Question 1:

Step a: Preprocessing

I start with converting each letter to its lowercase form, secondly with use if translator I remove punctuations. Since some letters in the document are not recognized with the default encoder, I used utf-8 encoding system. Then I omit stop words and create the unique set of terms as asked in the question.

Step b: Create Dictionary and Postings Lists

in this step I create the dictionary. Dictionary keys are all terms in all documents and each value is also a dictionary itself that contains “freq” for number of occurrences the word in all documents and a posting list which sores the document that contains the term and also the index of happening word in that document.

Step c: Apply Front-Coding to Compress the Dictionary

At first, I defined a function for the front-coding explained in the question: I have revised the previous dictionary the way it contains dictionaries and in each one contains 4 terms that encoded corresponding to the key, so the dictionary uses much less storage with the suitable compression.

Two examples are:

'apparent': {'apparent': {'freq': 1, 'postingList': {12: [199]}},
(3, 'eal'): {'freq': 1, 'postingList': {14: [602]}},
(3, 'ealing'): {'freq': 1, 'postingList': {7: [290]}}
(3, 'eals'): {'freq': 1, 'postingList': {14: [66]}}},

'appeared': {'appeared': {'freq': 3, 'postingList': {13: [541], 16: [264], 19: [476]}},
(5, 'se'): {'freq': 1, 'postingList': {13: [355]}},
(3, 'liances'): {'freq': 1, 'postingList': {10: [134]}},
(3, 'lied'): {'freq': 1, 'postingList': {17: [488]}}}

Step d: Apply Gamma Encoding to Compress Postings Lists

I defined two functions, one for delta encoding and one for gamma encoding and, finally I used ran them on the previous dictionary and the result is the final encoded format of the dictionary and is compressed well. Implementation is straight forward. Here’s two previous examples in this step:

'apparent': {'apparent': {'freq': 1, 'postingList': [{'1110100': [199]}]}},
(3, 'eal'): {'freq': 1, 'postingList': [{'1110110': [602]}]},
(3, 'ealing'): {'freq': 1, 'postingList': [{'11011': [290]}]},
(3, 'eals'): {'freq': 1, 'postingList': [{'1110110': [66]}]}},

'appeared': {'appeared': {'freq': 3, 'postingList': [{'1110101': [541]}, {'101': [264]}, {'101': [476]}]},
(5, 'se'): {'freq': 1, 'postingList': [{'1110101': [355]}]},
(3, 'liances'): {'freq': 1, 'postingList': [{'1110010': [134]}]},
(3, 'lied'): {'freq': 1, 'postingList': [{'111100001': [488]}]}}

Step e: Answering Queries Based on Compressed Dictionary and Postings Lists

In this step I defined a function that takes a term and find its posting list in the encoded dictionary. I used this function to get posting list for queries (I preprocessed the queries at first). Then based on AND, OR and NOT I handled different queries and printed the result for each query. Here's the result:

safety AND events : **[1, 5, 12, 13]**

governments OR benefits : **[7, 9]**

total NOT weight : **[]**

who AND cleanup : **[10]**

items OR cleanup : **[3, 10]**

financial AND rentals : **[9]**

residents AND regulations : **[7, 9]**

challenges OR costs : **[4, 6, 12, 14, 16, 17, 19]**

housing AND health : **[12]**

resources NOT wellness : **[4, 8, 9, 13, 14, 17]**

community AND challenges : **[4, 6, 12, 14, 16, 19]**

consequences OR driver : **[12, 13, 20]**

Charlie AND lottery : **[]**

impact NOT habits : **[4, 6, 7, 8, 17, 20]**

strategies OR safety : **[1, 5, 7, 12, 13, 16, 20]**

authorities AND festivals : **[]**

number OR cleanup : **[3, 10]**

who AND initiative : **[]**

items OR cleanup : **[3, 10]**

effects AND rentals : **[]**

members AND regulations : **[]**

housing OR well-being : **[7, 12, 17]**

services AND communities : **[]**

Question 2:

In this question we want to implement the Vector-Space Model.

Step a: Use Preprocessed Dataset from Previous Part

Step b: Create TF-IDF Matrix

We know we can calculate the tf-idf weight with the formula below:

$$w_{t,d} = (1 + \log tf_{t,d}) \cdot \log \frac{N}{df_t}$$

so, I used this formula to calculate the tf_idf matrix and store it in a dictionary in which the keys are all the terms and the values are a list of size 20 in which rach of elements are this list, represents tf_idf weight corresponding to a document.

Step c: Implement Vector-Space Model

It's a straightforward step in which I appended a 1 if the term is present in the corresponding document, and a 0 otherwise.

Step d: Retrieve Queries with and without Weight Normalization

Both methods are almost similar to each other. But, in weight normalization method we only divide d and q to d/||d|| and q/||q||.
finally, we print 4 nearest documents to the query. Here's some result in the normalized form:

Who organized the community cleanup event?
[10, 3, 9, 19]

What items were found during the cleanup?
[10, 3, 19, 7]

What are the financial impacts of short-term rentals?
[9, 7, 2, 18]

How do residents feel about rental regulations?
[7, 16, 9, 2]

What challenges arise from rising housing costs?
[17, 12, 14, 16]

Step e: Visualize Retrieved Documents based on Cosine Similarities

I plotted the cosine similarity of the 4 closest document to each query in a separated plot and results are available in the .ipynb file. Despite the huge difference between the query (lied on the x-axis) and the results, these documents are highly related to the query.

The angle between each vector and the x-axis is equal to cosine inverse of the similarity.

Question 2.1: How does normalization impact the retrieval results? Discuss any differences observed in the top-ranked documents between the normalized and nonnormalized methods.

Answer: in method without the normalization, longer documents like document no.14 is in more results and the opposite, shorter documents are not in most of the results. So normalizing causes to ignore the length of documents and return the result based on the relevancy and similarity not based on their length.

Question 2.2: Based on the visualization, what insights can you draw about the distribution of documents in vector space?

Answer: one thing that can be concluded from the visualization is that documents with similar themes tend to be closer to each other in vector space, indicating that their content is similar. Similarly, if we plot all documents, we can identify outliers and this can be used for anomaly detection and clustering. Finally By visualizing query vectors, we can assess how well the documents in our collection match a given query and whether they are appropriately represented in the vector space.

Question 2.3: Suggest one or two ways to further enhance the retrieval model based on your findings

Answer: if we used some methods instead of bag-of-words like word2vec or something like this, working and implementing would be more meaningful and easier. Another thing is that we can use dimensionality reduction methods like PCA for dimension reduction, this might be very helpful for bigger documents and higher amount of documents.

Question 3: Term-At-A-Time Processing

Step a: Use the TF-IDF Matrix

Step b: Implement Term-at-a-Time Processing

I defined a function in which creates a accumulator and calculate the relevance scores for each document.

Step c: Implement Term-at-a-Time Processing

In this step we use the function above and we create a min heap for sorting the result. Finally, we print 4 most similar document IDs. Here's the result:

What safety measures should be implemented to prevent injuries during community sports events?
[5, 13, 20, 7]
How can local governments balance the economic benefits of community events with the potential disruptions they may cause to residents?
[7, 4, 9, 8]
What was the total weight of debris removed?
[10, 16, 3, 6]
Who organized the community cleanup event?
[10, 19, 3, 8]
What items were found during the cleanup?
[10, 19, 3, 6]
What are the financial impacts of short-term rentals?
[7, 9, 6, 14]
How do residents feel about rental regulations?
[7, 9, 16, 17]
What challenges arise from rising housing costs?
[17, 12, 16, 14]
How does housing instability affect mental health in communities?
[12, 19, 7, 14]
What resources are available for supporting mental health in neighborhoods?
[12, 19, 8, 4]
How can community connections mitigate challenges faced by residents?
[6, 4, 19, 8]
What consequences did the reckless driver face after the chase?
[20, 12, 13, 4]
How did Charlie Jenkins change after winning the lottery?
[6, 13, 4, 10]
What impact did the book fair have on reading habits?
[8, 13, 16, 10]
What strategies can be employed to enhance safety at community sports activities?
[4, 14, 16, 13]
How can local authorities manage the trade-offs of community festivals for residents?
[8, 4, 19, 14]
What was the total number of volunteers who participated in the cleanup event?
[10, 3, 6, 8]

Question 3.1: Why is pivot-normalization used in the context of document retrieval?

Answer: Pivot normalization is used in document retrieval to adjust term frequencies relative to a reference value (like average document length) to correct biases. It ensures that longer documents or common terms don't dominate retrieval results, leading to fairer and more accurate document rankings.

Question 3.2: What steps do you take to choose an appropriate pivot value, and how does this choice affect the normalization process?

Answer: To choose an appropriate pivot value, you can use the average document length or median term frequency as a reference. The pivot value should reflect the central tendency of the dataset. The choice affects normalization by determining how term frequencies are scaled, influencing how documents are compared and ranked for relevance.

Question 3.3: How Does term-at-a-time-processing differ from document-at-a-time processing, and what are the advantages and disadvantages of using term-at-a-time processing in information retrieval?

Answer: in term-at-a-time-processing we process each query term one by one and we calculate the relevancy of it with each documents, but, in document-at-a-time-processing we use relevance for each document using all query terms before moving to the next document.
obviously, term-at-a-time-processing is much slower and can take some time for longer queries. But since it only process one word at a time it uses less storage and also can be parallelized easily.

Question 3.4: Discuss the role of a min-heap in the ranking process during document retrieval. How does using a min-heap enhance the efficiency of retrieving the top k documents?

Answer: creating a min heap is in order of O(nlogk) while reading off one documents is in order O(1), so creating a min heap will help us to get k top documents because it creates the min heap once in O(nlogk) and it can get top k documents in O(k) which is really efficient.