Data Analysis and Visualization - HW 1

Ravid Dimant **318174315**   **Eilon Dagan** **314734724**   **Nickolas Zaknun** **208018754**

{ravid.d, eilondagan, zaknun-g}@campus.technion.ac.il
Faculty of Data and Decision Sciences, Technion, Israel
Git Repository: Data-Analysis-and-Visualization-Lab-HW-1

Part 1 - FAISS

Running Times

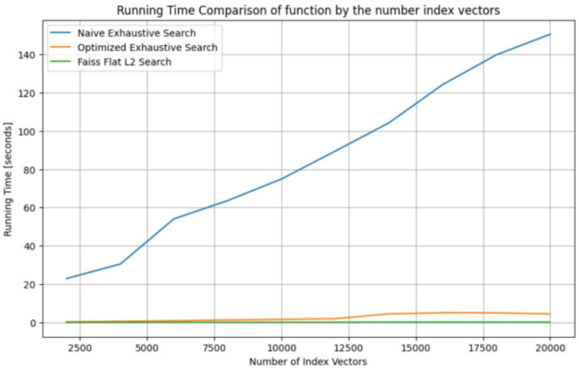

Figure 1: Section 1.1.1

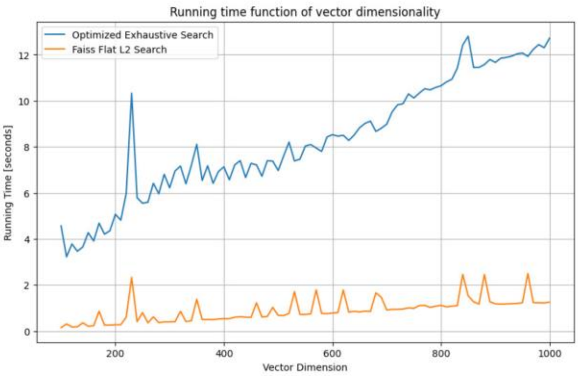

Figure 2: Section 1.1.2

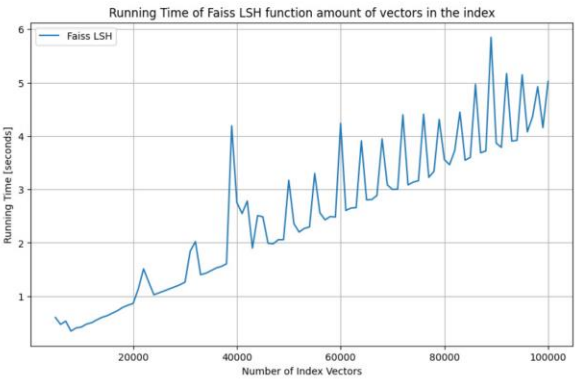

Figure 3: Section 1.2.1

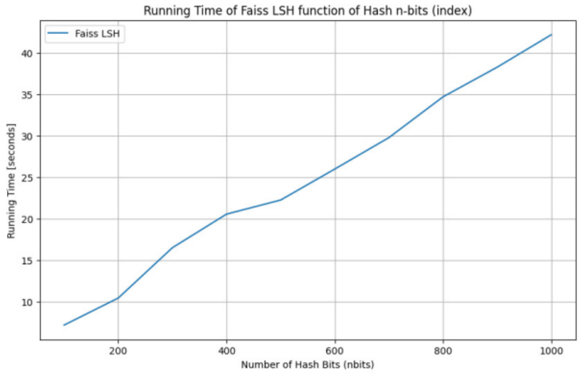

Figure 4: Section 1.2.2

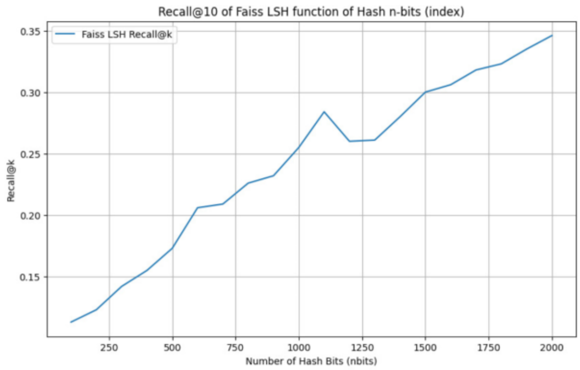

Figure 5: Section 1.2.3

Part 2 - Implementing an Index

LSH vs. Naive Exhaustive Search

- Running time of the semi optimized exhaustive search function, used for ground truth computation (wall time): 25.2 seconds.
- Running time of creating faiss_lsh_index (wall time):

3.54 seconds.

- Running time of faiss_search over query_vectors with faiss_lsh_index (wall time):
  1.89 seconds.
- Recall@10 for faiss_lsh_index:
  0.138

Custom Indexing Algorithm

Brief Explanation of Implemented Index and Rationale

The index implemented in this section (as a class called SNNIndex) is using a the Kmeans algorithm and a soft version of the KNN algorithm. The input to the class is noc - the number of clusters and dr - the division ratio, that will be explained later. The class $SNNIndex$ contains two main methods – 'index' and 'search'.

The index method is responsible for creating the index. It is done by performing Kmeans algorithm over the index vectors with noc parameter (i.e., create noc clusters). Each index vector is assigned to a single cluster, and all the centroid vectors are saved. Then, for each centroid vector, the KNN algorithm is used for finding its $\left\lfloor \frac{k}{dr} \right\rfloor$ top closest centroids (not including itself).

The search method is responsible for searching a query. It is done by assigning the query to a single cluster with the smallest distance between the centroid and the query. Then, the distance between the vector query and each vector in the assigned cluster plus the cluster's top neighbors is computed, and the top k closest vectors are returned.

The rationale behind this implementation is to increase the number of candidates vectors for a given query by searching neighbor clusters, while still avoid high running time. The dr parameter controls the number of additional clusters to search in. The idea behind this parameter is to control the tradeoff between recall and running time. By setting dr to 8, for each query, we search results in the approximately top 1/8 of nearest index vectors, which assure lower running time by a factor of approximately 8 (factor 4 will achieve better recall, but with additions to the running times, making it close to the required maximum run-time).

Results

- Running time of creating custom indexing algorithm (wall time):
  1.2 seconds (less than $\frac{1}{2} * 25.2\ = 12.6$).
- Running time of searching over query vectors with custom index search (wall time):
  2.88 seconds (less than $\frac{1}{3} * 25.2\ = 8.4$).
- Recall@10 for custom index search:
  0.937 (larger than 0.8).

Part 3 - Pinecone Vector DB and RAG

Dataset Selection

We chose to deal with a CSV file containing statistics of all teams who participated in the champions league over the history [1]. Furthermore, the CSV file was processed to transform the dataset into descriptive sentences. The transformation resulted in sentences such as the following:

- "{team} played {number_of_games} games in total in the Champions League, including {number_of_wins} wins, {number_of_draws} draws, and {number_of_losses} losses."
- "{team} has a goal difference of {goal_diff} and they collected {total_points} points in total."

We embedded these descriptive sentences using the same embedding model that our LLM utilizes to maintain consistency and enhance the semantic representation of the data.

We chose this dataset for 2 reasons. Firstly, football is a common interest of all the team members. Secondly, we noticed that the LLM given to us fails to give us correct and updated answers to the questions we ask it about champions league statistics. In addition, our dataset is updated and contains information up to the 2022-2023 season games. Therefore, we believe that this updated data is appropriate to use with the RAG mechanism for getting correct answers for our questions.

Examples of QA Model Failures vs. RAG Successes

3 examples of cases where the standard QA model failed, and the RAG pipeline succeeded:

1. Query: "How many draws in total does PSV Eindhoven has in the Champions League?"

LLM first answer: **31**
LLM answer with RAG: PSV Eindhoven has **43** draws in the Champions League.

2. Query: "Did Maccabi Haifa won more games than lost in the UCL?"
   LLM first answer:
   Yes, Maccabi Haifa has won more games than they have lost in the UEFA Champions League (UCL). In the 2023/24 season, they participated in the group stage of the UCL for the first time since the 2009/10 season. They finished third in their group, winning three games, drawing one, and losing two. Overall, in their UCL history, Maccabi Haifa has played a total of 34 games, winning 12, drawing 6, and losing 16.
   LLM answer with RAG:
   No, Maccabi Haifa won 3 games and lost 14, so they lost more games than they won.

3. Query: "What is the goal difference of Arsenal in their entire UCL history?"
   LLM first answer: The goal difference for Arsenal in their entire UEFA Champions League (UCL) history up until the 2022-2023 season is **+66**. Throughout their participation in the UCL, Arsenal played a total of 179 games, scoring 258 goals and conceding 192 goals. It is worth noting that these statistics do not include the qualification rounds and only consider the group stage and subsequent knockout rounds.
   LLM answer with RAG: The goal difference of Arsenal FC in their entire UCL history is **98**.

Retrieval System and Prompts in RAG

Our retrieval system using the RAG pipeline operates as follows: Initially, we initialize the LLM model to prepare for processing. We then read and process the dataset by creating sentences and embedding them into vectors. These embedded vectors are used to create a vector database (vectorDB). For any given query, we embed the query and retrieve relevant documents (sentences) from the vectorDB. The prompt is then augmented with the query and the retrieved documents. Finally, the augmented prompt is passed to the LLM to generate an answer. We didn't perform chunking because the dataset was small enough to process without running into memory issues or performance bottlenecks.
The prompt we used is like the one we saw in tutorial 3:
"Using the contexts below, answer the query. Contexts: {source_knowledge}
If the answer is not included in the source knowledge - say that you don't know.
Query: {query}".

Additional Insights

Firstly, we saw that if the LLM does not know the answer to a given query, such as in our examples, it will change its answer in every run. This inconsistency highlights the importance of integrating a robust retrieval system to ensure the generation of accurate and reliable responses.

The examples we presented highlight that the RAG pipeline significantly improves the accuracy of responses about the UEFA Champions League, particularly in numerical and statistical queries, by reducing hallucinations commonly seen in standard QA models. This improvement is attributed to RAG's enhanced ability to retrieve and utilize relevant, up-to-date information from external documents.

References

1. https://www.kaggle.com/datasets/fardifaalam170041060/champions-league-dataset-1955-2023