1. **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?**

Model / Preprocess Method	Vector model	BM25
Without preprocess	45.0	53.5
Removing HTML Tags	44.5	53.5
Removing Links	44.0	54.0
Removing Special Characters	49.0	58.0
Converting to Lowercase	81.5	89.0

Figure.1 Recall@3 Comparison of Vector Model and BM25 Across Preprocessing Methods

1. Removing HTML Tags: Removed HTML tags to reduce noise.
    - Example: " This is text. " → "This is text."
2. Removing Links: Removed links (e.g., starting with https://) to avoid irrelevant info.
    - Example: "Visit https://example.com for more." → "Visit for more."
3. Removing Special Characters: Removed non-alphanumeric characters, keeping only letters and numbers.
    - Example: "Hello! Version 2.0" → "Hello Version 20"
4. Converting to Lowercase: Converted text to lowercase for consistent matching.
    - Example: "Natural Language Processing" → "natural language processing"

According to fig 1, converting to lowercase had the most substantial positive impact on model performance, significantly raising Recall@3 from 49.0 to 81.5 for the vector model and from 58.0 to 89.0 for BM25. This result underscores the importance of case normalization, as it reduces mismatches caused by case differences, allowing for better alignment between query and document terms. Removing special characters also improved model performance,

boosting Recall@3 by 4-5% for both models by standardizing text and minimizing noise, which likely enhanced token matching. However, removing HTML tags and links had almost no impact, showing these elements were sparse or not relevant to model predictions in this dataset.

2. Please provide details on how you implemented the vector model and BM25.

Vector model

1. Building Vocabulary

   This method builds a vocabulary from the tokenized texts in the corpus. The vocabulary helps in mapping each unique word to an index, which is essential for constructing the TF-IDF matrices.

   - it collects all unique words from the tokenized documents.
   - it enumerates each word, assigning a unique index for use in the TF matrix.

2. Calculating Term Frequency (TF)

   To calculate the term frequency (TF) matrix, which represents the frequency of each word in each document, normalized by document length.

   - A zero matrix is created with dimensions [number of documents] x [vocabulary size], where each cell will hold the TF for a word in a document.
   - For each document, it iterates over tokens. If a token is in the vocabulary, it increments the corresponding cell in the matrix.Finally, it divides each row by the document length to obtain the normalized TF values.

3. Calculating Inverse Document Frequency (IDF)

   This method calculates the Inverse Document Frequency (IDF) values for each term in the vocabulary.

   - It first calculates the Document Frequency (DF) count for each word (i.e., the number of documents containing the word).Then, it calculates the IDF for each term using the formula:

     $$IDF(t) = log(\frac{N}{1+DF(t)})$$

4. Calculating TF-IDF Scores

The method ties everything together, building the vocabulary, calculating the TF and IDF values, and producing the final TF-IDF matrices for both the corpus and the queries.

- Call build_vocabulary on the tokenized corpus to create the vocab_index.
- Calculate the document term frequencies using compute_tf.
- Calculate the query term frequencies for the tokenized queries.
- Calculate IDF values based on the document TF matrix.
- Multiply the document and query TF matrices by the IDF values element-wise, resulting in two TF-IDF matrices: doc_tfidf_matrix for documents and query_tfidf_matrix for queries.

```
class TFIDFModel:
    def __init__(self):
        self.vocab_index = {}
        self.idf_values = None

    def build_vocabulary(self, tokenized_texts):
        vocab = set([token for tokens in tokenized_texts for token in
tokens])
        self.vocab_index = {word: i for i, word in enumerate(vocab)}

    def compute_tf(self, tokenized_texts):
        tf_matrix = np.zeros(
            (len(tokenized_texts), len(self.vocab_index)), dtype=int)
        for i, tokens in enumerate(tokenized_texts):
            for token in tokens:
                if token in self.vocab_index:
                    tf_matrix[i, self.vocab_index[token]] += 1
        doc_lengths = np.array([len(tokens) for tokens in
tokenized_texts])
        return tf_matrix / doc_lengths[:, None]

    def compute_idf(self, tf_matrix):
        N = len(tf_matrix)
        df_count = np.sum(tf_matrix > 0, axis=0)
        self.idf_values = np.log(N / (1 + df_count))

    def fit_transform(self, tokenized_corpus, tokenized_queries):
        self.build_vocabulary(tokenized_corpus)

        doc_tf_matrix = self.compute_tf(tokenized_corpus)
```

```
        query_tf_matrix = self.compute_tf(tokenized_queries)

        self.compute_idf(doc_tf_matrix)
        doc_tfidf_matrix = doc_tf_matrix * self.idf_values
        query_tfidf_matrix = query_tf_matrix * self.idf_values

        return doc_tfidf_matrix, query_tfidf_matrix
```

BM25

1. **Processing the Corpus**

   Calculating Each Word's Frequency, Document Count, and Average Document Length

   Document Count: In the compute_tf method, the code iterates through the entire corpus, calculating the number of documents (corpus_size), which represents how many documents are in the corpus.

   Word Frequency Calculation: For each document, a dictionary is created to store the frequency of each word in that document. This involves:

   - Counting the occurrences of each word in the document.
   - Storing these frequencies in the doc_freqs list, where each element corresponds to a word frequency dictionary for a document in the corpus.

   Average Document Length: The total number of words across all documents is computed, and then divided by the number of documents to obtain the average document length (avgdl).

2. **Calculating IDF Values**

   IDF (Inverse Document Frequency) measures the importance of a word within a document set. Generally, words that occur less frequently have higher IDF values, while common words have lower IDF values.

   The IDF value is computed using the following formula:

   $$IDF(t) = log(\frac{N-df+0.5}{df+0.5} + 1)$$

   where:

   - N is the total number of documents.

- df is the number of documents containing the word.

After the calculation, the IDF values for each word are stored in the `idf` dictionary for quick reference during query scoring.

4. Calculating Query Scores

Using the get_scores Method to Calculate Each Document's Relevance Score for the Query

The user calls the get_scores method with the query terms

For each query term, the relevance score for each document is computed based on the following steps:

For each document, the frequency of the query term is retrieved (stored in q_freq), resulting in an array that represents how many times the term appears in each document.

Applying the BM25 Formula: The BM25 formula is used to calculate the score, which is expressed as:

$$score(D) = \sum_{t \in Q} IDF(t) \frac{f(t,D)(k_1+1)}{f(t,D)+k_1(1-b+b\frac{|D|}{avgdl})}$$

where:

- f(t,D) is the frequency of term t in document D .
- |D| is the length (word count) of document D .

```
class BM25:
    def __init__(self, corpus, tokenizer=None, k1=1.5, b=0.75):
        self.k1 = k1
        self.b = b
        self.corpus_size = 0
        self.avgdl = 0
        self.doc_freqs = []
        self.idf = {}
        self.doc_len = []
        self.tokenizer = tokenizer
```

```
        if tokenizer:
            corpus = self._tokenize_corpus(corpus)

        nd = self.compute_df(corpus)
        self.compute_idf(nd)

    def compute_df(self, corpus):
        nd = {}
        num_doc = 0
        for document in corpus:
            self.doc_len.append(len(document))
            num_doc += len(document)

            frequencies = {}
            for word in document:
                if word not in frequencies:
                    frequencies[word] = 0
                frequencies[word] += 1
            self.doc_freqs.append(frequencies)

            for word, freq in frequencies.items():
                nd[word] = nd.get(word, 0) + 1

            self.corpus_size += 1

        self.avgdl = num_doc / self.corpus_size
        return nd

    def compute_idf(self, nd):
        for word, freq in nd.items():
            self.idf[word] = math.log(
                (self.corpus_size - freq + 0.5) / (freq + 0.5) + 1)

    def get_scores(self, query):
        score = np.zeros(self.corpus_size)
        doc_len = np.array(self.doc_len)
        for q in query:
            q_freq = np.array([(doc.get(q) or 0) for doc in
self.doc_freqs])
            score += (self.idf.get(q) or 0) * (q_freq * (self.k1 + 1)
/
                                               (q_freq + self.k1 * (1
- self.b + self.b * doc_len / self.avgdl)))
        return score
```

3. **Compare the strengths and weaknesses of the vector model and BM25. What factors might account for the differences in their performance?**

Vector Model (TF-IDF)

Strengths:

1. In settings with short, well-defined documents, TF-IDF often performs well in capturing word importance relative to document length and overall rarity.
2. TF-IDF scores are clear to understand why a term is considered important (frequent in the document, rare in the corpus).

Weaknesses:

1. TF-IDF doesn't account for the diminishing returns of term frequency—adding multiple repetitions of the same word will continue to increase its score.
2. TF-IDF has limited length normalization, so it may favor longer documents with more words, skewing results in favor of verbose content.
3. When queries contain multiple words or phrases, TF-IDF may assign higher scores to documents that match some query terms many times, rather than all query terms at least once.

BM25 Model

Strengths:

1. BM25 doesn't overly reward documents that repeat a query term excessively.
2. BM25's length normalization factor is tunable, enabling it to balance favorably between short and long documents.
3. BM25 handles both short and long queries well because it considers the relative frequency and impact of each term, maintaining balanced scores for relevant documents.
4. BM25's k1 (frequency influence) and b (length normalization) parameters allow fine-tuning for specific datasets, improving retrieval performance and relevance.

Weaknesses:

1. Due to its complexity, BM25 can be more computationally expensive than TF-IDF, especially in large-scale retrieval tasks.
2. BM25 scores are less interpretable than TF-IDF scores, making it harder to justify why certain terms or documents are ranked higher without diving into the math

behind the scoring.

Performance Differences and Contributing Factors

1. In cases with significant length variation among documents, BM25 typically outperforms TF-IDF according to figure 2. due to its dynamic length normalization. The vector model tends to overvalue longer documents, whereas BM25 compensates by tuning term weights based on document length.
2. BM25's handling of term frequency saturation means it’s better suited for documents where key terms are mentioned excessively. TF-IDF will continue to increase a document's score as a term frequency rises, while BM25 levels it off, making it more resistant to "keyword stuffing."
3. BM25 tends to perform better with both short and long queries, especially when relevance depends on meeting all query terms at least once. TF-IDF may overrate documents containing high-frequency terms but missing other essential terms, making BM25 more reliable for multi-term queries.

Model / Metric	Vector model	BM25
Recall@3	84.0	91.0

Figure 2. Comparison of Retrieval Performance Between Vector Model and BM25