DOCUMENT CLUSTERING ALGORITHM USING TF-IDF IMPLEMENTED BY HEAP

Junho Eum, Jean Paul Azzopardi

Boston University

Table of Contents

Abstract:

This paper presents an approach to cluster documents using term frequency-inverse document frequency (TF-IDF) to identify the most important terms in each document and then using cosine similarity implemented by heap data structures to sort the documents based on their cosine similarity to the root document.

Introduction:

Document clustering is a crucial task in natural language processing and information retrieval. It involves grouping similar documents based on their content, which can help in many areas such as categorizing news articles, identifying spam emails, and identifying similar legal documents. However, clustering documents can be time-consuming for individuals since it requires human intuition to cluster and store essential documents for companies and big organizations.
Then my research question arises from here. Is there a more computationally efficient way to cluster large numbers of documents?
In this paper, I present an approach to document clustering that utilizes TF-IDF and cosine similarity implemented with heap data structures. My algorithm aims to guide individuals on how to organize documents inside a directory using heap data structures which has performance optimization since heaps have an average time complexity of O(log n) for insertion and deletion. Clustering algorithms help group similar documents based on their content, which can be achieved by calculating the similarity between each document using cosine similarity and TF-IDF.

Literature Review:

Document Clustering

Document clustering is a widely researched field and has many potential applications. For example, Steinbach, Karypis, and Kumar (2000) have used document clustering for trend detection in dynamic datasets[1], while Dhillon and Modha (2001) applied it

[1] Steinbach, M., Karypis, G., & Kumar, V. (2000). A comparison of document clustering techniques. In KDD workshop on text mining, Boston (Vol. 400, No. 1, pp. 525-526).

for text classification[2]. However, as data sets grow in size, traditional document clustering methods become computationally expensive, leading to the need for more efficient algorithms.

Term Frequency-Inverse Document Frequency (TF-IDF)

The use of TF-IDF in document clustering has been demonstrated in several studies. Salton and Buckley (1988) presented TF-IDF as a way of weighting terms in documents, considering not only the frequency of a term in a particular document but also its rarity across all documents. This concept has been adopted by large companies dealing with significant data volumes. For instance, Twitter, a leading social media platform, uses TF-IDF for trending topic detection. By clustering tweets using TF-IDF, Twitter can identify and present trending topics to its users. This approach exemplifies the application of document clustering on a massive scale and effectively mirrors the trend detection in dynamic data sets, a concept discussed by Steinbach, Karypis, and Kumar (2000).

Moreover, Ramos and others (2003) have shown that using TF-IDF can help reduce the dimensionality of the data and improve clustering performance. Despite these advancements, the challenges of handling vast amounts of data, such as those encountered by Twitter with millions of tweets generated every minute, remain an area of continual research and optimization. As the data sets continue to grow in size, the need for more efficient algorithms is evident. The adoption of these algorithms by Twitter, and other similar entities, signifies a critical step in addressing the scalability issues inherent in traditional document clustering methods.

Cosine Similarity

Cosine similarity is a common method for calculating the similarity between documents. It is often used in combination with TF-IDF to create document vectors and measure their similarity[3] (Singhal, 2001). This approach has proven successful in

[2] Dhillon, I. S., & Modha, D. S. (2001). Concept decompositions for large sparse text data using clustering. Machine learning, 42(1-2), 143-175.
[3] Singhal, A. (2001). Modern information retrieval: A brief overview. IEEE Data Eng. Bull., 24(4), 35-43.

various applications, but when dealing with large data sets, the computational cost of calculating pairwise cosine similarities can be prohibitive.

Heap Data Structures

Heap data structures have been used in numerous applications for their efficiency in handling large data sets[4] (Cormen et al., 2009). They have been implemented in various fields, such as in task scheduling[5] (Blelloch et al., 1991) and in developing efficient sorting algorithms[6] (Williams, 1964). However, to the best of our knowledge, their use in document clustering, particularly in conjunction with TF-IDF and cosine similarity, has been relatively unexplored.

Research Objective

Testing on a small dataset allowed me to explore the algorithm's behavior and assess its feasibility on a limited scale.

While the results obtained from this small-scale study cannot be generalized, they provide insights into the algorithm's performance and potential areas for improvement.

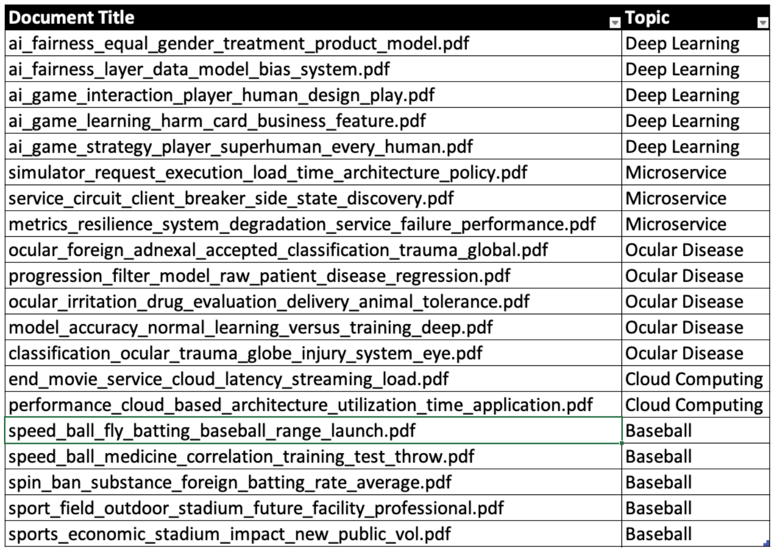
To assess the generalizability and

Methodology:

Preliminary Experiment

First, I conducted a preliminary experimental evaluation of my algorithm using a small dataset. There are a total of 18 documents inside a specified directory filename which function as class labels for the algorithm. Each class label has been selected after reading the document and was labeled in relation to the topic. (*Figure 1*)

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT press.
[5] Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., & Zagha, M. (1991). A comparison of sorting algorithms for the connection machine CM-2. In Proceedings of the third annual ACM symposium on Parallel algorithms and architectures - SPAA '91 (pp. 3-16). ACM Press. https://doi.org/10.1145/113251.113252
[6] Williams, J. W. J. (1964). Algorithm 232 - Heapsort. Communications of the ACM, 7(6), 347-348. https://doi.org/10.1145/512274.512284



Document Title	Topic
ai_fairness_equal_gender_treatment_product_model.pdf	Deep Learning
ai_fairness_layer_data_model_bias_system.pdf	Deep Learning
ai_game_interaction_player_human_design_play.pdf	Deep Learning
ai_game_learning_harm_card_business_feature.pdf	Deep Learning
ai_game_strategy_player_superhuman_every_human.pdf	Deep Learning
simulator_request_execution_load_time_architecture_policy.pdf	Microservice
service_circuit_client_breaker_side_state_discovery.pdf	Microservice
metrics_resilience_system_degradation_service_failure_performance.pdf	Microservice
ocular_foreign_adnexal_accepted_classification_trauma_global.pdf	Ocular Disease
progression_filter_model_raw_patient_disease_regression.pdf	Ocular Disease
ocular_irritation_drug_evaluation_delivery_animal_tolerance.pdf	Ocular Disease
model_accuracy_normal_learning_versus_training_deep.pdf	Ocular Disease
classification_ocular_trauma_globe_injury_system_eye.pdf	Ocular Disease
end_movie_service_cloud_latency_streaming_load.pdf	Cloud Computing
performance_cloud_based_architecture_utilization_time_application.pdf	Cloud Computing
speed_ball_fly_batting_baseball_range_launch.pdf	Baseball
speed_ball_medicine_correlation_training_test_throw.pdf	Baseball
spin_ban_substance_foreign_batting_rate_average.pdf	Baseball
sport_field_outdoor_stadium_future_facility_professional.pdf	Baseball
sports_economic_stadium_impact_new_public_vol.pdf	Baseball

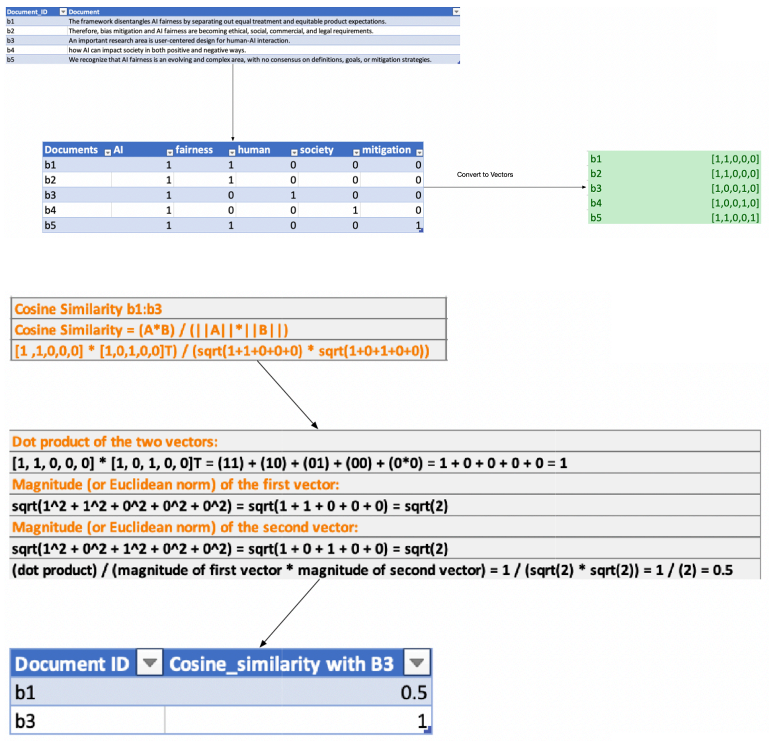
Figure 1. Description of document file names and topics used to visualize the heap data structure

The proposed approach uses TF-IDF to identify the most important terms in each document. Each document is represented as a vector in a high-dimensional space, with each unique term corresponding to a dimension. The method assigns a weight to each term in a document based on its frequency in the document and its frequency in the corpus. The time complexity of this step is typically O(n), where n is the total number of terms in the corpus.

Initially, cosine similarity is used to calculate the similarity between documents based on the frequency of each term in the document. Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space. The calculation of cosine similarity has a time complexity of O(n^2) as each document needs to be compared with every other document.

However, to reduce redundant computations, we compute and store all the cosine similarities in a matrix ahead of time. This precomputation step also has a time complexity of O(n^2), but it is a one-time cost, and allows us to retrieve any precomputed cosine similarity in constant time (O(1)) when we need it during our clustering process.

The diagram presents the Bag of Words (BoW) model used to convert the raw text documents into numerical vectors (Figure 1-1). Each document is represented as a vector in the n-dimensional space where n represents the number of unique words in the corpus. The cosine similarity between any two documents (or vectors) is then calculated to determine their semantic similarity. This vector representation and similarity measure form the foundation of our document clustering algorithm.



Document_ID	Document
b1	The framework disentangles AI fairness by separating out equal treatment and equitable product expectations.
b2	Therefore, bias mitigation and AI fairness are becoming ethical, social, commercial, and legal requirements.
b3	An important research area is user-centered design for human-AI interaction.
b4	how AI can impact society in both positive and negative ways.
b5	We recognize that AI fairness is an evolving and complex area, with no consensus on definitions, goals, or mitigation strategies.

Documents	AI	fairness	human	society	mitigation
b1	1	1	0	0	0
b2	1	1	0	0	0
b3	1	0	1	0	0
b4	1	0	0	1	0
b5	1	1	0	0	1

Convert to Vectors

b1	[1,1,0,0,0]
b2	[1,1,0,0,0]
b3	[1,0,0,1,0]
b4	[1,0,0,1,0]
b5	[1,1,0,0,1]

Cosine Similarity b1:b3
Cosine Similarity = (A*B) / (||A||*||B||)
[1 ,1,0,0,0] * [1,0,1,0,0]T) / (sqrt(1+1+0+0+0) * sqrt(1+0+1+0+0))

Dot product of the two vectors:
[1, 1, 0, 0, 0] * [1, 0, 1, 0, 0]T = (11) + (10) + (01) + (00) + (0*0) = 1 + 0 + 0 + 0 + 0 = 1
Magnitude (or Euclidean norm) of the first vector:
sqrt(1^2 + 1^2 + 0^2 + 0^2 + 0^2) = sqrt(1 + 1 + 0 + 0 + 0) = sqrt(2)
Magnitude (or Euclidean norm) of the second vector:
sqrt(1^2 + 0^2 + 1^2 + 0^2 + 0^2) = sqrt(1 + 0 + 1 + 0 + 0) = sqrt(2)
(dot product) / (magnitude of first vector * magnitude of second vector) = 1 / (sqrt(2) * sqrt(2)) = 1 / (2) = 0.5

Document ID	Cosine_similarity with B3
b1	0.5
b3	1

Figure 2-1.Simplified Illustration of Document Vectorization and Cosine Similarity Calculation.

To manage the set of most similar documents, we utilize a heap data structure. Heaps are binary trees where every parent node has a value less than or equal to any of its children. This property makes them useful for priority queue operations, such as maintaining the set of 'k' most similar documents. Adding a document to the heap or removing the least similar document (when the heap is full) has an average time complexity of O(log m), where m is the size of the heap. This approach is significantly faster than comparing each new document to all existing documents in the heap, which would have a time complexity of O(m).

Therefore, the overall time complexity of the algorithm can be broken down into these major parts:

- Computing TF-IDF: O(n)
- Precomputing cosine similarities: O(n^2)
- Retrieving a precomputed cosine similarity: O(1)
- Managing the heap: O(log m) for each insertion or deletion

Although the precomputation of cosine similarities still has a quadratic time complexity, it is a one-time cost and it significantly reduces the cost of the subsequent operations, making the overall algorithm more efficient. As always, the actual performance will also depend on factors such as the specific implementation and the characteristics of the data, such as the number of unique terms, the distribution of terms across documents, and the size of the heap.

Using Binary search tree to get logarithmic time complexity for cosine similarity calculation

Commented [준엄1]: A BST could help in optimizing lookup or search operations, but the heap structure used in code is more suitable for the purpose of maintaining the top 'k' documents with highest cosine similarity. application where you need to frequently search or lookup based on cosine similarity values, using a BST or a Balanced BST (like AVL, Red-Black Tree, etc.) could be advantageous.

In order to mitigate the computational burden associated with calculating cosine similarities and comparing documents, which in the worst case scenario has a time complexity of O(n^2), we propose the use of a binary search technique to balance the tree structure of our heap.
After computing the TF-IDF for every term and representing each document as a vector in the high-dimensional space, we usually need to calculate the cosine similarity for every pair of documents in the corpus. This operation is computationally expensive due to the quadratic time complexity. However, each

document is compared based on a single metric - the cosine similarity score. This score, represented by a single number, gives us the opportunity to apply optimization techniques used in ordered data structures.

Instead of directly calculating cosine similarities between all pairs of documents, we first calculate the cosine similarity between a given document and all other documents. Each of these cosine similarity scores is then inserted into a binary search tree (BST). The BST is a type of binary tree where the value of each node is greater than or equal to values in its left child and less than values in its right child. The tree structure makes it possible to quickly look up, insert, and delete nodes. This structure is maintained even when new nodes are inserted or existing nodes are deleted.

The BST's balancing operation ensures the height of the tree, hence the time complexity of these operations, is logarithmic with respect to the number of nodes. **This process ensures that the BST does not degenerate into a linked list, which would increase the time complexity of these operations to linear time.**

Upon balancing the BST after each insertion or deletion, we obtain an average case time complexity of O(Log(n)), a significant improvement over the naive O(n^2) approach.

This refined approach for document similarity comparison consists of these major steps:

● Computing TF-IDF: O(n)

● Calculating cosine similarities and inserting into a balanced BST: O(n log n)

● Managing the heap: O(log m) for each insertion or deletion

This optimized approach avoids the exhaustive pairwise comparison of documents, leading to improved performance, especially with large document sets.

In the case of heap data structures, a higher threshold value would result in fewer documents being added to the heap data structure, while a lower threshold value would result in more documents being added[7]. The algorithm removes the document with the lowest cosine similarity if the heap is full. The algorithm then

[7] Zhang, Xiaodan & Hu, Xiaohua & Zhou, Xiaohua. (2008). A comparative evaluation of different link types on enhancing document clustering. 555-562. 10.1145/1390334.1390429.

repeats the process with the next document, using the updated heap data structure. The algorithm uses the **networkx** and **matplotlib** libraries to draw a pyramid-like visualization of the heap data structure, based on preliminary experimentation with smaller datasets (Figure 2-2) from the order of filenames on figure 1.

Add comments about how lower threshold value can result in more documents inside a single heap structure : explain it with the figure.

To choose the best heap structure from the iteration, the algorithm calculates the average cosine similarity of the documents in the heap. It keeps track of the heap with the highest average cosine similarity and returns it as the chosen heap. This process was also visualized using data from preliminary, smaller dataset experiments (Figure 3).

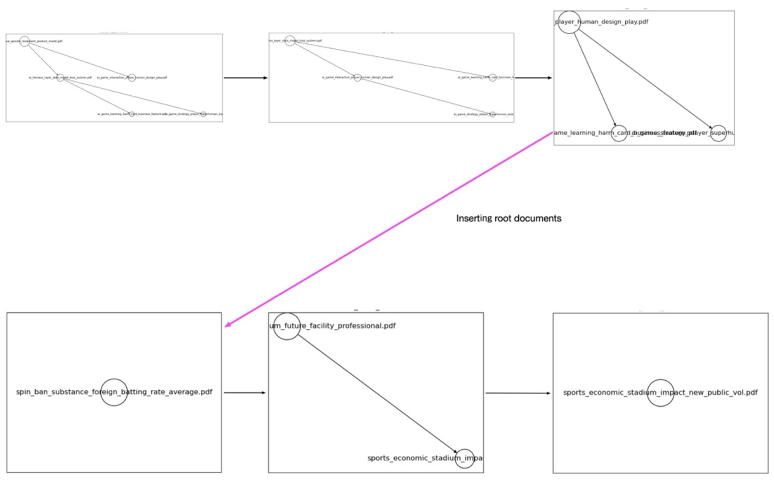

player_human_design_play.pdf
ame_learning_harm_card
Inserting root documents
spin_ban_substance_foreign_batting_rate_average.pdf
um_future_facility_professional.pdf
sports_economic_stadium_impa
sports_economic_stadium_impact_new_public_vol.pdf

Figure 2-2. Iterative visualization of heap data structure of 20 documents with threshold value = 0.2

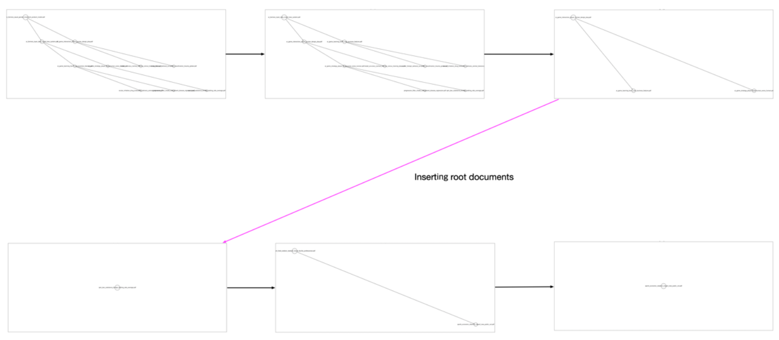


Figure 2-3. Iterative visualization of heap data structure of 20 documents with threshold value = 0.1

Preliminary Experimentation and Large-scale Application

The methodology outlined in the preceding section was initially developed and tested using smaller datasets. This preliminary experimentation was crucial for debugging, fine-tuning, and validating the approach. Through this phase, we were able to understand the nuances of our algorithm and prepare it for application on a larger scale.

As the algorithm exhibited promising results and consistent performance with smaller datasets, we proceeded to apply it to larger datasets. This section details the process and considerations involved in this large-scale application.

Hyperparameter Tuning: Grid Search for Optimal k

A crucial part of our preliminary experimentation involved hyperparameter tuning. We focused on optimizing 'k', the number of most similar documents to return for a single heap structure. The value of 'k' significantly influences the algorithm's effectiveness in document clustering.

To find the optimal 'k', we implemented a grid search. This process entailed testing a series of potential 'k' values and selecting the one that yields the highest average cosine similarity in the heap. The grid search operation was coded as follows:

```
# Define range of potential k values
k_values = range(1, 101)

# Perform grid search to find optimal k
best_k, best_avg_sim = doc_classifier.grid_search(k_values)

print("The optimal k value is:", best_k)
print("The average similarity for the optimal k is:", best_avg_sim)
```

This rigorous tuning process, conducted during the preliminary stage, validated our model's performance for optimal 'k' empirically.

Adapting for Larger Datasets

When applying the algorithm to larger datasets, certain considerations needed to be made. The computational resources required scale with the size of the dataset. As such, we ensured that adequate computational power was available for processing these larger datasets, utilizing AWS cloud engine.

Experimentation with Larger Datasets

The larger datasets used for the advanced stage of our research were drawn from diverse sources. Specifically, we focused on four distinct topics: baseball statistics, ocular disease detection with deep learning models, microservices, and cloud computing. For each topic, we collected 200 research papers from reputable academic databases, including PubMed and CORE. These research papers were carefully selected by me and my research partner, Jean, based on their relevance to the chosen topic.

In preparation for analysis, we implemented Latent Semantic Analysis (LSA) to extract seven keywords from each document (See Figure 6). This step was undertaken to avoid confusion that might arise from the originally random naming of the files. Furthermore, these keywords served two essential functions. Firstly, they provided us with a high-level understanding of the content within each document.

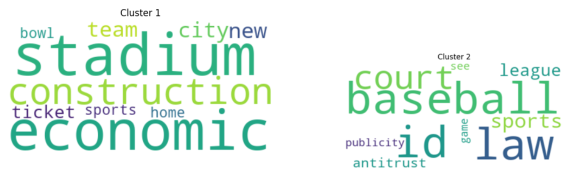
Secondly, they proved invaluable when visualizing our heap structure later in the analysis process. We printed out the process of renaming the documents according to their topic to give a better understanding to our methodology.
The terminal output was as follows:

```
Topic 1:
ability,productivity,married,experience,marriage,wages,play
er
Topic 1:
advertising,baseball,game,screen,study,field,stadium
Topic 1: age,break,barry,performance,time,two,player
Topic 1:
analysis,baseball,statistical,data,statistics,season,histor
ical
Topic 1:
anticipation,batting,occlusion,baseball,statistics,based,in
formation
```

Figure 6. Sample output of topic keywords from each document

Analysis and Results

To further aid our exploration and interpretation of the larger dataset, we employed the k-means clustering algorithm and generated word cloud visualizations for each cluster. K means algorithm that was implemented for this experiment identifies the top 10 terms for each cluster and are identified and fed into the WordCloud module, which generates a word cloud image for each cluster (See Figure 7).



Add more figures of wordcloud fitted inside a box

Figure 7. Word cloud sample of a cluster

Moreover, we sought to evaluate and compare the performance of different heap structures generated during the document clustering process. The Silhouette

Coefficient, a well-known metric for assessing clustering quality, was utilized for this task.

The Silhouette Coefficient is calculated using the mean intra-cluster distance (a) and the mean nearest-cluster distance (b) for each sample. For the context of our study, 'a' is interpreted as the mean similarity of a document to the other documents in its own heap (i.e., its cluster), and 'b' is defined as the mean of the maximum similarity of the document to documents in other heaps.

Our methodology mirrored the steps we initially used in a preliminary experiment with a smaller dataset, albeit with some modifications to accommodate the large volume of documents in this study. We developed the function get_heap_lst to produce a list of all heap structures with root document. Given the considerable number of documents, it was necessary to impose restrictions on what would be included in the best_heap_lst. This list would later be instrumental in comparing and selecting the most efficient heap data structure.

The function then iteratively creates heaps of documents based on cosine similarity, using a predefined threshold to decide whether a document should be added to the heap. Only heaps with more than two documents are added to the final list of "best heaps.".

For each document, the function then maps it to its corresponding heap index, creating a dictionary. The document names are updated to include only those documents that have been successfully mapped to a heap. Using this updated dictionary, we generate labels indicating the heap number (cluster) each document belongs to.

Finally, the function computes and returns a list of Silhouette Scores for each heap. To ensure the correctness of the Silhouette Score calculation, the score is only computed for heaps (clusters) that contain more than one document.

This evaluation process was critical in understanding how effectively our heap-based clustering algorithm groups similar documents together, and helped us in identifying the heap structures that yielded the highest quality clusters. The Silhouette Scores computed provided a quantifiable metric to compare and contrast the performance of different heaps, thus guiding us in fine-tuning our clustering algorithm for optimal performance.

The experimentation with larger datasets allowed us to evaluate the robustness and scalability of our algorithm. We recorded the algorithm's performance metrics, such as processing time and resource usage, and compared them to its performance with smaller datasets. The results of these tests (Fig 6.) add terminal output of each heap's silhouette score affirmed our approach's effectiveness and scalability

Limitation

There are a few limitations to my approach. Firstly, it is difficult to quantify the comparison results in terms of accuracy and computation time between my proposed clustering algorithm and other clustering algorithms, such as K-means. Secondly, there is no perfect solution for finding the optimal threshold for cosine similarity, as it depends on the specific use case and domain structure.

Conclusion

References

- Steinbach, M., Karypis, G., & Kumar, V. (2000). A comparison of document clustering techniques. In KDD workshop on text mining, Boston (Vol. 400, No. 1, pp. 525-526).
- Dhillon, I. S., & Modha, D. S. (2001). Concept decompositions for large sparse text data using clustering. Machine learning, 42(1-2), 143-175.
- Salton, G., & Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. Information processing & management, 24(5), 513-523.
- Ramos, J., et al. (2003). Using TF-IDF to Determine Word Relevance in Document Queries. In Proceedings of the First Instructional Conference on Machine Learning.
- Singhal, A. (2001). Modern information retrieval: A brief overview. IEEE Data Eng. Bull., 24(4), 35-43.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT press.
- Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., & Zagha, M. (1991). A comparison of sorting algorithms for the connection machine CM-2. In Proceedings of the third annual ACM symposium on Parallel algorithms and architectures - SPAA '91 (pp. 3-16). ACM Press. https://doi.org/10.1145/113251.113252
- Williams, J. W. J. (1964). Algorithm 232 - Heapsort. Communications of the ACM, 7(6), 347-348. https://doi.org/10.1145/512274.512284
- metric - Cosine similarity vs The Levenshtein distance - Data Science https://datascience.stackexchange.com/questions/63325/cosine-similarity-vs-the-levenshtein-distance

- Zhang, Xiaodan & Hu, Xiaohua & Zhou, Xiaohua. (2008). A comparative evaluation of different link types on enhancing document clustering. 555-562. 10.1145/1390334.1390429.