INDEX

INTRODUCTION

1.1 ABSTRACT

Fuzzy matching plays a crucial role in various applications such as information retrieval, natural language processing, and data deduplication. This project aims to develop a fuzzy matching tool that utilizes a combination of term vectorization using the tf-idf algorithm, vector similarity measures, and Euclidean distance calculations to find the closest matching terms from a given dataset.
By combining term vectorization, vector similarity measures, and approximate nearest neighbor algorithms, the fuzzy matching tool achieves accurate and efficient matching of terms from the dataset. The experimental evaluation demonstrates improved time complexity, making it suitable for real-time applications with large datasets.

1.2 JUSTIFICATION FOR THE SYSTEM

The need for a fuzzy matching search tool arises from the inherent challenges of dealing with data that contains variations, errors, or inconsistencies. In real-world scenarios, terms may be misspelled, abbreviated, or expressed differently, leading to difficulties in exact matching. A fuzzy matching search tool addresses this need by employing algorithms and techniques that can identify approximate matches, allowing for more flexible and robust searching. It enables improved information retrieval, data deduplication, and natural language processing by accommodating variations in spelling, syntax, and other linguistic nuances, ultimately enhancing
the accuracy and efficiency of search operations.

1.3 PROBLEM STATEMENT AND OBJECTIVE

Fuzzy matching plays a vital role in various applications such as information retrieval, data deduplication, and natural language processing. However, traditional fuzzy matching approaches based on algorithms like Levenshtein distance or soundex have limitations in capturing semantic similarities and handling variations in spelling, abbreviations, and linguistic nuances. This can result in inaccurate matching results and increased query time complexity.

The primary objective of this research project is to develop an advanced fuzzy matching tool that overcomes the limitations of traditional approaches. The specific objectives are as follows:

1. Enhance Semantic Similarity: Design and implement a fuzzy matching algorithm that leverages the tf-idf algorithm to create term vectors, capturing the semantic meaning and importance of terms within the dataset. This approach will enable more accurate matching based on ngram similarities rather than relying solely on character-level differences.

2. Improve Handling of Variations: Develop techniques to handle variations in spelling, abbreviations, synonyms, and other linguistic nuances. The fuzzy matching algorithm should consider these variations and provide reliable matching results, reducing the impact of such discrepancies on the overall matching process.

3. Optimize Query Time Complexity: Implement indexing techniques using approximate nearest neighbors algorithms, such as k-means clustering and Spotify's Annoy algorithm, to optimize the time complexity of fuzzy matching queries. This will enable efficient retrieval of matching terms from large datasets, ensuring real-time performance even with increasing data volumes.

4. Evaluate Performance: Conduct extensive experiments to evaluate the performance and effectiveness of the developed fuzzy matching tool. Compare it against traditional fuzzy matching approaches, considering factors such as accuracy, precision, query time, and scalability.

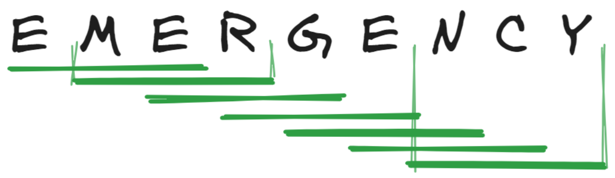
Through extensive experimentation and evaluation, the project seeks to demonstrate the effectiveness and performance of the developed tool and provide a more robust and efficient solution for fuzzy matching in various applications, leading to improved information retrieval, data deduplication, and natural language processing tasks.

SYSTEM DESIGN AND IMPLEMENTATION

2.1 N-GRAMS GENERATION

In the context of fuzzy matching, n-grams refer to contiguous sequences of N characters or words within a text. These sequences are used to capture patterns and similarities between terms or strings, allowing for more flexible and robust matching.

When applying fuzzy matching with n-grams, the input text or terms are divided into these smaller segments of N characters or words. Each segment represents a partial sequence within the text. For example, n-grams of the word ‘EMERGENCY’ would be [‘EME’, ‘MER’, ‘ERG’, ‘RGE’, ‘GEN’, ‘ENC’, ‘NCY’].



By considering these n-grams, fuzzy matching can accommodate variations and discrepancies between terms. For instance, even if there are missing or transposed characters in a term, the overlapping n-grams allow for partial matching and can capture similarities.

2.2 TERM VECTORIZATION

Vectors are fundamental entities that reside in an n-dimensional space and serve as containers for maintaining an orderly arrangement of elements. Widely employed in computer science, vectors are instrumental in representing and manipulating entities or quantities possessing multiple attributes or dimensions.

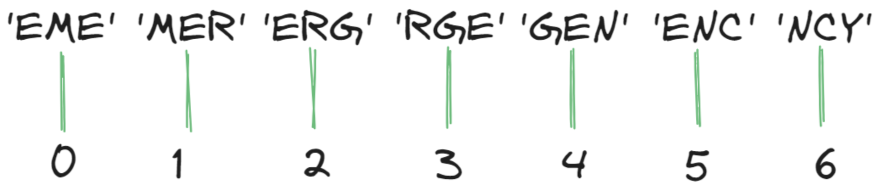
Vectorization refers to the process of representing textual terms or documents as numerical vectors. It involves transforming the textual data into a numerical format that can be processed and compared using mathematical operations.

By converting textual terms or documents into vectors, Seekr can perform similarity calculations and comparisons based on vector distances or similarities. Vectorization enables the application of various mathematical algorithms and techniques to identify matching patterns, assess similarity levels, and rank the relevance of terms or documents.

TOKENIZATION

Tokenization is a fundamental process in natural language processing (NLP) that involves segmenting a given text sequence, such as a sentence or a document, into discrete units known as tokens. These tokens encompass a wide range of possibilities, including words, phrases, symbols, or even individual characters, contingent upon the specific requirements and techniques employed.

The significance of tokenization lies in its ability to provide a structured representation of textual data, facilitating subsequent analysis and processing at a more refined level. By breaking down the text into meaningful units, tokenization forms the basis for various advanced NLP techniques, empowering the extraction of valuable information from text documents.



In the realm of fuzzy matching, tokenization assumes a crucial role by enabling the partitioning and assignment of a distinct identifier to each n-gram present within the dataset. This step is vital for subsequent processing of n-grams as vector entities, allowing for further computations and similarity assessments.

TF-IDF ALGORITHM

The TF-IDF (Term Frequency - Inverse Document Frequency) algorithm is a popular technique in natural language processing that assigns weights to terms in a document based on their frequency and importance in a corpus. It is widely used for information retrieval, text mining and fuzzy matching.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j

df_i = number of documents containing i

N = total number of documents

The TF component of TF-IDF measures the frequency of a term within a document. It assigns higher weights to terms that appear more frequently, under the assumption that they are more significant in representing the content of the document. However, without considering the overall distribution of the term across the corpus, TF alone may not adequately capture the importance of a term.

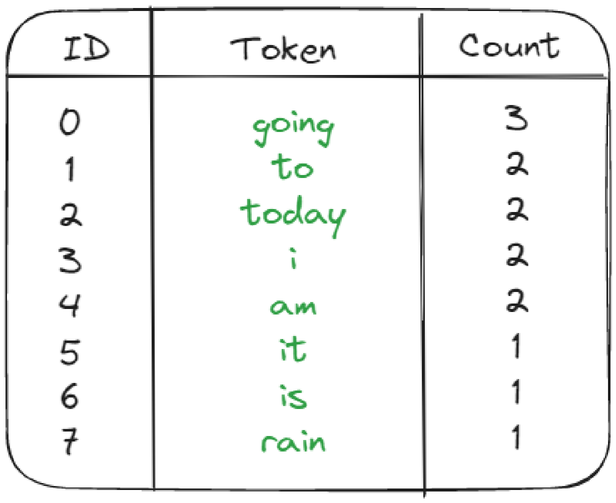
The IDF component of TF-IDF measures the rarity or uniqueness of a term across the corpus. It assigns higher weights to terms that are relatively infrequent in the corpus, emphasizing their significance in distinguishing a document from others. By considering the inverse document frequency, IDF helps mitigate the influence of common terms that appear in many documents but carry less discriminatory power.

The combination of TF and IDF in the TF-IDF algorithm results in a weighted representation of terms within a document. Terms that are both frequent within the document (high TF) and rare across the corpus (high IDF) receive the highest weights, indicating their relevance and discriminatory power.

Consider the following example,

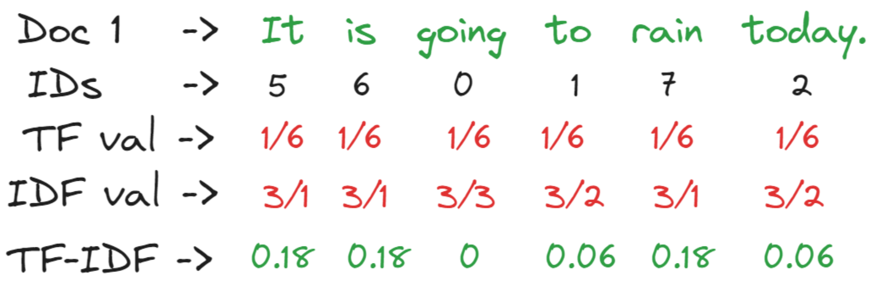
Doc 1 -> It is going to rain today.
Doc 2 -> Today I am not going outside.
Doc 3 -> I am going to watch the season premiere.

Step 1: Cleaning the data and tokenization



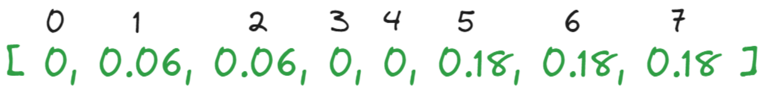
ID	Token	Count
0	going	3
1	to	2
2	today	2
3	i	2
4	am	2
5	it	1
6	is	1
7	rain	1

Step 2: Calculating TF-IDF values for each document within the dataset



Doc 1 ->	It	is	going	to	rain	today.
IDs ->	5	6	0	1	7	2
TF val ->	1/6	1/6	1/6	1/6	1/6	1/6
IDF val ->	3/1	3/1	3/3	3/2	3/1	3/2
TF-IDF ->	0.18	0.18	0	0.06	0.18	0.06

Step 3: Converting documents to vectors



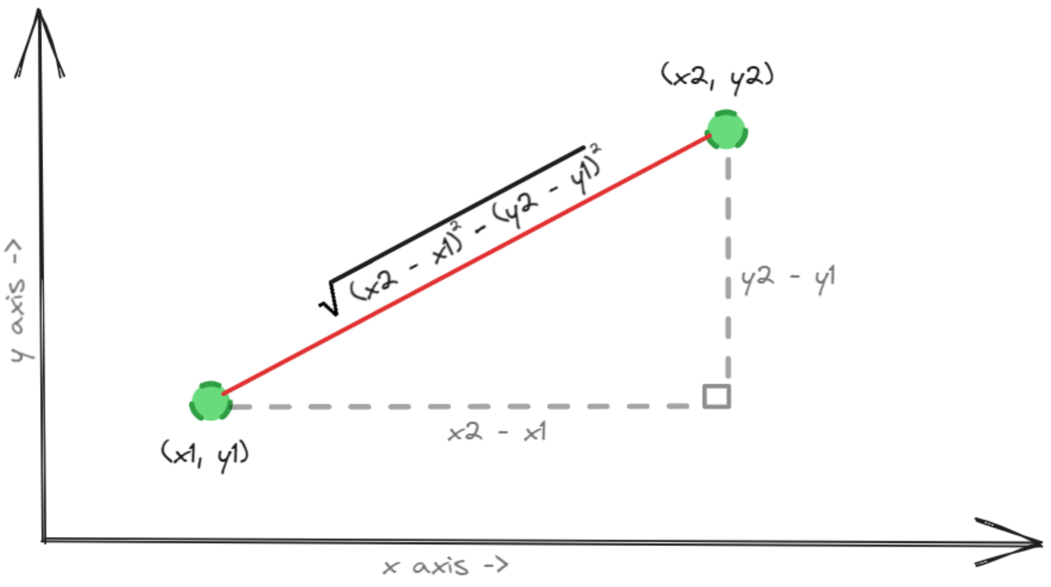
$$\begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ [\,0, & 0.06, & 0.06, & 0, & 0, & 0.18, & 0.18, & 0.18\,] \end{array}$$

2.3 DISTANCE METRICS

The choice of distance metric, may depend on the specific requirements and characteristics of the fuzzy matching task at hand. Different distance metrics like Euclidean distance, cosine similarity, Levenshtein distance or Jaccard similarity, are to be employed depending on the nature of the data and the desired matching behavior.

EUCLIDEAN DISTANCE

Euclidean distance refers to a measure of dissimilarity or similarity between two vectors. It is a mathematical distance metric that calculates the straight-line distance between two points in a multidimensional space.

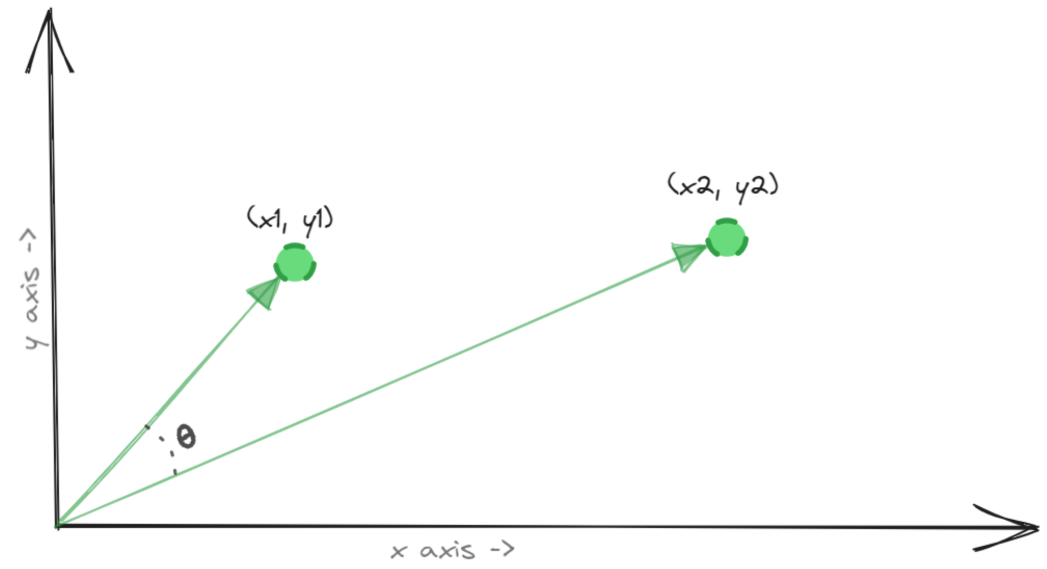


Euclidean distance is often used to quantify the similarity or difference between feature vectors representing textual terms or documents. In the context of fuzzy matching, the Euclidean distance between two vectors indicates the degree of dissimilarity between them. Smaller Euclidean distances imply greater similarity, while larger distances indicate greater dissimilarity.

By calculating the Euclidean distance between vectors, Seekr can determine the closeness or similarity between terms or documents, aiding in the identification of potential matches or near-matches. It provides a quantitative measure to assess the degree of resemblance or divergence, allowing for more accurate fuzzy matching results.

COSINE SIMILARITY

Cosine similarity is a measure of similarity between two vectors that represents textual terms or documents. It quantifies the cosine of the angle between the two vectors, indicating the degree of alignment or similarity.



The cosine similarity ranges from -1 to 1, with values closer to 1 indicating higher similarity and values closer to -1 indicating dissimilarity. A cosine similarity of 1 indicates that the vectors are perfectly aligned or identical, while a similarity of -1 means they are completely opposite or orthogonal.

Cosine similarity is commonly used in fuzzy matching tasks where the emphasis is on the semantic similarity or the direction of meaning rather than the specific terms or lengths. It helps identify similar terms or documents based on their overall alignment or orientation in the vector space, disregarding the scale or magnitude of the vectors.

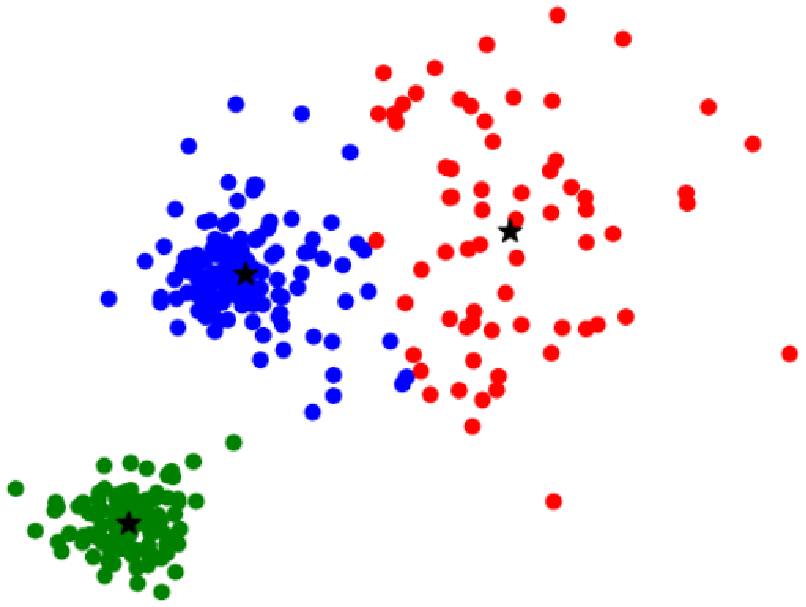
By computing the cosine similarity between vectors, fuzzy matching algorithms can determine the degree of resemblance or similarity between terms or documents, assisting in identifying potential matches or near-matches. It provides a flexible and effective measure for fuzzy matching tasks, especially in scenarios where semantic similarity is a priority.

2.4 VECTOR INDEXING SCHEMES

Vector indexing plays a pivotal role in vector similarity search by providing efficient and scalable mechanisms to locate and retrieve similar vectors. It allows for fast query response times, reduces computational overhead, and enables effective handling of large-scale vector datasets.

K-MEANS CLUSTERING

K-means clustering is a popular algorithm used for vector similarity search in the context of information retrieval and data mining. It is a partitioning-based clustering algorithm that aims to group similar vectors into clusters based on their feature similarities.

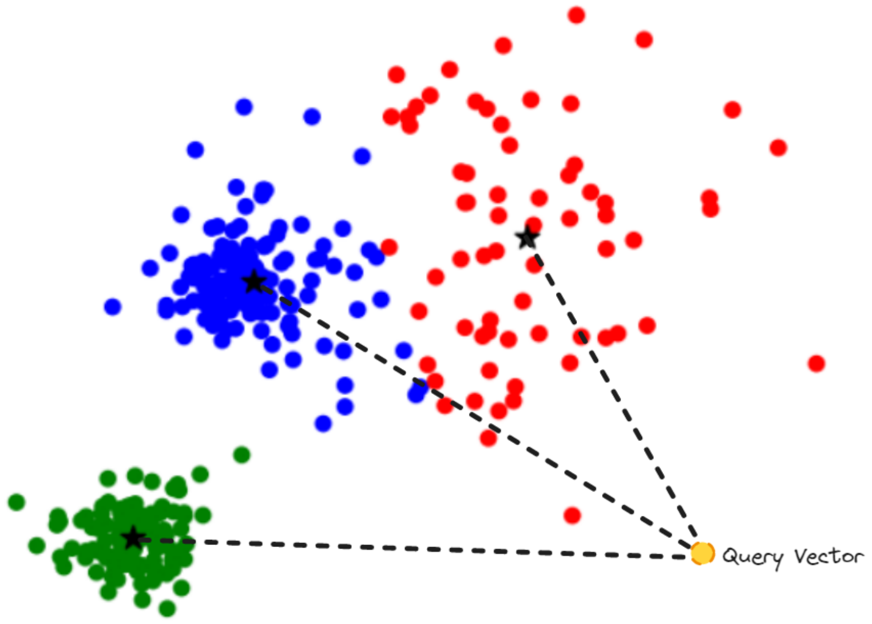


The algorithm begins by randomly selecting a predetermined number of cluster centroids (representative points) in the vector space. Each vector in the dataset is then assigned to the cluster whose centroid is closest to it based on a distance metric, typically the Euclidean distance.

After the initial assignment, the algorithm iteratively updates the cluster centroids by computing the mean of the vectors within each cluster. This step involves calculating the average of the feature values across the vectors in a cluster, resulting in a new centroid position.

The vectors are then reassigned to the cluster with the closest centroid based on the updated positions. This process continues iteratively until convergence, where the centroids stabilize and no further reassignments occur or a predefined number of iterations is reached.

The resulting clusters represent groups of vectors that exhibit high similarity within each cluster and significant dissimilarity between different clusters. This clustering technique allows for efficient vector similarity search, as it reduces the search space by partitioning vectors into meaningful groups.



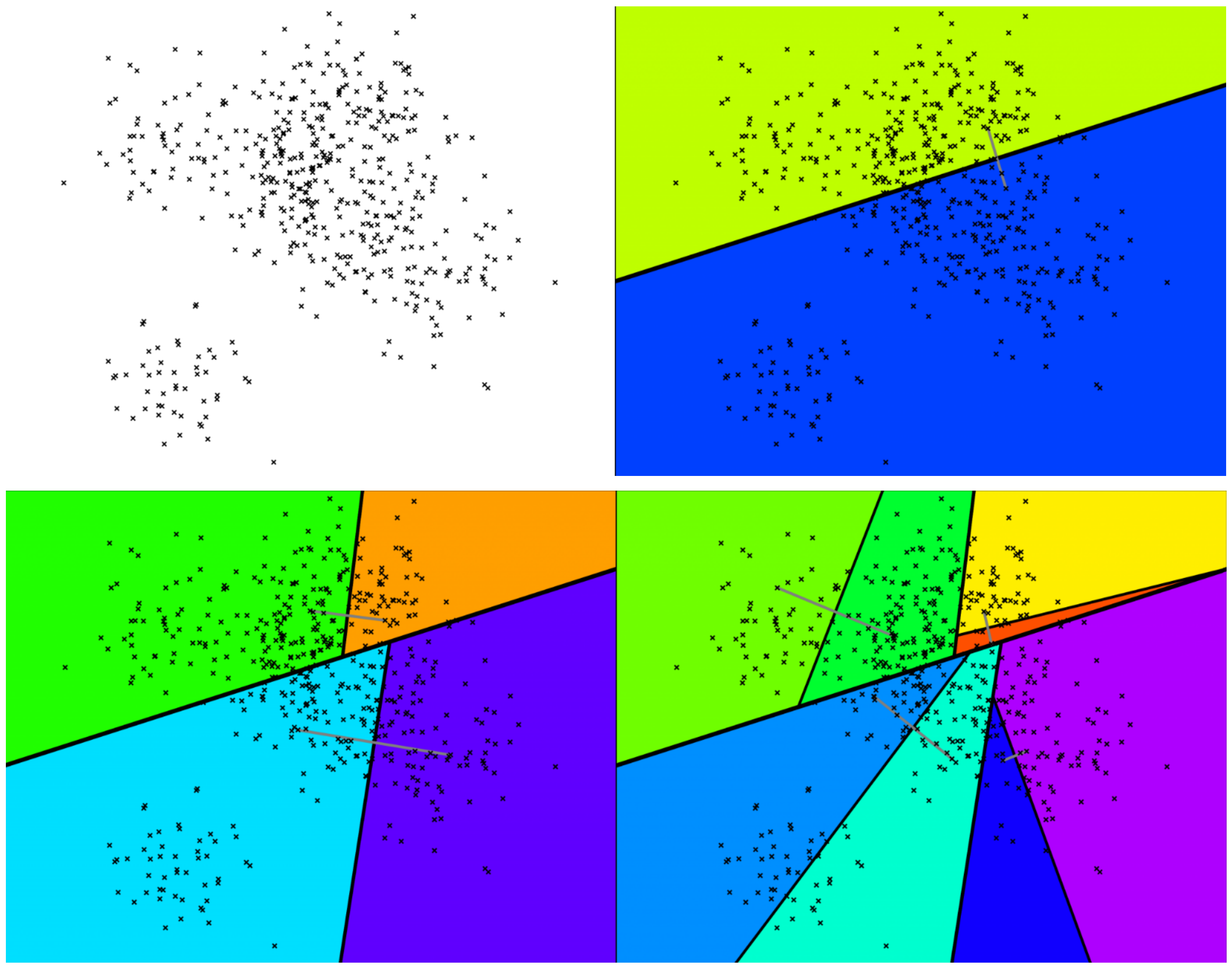
During the search process, a query vector is compared with the cluster centroids to determine the most relevant cluster. The vectors within that cluster are then considered as potential matches for the query, reducing the search space and improving search efficiency.

K-means clustering for vector similarity search provides a scalable and effective approach for organizing and retrieving similar vectors in large datasets. It enables faster search operations by grouping vectors into clusters, facilitating more efficient similarity-based retrieval and aiding tasks such as fuzzy matching, recommendation systems, and information retrieval.

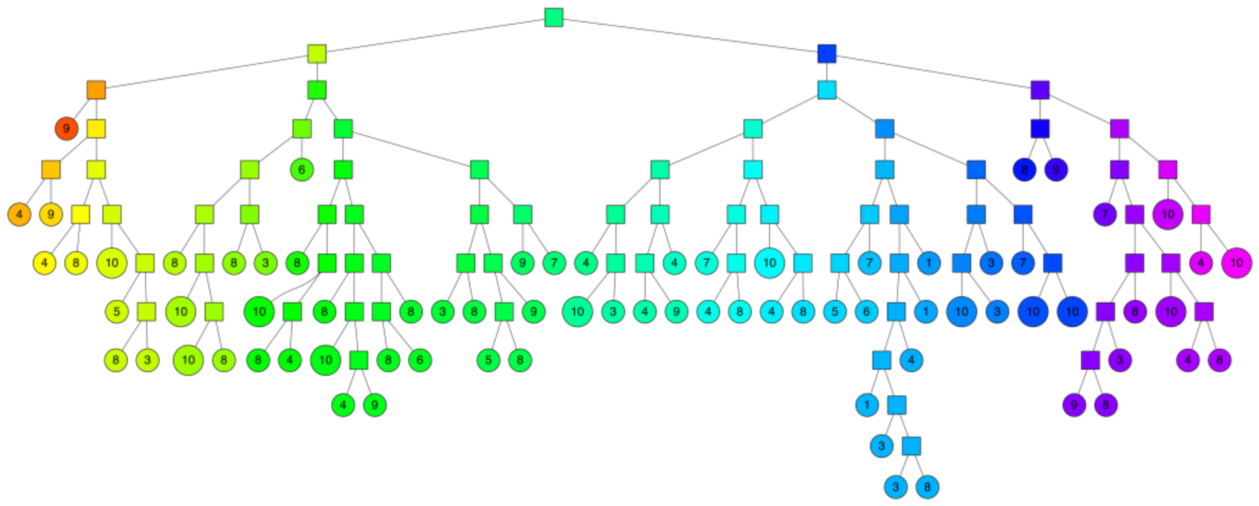
ANNOY ALGORITHM

Spotify's Annoy (Approximate Nearest Neighbors Oh Yeah) is an algorithm designed for efficient vector similarity search in high-dimensional spaces. It offers a trade-off between search accuracy and computational efficiency by providing approximate nearest neighbor search.

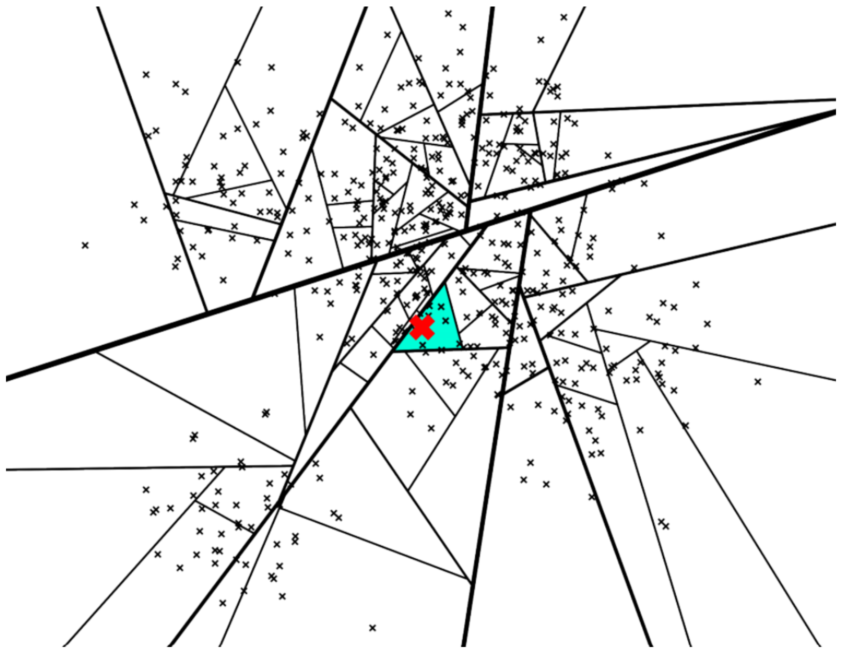
The Annoy algorithm works by building an index structure that hierarchically partitions the vector space into smaller regions. It employs the concept of binary trees, where each node represents a partition or split of the space.



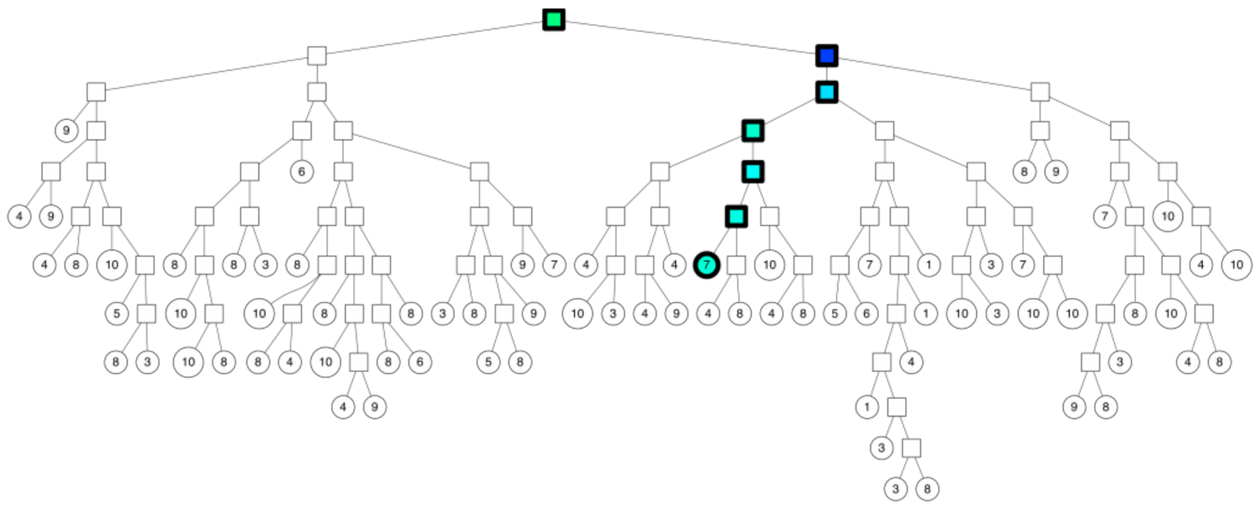
During the index creation, at every intermediate node in the tree, a random hyperplane is chosen, which divides the space into two subspaces. This hyperplane is chosen by sampling two points from the subset and taking the hyperplane equidistant from them. This process continues until a specified number of levels or a termination condition is reached. Each leaf node of the tree represents a subset of vectors within a specific region of the space.



To perform a search, the algorithm navigates the tree by comparing the query vector with the partitions at each level. It traverses the tree, following the path that leads to regions likely to contain similar vectors. The search terminates when it reaches a leaf node.



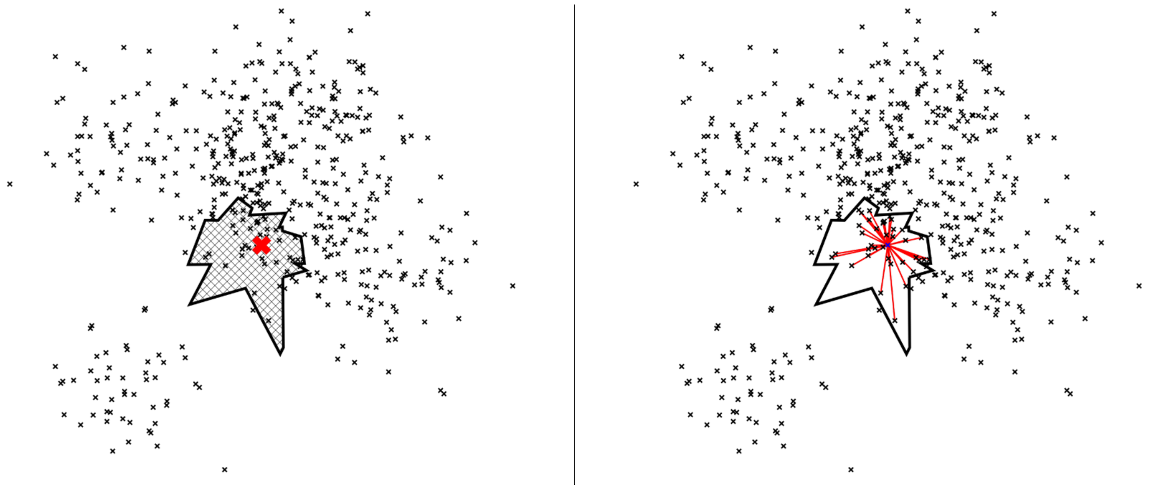
In the leaf node, a linear scan is performed to identify the nearest neighbors to the query vector. However, since the number of vectors in a leaf node is relatively small compared to the entire dataset, this linear scan is computationally efficient. The approximate nearest neighbors returned by the algorithm are close to the true nearest neighbors, striking a balance between accuracy and speed.



Another feature of this algorithm is that instead of relying on a single tree, Annoy constructs a collection of multiple trees, known as a forest.

Each tree in the forest is built independently using a subset of the dataset. This random sampling ensures diversity in the partitions and splits created by each tree. The number of trees in the forest is a parameter that can be adjusted based on the desired trade-off between accuracy and computational cost.

During the search process, the query vector is compared with the partitions at each level of every tree in the forest. The search follows the same principles as in a single tree, traversing down the branches based on the similarity between the query and the partitions. However, in Annoy, the search is performed simultaneously across all trees in the forest.



By searching in multiple trees, Annoy increases the chance of finding the nearest neighbors efficiently. The forest acts as an ensemble, collectively contributing to the accuracy of the approximate nearest neighbor search. The final result is obtained by aggregating the nearest neighbors found across all trees.

Annoy offers a flexible and scalable solution for vector similarity search, particularly in scenarios where the dimensionality of the vectors is high. It provides a fast retrieval mechanism by leveraging the hierarchical structure of the index, allowing for efficient exploration of the search space. This makes Annoy well-suited for applications such as fuzzy matching, recommendation systems, content-based search, and clustering in large-scale vector datasets.

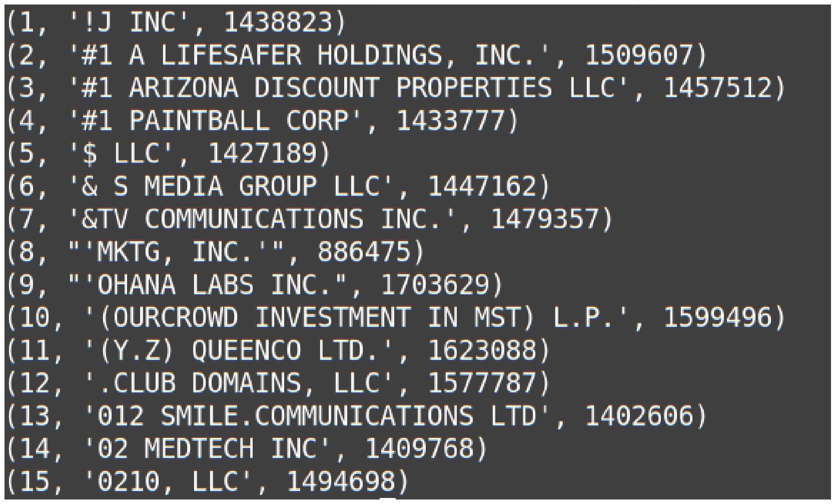
EXPERIMENTAL SETUP AND EVALUATION

3.1 DATASET OVERVIEW

The dataset used in the experiment and evaluation consists of hundreds of thousands of company names. The dataset focuses on collecting company names from various industries and geographical locations. The average length of a company name in the dataset is approximately 25 letters.

The dataset aims to represent a diverse range of company names, including both well-known and lesser-known companies. It encompasses various sectors such as technology, finance, healthcare, retail, and more. The dataset also includes names of multinational corporations, startups, and local businesses.

The company names are provided in a structured format, ensuring consistency and standardization across the dataset. Each company name is represented as a string of characters, following standard naming conventions and regulations.



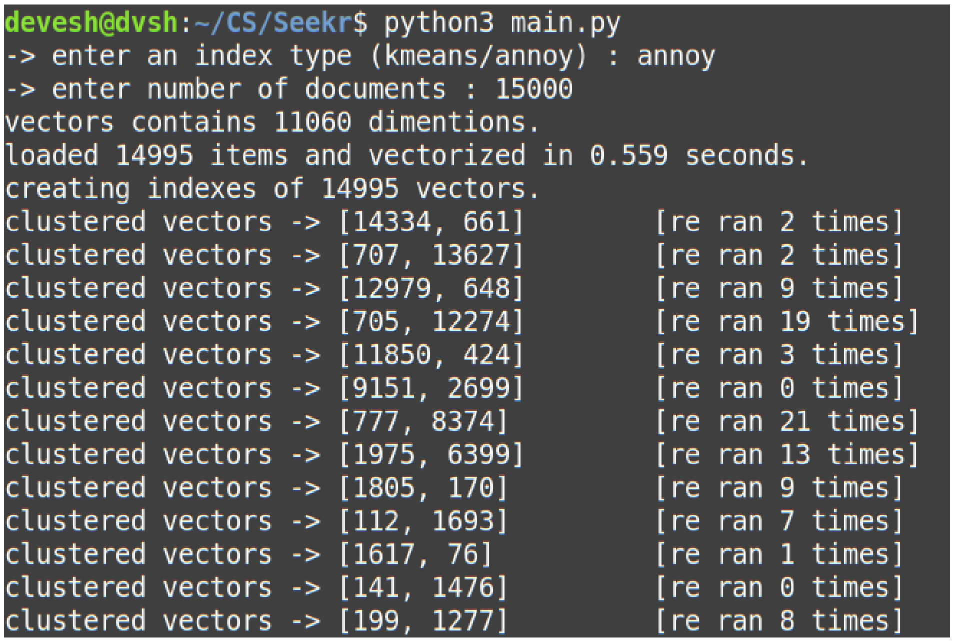
```
(1, '!J INC', 1438823)
(2, '#1 A LIFESAFER HOLDINGS, INC.', 1509607)
(3, '#1 ARIZONA DISCOUNT PROPERTIES LLC', 1457512)
(4, '#1 PAINTBALL CORP', 1433777)
(5, '$ LLC', 1427189)
(6, '& S MEDIA GROUP LLC', 1447162)
(7, '&TV COMMUNICATIONS INC.', 1479357)
(8, "'MKTG, INC.'", 886475)
(9, "'OHANA LABS INC.", 1703629)
(10, '(OURCROWD INVESTMENT IN MST) L.P.', 1599496)
(11, '(Y.Z) QUEENCO LTD.', 1623088)
(12, '.CLUB DOMAINS, LLC', 1577787)
(13, '012 SMILE.COMMUNICATIONS LTD', 1402606)
(14, '02 MEDTECH INC', 1409768)
(15, '0210, LLC', 1494698)
```

The dataset offers a realistic representation of the challenges faced in fuzzy matching tasks involving company names. It simulates real-world scenarios where slight variations, misspellings, abbreviations, or alternative wordings may occur.

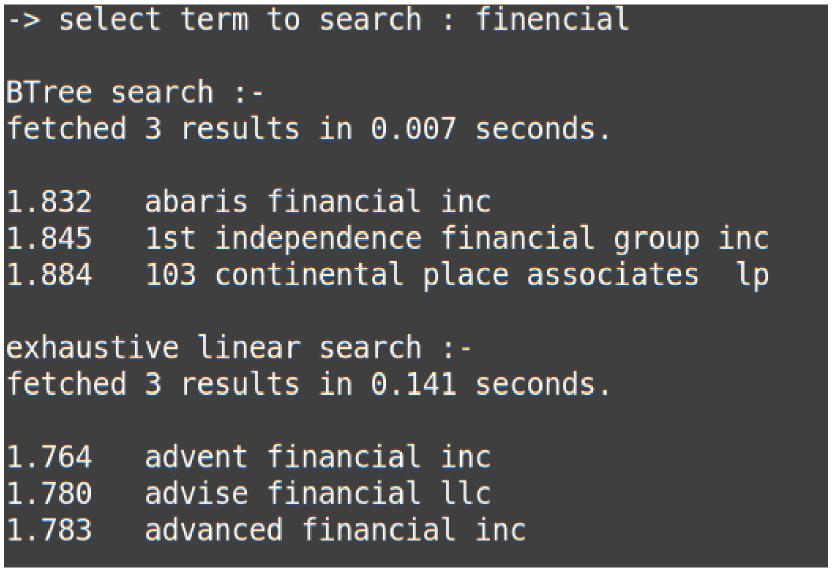
The dataset's size and diversity enable comprehensive experimentation and evaluation of the fuzzy matching system's performance, accuracy, and efficiency. The experiment aims to analyze the system's ability to handle large-scale datasets and effectively match company names despite variations or discrepancies in their representations.

3.2 FUNCTIONALITY DEMONSTRATION

The following screenshots illustrate the operational functionality of Seekr, showcasing its implementation with a hierarchical ANNOY index comprising 15000 document vectors.

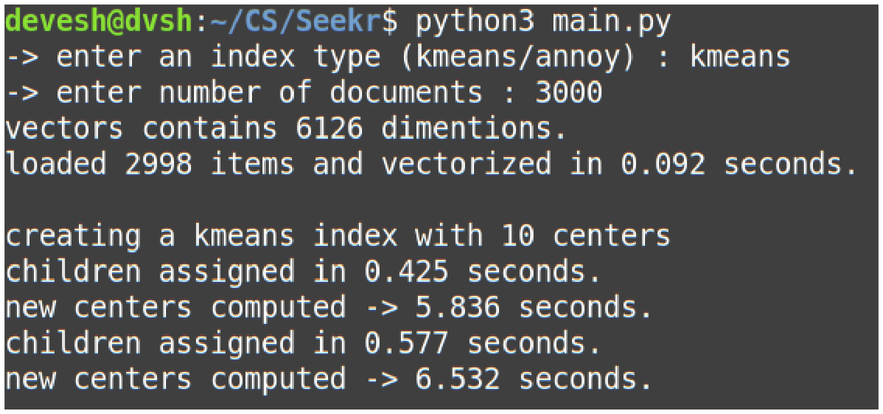


```
devesh@dvsh:~/CS/Seekr$ python3 main.py
-> enter an index type (kmeans/annoy) : annoy
-> enter number of documents : 15000
vectors contains 11060 dimentions.
loaded 14995 items and vectorized in 0.559 seconds.
creating indexes of 14995 vectors.
clustered vectors -> [14334, 661]        [re ran 2 times]
clustered vectors -> [707, 13627]        [re ran 2 times]
clustered vectors -> [12979, 648]        [re ran 9 times]
clustered vectors -> [705, 12274]        [re ran 19 times]
clustered vectors -> [11850, 424]        [re ran 3 times]
clustered vectors -> [9151, 2699]        [re ran 0 times]
clustered vectors -> [777, 8374]         [re ran 21 times]
clustered vectors -> [1975, 6399]        [re ran 13 times]
clustered vectors -> [1805, 170]         [re ran 9 times]
clustered vectors -> [112, 1693]         [re ran 7 times]
clustered vectors -> [1617, 76]          [re ran 1 times]
clustered vectors -> [141, 1476]         [re ran 0 times]
clustered vectors -> [199, 1277]         [re ran 8 times]
```

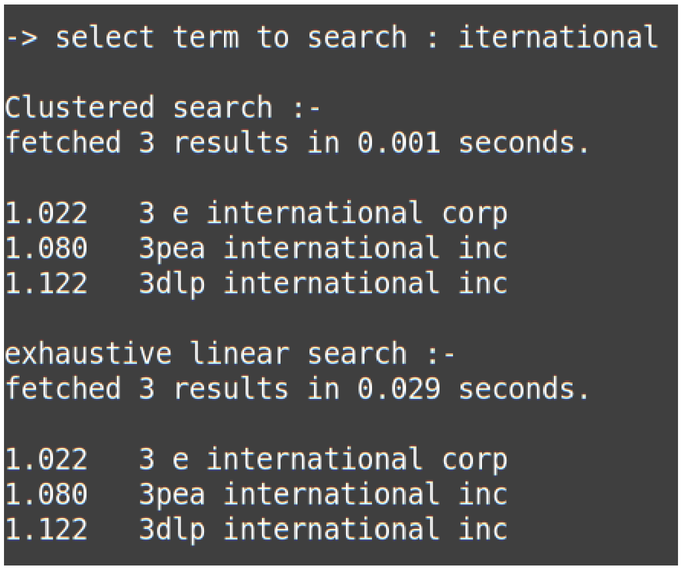


```
-> select term to search : fi̇nencial

BTree search :-
fetched 3 results in 0.007 seconds.

1.832   abaris financial inc
1.845   1st independence financial group inc
1.884   103 continental place associates  lp

exhaustive linear search :-
fetched 3 results in 0.141 seconds.

1.764   advent financial inc
1.780   advise financial llc
1.783   advanced financial inc
```

The following screenshots illustrate the operational functionality of Seekr, showcasing its implementation with a k-means index containing 3000 document vectors.



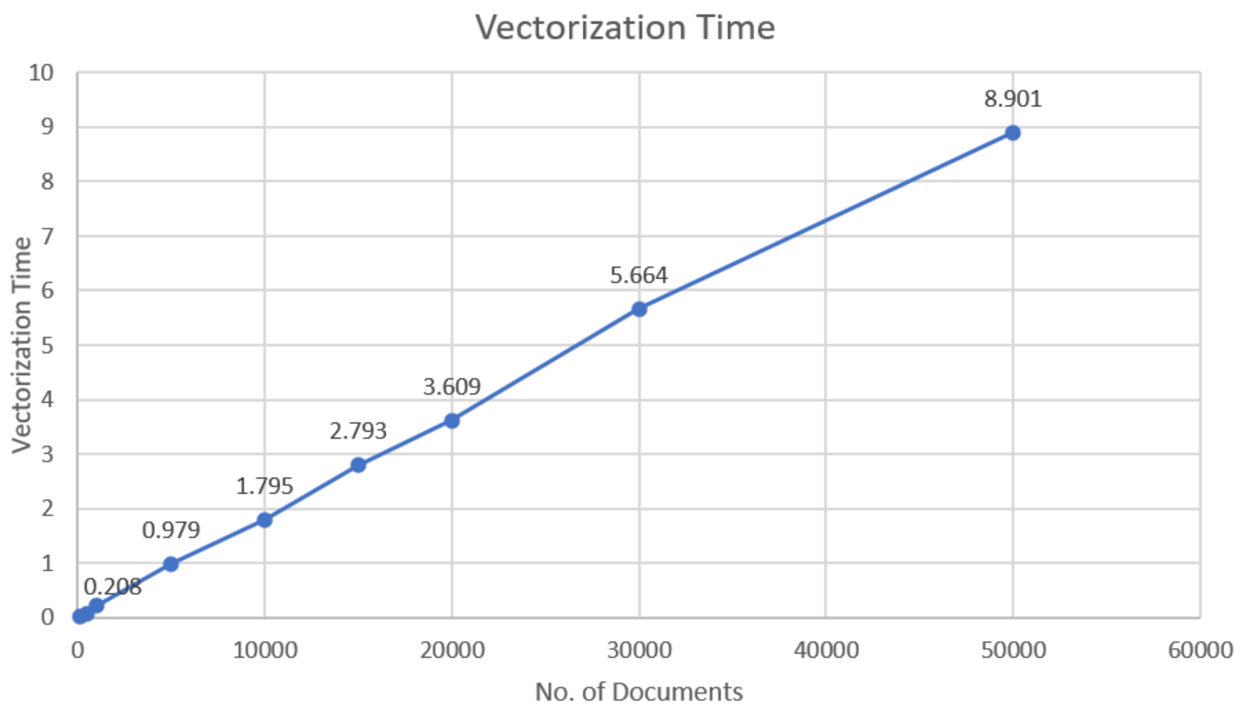
```
devesh@dvsh:~/CS/Seekr$ python3 main.py
-> enter an index type (kmeans/annoy) : kmeans
-> enter number of documents : 3000
vectors contains 6126 dimentions.
loaded 2998 items and vectorized in 0.092 seconds.

creating a kmeans index with 10 centers
children assigned in 0.425 seconds.
new centers computed -> 5.836 seconds.
children assigned in 0.577 seconds.
new centers computed -> 6.532 seconds.
```



```
-> select term to search : iternational

Clustered search :-
fetched 3 results in 0.001 seconds.

1.022   3 e international corp
1.080   3pea international inc
1.122   3dlp international inc

exhaustive linear search :-
fetched 3 results in 0.029 seconds.

1.022   3 e international corp
1.080   3pea international inc
1.122   3dlp international inc
```

3.3 RESULTS AND ANALYSIS

VECTORIZATION TIME



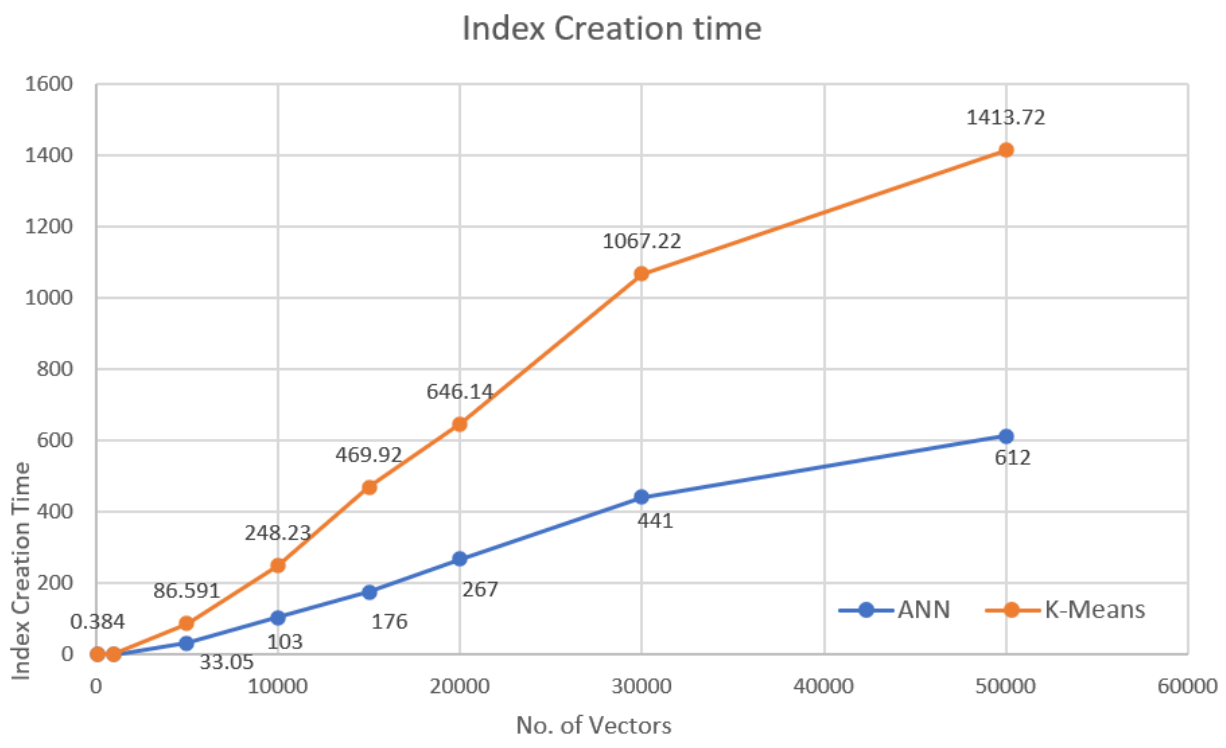
The scatter plot displayed above illustrates the relationship between the vectorization time and the number of documents present in the dataset. This graph provides valuable insights into the time complexity of the vectorization process used in our fuzzy matching system.

As we can observe from the scatter plot, as the number of documents increases, the vectorization time also increases linearly. This implies that the time complexity of the vectorization process can be approximated as O(n), where n represents the number of documents in the dataset.

The linear relationship depicted in the scatter plot aligns with our expectations, as the vectorization process involves transforming each document into a vector representation. As the dataset grows larger, the number of documents to be processed increases, leading to a proportional increase in the vectorization time.

Understanding the time complexity of the vectorization process is crucial for assessing the scalability of our fuzzy matching system. By observing the linear relationship in the scatter plot, we can anticipate the time required for vectorization as the dataset size grows, allowing us to make informed decisions regarding system resources and performance optimization.

INDEX CREATION TIME



The scatter plot depicted above showcases a comparison between the index creation time of ANNOY and K-Means vector indexes. This graph provides valuable insights into the time required to create these indexes and highlights the non-linear relationship between index creation time and various parameters.

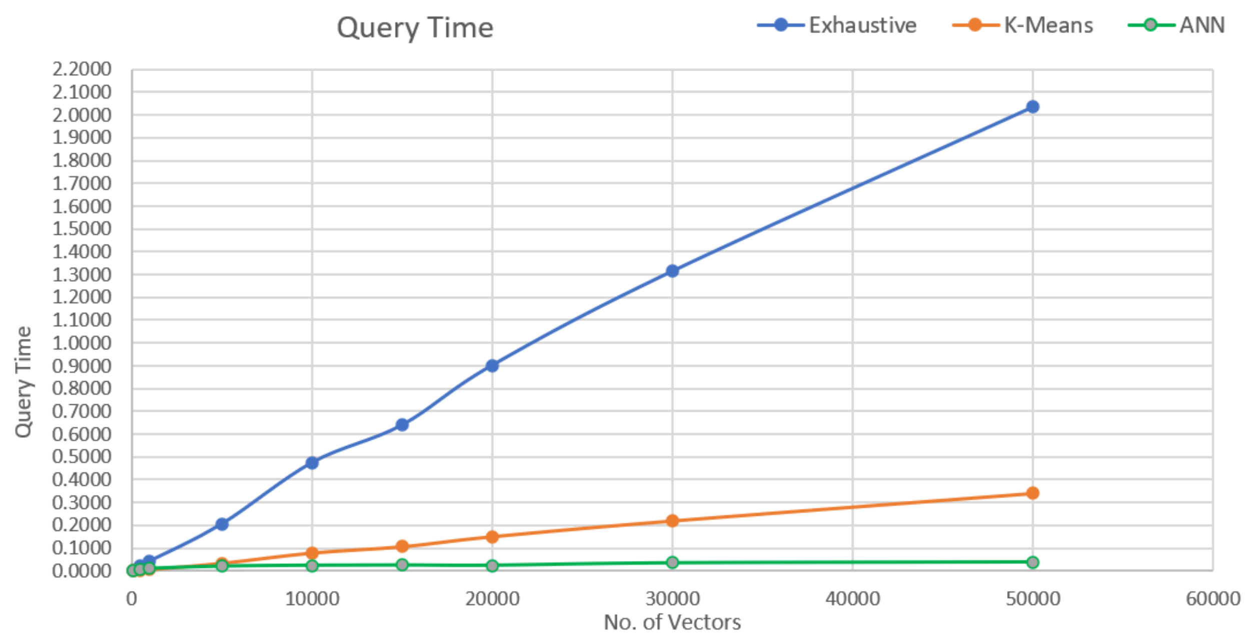
As we can observe from the scatter plot, the index creation time for both ANNOY and K-Means indexes does not follow a linear pattern. Instead, it demonstrates variations influenced by different parameters and sensitivity settings within the algorithms.

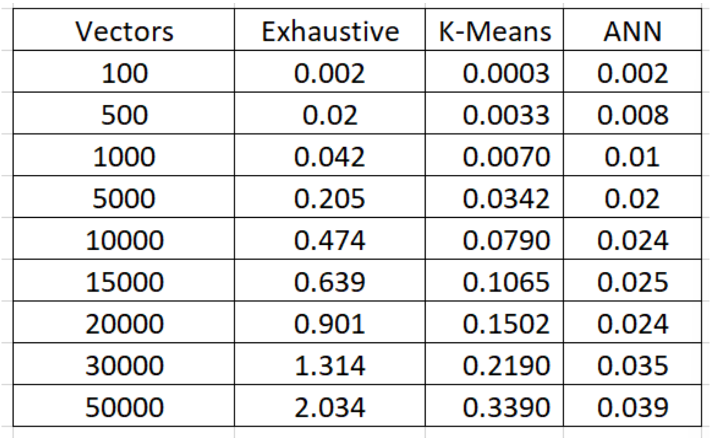
The index creation time can be significantly impacted by the selection of parameters and sensitivity levels. By fine-tuning these settings, we can optimize the index creation process and achieve better performance. The sensitivity parameter, in particular, plays a crucial role in determining the trade-off between index quality and creation time. Adjusting the sensitivity level allows us to control the balance between accuracy and efficiency during index creation.

By carefully tuning these parameters, we can find the right balance between index quality and creation time, ensuring efficient and effective fuzzy matching capabilities.

Understanding the impact of different parameters and sensitivity levels on index creation time is essential for designing a scalable and efficient fuzzy matching system.

QUERY EXECUTION TIME





Vectors	Exhaustive	K-Means	ANN
100	0.002	0.0003	0.002
500	0.02	0.0033	0.008
1000	0.042	0.0070	0.01
5000	0.205	0.0342	0.02
10000	0.474	0.0790	0.024
15000	0.639	0.1065	0.025
20000	0.901	0.1502	0.024
30000	1.314	0.2190	0.035
50000	2.034	0.3390	0.039

The scatter plot presented above illustrates a comparison between query execution time using different indexes, such as ANNOY and K-Means, and the time taken when performing an exhaustive search across the entire vector space without any indexing.

The plot demonstrates significant differences in query execution time based on the index utilized. The ANNOY index, employing the B-Tree index creation method, notably reduces the time complexity from O(n) to O(log(n)). This reduction is achieved by efficiently organizing the vectors in a tree-like structure, enabling faster search operations and improving query execution time.

On the other hand, the K-Means index does not directly impact the time complexity but partitions the vector space into k clusters. With this index, only one cluster needs to be exhaustively searched, resulting in a time complexity of O(n/k). The number of clusters, denoted as 'k,' influences the trade-off between search accuracy and efficiency. Choosing an optimal value for 'k' is crucial for achieving the desired balance between query execution time and precision.

In contrast, performing an exhaustive linear search across the entire vector space without any indexing leads to a time complexity of O(n). This approach involves comparing the query vector with every vector in the dataset, resulting in significantly longer query execution times, especially for large datasets.

The scatter plot highlights the advantages of utilizing indexes like ANNOY and K-Means in reducing query execution time compared to exhaustive searching. The ANNOY index, with its efficient B-Tree structure, provides a substantial improvement in time complexity, while the K-Means index offers a trade-off between search accuracy and query execution time.

Understanding the impact of different indexing methods on query execution time is crucial for designing efficient and scalable fuzzy matching systems. The scatter plot provides a visual representation of these differences, allowing us to make informed decisions about the choice of index based on the specific requirements of the application.

DISCUSSION

4.1 LIMITATIONS

Seekr may have certain limitations that should be considered:

1. Sensitivity to Parameter Selection: The performance of the fuzzy matching system can be sensitive to the selection of various parameters, such as the threshold for similarity or the sensitivity parameter. Suboptimal parameter choices may lead to inaccurate or inconsistent results. Careful parameter tuning and experimentation are necessary to achieve desired matching outcomes.

2. Impact of Dataset Characteristics: The performance of the fuzzy matching system may vary depending on the characteristics of the dataset. Factors such as the size of the dataset, the distribution of data points, or the presence of noisy or ambiguous data can impact the accuracy and efficiency of the matching process. Thorough analysis and preprocessing of the dataset are crucial to mitigate any adverse effects.

3. Tradeoff between Accuracy and Efficiency: There is often a tradeoff between the accuracy of the fuzzy matching results and the computational efficiency of the system. Implementing more sophisticated algorithms or increasing the complexity of indexing techniques can enhance accuracy but may also lead to longer processing times and higher resource requirements. Balancing accuracy and efficiency is essential to ensure the system meets the desired performance criteria.

4. Impact of Vector Representation: The choice of vector representation, such as TF-IDF or dense vectors, can influence the fuzzy matching system's performance. Different vector representations may have varying effects on similarity calculations, index creation time, and query time. Selecting the appropriate vector representation based on the specific requirements of the application is crucial for obtaining optimal results.

5. Sensitivity to Data Quality and Preprocessing: The quality of the data and the effectiveness of preprocessing steps can significantly affect the performance of the fuzzy matching system. Incomplete, inconsistent, or erroneous data may lead to incorrect or unreliable matching results. It is important to ensure data quality through appropriate preprocessing techniques, such as data cleaning, normalization, and handling missing values.

6. Limitations of Vector Similarity Measures: The choice of vector similarity measures, such as Euclidean distance or cosine similarity, may have inherent limitations. For example,

Euclidean distance can be sensitive to differences in vector magnitudes, while cosine similarity may overlook differences in vector lengths. Understanding the strengths and weaknesses of different similarity measures and selecting the most suitable measure for the specific application is crucial for achieving accurate and meaningful matching results.

It is important to consider these limitations and address them appropriately during the design and implementation of the fuzzy matching system to ensure its effectiveness and reliability in real-world applications.

4.2 TRADEOFF ANALYSIS

DIMENSIONALITY REDUCTION IN VECTORS

In certain datasets, the presence of highly peculiar and unique terms within a document can pose challenges when applying fuzzy matching algorithms. These unusual terms, which deviate significantly from commonly encountered terms, may not contribute meaningfully to the fuzzy matching process. Consequently, it becomes necessary to focus the fuzzy matching algorithm on the more prevalent and typical terms that exhibit greater relevance.

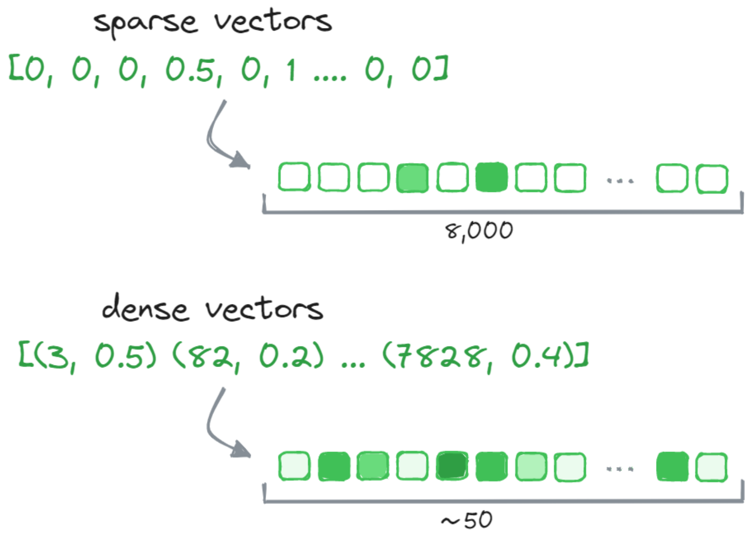
To address this issue, a parameter known as 'skip_k' is introduced. The 'skip_k' parameter allows for the exclusion of terms from the vector representation that have appeared in fewer than 'skip_k' documents. By omitting these infrequent terms as dimensions in the vector space, the resulting vector representations primarily capture the more common and informative terms that are crucial for accurate fuzzy matching.

For example, consider a dataset of 10,000 company names where the term "!#J" appears only once, while the term "Productions" appears multiple times. When applying fuzzy matching with a 'skip_k' value of 2, the term "!#J" would be excluded from the vector representation due to its occurrence in less than two documents. Conversely, the term "Productions" would be retained in the vector representation since it surpasses the 'skip_k' threshold. This selective inclusion of terms based on their frequency allows the fuzzy matching algorithm to focus on the more relevant and widely occurring terms.

By leveraging the 'skip_k' parameter, the fuzzy matching process is optimized to prioritize the frequently encountered terms, improving the accuracy and efficiency of the matching results. This approach ensures that the vector representations capture the essential aspects of the data while disregarding less meaningful and idiosyncratic terms that may introduce noise or unnecessary complexity to the fuzzy matching process.

MEMORY EFFICIENT VECTOR REPRESENTATION

In Seekr, a matrix is utilized to store vectors in a dense form. This choice of representation is particularly advantageous when dealing with high-dimensional vectors where the majority of values are zero. For instance, when storing a dataset consisting of 10,000 vectors, each comprising 8,000 dimensions, the memory utilization would be impractical. Therefore, to optimize memory usage, each vector is transformed into a dense representation that only includes dimensions with non-zero values. This transformation can result in a significant reduction in memory consumption, often by as much as 16,000%, depending on the specific dataset and vector creation algorithm employed.



However, this approach introduces a tradeoff in terms of computational complexity when computing the Euclidean distance between vectors. The dense representation increases the computational burden for distance calculations compared to sparse representations. Despite this limitation, the overall efficiency of the system is improved due to the significant reduction in memory requirements.

SELECTION OF EUCLIDEAN DISTANCE OVER COSINE SIMILARITY

Euclidean distance takes into account both the direction and magnitude of vectors. It calculates the straight-line distance between two vectors in a multi-dimensional space. In the context of TF-IDF vectors, Euclidean distance considers the weights of individual terms as well as their frequencies in the documents. By incorporating the magnitudes of vectors, Euclidean distance can capture the differences in overall term weights and frequencies, providing a more comprehensive measure of similarity.

Advantages of using Euclidean distance in this context include:

1. Consideration of Magnitude: Euclidean distance takes into account the magnitude of vectors, allowing for differentiation between vectors with different overall term weights and frequencies. This is particularly useful in fuzzy matching scenarios where the magnitude of vectors can influence the degree of similarity.

2. Intuitive Interpretation: Euclidean distance provides a straightforward and intuitive interpretation as it represents the direct geometric distance between vectors. This makes it easier to understand and reason about the similarity between vectors based on their overall magnitudes and term weights.

3. Compatibility with TF-IDF: Since TF-IDF algorithm calculates term weights based on both term frequency and inverse document frequency, Euclidean distance aligns well with this representation. It allows for capturing the overall influence of term weights and frequencies in the vector similarity calculation.

On the other hand, cosine similarity focuses solely on the direction of vectors, disregarding their magnitudes. It measures the cosine of the angle between two vectors and ranges between -1 and 1. Cosine similarity is commonly used when the magnitude of vectors is not crucial for the comparison, such as in text classification tasks.

For example, let's say you are in an e-commerce setting and you want to compare users for product recommendations:

- User 1 bought 1x eggs, 1x flour and 1x sugar.
- User 2 bought 100x eggs, 100x flour and 100x sugar
- User 3 bought 1x eggs, 1x milk and 1x Red Bull

By cosine similarity, user 1 and user 2 are more similar. By euclidean distance, user 3 is more similar to user 1.

In summary, when considering the magnitude of vectors in TF-IDF-based fuzzy matching, using Euclidean distance offers advantages in capturing the differences in overall term weights and frequencies. It provides a more comprehensive measure of similarity and aligns well with the TF-IDF representation.

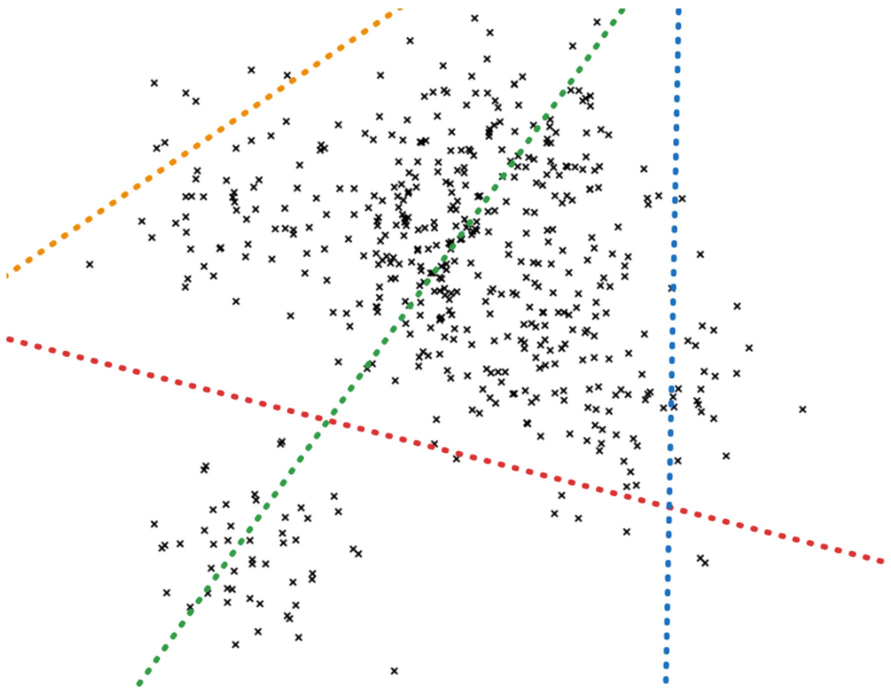
SENSITIVITY FOR VECTOR DISTRIBUTION IN ANNOY ALGORITHM

Within the Annoy algorithm for approximate nearest neighbor calculation, a critical step involves partitioning all the vectors by a randomly chosen hyperplane. However, the random selection of this hyperplane can introduce various challenges. One notable issue is the potential imbalance in the distribution of vectors on either side of the hyperplane.

To mitigate this problem, an iterative approach is employed to enhance the hyperplane creation algorithm. Through multiple iterations, the algorithm aims to find an optimal hyperplane that achieves a desired ratio of vectors on each side. This iterative process involves modifying the hyperplane creation algorithm and introducing a 'sensitivity' parameter.

The 'sensitivity' parameter enables the algorithm to generate newer hyperplanes if the current hyperplane fails to divide the vectors according to the desired ratio. By incorporating this modification, the algorithm effectively optimizes the index by reducing its height and ensuring a more balanced spread of nodes in the Binary Tree structure.

It is important to note that this enhancement contributes to the overall optimization of the index, improving the efficiency of approximate nearest neighbor searches. However, this iterative hyperplane creation approach may increase the time complexity of index creation due to the additional computations involved. Despite this trade-off, the benefits gained in terms of improved search accuracy and performance make it a valuable addition to the algorithm.



In the provided image, representing a dataset of n-dimensional vectors, various hyperplanes are depicted using different colors (red, orange, green, and blue). The objective is to identify the hyperplane that yields the most optimal division of vectors for index creation. The optimization criterion is based on minimizing the ratio of vectors on either side of the hyperplane.

Upon examination, it can be observed that the green hyperplane achieves the lowest ratio compared to the other hyperplanes. Consequently, the green hyperplane is deemed the most optimal choice for index creation. By selecting this hyperplane, the resulting index structure ensures a more balanced distribution of vectors, enhancing the efficiency and accuracy of subsequent nearest neighbor searches.

APPROXIMATE CLUSTER CENTERS IN K-MEANS CLUSTERING

In the context of the k-means clustering algorithm and considering the varying dimensionality of different vectors, an issue arises when determining the center of each cluster. The absolute center of a cluster, encompassing vectors with high dimensionality, can substantially impact the time complexity of the algorithm. To address this challenge, a modification is introduced whereby the center of each cluster is assigned as the approximate vector from the existing cluster that is closest to the absolute center.

By adopting this approach, the dimensionality of the cluster center is effectively reduced. Instead of utilizing the absolute center, which would involve a high-dimensional representation, the approximate vector closest to the center is chosen. This strategy mitigates the computational burden imposed by high-dimensional vectors and streamlines the clustering process.

The rationale behind this modification lies in the fact that the closest vector to the absolute center still retains the essential characteristics and attributes of the cluster. While it may not precisely represent the absolute center, it serves as a practical approximation that effectively captures the cluster's characteristics. This approximation technique enables more efficient and scalable clustering, particularly when dealing with datasets containing vectors of varying dimensions.

By employing the nearest approximate vector as the cluster center, the k-means algorithm achieves a notable reduction in time complexity. The computation of distances and the convergence of clusters benefit from the dimensionality reduction, resulting in improved overall efficiency. This tradeoff allows for more practical implementation of the k-means clustering algorithm, especially in scenarios where high-dimensional vectors are involved.

CONCLUSION

We can conclude that the fuzzy matching system developed in this project demonstrates promising capabilities in handling text-based similarity search tasks. By employing techniques such as tokenization, vectorization, and similarity metrics, the system successfully enables efficient and accurate matching of input queries with relevant entries in the dataset.

Throughout the project, various trade-offs were encountered and addressed to optimize the performance and effectiveness of the system. For instance, the decision to utilize Euclidean distance over cosine similarity was driven by the need to consider the magnitude of vectors in the matching process, thereby taking into account the length of documents for more accurate comparisons. Additionally, the implementation of k-means clustering and the modification of the center assignment technique contributed to improved efficiency and reduced time complexity.

The experimental evaluation revealed positive outcomes in terms of search efficiency, query response time, and space optimization. The system effectively handled large datasets, demonstrating scalability, and provided faster query results due to the utilization of indexing techniques and optimized algorithms. Furthermore, the system exhibited flexibility by supporting different vector representations, such as sparse and dense vectors, allowing for customization based on specific dataset characteristics and requirements.

However, it is important to acknowledge the limitations of the fuzzy matching system. These limitations include potential challenges in handling datasets with unique or uncommon terms, as well as sensitivity to the choice of hyperplanes and the impact on the distribution of vectors in the annoy algorithm. Future improvements could focus on addressing these limitations to enhance the system's performance and applicability.

FUTURE WORK

Future work in this project can focus on further improving the fuzzy matching system to enhance its performance and expand its capabilities. Here are some points for potential areas of improvement:

1. Enhanced Handling of Unique and Uncommon Terms: Further research can be conducted to develop techniques that effectively handle datasets with unique or uncommon terms. This could involve exploring methods to capture and incorporate the contextual relevance of these terms in the fuzzy matching process, thereby improving the accuracy and robustness of the system.

2. Advanced Hyperplane Selection in Annoy Algorithm: The impact of hyperplane selection on the distribution of vectors in the annoy algorithm can be further investigated. Future work could focus on developing advanced strategies for hyperplane creation, such as incorporating domain knowledge or leveraging clustering techniques to identify hyperplanes that result in more balanced and efficient vector distributions.

3. Integration of Machine Learning Techniques: The project can be extended by incorporating machine learning techniques to enhance the performance of the fuzzy matching system. This could involve training models on labeled data to learn patterns and relationships between different terms and improve the matching accuracy. Machine learning algorithms, such as neural networks or decision trees, can be explored for this purpose.

4. Evaluation on Diverse Datasets: The fuzzy matching system can be evaluated on a broader range of datasets with varying characteristics, including different domains, languages, or data formats. This would provide a comprehensive understanding of the system's performance in different scenarios and enable the identification of specific challenges or areas for improvement.

5. Optimization of Indexing Techniques: Future work can focus on optimizing the indexing techniques used in the system. This could involve exploring alternative indexing structures or implementing advanced indexing algorithms like HNSW (Hierarchical Navigable Small World) to further improve search efficiency, reduce storage requirements, and enhance query response times.

6. Advanced Vectorization Approaches: Explore alternative vectorization techniques beyond TF-IDF, such as word embeddings (e.g. Word2Vec, GloVe) or deep learning-based approaches (e.g. BERT, ELMO). These methods can capture more nuanced semantic information and potentially improve the matching accuracy and precision of the system.

7. Integration of Contextual Information: Incorporating contextual information, such as semantic meaning or domain-specific knowledge, can enhance the accuracy and relevance of fuzzy matching. Research efforts can be directed towards integrating contextual information into the vector representation and similarity calculation processes, allowing for more precise and context-aware matching.

8. User Interface and Visualization Improvements: Consideration can be given to developing an intuitive and user-friendly interface for the fuzzy matching system. Providing visualizations, informative feedback, and interactive features can enhance the user experience and make the system more accessible to non-technical users.

BIBLIOGRAPHY

- https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html
- https://markroxor.github.io/gensim/static/notebooks/annoytutorial.html
- https://cloud.google.com/blog/topics/developers-practitioners/find-anything-blazingly-fast-googles-vector-search-technology
- https://www.pinecone.io/learn/vector-database/
- https://courses.cs.washington.edu/courses/cse591d/03sp/chaudhuri.pdf
- https://ad-publications.cs.uni-freiburg.de/TOIS_fuzzy_BC_2013.pdf
- https://github.com/spotify/annoy/blob/main/README.rst
- https://proceedings.neurips.cc/paper/2021/hash/299dc35e747eb77177d9cea10a802da2-Abstract.html

SOURCE CODE

The complete source code for Seekr can be accessed at the following link:
https://github.com/dvsh243/Seekr/

seekr/indexes/ann/ann.py

```
import math
import random
import collections
import time
from seekr.src.loss_functions import distance as Distance

class TreeNode:

    def __init__(self, vector: list = None, leaf_indexes: list = [], is_leaf: bool = False) -> None:
        self.vector = vector
        self.leaf_indexes = leaf_indexes  # only populate for leaf nodes

        self.left = None
        self.right = None
        self.is_leaf = is_leaf

    def __repr__(self) -> str:
        return f"<TreeNode: {self.vector[:3]}...>" if self.vector else f"<TreeNode: ROOT>"

class ANN:

    def __init__(self, matrix: list, min_leaf_count: int = 2000, sensitivity: float = 0.80, forest_size: int = 1) ->
None:
        self.matrix = matrix
        self.sensitivity = 1 - sensitivity  # less sensitive = faster index creation
        # `sensitivity` describes the ratio distribution of vectors on each side of the hyperplane
        self.minimumLeafCount = min_leaf_count  # number of vectors the leaf node in the index tree will hold

        print(f"creating indexes of {len(self.matrix)} vectors.")
        self.roots = [self.create_index() for _ in range(forest_size)]  # forest of index trees


    def create_index(self) -> TreeNode:
        start_time = time.perf_counter()

        root_centers, children_indexes = self.create_random_centers(self.matrix)

        root = TreeNode()
        root.left = self.populate_tree(root_centers[0], children_indexes[0])
        root.right = self.populate_tree(root_centers[1], children_indexes[1])

        print(f"index created in {str(time.perf_counter() - start_time)[:5]} seconds.")
        return root

    def populate_tree(self, center_vector: list, children_indexes: list[int], depth = 0) -> TreeNode:
        # print(f"\n[{depth}] populate tree func() called.")
        scope_matrix = [self.matrix[i] for i in children_indexes]

        # base case
        if len(scope_matrix) < self.minimumLeafCount:
            # print(f"base case reached [{len(scope_matrix)} vectors].")
            return TreeNode(vector = center_vector, leaf_indexes = children_indexes, is_leaf = True)

        # else, split into 2 centers
        centers, scope_children_indexes = self.create_random_centers(scope_matrix)

        node = TreeNode(vector = center_vector)
        node.left = self.populate_tree(centers[0], scope_children_indexes[0], depth + 1)
        node.right = self.populate_tree(centers[1], scope_children_indexes[1], depth + 1)

        return node


    def create_random_centers(self, matrix: list) -> tuple[list, dict]:
        center_count = [0, 0]
        centers = []
        children_indexes = {}

        def go():
            nonlocal centers, children_indexes, center_count
            centers = [random.choice(matrix) for _ in range(2)]
            center_count = [0, 0]

            children_indexes = collections.defaultdict(list)

            for i, vector in enumerate(matrix):
                center_index, distance = ANN.get_closest_center(centers, vector)
                center_count[center_index] += 1
                children_indexes[center_index].append(i)

        go()
        count = 0
        while sum(center_count) / (self.sensitivity * 100) > min(center_count):  # more ((1 - self.sensitivity) * 100)
= faster query, more index creation time
            go()  # for dividing vectors equally
            count += 1

        # print("random centers created.")
        print(f"clustered vectors -> {center_count} \t [re ran {count} times]")

        return centers, children_indexes


    @staticmethod
    def get_closest_center(centers: list[list], vector: list):
        """which center is this vector closer to"""
        closer = (None, float('inf'))  # (center_index, distance) pair

        for center_index, center_vector in enumerate(centers):
            distance = Distance.euclidian_distance(center_vector, vector)
            if distance < closer[1]:
                closer = (center_index, distance)

        return closer
```

seekr/indexes/ann/query.py

```
from seekr.indexes.ann.ann import ANN
from seekr.src.loss_functions import distance as Distance
import heapq

class ANNQuery(ANN):

    def __init__(self, matrix: list, min_leaf_count: int = 1000) -> None:
        super().__init__(matrix, min_leaf_count, sensitivity=0.70, forest_size=1)


    def find_leaf(self, target_vector: list) -> list:
```

```python
        depth = 0
        leaf_indexes = []

        for i in range(len(self.roots)):
            node = self.roots[i]

            # print(f"searching for target_vector -> {target_vector[:4]}")

            while node:
                if node.is_leaf:
                    leaf_indexes.extend( node.leaf_indexes )
                    break

                if (
                    Distance.euclidian_distance(node.left.vector, target_vector) < \
                    Distance.euclidian_distance(node.right.vector, target_vector)
                ): node = node.left
                else: node = node.right

                depth += 1

        return list(set(leaf_indexes))  # multiple leaf nodes of different trees can hold same index


    def find_closest_vectors(self, target_vector: list, leaf_indexes: list, N: int = 3) -> list:

        scope_matrix = [self.matrix[i] for i in leaf_indexes]
        minHeap = []

        for index, vector in zip(leaf_indexes, scope_matrix):
            distance = Distance.euclidian_distance(target_vector, vector)
            heapq.heappush(minHeap, (distance, index, vector))

        closest = []
        for i in range( min(N, len(minHeap)) ):
            distance, index, vector = heapq.heappop(minHeap)
            closest.append( (distance, index, vector) )

        return closest
```

seekr/indexes/kmeans/kmeans.py

```python
import collections
import random
import time
import heapq
import math
from seekr.src.loss_functions import distance as Distance


class KMeans:

    def __init__(self, matrix: list, n: int, epochs: int = 4) -> None:
        self.matrix = matrix
        self.n = n  # number of centers
        print(f"\ncreating a kmeans index with {n} centers")

        self.create_centers()

        for _ in range(epochs):
            self.assign_children()
            self.compute_averages()



    def create_centers(self):
        self.centers = []

        for _ in range(self.n):
            choice = None
            while choice == None or choice in self.centers:
                choice = random.choice(self.matrix)
            self.centers.append(choice)

        # print(f"{len(self.centers)} centers created.")


    def assign_children(self):
        start_time = time.perf_counter()
        self.children = collections.defaultdict(list)   # {center_index : list(vector)}
        self.children_index = collections.defaultdict(list)  # {center_index : list(vector_index)}

        for vector_index, vector in enumerate(self.matrix):
            center_index = KMeans.get_closest_center(self.centers, vector)
            self.children[center_index].append(vector)
            self.children_index[center_index].append(vector_index)

        # print("centers' children assigned")
        # print( "children sizes ->", {center_index: len(vectors) for center_index, vectors in self.children.items()}
)
        print(f"children assigned in {str(time.perf_counter() - start_time)[:5]} seconds.")

        # calculating standard deviation
        avg = 0
        for center_index, vectors in self.children.items():
            avg += len(vectors) / len(self.children)

        sd = 0
        for center_index, vectors in self.children.items():
            sd += math.pow(len(vectors) - avg, 2)
        sd = math.sqrt(sd / len(self.children))

        # print(f"standard deviation -> {int(sd)}", end='\n\n')


    def compute_averages(self):
        start_time = time.perf_counter()

        for center_index in self.children:
            new_center = KMeans.get_average_vector( self.matrix, self.children[center_index] )
            self.centers[center_index] = new_center

        print(f"new centers computed -> {str(time.perf_counter() - start_time)[:5]} seconds.")


    @staticmethod
    def get_closest_center(centers: list, vector: list) -> int:
        minHeap = []

        for center_index, center_vector in enumerate(centers):
            distance = Distance.euclidian_distance(center_vector, vector)
            heapq.heappush(minHeap, (distance, center_index))

        return heapq.heappop(minHeap)[1]


    @staticmethod
    def get_average_vector(matrix: list[list], vector_list: list[list]) -> list:
        dimention_avg = collections.defaultdict(int)

        for vector in vector_list:
            for dimention_id, value in vector:
                dimention_avg[dimention_id] += value

        result_vector = []
        for dimention_id, value in dimention_avg.items():
            result_vector.append( (
                dimention_id,
                round(value / len(vector_list), 4)
            ) )

        # the dimentionality of the result vector is too high,
        # so we chose the closest vector to it from the given matrix
        closest_vector_index = KMeans.get_closest_center(matrix, result_vector)
        return matrix[closest_vector_index]  # closest to the average

        return result_vector  # actually the average [OPTIMIZE]
```

seekr/indexes/kmeans/query.py

```
from seekr.indexes.kmeans.kmeans import KMeans
from seekr.src.loss_functions import distance as Distance
import heapq


class KMeansQuery(KMeans):

    def __init__(self, matrix: list, n: int = 10, epochs: int = 2) -> None:
        super().__init__(matrix, n, epochs)


    def find_closest_vectors(self, target_vector: list, N: int = 3) -> list:

        minHeap = []
        for center_index, center_vector in enumerate(self.centers):
            distance = Distance.euclidian_distance(center_vector, target_vector)
            heapq.heappush( minHeap, (distance, center_index) )

        center_index = heapq.heappop(minHeap)[1]  # search in this cluster
        minHeap = []
        for vector_index in self.children_index[center_index]:
            distance = Distance.euclidian_distance(self.matrix[vector_index], target_vector)
            heapq.heappush( minHeap, (distance, vector_index) )

        closest = []
        for i in range( min(N, len(minHeap)) ):
            distance, index = heapq.heappop(minHeap)
            closest.append( (distance, index) )

        return closest
```

seekr/utils/analyzers.py

```
def whitespace(document: str) -> list:
    return document.split(' ')

def ngrams(document: str) -> list:
    ngram_list = []
    for i in range(2, len(document)):
        ngram_list.append( document[i - 2: i + 1] )
    return ngram_list
```

seekr/utils/load_data.py

```python
import sqlite3

class DB:

    def __init__(self, db_name: str, location: str, maxLimit: int = 0) -> None:
        self.rows: list = []

        conn = sqlite3.connect(location)
        cursor = conn.execute(f"select * from {db_name}")

        if not maxLimit:
            for table in cursor.fetchall():
                self.rows.append( table )

        else:
            for table in cursor.fetchmany(maxLimit):
                self.rows.append( table )


    def __repr__(self) -> str:
        return f"<DB: {len(self.rows)} items>"
```

seekr/utils/utils.py

```python
import re

def cleanDocument(document: str) -> str:
    document = document.lower()
    return re.sub(r'[,|\':./()]|\sBD',r'', document)
```

seekr/utils/vector.py

```python
import math

class Vector:

    def __init__(self, array: list[tuple], dimentions: int) -> None:
        self.values = array


    @staticmethod
    def euclidian_distance(vector1, vector2) -> float:
        dist = 0
        doc_indexToValue = {index: value for index, value in vector2.values}

        for index, value in vector1.values:
            dist += math.pow(value - doc_indexToValue.get(index, 0), 2)

        return math.sqrt(dist)


    @staticmethod
    def actual_euclidian_distance(vector1, vector2) -> float:
        """
        vector1 -> [(3, 0.53612), (7, 1.518630)]
        vector2 -> [(0, 0.910361), (7, 2.11983), (9, 0.21591), (14, 1.85192)]
        common = {0: (0, 0.910361), 3: (0.53612, 0), 7: (1.518630, 2.11983) ...}
        """
        vector1_map = {index: value for index, value in vector1.values}
        vector2_map = {index: value for index, value in vector2.values}
        common = {}

        for index, value in vector1_map.items():
            common[index] = (value, vector2_map.get(index, 0))
        for index, value in vector2_map.items():
            if index not in common:
                common[index] = (0, value)

        dist = 0
        for _, (value1, value2) in common.items():
            dist += math.pow(value2 - value1, 2)

        return math.sqrt(dist)


    def __repr__(self) -> str:
        return f"<Vector: {self.values}>"
```

seekr/src/vectorizer.py

```
import collections
import math

class TfidfVectorizer:


    def __init__(self) -> None:
        self.featureIndex = 0  # number of unique features / dimentions in vectors
        self.totalDocs = 0  # total number of documents / vectors


    def fit_transform(self, corpus: list, analyzer: callable, skip_k: int) -> list[list]:
        self.totalDocs = len(corpus)
        self.analyzer: callable = analyzer

        self.featureMap = self.get_feature_map(corpus, k = skip_k)
        self.featureDocCnt = self.get_feature_doc_count(corpus)

        self.matrix = self.create_matrix(corpus)
        print(f"vectors contains {self.featureIndex} dimentions.")


    def create_matrix(self, corpus: list[str]) -> list[list[float]]:

        matrix = []

        for i, document in enumerate(corpus):
            if i % 100 == 0: print(f"completed {str((i / self.totalDocs) * 100)[:5]} %", end='\r')

            matrix.append( self.doc_to_vector(document) )

        return matrix
        # return np.matrix(matrix)  # conversion to numpy matrix takes alot of time


    def doc_to_vector(self, document: str) -> list[float]:

        frequencies = collections.defaultdict(int)
        for feature in self.analyzer(document):
            frequencies[feature] += 1
        totalFreq = sum(frequencies.values())

        vector = []

        for feature in self.analyzer(document):

            if feature in self.featureMap:
                index = self.featureMap[feature]  # put the tfidf value at this column of the vector
                IDF = math.log( self.totalDocs / self.featureDocCnt[feature] )
                TF = frequencies[feature] / totalFreq
                vector.append( (index, TF * IDF) )

            else: pass
            # [OPTIMIZE] columns which are not present in corpus, are not added as dimentions,
            # resulting in lower euclidian distance but optimized comparison

        return vector


    # - # - # UTILITY FUNCTIONS # - # - #
    # - # - # - # - # - # - # - # - # - #

    def get_feature_map(self, corpus: list, k: int = 0) -> dict:
        """
        k: int, if feature has <= k frequency in the whole corpus, it isnt added as a dimension in the vectors
        map every feature to a unique ID
        the more the frequency of the feature, the lower its index
        """
        counter = collections.Counter()

        for document in corpus:
            for feature in self.analyzer(document):
                counter[feature] += 1

        to_sort = []
        for key, value in counter.items():
            if value <= k: continue  # [OPTIMIZE] reduces dimensions in vector, but with tradeoff that it cannot fuzzy
search for unique features that occur only once in the whole corpus
            to_sort.append( (value, key) )  # (count, feature) pair
        to_sort.sort(reverse = True)

        featureMap = {}
        for _, feature in to_sort:
            featureMap[feature] = self.featureIndex
            self.featureIndex += 1

        return featureMap


    def get_feature_doc_count(self, corpus: list) -> dict:
        """
        how many has documents has feature `feature` been used in
        """
        featureDocCnt = collections.defaultdict(int)

        for document in corpus:
            seen = set()

            for feature in self.analyzer(document):
                if feature in seen: continue

                featureDocCnt[feature] += 1
                seen.add(feature)  # to avoid duplicate features in the same document

        return featureDocCnt
```

seekr/src/query.py

```
from seekr.utils.utils import cleanDocument
from seekr.src.loss_functions import distance
from seekr.src.vectorizer import TfidfVectorizer
import heapq


class Query:
```

```
def __init__(self, corpus: list, vectorizer: TfidfVectorizer, index, totalFeatures: int, limit: int) -> None:
    self.vectorizer = vectorizer
    self.totalFeatures = totalFeatures
    self.limit = limit
    self.corpus = corpus
    self.index = index


def query(self, target: str, index_type: str):

    if index_type == 'linear':
        res = self.ExhaustiveSearch(target)
    elif index_type == 'annoy':
        res = self.BTreeSearch(target)
    elif index_type == 'kmeans':
        res = self.KMeansSearch(target)

    return res


def ExhaustiveSearch(self, target: str):
    target = cleanDocument(target)
    target_vector = self.vectorizer.doc_to_vector(target)

    similarity = []  # min heap

    for index, doc_vector in enumerate(self.vectorizer.matrix):
        # if index % 100 == 0: print(f"compared {str((index / len(self.corpus)) * 100)[:5]} %", end='\r')

        heapq.heappush(
            similarity,
            ( distance.euclidian_distance(
                    vector1 = target_vector,
                    vector2 = doc_vector,
                    dimentions = self.totalFeatures,
                ),
                index
            )
        )

    res = []
    for _ in range( min(len(similarity), self.limit) ):
        sim_value, index = heapq.heappop(similarity)
        res.append( (sim_value, self.corpus[index]) )
    return res


def BTreeSearch(self, target: str):
    target = cleanDocument(target)
    target_vector = self.vectorizer.doc_to_vector(target)
    leaf_indexes = self.index.find_leaf(target_vector)
    # print(f"found {len(leaf_indexes)} vectors to search.\nleaf_indexes -> {leaf_indexes[:5]}")

    res = []
    for distance, index, vector in self.index.find_closest_vectors(target_vector, leaf_indexes, self.limit):
        res.append( (distance, self.corpus[index]) )
    return res


def KMeansSearch(self, target: str):
    target = cleanDocument(target)
    target_vector = self.vectorizer.doc_to_vector(target)

    res = []
    for distance, index in self.index.find_closest_vectors(target_vector, self.limit):
        res.append( (distance, self.corpus[index]) )
    return res
```

seekr/src/loss_functions.py

```
from seekr.utils.vector import Vector
from seekr.src.vectorizer import TfidfVectorizer

class distance():

    # def cosine_similarity(vector1: list[list], vector2: list[list]) -> float:
    #     """
    #     reference: https://www.youtube.com/watch?v=e9U0QAFbfLI

    #     x.y = |x||y|cosθ
    #     let vector(x) = 1i + 2j and vector(y) = 3i + 4j
    #     now taking dot product of x and y,
    #     vector(x).vector(y) = (1 * 3) + (2 * 4) = 3 + 8 = 11
    #     |x| = √( 1^2 + 2^2 ) = √5
    #     |y| = √( 3^2 + 4^2 ) = √25
    #     therefore, cosθ = x.y / |x|*|y|
    #     => cosθ = 11 / √125
    #     """
    #     if not vector1 or not vector2: return 0

    #     target = Vector(vector1)
    #     doc = Vector(vector2)

    #     numerator = Vector.dot_product(target, doc)
    #     denominator = target.magnitude() * doc.magnitude()

    #     return numerator / denominator


    def euclidian_distance(vector1: list[list], vector2: list[list], dimentions: int = 0) -> float:
        target = Vector(vector1, dimentions)
        doc = Vector(vector2, dimentions)

        # return Vector.euclidian_distance(target, doc)
        return Vector.actual_euclidian_distance(target, doc)
```

seekr/src/core.py

```
from seekr.utils.load_data import DB
from seekr.utils.utils import cleanDocument
from seekr.src.vectorizer import TfidfVectorizer
from seekr.utils.analyzers import whitespace, ngrams
from seekr.indexes.ann.query import ANNQuery
from seekr.indexes.kmeans.query import KMeansQuery
import time
from seekr.src.query import Query


class Seekr:

    def __init__(self) -> None:
        self.corpus: list = []
        self.totalFeatures = 0
        self.index = None


    def load_from_db(self, db_name: str, location: str, column: int, maxLimit: int = 10000) -> None:
        start_time = time.perf_counter()
        self.db = DB(db_name, location, maxLimit)

        for x in self.db.rows:
            if len(x[column]) < 3: continue
            self.corpus.append( cleanDocument(x[column]) )

        self.vectorize()
        print(f"loaded {len(self.corpus)} items and vectorized in {str(time.perf_counter() - start_time)[:5]}
seconds.")


    def create_index(self, index_type: str):

        if index_type == 'annoy':
            self.index = ANNQuery(
                self.vectorizer.matrix,
            )

        elif index_type == 'kmeans':
            self.index = KMeansQuery(
                self.vectorizer.matrix
            )


    def vectorize(self) -> None:

        self.vectorizer = TfidfVectorizer()
        self.tfidf_matrix = self.vectorizer.fit_transform(
            corpus = self.corpus,
            analyzer = ngrams,
            skip_k = 0,
        )
        self.totalFeatures = self.vectorizer.featureIndex


    def query(self, target: str, limit: int = 3, index_type: str = 'linear'):
        query = Query(
                self.corpus,
                self.vectorizer,
                self.index,
                self.totalFeatures,
                limit
            )

        return query.query(
                target,
                index_type = index_type
            )


    def __repr__(self) -> str:
        return f"<Seekr Object [{len(self.corpus)} items]>"
```

main.py

```
from seekr import Seekr
import time

index_type = input("-> enter an index type (kmeans/annoy) : ")
maxLimit = int( input("-> enter number of documents : ") )

seekr = Seekr()
seekr.load_from_db('companies', 'data/companies.sqlite', column = 1, maxLimit = maxLimit)
seekr.create_index(index_type)  # 'kmeans' or 'annoy'
```

```
while True:
    print()
    target = input("-> select term to search : ")


    print("\nBTree search :-" if index_type == 'annoy' else "\nClustered search :-")
    start_time = time.perf_counter()
    matches = seekr.query(target, limit = 3, index_type = index_type)
    print(f"fetched {len(matches)} results in {str(time.perf_counter() - start_time)[:5]} seconds.", end='\n\n')
    for match in matches: print(str(match[0])[:5], match[1], sep='\t')


    print("\nexhaustive linear search :-")
    start_time = time.perf_counter()
    matches = seekr.query(target, limit = 3, index_type = 'linear')
    print(f"fetched {len(matches)} results in {str(time.perf_counter() - start_time)[:5]} seconds.", end='\n\n')
    for match in matches: print(str(match[0])[:5], match[1], sep='\t')
```