式碼作為負樣本 用這種高質量的「難題」去訓練模型，強迫它去學習更深層次、更細微的語義差別，從而使其在面對模稜兩可的查詢時，具備更強的判斷力
   這也是分數能從 0.85200 進一步提升到 0.87200 的關鍵

因此，微調不僅是有效的，而且微調的「策略」也至關重要，它直接決定了模型性能的上限

補充實驗細節：

根據實驗記錄，尋找最佳密集模型的過程如下：

- **模型選擇**: 橫向比較了四個不同的預訓練模型 (unixcoder-base, graphcodebert-base, codebert-base, codet5p-220m)。實驗證明 microsoft/unixcoder-base 在本地和 Kaggle 上的表現均為最佳，其 Kaggle 分數為 0.85200，遠超其他模型。這次實驗也驗證了本地評估機制的可靠性。

為了尋找更強大的基底模型，並驗證本地評估機制，對不同的預訓練模型進行了微調，並以嘗試以**本地驗證集**和 **Kaggle 公開分數**的表現作為雙重指標進行評估

模型 (Model)	本地分數 (Local Score)	Kaggle 分數 (Kaggle Score)	排名 (Kaggle/本地)
`microsoft/unixcoder-base`	**0.88000**	**0.85200**	**1 / 1**
`microsoft/graphcodebert-base`	0.80000	0.75200	2 / 2
`microsoft/codebert-base`	0.78000	0.65200	3 / 3
`Salesforce/codet5p-220m`	0.70000	0.60800	4 / 4

- **困難負樣本挖掘**: 在確定 unixcoder-base 為最佳基模型後，引入了困難負樣本挖掘策略。這使 Kaggle 分數從 0.85200 提升至 0.87200，證明了讓模型學習區分相似但錯誤的樣本是有效的。
- **負樣本採樣策略探索**: 這是最關鍵的優化步驟。
    - **單負樣本策略**: 實驗發現，從「Top 5 困難樣本」中隨機抽樣（Kaggle: 0.92800）比從「Top 50」（Kaggle: 0.87200）或只用「Top 1」（Kaggle: 0.90800）效果更好，揭示了**難度與多樣性之間的平衡點**。
    - **多負樣本策略**: 透過數據增強的方式，為一個正樣本匹配 4 個負樣本，進一步提升了性能。其中，「分層抽樣」（保證難、中、易的困難樣本都被抽到）取得了 **0.92400** 的 Kaggle 分數，成為冠軍策略。這證明了**樣本的多樣性**對於模型泛化至關重要。

在確定了 `microsoft/unixcoder-base` 作為最佳基底模型後，本節旨在探索不同的困難負樣本採樣策略，以找到最能提升模型分辨能力的訓練方法。

單一負樣本策略 (N=1)

為每個正樣本嘗試以不同方式抽取1個負樣本組成三元祖。

策略 (Strategy)	本地分數 (Local Score)	Kaggle 分數 (Kaggle Score)
從 Top 50 隨機抽樣	**0.9600**	0.87200
僅使用最難樣本 (Top 1)	0.9400	0.90800
從 Top 5 隨機抽樣	0.9400	**0.92800**

多重負樣本策略 (N=4)

此策略透過數據增強，為同一個 `(查詢, 正樣本)` 配對多次不同的負樣本，在整個 Epoch 中強化模型對多樣化困難樣本的應對能力。

策略 (Strategy)	本地分數 (Local Score)	Kaggle 分數 (Kaggle Score)
從 Top 50 隨機抽樣	**0.9800**	0.89200
分層抽樣	**0.9800**	**0.92400**
只取最難的 Top 4	0.8600	0.84400

問題三：稀疏 vs. 密集檢索及未來改進

在文字到程式碼檢索任務中，比較稀疏檢索和密集檢索的差異與性能。除了這些方法，還有哪些方法（例如：**Retrieve-and-Re-rank**）可以進一步提升檢索性能？

性能與差異比較： 在此次作業中，**密集檢索的性能全面且顯著地超越了稀疏檢索**

- 最強的**稀疏模型**（TF-IDF 優化版）在 Kaggle 上的分數為 **0.73600**
- 最強的**密集模型**（Unixcoder + 困難負樣本 + 多重負樣本分層抽樣）在 Kaggle 上的分數達到了 **0.92400**

兩者的核心差異在於：

- **稀疏檢索 (TF-IDF)** 基於「關鍵字匹配」
  它快速、高效、可解釋性強，但無法理解語義
  如果查詢中的詞彙（如 "add"）沒有出現在目標程式碼中（用了 "sum"），它就可能失敗
- **密集檢索 (Unixcoder)** 基於「語義理解」
  通過深度學習模型將查詢和程式碼映射到同一個語義空間，即使沒有共享的關鍵字，只要語義相關，就能成功檢索
  這是其性能遠超稀疏模型的根本原因

未來可嘗試的改進方法：

1. **混合檢索 (Hybrid Retrieval)** 這是在實驗中嘗試過的方法
   理論上，它可以結合稀疏模型的「精準匹配」能力和密集模型的「語義理解」能力
   實驗中嘗試使用了更穩健的 RRF 演算法進行融合，但在本地驗證集上，混合後的結果(0.8400)反而低於單獨的密集模型(0.9600)
   經過分析得到的結論是：**當其中一個模型（密集模型）的性能過於強大時，另一個較弱模型（稀疏模型）的貢獻會成為「噪音」而非「補充」，從而導致性能下降**
2. **檢索再排序 (Retrieve-and-Re-rank)** 這是目前業界最主流、最高效的頂級性能方案，它分為兩階段：
   - **第一階段：召回 (Retrieve)** 使用一個**快速**的模型（如 TF-IDF 或本次實驗中的 Bi-Encoder）從龐大的文檔庫中，快速篩選出一個較大的候選集（例如 Top 100） 這個階段追求「快」和「全」，目標是確保正確答案大概率在這個候選集裡
   - **第二階段：精排 (Re-rank)** 使用一個**強大但緩慢**的 **Cross-Encoder** 模型，對這 100 個

候選者進行精細的二次排序

Cross-Encoder 會將 (查詢, 候選程式碼)對同時輸入模型，進行深度的注意力交互後輸出兩者是否是同一類的分數，其排序精度遠高於目前使用的 Bi-Encoder（分開編碼查詢和程式碼）

最終從這 100 個中選出 Top 10 作為最終答案

補充實驗細節：

根據實驗記錄，對進階方法的探索結論如下：

- **混合檢索 (RRF) 的失敗**：實驗中，將本地分數 ~0.74 的 TF-IDF 模型與分數高達 0.96 的密集模型進行 RRF 融合，最終得分不升反降至 0.84。這證明了，只有當多個模型**實力相近或能力互補**時，融合才有意義。當實力差距過大，弱者只會成為噪音
- **Re-rank 策略的挑戰**：
    i. **使用預訓練 Cross-Encoder**: 直接使用通用的 ms-marco-MiniLM-L-6-v2 作為精排器，本地分數從 0.9600 下降到 0.8800。分析認為這是**領域不匹配**導致的，一個通用的文本精排器無法理解程式碼檢索的細微差別。
    ii. **微調 Cross-Encoder**: 接著嘗試用作業的數據集對 Cross-Encoder 進行微調，但分數依然是 0.8800，沒有任何提升。結論是，對於一個大型預訓練模型，僅有約 1500 個訓練樣本對的**數據量遠遠不足**以讓它學會新的、有意義的領域知識。

更多細節：https://github.com/megrez33281/Generative-Information-Retrieval