**1.**

For an Information Retrieval (IR) system, the metrics of Precision and Recall are used to evaluate its performance in retrieving relevant documents from a database. Here's what each term means:

1. **Precision**: The fraction of retrieved documents that are relevant. It is calculated as:

Precision=Number of relevant documents retrieved/Total number of documents retrieved

1. **Recall**: The fraction of relevant documents that have been retrieved over the total amount of relevant documents available. It is calculated as:

Recall=Number of relevant documents retrieved/Total number of relevant documents

**a. Precision = 1**:

* This situation implies that every document retrieved by the IR system is relevant, but it does not indicate anything about whether all relevant documents have been retrieved. Achieving a precision of 1 is possible if the system is highly conservative in retrieving documents, ensuring only relevant ones are retrieved. However, this may not necessarily result in high recall, as some relevant documents might not be retrieved at all.

**b. Recall = 1**:

* This implies that all relevant documents have been retrieved. However, this metric does not control for the number of irrelevant documents that might also have been retrieved along with the relevant ones. Hence, while recall can be 1, indicating maximum coverage of relevant documents, the precision could be lower if many irrelevant documents are also retrieved.

**c. Both Precision and Recall = 1**:

* Achieving both precision and recall at 1 simultaneously means that the IR system retrieves exactly all the relevant documents and no irrelevant ones. This is the ideal scenario, indicating perfect retrieval performance. It is theoretically possible but practically very challenging, especially in large and complex datasets where distinguishing relevant from irrelevant information perfectly can be very difficult.

In conclusion:

* **Precision = 1** is possible if the system retrieves only relevant documents, potentially at the cost of not retrieving all relevant documents.
* **Recall = 1** is possible if the system retrieves all relevant documents, but may also retrieve irrelevant ones, affecting precision.
* **Both Precision and Recall = 1** is theoretically possible but practically difficult to achieve, especially in systems dealing with ambiguous, varied, or extensive sets of data.

**2.**

**Step 1: Create Vocabulary**

First, determine all unique words across the documents and queries:

* Documents:
  + "fast car win more race"
  + "sport car and fast car win"
  + "car win major car race"
  + "formula race win major car bet"
* Queries:
  + "formula"
  + "sport fast drive"

**Vocabulary (sorted alphabetically):** "and", "bet", "car", "drive", "fast", "formula", "major", "more", "race", "sport", "win"

**Step 2: Vector Representation**

Each document and query is converted into a vector based on this vocabulary. The vectors are filled with term frequencies (TF)—the number of times each word appears in the document/query.

**Document Vectors:**

1. "fast car win more race" → [0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1]
2. "sport car and fast car win" → [1, 0, 2, 0, 1, 0, 0, 0, 0, 1, 1]
3. "car win major car race" → [0, 0, 2, 0, 0, 0, 1, 0, 1, 0, 1]
4. "formula race win major car bet" → [0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1]

**Query Vectors:**

1. "formula" → [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
2. "sport fast drive" → [0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0]

**Step 3: Calculate Cosine Similarity**

**Document and Query Vectors:**

**Document Vectors**:

1. [0,0,1,0,1,0,0,1,1,0,1]
2. [1,0,2,0,1,0,0,0,0,1,1]
3. [0,0,2,0,0,0,1,0,1,0,1]
4. [0,1,1,0,0,1,1,0,1,0,1]

**Query Vectors**:

1. [0,0,0,0,0,1,0,0,0,0,0]
2. [0,0,0,1,1,0,0,0,0,1,0]

**Calculations for Each Pair:**

Query 1 with All Documents:

1. **Query 1 with Document 1**:
   * Dot product = 00
   * Norm(Query 1) = 11, Norm(Document 1) = 55​
   * Cosine Similarity = 0/(5×1)=00/(5​×1)=0
2. **Query 1 with Document 2**:
   * Dot product = 00
   * Norm(Query 1) = 11, Norm(Document 2) = 88​
   * Cosine Similarity = 0/(8×1)=00/(8​×1)=0
3. **Query 1 with Document 3**:
   * Dot product = 00
   * Norm(Query 1) = 11, Norm(Document 3) = 55​
   * Cosine Similarity = 0/(5​×1)=0
4. **Query 1 with Document 4**:
   * Dot product = 11
   * Norm(Query 1) = 11, Norm(Document 4) = 66​
   * Cosine Similarity = 1/(6​×1)≈0.408

Query 2 with All Documents:

1. **Query 2 with Document 1**:
   * Dot product = 11 (from 'fast')
   * Norm(Query 2) = 33​, Norm(Document 1) = 55​
   * Cosine Similarity = 1/(5​×3​)≈0.258
2. **Query 2 with Document 2**:
   * Dot product = 22 (from 'fast' and 'sport')
   * Norm(Query 2) = 33​, Norm(Document 2) = 88​
   * Cosine Similarity = 2/(8​×3​)≈0.408
3. **Query 2 with Document 3**:
   * Dot product = 00
   * Norm(Query 2) = 33​, Norm(Document 3) = 55​
   * Cosine Similarity = 0/(5​×3​)=0
4. **Query 2 with Document 4**:
   * Dot product = 00
   * Norm(Query 2) = 33​, Norm(Document 4) = 66​
   * Cosine Similarity = 0/(6​×3​)=0

**3.**

Storing term indexes in a sorted manner significantly enhances the efficiency of searching by allowing the use of advanced search algorithms such as binary search, which can drastically reduce the time complexity compared to a linear search in an unsorted list.

**Binary Search vs. Linear Search:**

* **Binary Search:** Operates by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. This method is efficient with a time complexity of �(log⁡�)*O*(log*n*), but it requires the list to be sorted.
* **Linear Search:** Scans one item at a time, without any assumption about the order of the elements, leading to a time complexity of �(�)*O*(*n*).

**Example:** Imagine you have an index of terms from a set of documents as follows:

* **Unsorted Index:** ["zebra", "apple", "mango", "cherry", "banana"]
* **Sorted Index:** ["apple", "banana", "cherry", "mango", "zebra"]

If you want to check if the term "cherry" is in the list:

* In the **unsorted index**, you might end up checking every term until you find "cherry" or exhaust the list, leading potentially to checking all five terms.
* In the **sorted index**, you can use binary search:
  1. Start by comparing "cherry" to "cherry" (middle of the list), finding the term in just one step.

Thus, sorting provides a dramatic improvement in search speed, especially as the size of the data grows.

**4.**

Stemming, which involves reducing words to their base or root form, is not universally beneficial in every situation due to several limitations:

**Advantages of Stemming:**

1. **Improved Search Efficiency**: Stemming can help consolidate different forms of a word (like "drive," "drives," "driving") into a single term, making it easier to match queries with documents that use different forms of the same word.
2. **Reduced Space**: By collapsing words into their stems, stemming can reduce the size of the vocabulary in a dataset, which simplifies data management and processing.

**Disadvantages of Stemming:**

1. **Loss of Meaning**: Stemming can sometimes remove important parts of a word, leading to a loss in the specificity or change in meaning, which can affect the accuracy of search and analysis.
2. **Over-Stemming and Under-Stemming Errors**:
   * **Over-Stemming**: Different words with unrelated meanings are reduced to the same stem. This can cause irrelevant information to be retrieved in a search query.
   * **Under-Stemming**: Similar words that should be reduced to the same stem are not, possibly leading to missed connections in search results.
3. **Contextual Ambiguity**: The same stem can have multiple meanings depending on the context, which can complicate interpretation and relevance assessments in text processing tasks.

**Example of a Problematic Stemmed Query**

**Document 4 from Question 2**: "formula race win major car bet"

A problematic stemmed query could be: "winning majorly". This query, when stemmed, might include words like "win" and "major", which also appear in Document 4.

**Stemming Process:**

* "winning" → "win"
* "majorly" → "major"

**Analysis:**

* **Document 4 contains**: "win", "major"
* **Stemmed Query**: "win", "major"

Here, the stemmed query matches terms in Document 4 due to the shared stems "win" and "major." However, the intent of the query "winning majorly" might be entirely different, perhaps relating to achieving a significant victory in any context, not specifically related to a "formula race" or a "car bet." This example shows how stemming, while useful for broadening search matches, can also lead to retrieving documents that are contextually irrelevant to the actual intent of the query.

This demonstrates one of the limitations of stemming—while it can help in retrieving more documents by broadening the search, it can also lead to false positives, retrieving documents that match on a superficial level but are not truly relevant to the user's needs. For more nuanced or precise search requirements, other techniques like lemmatization or semantic search might be more appropriate.

**5.**

**a. Calculate the TF-IDF scores of query with document 1 and 4 using natural (raw) term frequency count, no document frequency, and cosine similarity**

For this part, we calculate the TF-IDF scores of the second query ("sport fast drive") against Documents 1 ("fast car win more race") and 4 ("formula race win major car bet"). We are instructed to use natural term frequency counts (raw counts of terms in documents and queries) and no document frequency adjustments (i.e., we don't consider how many documents contain the terms, which simplifies to using just term frequency).

Step 1: Term Frequency (TF)

* **Document 1:** [0,0,1,0,1,0,0,1,1,0,1][0,0,1,0,1,0,0,1,1,0,1] (based on vocabulary: "and", "bet", "car", "drive", "fast", "formula", "major", "more", "race", "sport", "win")
* **Document 4:** [0,1,1,0,0,1,1,0,1,0,1][0,1,1,0,0,1,1,0,1,0,1]
* **Query:** [0,0,0,1,1,0,0,0,0,1,0][0,0,0,1,1,0,0,0,0,1,0] (for "sport fast drive")

Step 2: Calculate TF-IDF Scores

Since we are not using document frequency (DF) and assuming inverse document frequency (IDF) as 1 (since no DF is used), the TF-IDF vector will be the same as the TF vector in this specific case.

Step 3: Cosine Similarity

Cosine similarity between the TF vectors of the query and each document will give us the score. Cosine Similarity=A⋅B∥A∥∥B∥Cosine Similarity=∥**A**∥∥**B**∥**A**⋅**B**​ where A**A** and B**B** are vectors, A⋅B**A**⋅**B** is the dot product, and ∥A∥∥**A**∥ and ∥B∥∥**B**∥ are norms of the vectors.

Let's calculate this now.

**Results for Part (a):**

* **Cosine Similarity between Query ("sport fast drive") and Document 1 ("fast car win more race")**: 0.2580.258
* **Cosine Similarity between Query and Document 4 ("formula race win major car bet")**: 0.00.0

Document 1 has some similarity with the query due to the presence of the term "fast," whereas Document 4 does not share any common terms with the query relevant to "sport" or "drive," resulting in a cosine similarity of zero.

**b. Calculate the TF-IDF scores for all documents using logarithmic term frequency and inverse document frequency**