# Comparative Analysis of SCANN and FAISS for Efficient Similarity Search in Gene Embeddings

Gavin Hearne, Kieran Lynch, Cimi Patani, Harrison Muller, Saleh Refahi

## Abstract

Due to insufficient metadata and outdated heuristics, the rapid expansion of DNA sequencing data has outpaced the capabilities of traditional methods. To address this, efficient computational tools capable of handling massive datasets are essential. Similarity search algorithms play a crucial role in this endeavor, aiding in identifying similarities across DNA sequences and protein functionalities. While widely used tools like BLAST have been fundamental, they may encounter challenges such as prolonged alignment times and difficulty detecting novel sequences.

To improve efficiency, we evaluated the FAISS and ScaNN similarity search packages. FAISS, developed by Facebook AI, constructs an index from input vectors to facilitate fast similarity searches, while ScaNN offers speed enhancements and a unique loss function. Our objective was to determine which algorithm better performs similarity searches and enables the detection of in and out-of-domain sequences and novelty.

Using representation learning to express genetic sequences, we employed FAISS and ScaNN to generate distance and index matrices. FAISS and ScaNN proved ineffective at detecting novelties in large datasets despite our efforts. However, comparative analysis revealed that FAISS slightly outperforms ScaNN in accuracy, sensitivity, and specificity. FAISS also excels in indexing time but lags behind ScaNN in search time efficiency. Additionally, ScaNN demonstrates superior CPU and memory usage efficiency.

**Github Page: https://github.com/hrm53/ECES-450-Project-4**

## Background

Metagenomic data present an annotation challenge, with a notable portion of sequences likely remaining unannotated due to their novelty or significant divergence from known Olson et al. (2024). In addressing the annotation challenge posed by metagenomic data, the application of cutting-edge techniques such as deep learning, and specifically representation learning for biological sequences, neural networks compute low-dimensional vector representations for sequences, enabling the clustering of similar sequences and separating dissimilar ones in space. This approach involves computing embeddings for each sequence, searching for near neighbors in target embeddings datasets, and extracting relevant sequences Schütze et al. (2022). The literature extensively covers tools like FAISS Johnson et al. (2019)and SCANN Guo et al. (2020) in the context of similarity search, particularly emphasizing max inner product and nearest neighbor search Olson et al. (2024); Schütze et al. (2022). In our case, as we explore query embeddings, both FAISS and SCANN are considered potential tools for identifying similar embeddings within our large dataset.

While not precisely tailored to bioinformatics applications, FAISS provides several features beneficial for distance calculations and clustering for large datasets in different domains. FAISS is a library built to implement approximate nearest neighbor search (ANNS) algorithms on vector embeddings in a way adaptable to multiple use cases. To do this, FAISS employs indexing methods that perform preprocessing, compression, and non-exhaustive search on database vectors. A query vector is submitted to perform the nearest neighbor search, and the closest database vector is returned in terms of the Euclidean distance or highest dot product Douze et al. (2024). FAISS is unique compared to its single-index counterparts in similarity searching because of the number of features it provides to speed up the processing of big data sets and reduce memory overhead. As can be seen in Figure 1, FAISS implements several nearest-neighbor search algorithms, including vector quantizers (k-means), scalar quantizers (locality-sensitive hashing), etc. Methods such as locality-sensitive hashing have successfully addressed large datasets but may suffer from massive

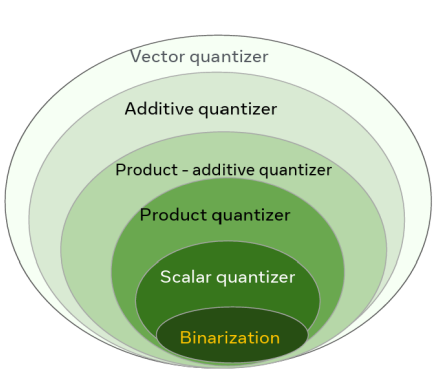


Figure 1: FAISS Quantizer Hierarchy Douze et al. (2024)

memory overhead under specific applications Johnson et al. (2019). To better execute these methods, FAISS includes preprocessing features that can better take advantage of the data during the quantization, allowing for an efficient implementation of numerous search algorithms Douze et al. (2024). Of note is that FAISS includes product quantization (PQ) algorithms, enabling dramatic data compression and speeding up ANNS by splitting vectors into sub-vectors and storing their IDs in centroids. This massively reduces the size of the vectors in memory while also allowing for the implementation of clustering Jégou et al. (2011). Another significant advantage of FAISS is that it can be utilized on both the CPU and GPU depending on user applications, with its limited memory usage allowing for quick access to the cache and more compatibility with multi-threaded applications Johnson et al. (2019). Due to these features, FAISS has become a hotbed of research and is used in numerous studies, including matching word embeddings to improve translation and finding similarities amongst millions of images without annotations Johnson et al. (2019). Furthermore, a practical application of FAISS in bioinformatics through the ClusTCR tool underscores its versatility and efficiency. ClusTCR's use of FAISS for clustering large sets of CDR3 sequences demonstrates a 50-fold increase in performance over comparative algorithms, handling up to 50,000 sequences with reduced pairwise comparisons. This showcases FAISS's ability to improve runtime and scalability in specialized clustering tasks significantly, highlighting its potential for broad application in data-intensive fields Valkiers et al. (2021). With genome sequences becoming larger and larger and metadata becoming increasingly challenging to generate, the speed provided by FAISS has made it a prime candidate for homology searches and genomic sequencing.

Google's ScaNN has proven to be a similarly powerful technique for searching vector embeddings, outperforming other techniques in queries/second and recall on several benchmarks Guo et al. (2020); Aumüller and Ceccarello (Aumüller and Ceccarello). Although, in some ways, it operates under techniques comparable with FAISS - both implement clustering-based approaches as their primary data structures - in testing, it performs quite differently. This separation is the result of several novel techniques that ScaNN employs in combination: PQ is implemented utilizing anisotropic loss and score-aware loss functions to maximize the accuracy of the quantizations, while Single Instruction Multiple Data (SIMD) in-register lookup tables and vector quantization-based trees increase run speed as described by André et al. (2019). First, the score aware loss function works by weighing the quantization based on the inner product between the query $q$ and datapoints $x$ Guo et al. (2020). This leads to the second novelty that ScaNN employs: the anisotropic loss function. Quantization error can be decomposed into orthogonal and parallel components, of which the parallel component is more heavily weighted. While PQ Jégou et al. (2011) alone is a common technique for vector quantization due to its scalability, adding these two techniques can sometimes result in significantly improved results Guo et al. (2020). Due to its impressive performance in large datasets (it can scale into the billions), ScaNN has been used to improve capabilities in the fields of text retrieval Xiong et al. (2020) and of particular interest to us, language modeling Borgeaud et al. (2022). Furthermore, ScaNN enables efficient entity linking for extensive datasets through its vector quantization and partitioning techniques, optimizing speed and accuracy. Its scalable approach is proven to handle 700 million mentions FitzGerald et al. (2022). This efficiency is critical for bioinformatics, facilitating rapid, precise analysis of large genetic datasets. This capability mirrors the need for scalable, accurate tools, particularly in gene

annotation, which parallels matching biological sequences to database entries. Both fields benefit from ScaNN's capability to manage large datasets, making it a vital tool for accurate data linkage and annotation in extensive databases.

One of ScaNN's many features is its use of asymmetric distance calculation between query and database. This technique provides significant speedup as the database size approaches tens of thousands of records (Jégou et al. (2011)). Furthermore, rather than quantizing both the query vector and the database vector, ScaNN only quantizes the database vector. This yields a better estimate of the distance between the query and database vectors. It has been noted that this asymmetric methodology has enjoyed a higher precision than other symmetric distances such as Hamming Gordo et al. (2014); Jégou et al. (2011). ScaNN utilizes this distance to supplement its scoring criteria at larger database sizes. While not applicable to us, it is worthwhile to note that ScaNN would have the facilities to support an effective and accurate search if our database grew.

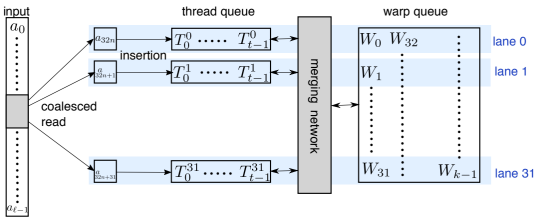


Figure 2: Overview of WarpSelect Douze et al. (2024)

Several challenges exist in using FAISS and ScaNN to address massive datasets. One such challenge is the implementation of FAISS on the GPU to increase its speed and optimize memory usage. Researchers point out that implementing FAISS on the GPU poses several challenges due to the number of irregularities and memory dependencies that result from GPU architecture. As a result, studies are also being conducted on constructing additional methods to effectively implement FAISS on the GPU, such as the WarpSelect algorithm, shown in Figure 2. Warpselect helps implement FAISS on the GPU by storing state entirely in registers, allowing for a single pass over the data to avoid challenges such as thread synchronization Johnson et al. (2019).

Table 1: Faiss and ScaNN comparison

| | **FAISS** | **ScaNN** |
|---|---|---|
| Developer | Facebook | Google |
| Ease of Use | FAISS offers numerous options and features that give the user much freedom in its operation. Because it was released earlier, its performance has been better studied, and there are more third-party implementations | ScaNN can be simpler to implement due to fewer hyperparameters and TensorFlow integration. So, for any operations that utilize TensorFlow or do not require the supplemental features of FAISS, ScaNN can be much simpler to implement |
| Features | Product Quantization; CPU and GPU implementations; multiple indexing methods including Hierarchical Navigable Small World graphs (HNSW) and Inverted file with approximate distance computation (IVFADC); support for exact search methods | Product Quantization; CPU and GPU implementations |
| Distance metrics | L2, inner product | brute-force, asymmetric distance |

# Materials and Methods

In this project, we attempt to apply FAISS and ScaNN to similarity searches for an embedding of genome sequences from a wide variety of sequences. As technology for retrieving similar sequences improves, the size of sequence datasets has begun to outpace traditional similarity search algorithms. To address these growing pains, our group applied FAISS and ScaNN, which utilize index-based compression and search methods to vastly improve memory and processing power. By using and comparing these packages, we hope to provide an alternative to traditional methods for analyzing massive datasets to improve the speed and accuracy of sequence searches. Figure 3 illustrates the pipeline of our studies, starting from inferring the embedding vector of each sequence and utilizing both FAISS and ScaNN to index and search query sequences against the training set. Following the provided pipeline, we want to determine if a difference exists between the distances

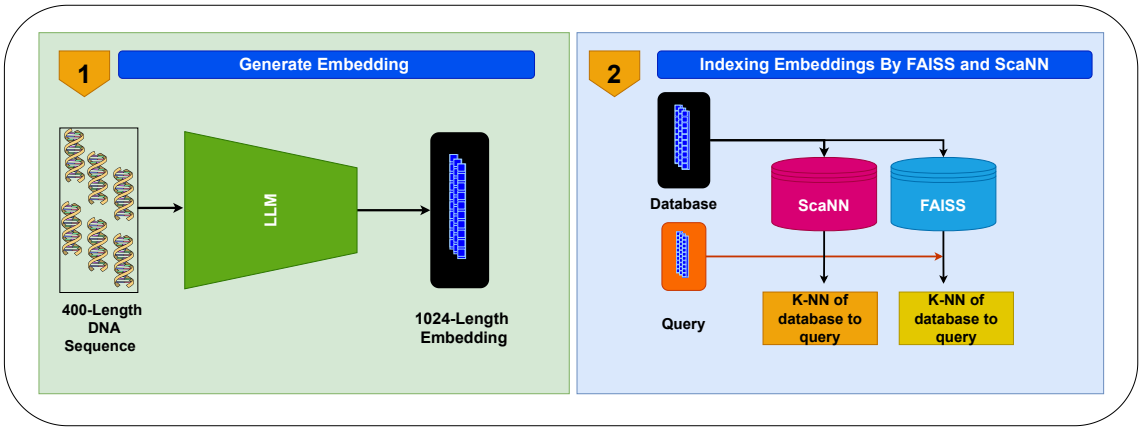


Figure 3: A snapshot of our pipeline

of 7K sequences in an in-domain test set and 7K sequences in an out-of-domain test set, known as the "taxa out" set. This set includes sequences not belonging to species in the training data. The distance scores generated from FAISS and ScaNN will be the basis for developing a heuristic for detecting novelty on the phylum level. In conjunction with the novelty determination, we will cluster similar sequences based on the distances found during indexing.

To accomplish these tasks using FAISS, several unknowns must be addressed to achieve the best implementation. Firstly, the correct indexing methods and parameters must be determined to produce a high level of accuracy while also considering an acceptable level of performance. FAISS offers many different indexing methods, each with its application and caveats based on the size of the database under analysis and the available hardware. Determining the best method to use requires an analysis of various index methods, including brute force flat methods, local search hashing, product quantization, etc. For each indexing method potentially applicable to the dataset, metrics such as accuracy, speed, homogeneity, and so on must be determined. Determining the best indexing method allows for the most efficient implementation and must be determined before generating any indexes. A sweep of Flat, IVF, and PQ indexes is performed to address the best indexing method and search parameters. The Flat index is computationally the simplest provided by FAISS, utilizing brute force methods to calculate distances and storing vectors into a flat array. IVF Indexes were also tried, which cluster the database vectors into a set of user-defined cells using a user-specified coarse quantizer and perform the query process using a provided number of probes. This is optimal for larger datasets, as it reduces the memory used. The final type of index of interest that the team tested was product quantization (PQ), which separates the input vector into $M$ sub-vectors and works on them separately. This allows for a more significant reduction in memory usage while also sacrificing accuracy. These indexes will be tested using a for loop that calls the built-in indexfactory() function. This function generates a custom index from a user-provided string, which can include a preprocessing method, quantizer, and encoder.

Along with these indexes, multiple different preprocessing options and quantizers will be used to gauge their effect on the accuracy and performance of IVF and PQ indexing methods. The preprocessing methods include PCA, which reduces the dimensions of the database matrix to improve processing speed, and OPQ, which rotates the input vectors to help increase the accuracy of PQ indexing. The number of probes hyperparameter, probes, will also be swept in a separate

loop from 100 to 1000 to gauge its effect on search speed. From the results of these sweeps, the team will evaluate accuracy, speed, and computer performance metrics (CPU and memory) to determine the most efficient and effective search parameters and index methods.

ScaNN is, on a broad level, a tool similar to FaiSS. As such, it faces many of the same challenges of selecting indexing methods and parameter tuning, as well as the metrics needed to quantify the algorithm's performance. These two tools differ, however, in their specific parameters. Of particular note is the trade-off between ScaNN's anisotropic loss function and its brute-force scoring method, which needs some trade-offs to determine which performs best for the provided embeddings.

Using the results from both FAISS and ScaNN, the distances generated can be used to determine differences between the in-domain and out-domain sequences, providing the information needed to determine accuracy. Clustering can also be performed with each package to further address the similarity between sequences. By selecting these values, our group seeks to provide information on the best package to analyze large-scale metagenomic databases and develop methods for studying them for novelty.

# Results

## Parameter Tuning of Faiss and ScaNN

Our team has generated various results for indexing the LLM embeddings. For FAISS, multiple indexes and parameters were tested to gauge their speed and memory usage. First, numerous indexing methods were tested using the **indexfactory()** function. As seen in Figure 4, the indexes tested included the flat, IVF, and product quantization with differing pre-processors and values for the number of query nprobes.

The results of running each of these indexing methods can be seen in Figures 4, 5, and 6. From Figure 4, we can see that the Flat indexes are better in accuracy, with the highest accuracy of around 0.36 belonging to a Flat index with a 64-bit PCA pre-processor. Meanwhile, IVF and PQ methods generally perform worse in accuracy than the Flat index method. However, This is expected as these methods' compression and subvector computations can lead to more inaccurate results than brute-force similarity searches.

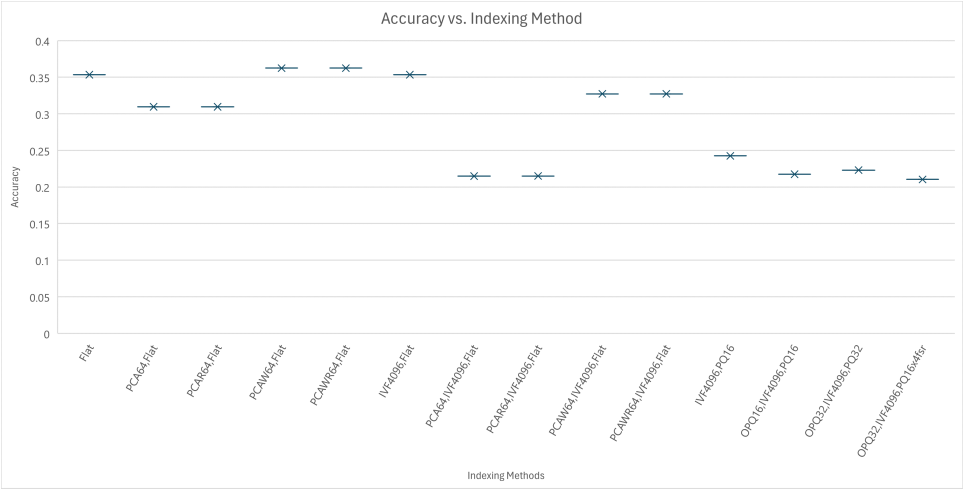


Figure 4: FAISS Index Accuracy vs. Indexing Method

Figures 5 and 6 also convey essential information about the speed of each indexing method. Based on Figure 5, Flat operates at a deficient speed during indexing, while IVF, especially PQ, spends much more time in this stage. This is expected, as Flat stores each index into a fixed-size array, requiring very little extra computation. IVF and PQ, however, partition the feature space to create smaller subspaces and subvectors that can be queried faster. As expected, Figure 6 shows PQ performs with the lowest CPU time, followed by Flat indexes and IVF. PQ's increased speed during the query stage results from the subvector querying and data compression, which allows for a much faster search that is both computationally more efficient and more sensitive to cache

locality. Flat requires more memory operations and does not efficiently manipulate the data set. However, it is essential to note that the Flat indexes with PCA pre-processors perform relatively well compared to PQ, with values ranging from 7 to 9 seconds for differing values of nprobe, while PQ averages at around 1 second. This is due to the compression of the data set that PCA provides by reducing the matrix dimensions of the dataset, allowing for quicker computations. This conveys that utilizing a PCA pre-processor with a Flat index could provide the balance between accuracy and speed required for this experiment. Additionally, the use of nprobe did not appear to significantly affect the timing of each index method during the query process. Ergo, this parameter does not need to be utilized during the search process.

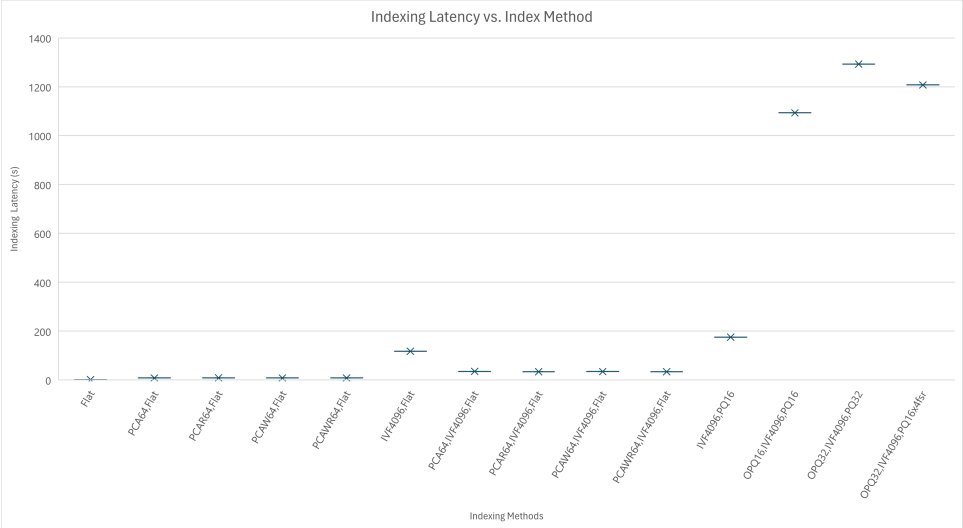


Figure 5: FAISS Index Indexing Time vs. Indexing Method

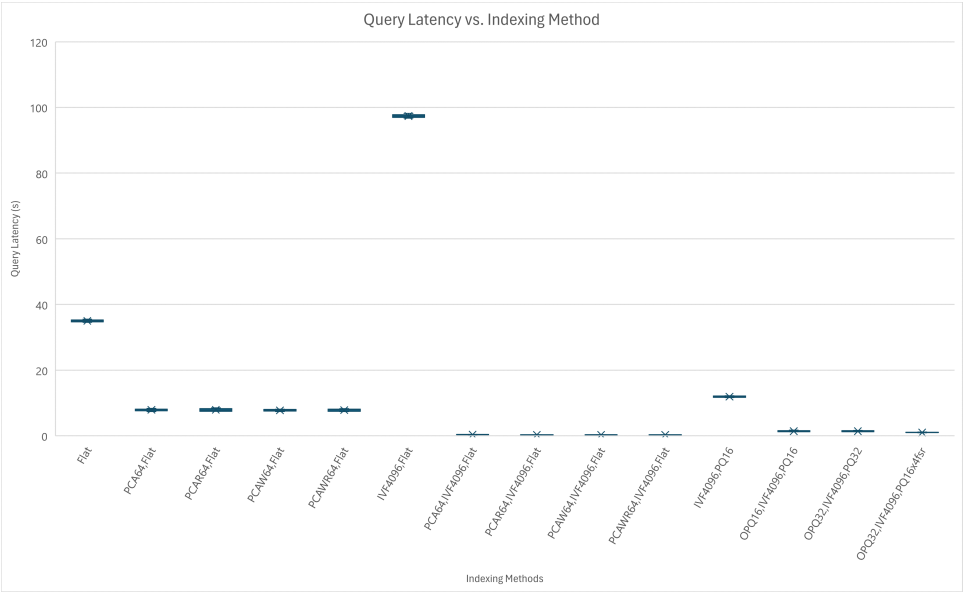


Figure 6: FAISS Index Query Time vs. Indexing Method

These results give a good idea of which methods are more desirable for analyzing the dataset and provide a good comparison method with ScaNN. Of note are additional metrics such as homogeneity and completeness, which remained constant throughout this process since they were generated with FAISS's separate built-in clustering objects Kmeans(). Based on the parameter tuning results, our group determined that utilizing a Flat index with a PCA pre-processor provides the best results in terms of accuracy and speed. While the memory overhead is more significant for Flat than other methods, the team deduced that the difference was insufficient to prevent the Flat index's usage.

Although ScaNN lacks equivalent tools for quickly testing multiple indexing methods, the comparatively low number of features is a benefit here as it can be tuned by hand relatively quickly. ScaNN performs vector searches in three phases. First is partitioning, an optional stage used to partition the data, ideally to reduce the number of items that must be scored to find a top result. This is done to reduce computational complexity and decrease runtimes when querying the index, as to perform a vector search, one only needs to calculate the distances between the query and all vectors within each partition rather than the entire dataset. Next, ScaNN computes the distances between the query and dataset vectors. This can be done in two ways: brute force calculation (with a basic quantizer included to reduce latency in scenarios where memory bandwidth is limited) and the novel usage asymmetric hashing (AH), which employs the aforementioned anisotropic quantization. Last is another optional step called rescoring, which more accurately computes the best k distances to get a more accurate score for each top hit, reducing speed in favor of greater accuracy. This experiment compared five methods: Pure brute force scoring (with and without quantization), asymmetric hashing (with and without reordering), and partitioning combined with asymmetric hashing and reordering. The results for these methods are shown in figures 7, 8, and 9.

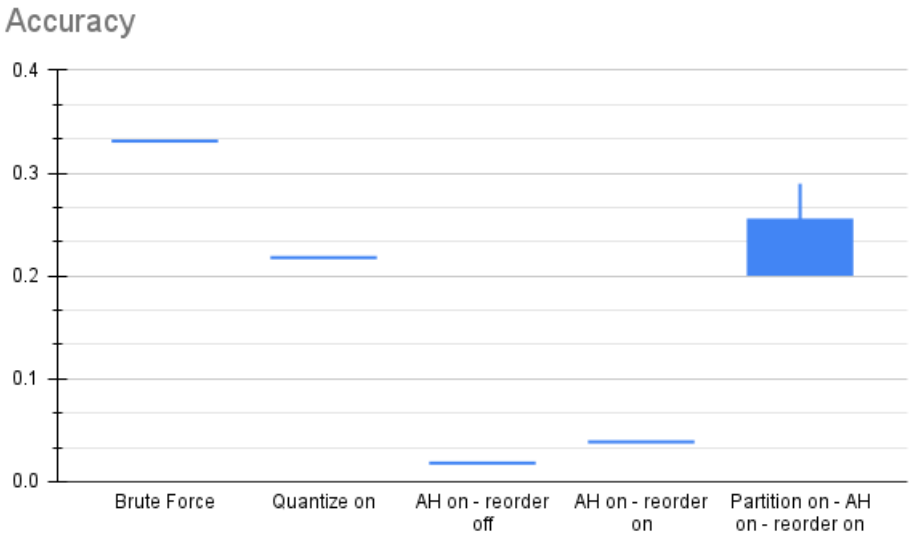


Figure 7: Scann accuracy vs. method

Figure 7 shows the changes in accuracy based on the different methods. It is clear here that pure brute force scoring performs the best on our dataset, likely because the number of data points is relatively small. Accuracy alone does not tell the whole story; however, it is demonstrated primarily in figures 8 and 9. While it is clear that enabling partitioning causes the index build time to increase massively, this is a favorable trade-off due to the decreased query latency. While it may be difficult to tell in the diagrams, brute-for trade-offs took nearly 50 times longer than the final partitioned method, achieving a maximum accuracy of just 6% less than brute force. Because of this, we determined the best option was to utilize the full stack method, combining partitioning with AH and reordering to minimize runtimes.

Multiprocessing was implemented to test multiple parameters at once. This created 30 different Scann instances simultaneously so we could efficiently iterate through different parameter combinations. All of the combinations were only tested once as we tried to make a pseudo-exhaustive testing framework so we could try to test the most promising combinations of parameters further.

It was found that the dimensions_per_block parameter resulted in a high amount of variance between accuracies. For instance, it is seen in Table 2 that only changing the dimensions per block parameter from 3 to 2 resulted in losing 0.03 percent accuracy. This can be seen in the first vs. ninth rows. This particular parameter dictates the compression done to the dataset. Interestingly, the higher the parameter, the higher the compression, but it would make more sense for the lower values of this parameter to yield higher accuracy. This may be due to the stochastic nature of searching. Thus, future work will attempt to get the average accuracy across this parameter. It

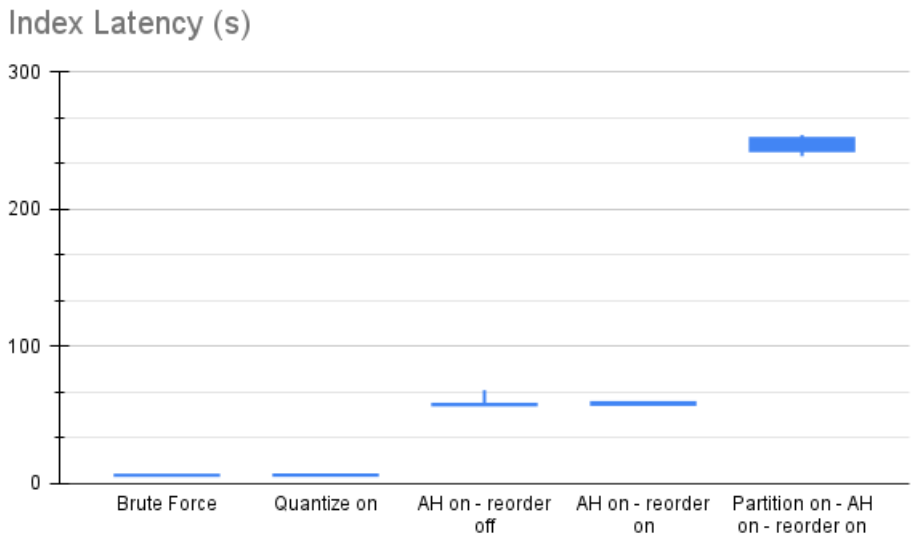


Figure 8: Scann index build time vs. method

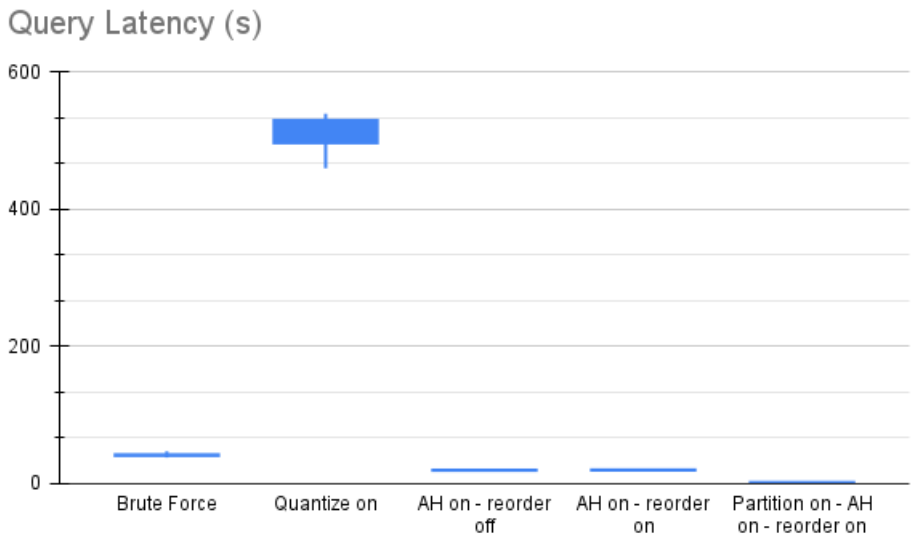


Figure 9: Scann index query time vs. method

is also interesting to note that as Table 3 shows, there is a minimum effect of parameters on the amount of memory that needs to be allocated to the ScaNN searcher. Thus, in the future, it makes sense not to concern ourselves with memory requirements but rather to hone in on the run times. The parameter, num_leaves_to_search, plays a significant role in query times. As we increase the number of leaves to search, the query_time increases almost linearly, as seen in Figure 10. It needs to be evaluated if the non-guaranteed accuracy is worth the guaranteed increase in time. Ultimately, the best results were found to come from little change. By implementing the near default parameters written by google, we achieved our best performance with accuracy up to 31% and indexing and query times of on average 250 and 4 seconds respectively. Any deviations from this (switching the training iterations, distance metrics, or overall process) tended to cause accuracy to decrease and runtimes to increase. Ultimately, the optimal pipeline was found to be as follows: Build the index with 10 iterations and a dot product distance metric. Then, partition the database with 2000 leaves and training sample size=250000. Then, utilize asymmetric hashing, with the recommended dimensions per block = 2 and anisotropic quantization threshold=0.2.

| Dim/Block | Leaves_Searched | Num_Leaves | Quantization_Threshold | Train_Iter | Accuracy |
|---|---|---|---|---|---|
| 3 | 59 | 238 | 0.3 | 10 | 0.153409 |
| 1 | 59 | 238 | 0.6 | 10 | 0.148008 |
| 1 | 119 | 238 | 0.6 | 10 | 0.147938 |
| 3 | 59 | 238 | 0.0 | 10 | 0.146535 |
| 2 | 119 | 238 | 0.9 | 10 | 0.144641 |
| 2 | 59 | 238 | 0.9 | 10 | 0.144571 |
| 1 | 59 | 119 | 0.6 | 10 | 0.122405 |
| 1 | 119 | 119 | 0.6 | 10 | 0.122194 |
| 2 | 59 | 119 | 0.3 | 10 | 0.122054 |
| 2 | 119 | 119 | 0.3 | 10 | 0.121984 |

Table 2: ScaNN parameter sweep, for the parameters most relevant to accuracy. Indexing was created on 165615 samples, and the top 10 accuracies over 14,256 samples were chosen

Finally, reorder, and the resulting index can be queried with leaves to search=100.

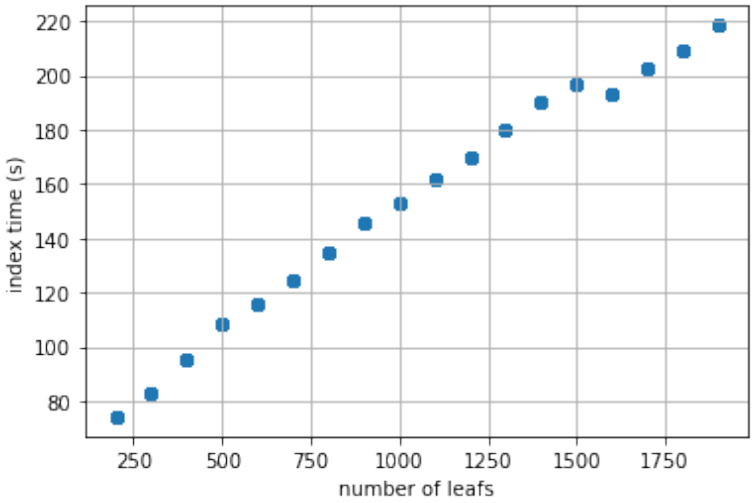


Figure 10: Number of leafs calculated vs Indexing time

The comparison Figure 23 illustrates the performance disparities between the Faiss and ScaNN methods across various metrics. Regarding accuracy, Faiss exhibits superiority, boasting a higher percentage (35.4%) compared to ScaNN (31%). This trend extends to specificity and sensitivity, where Faiss outperforms ScaNN with percentages of 93.7% and 92.9% for specificity and 92.9% and 92.5% for sensitivity, respectively. This is the most prominent advantage the FAISS has over ScaNN.

Examining resource utilization during indexing, Faiss generally utilizes more resources than ScaNN, which displays lower CPU usage (3.6%) compared to FAISS (10.9%). FAISS and ScaNN also showcase similar efficiency in memory usage during indexing, utilizing 678.3MB in contrast to ScaNN's 646.92MB. The story is significantly different, however, when it comes to runtime. When creating the index Faiss excels in timing, completing the process in 1.6 seconds, whereas ScaNN takes significantly longer at 240.9 seconds. However, this trend is reversed during the searching phase, where ScaNN demonstrates its advantage with lower CPU usage (20%) compared to FAISS (62.8%). ScaNN also demonstrates efficiency in memory usage of 55.798MB and timing (2.1 seconds) during searching, even outperforming FAISS, which requires 58MB and 8.177 seconds, respectively. In summary, both have merits. FAISS consistently exhibits superior performance in accuracy and indexing timing, whereas ScaNN demonstrates its capabilities to quickly and efficiently search indexes. Both have demonstrated strong capabilities, although FAISS's better accuracy and significantly shorter indexing times likely make it a better choice for extensive dataset indexing and searching tasks than ScaNN.

| Dim/Block | Leaves_Search | Leaves | Q_Thresh | Train_Iter | Query_Time(s) | Build_Time (s) | BuildMem (MB) | PredictMem (MB) |
|---|---|---|---|---|---|---|---|---|
| 3 | 59 | 238 | 0.0 | 10 | 42.343 | 259.752 | 646.942 | 55.798 |
| 1 | 59 | 238 | 0.0 | 10 | 40.458 | 262.846 | 646.942 | 55.798 |
| 1 | 59 | 119 | 0.0 | 10 | 66.610 | 250.684 | 646.942 | 55.798 |
| 1 | 59 | 119 | 0.0 | 10 | 55.649 | 262.999 | 646.942 | 55.798 |
| 3 | 119 | 119 | 0.0 | 20 | 75.226 | 256.698 | 646.942 | 55.798 |
| 1 | 59 | 119 | 0.0 | 20 | 54.117 | 260.336 | 646.942 | 55.798 |
| 3 | 59 | 238 | 0.3 | 10 | 50.445 | 258.313 | 646.942 | 55.798 |
| 3 | 59 | 238 | 0.3 | 10 | 46.139 | 266.552 | 646.942 | 55.798 |
| 2 | 59 | 119 | 0.3 | 10 | 54.067 | 272.540 | 646.942 | 55.798 |
| 2 | 119 | 119 | 0.3 | 10 | 72.038 | 271.440 | 646.942 | 55.798 |

Table 3: How changing parameters influence the build/search time and the build/search memory requirements.

## Exploring Novelty in Phylogenetic Structures: Distance Projection

We assessed the distinction between the distribution of the distances of Faiss embeddings, characterized by its high-dimensional structure with a shape of (14256, 165615), representing a comprehensive distance matrix that encapsulates the phylogenetic similarities across various organisms. We employed advanced dimensionality reduction and clustering techniques to elucidate the underlying structure and novelty within the data. Our objective was to discern the novelty within this dataset, aiming to identify organisms at a new taxonomic level that exhibit unique genetic signatures distinct from those in known phyla to the training set.

By assessing the distribution of distances within and across domains in our dataset, we sought to uncover the extent of novelty and the evolutionary divergence of organisms. The distance distribution histogram for Faiss, as shown in 15, shows a heavily right-skewed distribution, indicating that most distance values are clustered around the lower end of the scale. The ScaNN distance distribution, as shown in 16, shows a narrower and more centered spread of values than the one from FAISS. Here, the distribution is still right-skewed but less pronounced, with a tighter cluster of distance values. For Faiss, the minimum distance is 0.0, indicating some items are identical or extremely close in the dataset, while the maximum distance is quite far at 34.0929, suggesting some very dissimilar pairs. The mean distance is relatively low at 2.196, again supporting the fact that many items are close to each other. The median being lower than the mean (1.231) confirms the right skew of the distribution as it indicates that more than half of our distance values are below the mean. The standard deviation is 2.593, which is relatively high, indicating a considerable spread of distances from the mean. For ScaNN, the minimum distance is 6.8944, the maximum is lower at 9.380 compared to FAISS's range, the mean is very close to the median (8.820 and 8.832, respectively), and the standard deviation is relatively small at 0.0763. This tighter distribution implies that the ScaNN algorithm produces more consistent distances between items, suggesting a higher precision in maintaining relative distances between points in the gene embeddings. It's interesting to note the significantly smaller range of distances compared to FAISS, indicating less variability and potentially fewer outliers or less diversity in the dataset as interpreted by ScaNN.

We applied the Elbow Method with K-means clustering to navigate this high-dimensional genetic landscape. We evaluated the Within-Cluster Sum of Squares (WCSS) to ascertain the most appropriate number of clusters. This analysis indicated a progressive reduction in WCSS as we explored from 1 to 19 clusters, with the rate of decrease becoming markedly less pronounced beyond 5 clusters, hinting at an optimal clustering solution for further investigation.

We leveraged UMAP (Uniform Manifold Approximation and Projection) to reduce the dimensionality of our distance projection data, achieving a more tractable 2-dimensional representation that enabled us to visually explore the dataset's inherent structure using the UMAP function. Figure 22 shows the result of UMAP, colorized based on the in- and out-domain categories and phyla. As we know, the out-domain category comprises those from different phyla than the training set. We could observe that for both SCANN and Faiss distances, there is no clear distinction based on in-domain and out-domain sets. This could be due to UMAP preserving its specific features in its projection. However, when colored based on their phyla, it is interesting that we observe a distinction because the distance projection is fascinating. We could see some clusters of phyla, indicating that similar sequences from the same phyla may have similar distances to other phyla.

The Kernel Density Estimation (KDE) plots, as shown in Figure 17, for in-domain and out-domain categories within our dataset, offer insightful visualizations of the distribution of the best hit distances. Based on this figure, we observe that the KDE plots of SCANN for categories

map to each other, displaying distinct distributions. Similarly, for Faiss, we observe two different distributions of in-domain and out-domain categories. The plot for out-domain categories is skewed to the left, predominantly containing points from the in-domain. To determine an optimal threshold for classification, we utilize the ROC plot to maximize the difference between True Positive Rate (TPR) and False Positive Rate (FPR). This threshold is reflected in the KDE plot. Additionally, we calculate the Area Under the Curve (AUC) in the ROC curve for Faiss and SCANN. Notably, Faiss's AUC is significantly higher than that of SCANN. We obtained an AUC of 0.73 for Faiss and 0.58 for SCANN, which shows Faiss's superiority over SCANN.

# Discussion

We observed that Faiss distance projection embeddings outperformed SCANN, with Faiss exhibiting a significantly higher Area Under the Curve (AUC) of 0.73 compared to SCANN's 0.58, indicating its superiority in capturing genetic diversity and novelty. However, when considering other performance metrics, SCANN proves to be much better regarding search time, memory efficiency, and CPU usage than Faiss. This makes SCANN more convenient for handling high-dimensional and large datasets.

The results of our experiment present several significant implications. First and foremost, our research reveals that for greater accuracy results, FAISS should be used to perform dense similarity searches. In contrast, ScaNN should be used when speed and hardware are of significant concern due to the optimizations that make it capable of more efficient searching. While unable to correctly detect novelty, we still believe these libraries could provide much-needed processing power to address large sequence databases effectively.

It should be noted that the packages were not run on the GPU for this experiment, as our group could not get proper access to these features. Ergo, the question of FAISS and ScaNN's true performance capabilities for analyzing large genomic datasets has been left unanswered. As a result, we recommend additional research be done to determine the performance of FAISS and ScaNN on multiple GPUs. This may provide helpful information on how to utilize each package and may change the results we found for this experiment. Additional checks should also be performed on different data sets to verify the inability of these packages to detect novelty within metagenomic datasets.

Alongside testing the GPU capabilities of both packages and verifying the results of these experiments, tests should be conducted to determine their applicability for other applications in bioinformatics and for use with other popular packages used in the field today.

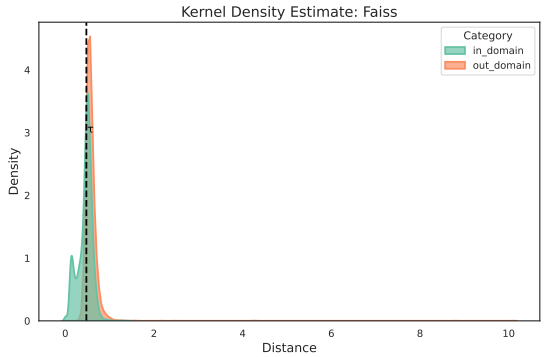


Figure 11: Kernel density estimation plot of Faiss

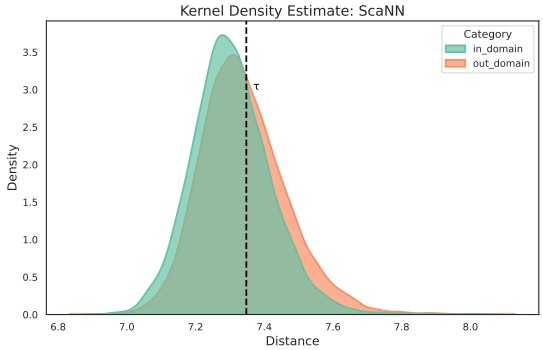


Figure 12: Kernel density estimation plot of ScaNN

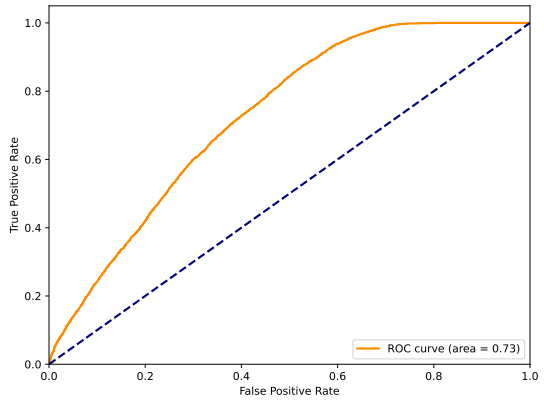


Figure 13: ROC plot of Faiss

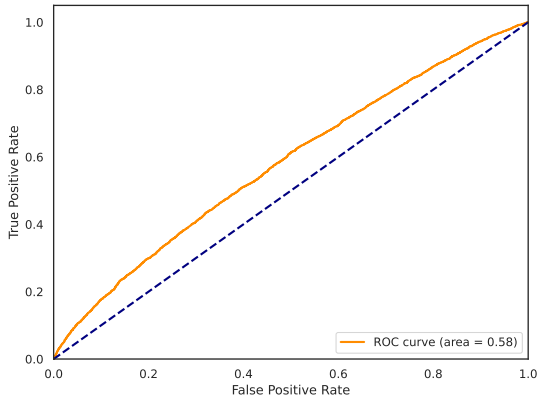


Figure 14: ROC plot of ScaNN

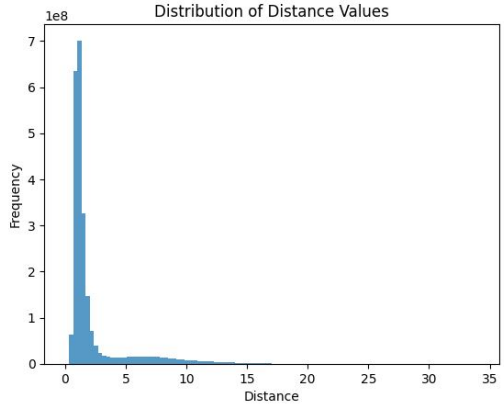


Figure 15: Histogram Representation of Distribution of Distances: Faiss

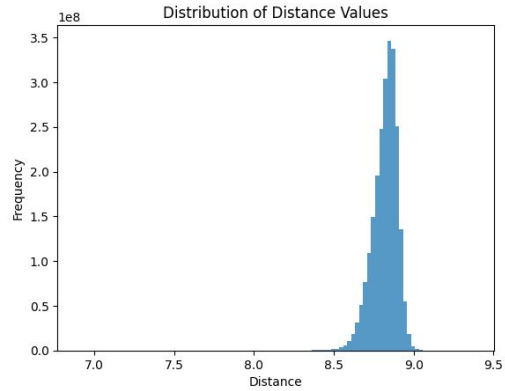


Figure 16: Histogram Representation of Distribution of Distances: ScaNN

Figure 17: Visualization of kernel density estimation plots and ROC plots for the Faiss and ScaNN models.

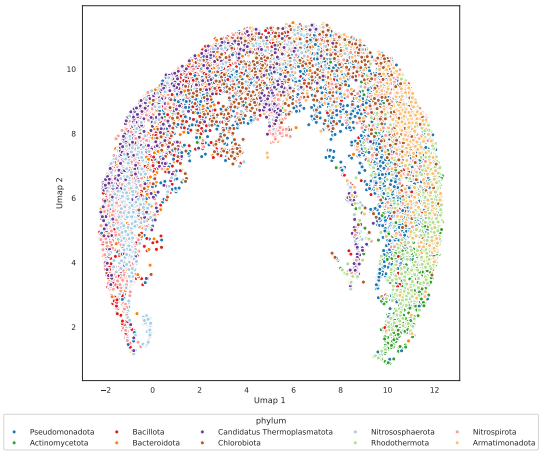


Figure 18: UMAP-Faiss distances Colorized Based on Phyla

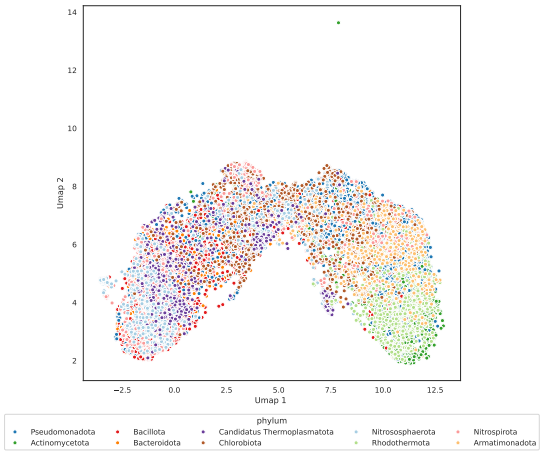


Figure 19: UMAP-ScaNN distances Colorized Based on Phyla

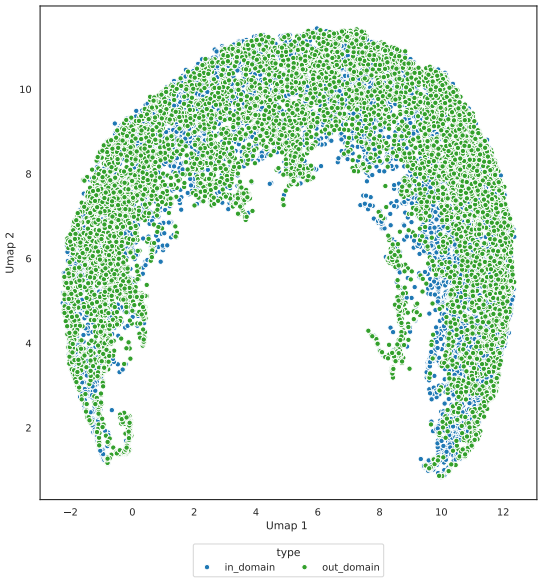


Figure 20: UMAP-Faiss distances Colorized Based on Category

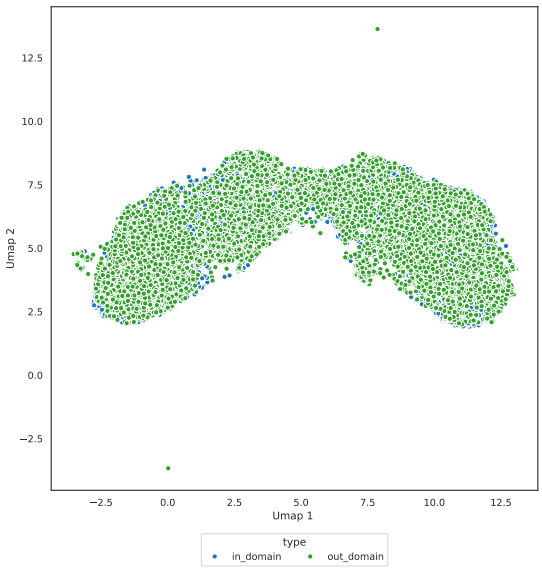


Figure 21: UMAP-ScaNN distances Colorized Based on Category

Figure 22: UMAP Projection: Comprehensive visualization showcasing UMAP projections at both the phylum level and category( in-domain/out-domain) distances.

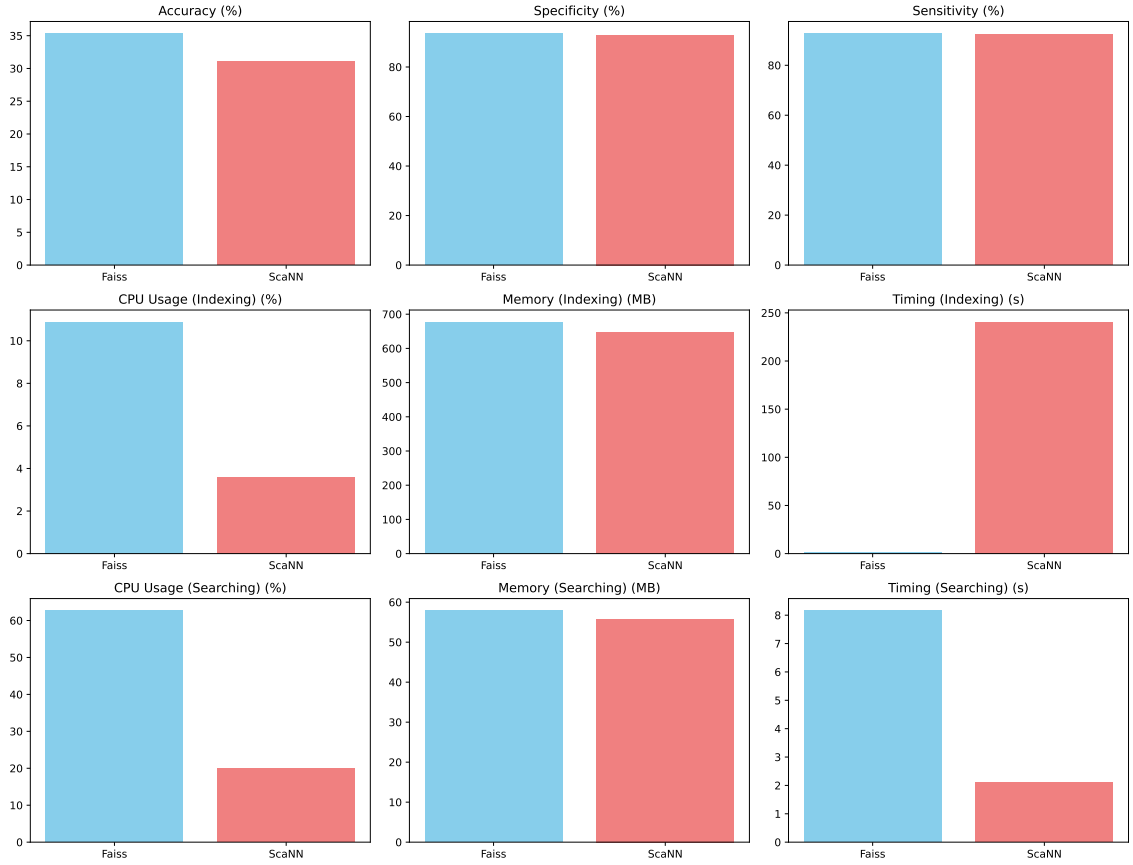


Figure 23: Performance Metrics Comparison between Faiss and SCANN.

# References

André, F., A.-M. Kermarrec, and N. Le Scouarnec (2019). Quicker adc: Unlocking the hidden potential of product quantization with simd. *IEEE transactions on pattern analysis and machine intelligence 43*(5), 1666–1677.

Aumüller, M. and M. Ceccarello. Recent approaches and trends in approximate nearest neighbor search, with remarks on benchmarking. *Data Engineering*, 89.

Borgeaud, S., A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. B. Van Den Driessche, J.-B. Lespiau, B. Damoc, A. Clark, et al. (2022). Improving language models by retrieving from trillions of tokens. In *International conference on machine learning*, pp. 2206–2240. PMLR.

Douze, M., A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P. E. Mazare, M. Lomeli, L. Hosseini, and H. Jegou (2024). *The FAISS Library*.

FitzGerald, N., J. A. Botha, D. Gillick, D. M. Bikel, T. Kwiatkowski, and A. McCallum (2022). Moleman: Mention-only linking of entities with a mention annotation network.

Gordo, A., F. Perronnin, G. Yunchao, and S. Lazebnik (2014, Jan). Asymmetric distances for binary embeddings. *IEEE Transactions on Pattern Analysis and Machine Intelligence 36*(1), 33–47.

Guo, R., P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar (2020). Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pp. 3887–3896. PMLR.

Johnson, J., M. Douze, and H. Jégou (2019). Billion-scale similarity search with gpus. *IEEE Transactions on Big Data 7*(3), 535–547.

Jégou, H., M. Douze, and C. Schmid (2011). Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence 33*(1), 117–128.

Olson, D. R., D. Demekas, T. Colligan, and T. Wheeler (2024). Near: Neural embeddings for amino acid relationships. *bioRxiv*, 2024–01.

Schütze, K., M. Heinzinger, M. Steinegger, and B. Rost (2022). Nearest neighbor search on embeddings rapidly identifies distant protein relations. *Frontiers in Bioinformatics 2*, 1033775.

Valkiers, S., M. Van Houcke, K. Laukens, and P. Meysman (2021, 06). ClusTCR: a python interface for rapid clustering of large sets of CDR3 sequences with unknown antigen specificity. *Bioinformatics 37*(24), 4865–4867.

Xiong, L., C. Xiong, Y. Li, K.-F. Tang, J. Liu, P. Bennett, J. Ahmed, and A. Overwijk (2020). Approximate nearest neighbor negative contrastive learning for dense text retrieval. *arXiv preprint arXiv:2007.00808*.