What kind of pre-processing did you apply to the document data or question text? Additionally, please discuss how different preprocessing methods affected the performance of the models?

1.我進行了兩項前處理，我在以下的(1)、(2)中解釋：

(1).Remove HTML tags :

因為在Document中有含有許多對訓練沒有幫助的HTML標籤，
所以決定將它移除。
(使用bs4中的BeautifulSoup函數判斷HTML標籤，並且使用
soup.get_text( ) 函數留下純文本)

```
soup = BeautifulSoup(document_html, 'html.parser')
text = soup.get_text()
```

(2).Lowercasing 以及Non-Alphabetic Character Removal:

因為同樣的詞彙並不會因為大小寫改變意思，如：Apple、apple，
將所有字母改為小寫表示，並且，移除數字、標點符號等，留下
純文字及空格讓模型可以單就語意做判斷並且避免大小寫所造成
的差異影響到模型的訓練。

```
text = re.sub(r'[^a-zA-Z\s]', '', text).lower()
```

2.對於不同的前處理我繪製了以下表格作比較，其中第二和第三欄分別為單獨進行該
前處理後得出的socre，從該途中可以得知，在此訓練中
只用了上述的Lowercasing 以及Non-Alphabetic Character Removal前處理
即可獲得很不錯的效能。

前處理方式	public score
什麼都不做	0.51
HTML 標籤處理	0.52
Lowercasing + Non-Alphabetic Character Removal	0.94

Please provide details on how you implemented the vector model and BM25. Compare the strengths and weaknesses of the vector model and BM25.

Vector model:

1.文本預處理：使用clean_html函數，去除HTML標籤、非文本字符，並轉換大小寫

```
# 定義預處理函數
def clean_html(document_html, is_html=False):
    if is_html:
        soup = BeautifulSoup(document_html, 'html.parser')
        text = soup.get_text()
    else:
        text = document_html
    text = re.sub(r'[^a-zA-Z\s]', '', text).lower()
    return text
```

2.建立詞彙表： 從文檔和訓練查詢中收集所有詞彙，並建立詞彙表（vocabulary），每個詞彙對應一個索引值

```
# 建立詞彙表（僅根據訓練集和文檔中的詞彙）
docs_tokens = documents_data['cleaned_text'].apply(lambda x: x.split())
train_questions_tokens = train_questions['cleaned_question'].apply(lambda x: x.split())
all_tokens = [token for tokens in docs_tokens for token in tokens] + [token for tokens in train_questions_tokens for token in tokens]
vocabulary = list(set(all_tokens))
vocab_size = len(vocabulary)
```

3.文檔-詞矩陣 (DTM) 和查詢-詞矩陣 (QTM)： 將每個文檔和查詢轉換為對應的詞矩陣，其中每一行對應一篇文檔或查詢，每一列對應詞彙表中的一個詞

```
for i, tokens in enumerate(docs_tokens):
    token_indices = [word_to_index[token] for token in tokens if token in word_to_index]
    counts = np.bincount(token_indices, minlength=vocab_size)
    DTM[i] = counts
```

4.計算IDF：這段程式碼先計算每個詞出現在多少篇文檔中(DF)，再通過文檔總數除以詞出現的文檔數來計算每個詞的逆文檔頻率（IDF），其中 1+ DF是為了防止除以 0 的錯誤

```
# 計算IDF（使用訓練集和文檔）
DF = np.sum(DTM > 0, axis=0)  # 包含每個詞的文檔數
N = num_docs
IDF = np.log(N / (1 + DF))  # 防止除以0
```

5.計算TF：先對每篇文檔的詞總數進行計算，防止出現詞總數為 0 的情況。接著，計算詞在文檔中的頻率，並以此生成文檔和查詢的詞頻矩陣（TF_docs 和 TF_train_queries）

```
# 計算TF
doc_lengths = np.sum(DTM, axis=1, keepdims=True)  # 文檔中詞的總數
doc_lengths[doc_lengths == 0] = 1  # 防止除以0
TF_docs = DTM / doc_lengths

train_query_lengths = np.sum(QTM_train, axis=1, keepdims=True)
train_query_lengths[train_query_lengths == 0] = 1
TF_train_queries = QTM_train / train_query_lengths
```

6.計算TF-IDF:根據公式計算document和train data的TF-IDF

```
# 計算TF-IDF for documents and train queries
TFIDF_docs = TF_docs * IDF
TFIDF_train_queries = TF_train_queries * IDF
```

BM25：

1.定義文本清理函數並清理資料：定義函數 clean_text用來清理文本資料。如果文本包含 HTML 標籤，就用 BeautifulSoup 解析並提取純文字。然後用正則表達式移除非字母的字符，把文字轉成小寫並去除常見的停用詞，最後把這個函數應用到文檔和問題資料上，得到乾淨的文本內容，

```
stop_words = set(['the', 'is', 'in', 'at', 'of', 'and', 'a', 'to'])

# 定義文本預處理函數
def clean_text(text, is_html=False):
    if is_html:
        soup = BeautifulSoup(text, 'html.parser')
        text = soup.get_text()
    text = re.sub(r'[^a-zA-Z\s]', '', text).lower()
    return text

# 預處理文檔和查詢
documents_data['cleaned_text'] = documents_data['Document_HTML'].apply(lambda x: clean_text(x, is_html=True))
train_questions['cleaned_question'] = train_questions['Question'].apply(clean_text)
test_questions['cleaned_question'] = test_questions['Question'].apply(clean_text)
```

2. 建立詞彙表和詞與索引的映射：

將清理後的文本分詞，收集所有詞彙，然後建立一個詞彙表（vocabulary）每個詞彙都對應一個唯一的索引值，這樣就可以在後續的矩陣運算中使用詞的索引來表示詞

```
# 建立詞彙表（基於訓練集和文檔）
docs_tokens = documents_data['cleaned_text'].apply(lambda x: x.split())
train_tokens = train_questions['cleaned_question'].apply(lambda x: x.split())

# 計算所有詞的出現次數
all_tokens = [token for tokens in docs_tokens for token in tokens] + [token for tokens in train_tokens for token in tokens]
all_word_counts = pd.Series(all_tokens).value_counts()

# 建立詞彙表
vocabulary = all_word_counts.index.tolist()
vocab_size = len(vocabulary)

# 詞彙到索引的映射
word_to_index = {word: idx for idx, word in enumerate(vocabulary)}
```

3. 建立文檔-詞矩陣（DTM）：

創建一個大小為文檔數量 x 詞彙表大小的矩陣 DTM，用於記錄每個文檔中每個詞的出現次數，同時計算每個文檔的總詞數，透過這個矩陣，就能方便地知道每個詞在每個文檔中出現了多少次

```
# 構建文檔-詞矩陣 (DTM)
num_docs = len(documents_data)
DTM = np.zeros((num_docs, vocab_size))
doc_lengths = np.zeros(num_docs)

for i, tokens in enumerate(docs_tokens):
    token_indices = [word_to_index[token] for token in tokens if token in word_to_index]
    counts = np.bincount(token_indices, minlength=vocab_size)
    DTM[i] = counts
    doc_lengths[i] = np.sum(counts)
```

5. 計算平均文檔長度和逆文檔頻率（IDF）：
計算所有文檔的平均長度，接著計算每個詞的文檔頻率（DF），也就是有多少文檔包含該詞

```
# 計算平均文檔長度
avg_doc_length = np.mean(doc_lengths)

# 計算IDF（逆文檔頻率）
df = np.sum(DTM > 0, axis=0)
N = num_docs
IDF = np.log((N - df + 0.5) / (df + 0.5) + 1)
```

6. 設定 BM25 的參數並計算文檔的 BM25 權重：
設定 BM25 的兩個主要參數 k1 和 b，其中 k1 控制詞頻的飽和程度，b 控制文檔長度對詞頻的影響，接著計算每個文檔中每個詞的 BM25 權重

```
# 設定 BM25 的參數
k1 = 1.5
b = 0.75

# 計算 BM25 文檔矩陣
BM25_docs = np.zeros((num_docs, vocab_size))

for i in range(num_docs):
    for j in range(vocab_size):
        tf = DTM[i, j]
        if tf > 0:
            numerator = tf * (k1 + 1)
            denominator = tf + k1 * (1 - b + b * (doc_lengths[i] / avg_doc_length))
            BM25_docs[i, j] = IDF[j] * (numerator / denominator)
```

7. 處理測試查詢並計算 BM25 查詢向量：將測試查詢分詞，然後建立一個查詢矩陣，記錄每個查詢中每個詞的出現次數乘上之前計算的 IDF 值，得到查詢的 BM25 向量

```
# 處理測試查詢
test_tokens = test_questions['cleaned_question'].apply(lambda x: x.split())
num_test_queries = len(test_questions)
BM25_test_queries = np.zeros((num_test_queries, vocab_size))

for i, tokens in enumerate(test_tokens):
    token_indices = [word_to_index[token] for token in tokens if token in word_to_index]
    query_term_freq = np.bincount(token_indices, minlength=vocab_size)
    BM25_test_queries[i] = query_term_freq * IDF
```

8. 計算查詢與文檔之間的相似度並檢索結果：使用矩陣點積計算查詢與所有文檔之間的相似度，得到一個相似度矩陣，然後對於每個查詢根據相似度排序，選擇最相關的前三個文檔

```
# 計算查詢與文檔的 BM25 相似度
similarity_matrix = np.dot(BM25_test_queries, BM25_docs.T)

# 對每個查詢找到最相關的前三個文檔
top_k = 3
top_k_indices = np.argsort(similarity_matrix, axis=1)[:, -top_k:][:, ::-1]
predicted_docs = documents_data['Document ID'].values

# 提取預測結果
results = [predicted_docs[indices] for indices in top_k_indices]
```

strengths and weaknesses of the vector model and BM25.

	Vector model	BM25
原理	基於 TF-IDF，使用詞頻與逆文檔頻率，透過餘弦相似度計算查詢與文檔的相似性。	基於概率檢索模型，考慮詞頻、逆文檔頻率和文檔長度，調整詞頻飽和與文檔長度的影響。
優勢	**簡單易實現**：計算過程直觀，主要依賴線性代數運算 **計算效率高**：適合大型數據集的快速計算 **可擴展性強**：容易結合其他技術，如詞嵌入或主題模型	**高檢索性能**：在資訊檢索任務中表現優異，能提供更精確的相關性評估 **考慮詞頻飽和**：避免高頻詞對相似度的過度影響 **文檔長度正規化**：調整不同長度文檔的差異，減少偏倖
劣勢	**忽略詞頻飽和**：對高頻詞可能過度強調其重要性 **對文檔長度敏感**：長文檔可能因詞數多而得到較高的相似度 **無法捕捉詞序與語義**：缺乏對詞序和深層語義的理解	**計算複雜度較高**：公式較為複雜，計算量大於向量模型 **參數依賴性強**：需要調整 k1和 b 等參數，增加模型調參的複雜性 **無法捕捉詞序與語義**：同樣基於詞袋模型，忽略詞序和語義關係
適用場景	適合需要快速檢索且資源有限的應用場景 適用於教學或快速原型設計	適用於對檢索質量要求較高的應用，如搜索引擎和專業資訊檢索系統
參數依賴性	**參數少**：不需要調整太多參數，實現相對簡單	**參數多**：需要調整 k1和 b 等參數，以獲得最佳性能
對新詞處理	對於未在訓練集中出現的詞，處理能力有限	同樣對新詞（未見過的詞）處理有限，需要詞彙表的支持
實踐應用	在資訊檢索的早期階段廣泛使用 作為基礎模型，方便理解和實現	現代資訊檢索系統的主流模型之一 被廣泛認為是強大的基線模型，效果優異

What factors might account for the differences in their performance?

可能影響的factors如以下所示：

1. 詞頻處理（TF Scaling）
向量模型：直接使用詞頻，無法處理詞頻飽和現象，詞頻越高，詞的重要性越大
BM25：限制高頻詞的影響，通過 k1 參數處理詞頻飽和，使詞頻對相似度的影響更合理

2. 文檔長度的影響
向量模型：沒有明確調整文檔長度，長文檔可能因為詞數多而得到較高的相似度
BM25：使用 b 參數進行文檔長度正規化，避免長文檔在比較中佔優勢

3. 逆文檔頻率（IDF）
向量模型：IDF 計算較簡單，難以處理低頻詞與高頻詞的極端差異
BM25：IDF 計算更精細，稀有詞彙影響更大，有助於提升檢索結果的準確性

4. 參數控制
向量模型：不需要太多參數，靈活性較差
BM25：提供 k1和b 參數，允許針對不同數據集調整模型性能

5. 查詢和文檔相似度計算方式
向量模型：使用餘弦相似度，僅比較查詢和文檔的詞頻向量，忽略詞的實際匹配程度
BM25：更關注詞在查詢和文檔中的具體出現次數和位置，能更精確評估匹配度

6. 對常見詞與稀有詞的處理
向量模型：對常見詞與稀有詞區分不明顯，常見詞影響較大
BM25：限制常見詞的影響，稀有詞權重更高，檢索效果更佳

7. 計算複雜度
向量模型：計算簡單，適合大規模數據，但檢索準確性不如 BM25
BM25：計算複雜，資源需求較高，但檢索結果更精確

8. 數據集特徵的適應性
向量模型：適合文檔長度差異小、詞頻分佈均勻的數據集
BM25：適合文檔長度差異大、詞頻不均勻的數據集，檢索準確性更高