**Documentation of the Solution**

**For the Research Engineer Position at Emplifi**

**Exploring and Clustering Facebook Topics for Hierarchical Word Clouds**

Javad M. Rad

Email: Javad\_mohamadi\_rad@yahoo.com

Phone: +420 721 024 505

17 January 2025

**Introduction**

This document explains a practical approach to analyzing a collection of Facebook posts from the NBC News World page. Each post contains a list of topics. Our goal is to create a system where we can view all topics in a single word cloud, then drill down into smaller, thematic clouds. Each smaller cloud will contain related topics. This allows users to explore the many themes discussed on the page in a structured, interactive way.

In the following sections, I describe the process in detail: how data is prepared, how topics are grouped, why we selected certain methods, how we validate correctness, strengths and weaknesses of the approach, and how we could modify the solution for larger data or quick response times.

**Data Structure and Workflow**

We start by loading a JSON file called posts\_with\_topics.json. This file contains many elements. Each element has a unique postId field (like “219012311450917\_1925315777487220”) and a list of topics. Some posts might have three to five topics, others might have more. Once we read all posts, we gather all their topics into one master list. We remove duplicates by using a set, then convert it back into a list so we can use indexes.

After gathering all topics, we calculate how often any two topics appear together. For every post, we take every pair of its topics and record one count in a two-dimensional array. This array is called a co-occurrence matrix. In this matrix, a cell [i, j] shows how many times topic i and topic j co-occurred in the same post. If two topics appear together in many posts, we expect [i, j] to be a large number. The co-occurrence matrix in our code is obtained by the following:

sub\_matrix = co\_matrix[np.ix\_(topic\_indices, topic\_indices)]

Using the co-occurrence matrix, we build a measure of how *“close”* the topics are. Co-occurrence can be inverted into a distance measure. For example, if two topics have high co-occurrence, then the distance between them is small. This distance information is important for clustering. This is achieved by subtracting the co-occurrence values from the maximum value in the matrix. This transformation ensures that topics with strong relationships (high co-occurrence) have smaller distances, while topics with weaker relationships (low co-occurrence) have larger distances, making the data ready for clustering algorithms that rely on distance metrics. This is implemented in our code by the following:

co\_max = sub\_matrix.max()

dist\_matrix = co\_max - sub\_matrix

Regarding the clustering, we prefer hierarchical clustering because it naturally creates a tree of clusters that can easily be turned into smaller, more specific groups when we go deeper in the tree.

The next step involves doing a top-level clustering (for instance, creating five or six groups) so we get broad themes. If those groups are still very large, we can repeat the clustering within each group to get subgroups. We keep going until our tree is sufficiently deep, or until each group is too small to be split further. The final structure is a nested representation of topics similar to a tree, organized in bigger groups, which can themselves be split into smaller ones.

In the last step, we prepare a JSON output that captures this hierarchy. Every node in the JSON structure has (1) a list of the topics in that group, including how frequently they appear, and (2) subclusters for further subdivisions. This JSON file can drive an interactive web application. This structure enables us to generate a word cloud from each node’s topics. When a user clicks on a node, the front end can show subclusters in separate word clouds.

**Reasoning Behind the Chosen Methods**

I chose to use a co-occurrence matrix because we already have labeled topics for each post. This approach is direct, does not require more complicated natural language processing, and quickly measures how often two topics appear together.

Hierarchical clustering, such as *Agglomerative Clustering*, is well suited to building a tree-like structure. It recursively refines clusters until it finds meaningful subgroups. This method also gives us full control over how many levels deep we want to explore. The followings are some important parameters used in recursions during the clustering:

max\_depth=3,

min\_cluster\_size=5,

n\_subclusters=4

max\_depth = 3 specifies the level we would like to go deep in the hierarchical clustering.

min\_cluster\_size=5, sets the minimum number of the topics in the cluster for further splitting.

n\_subclusters=4 determines the number of subclusters at each recursion level. In our code, it is set to 4.

Discussing other methods of approaching this problem, some other methods like topic modeling (for instance, LDA) or graph-based community detection, could also work. However, those are more complex and often need raw text or more advanced similarity metrics. Because we already have topic labels, the simpler co-occurrence-based approach is faster to implement and easy to explain.

Among the various clustering techniques, we can compare some methods based on several key aspects. Each method has distinct strengths and limitations, making them suitable for different use cases depending on the nature of the data and the clustering objectives. The table below summarizes these comparisons in detail.

|  |  |  |  |
| --- | --- | --- | --- |
| Aspect/Methods | K-Means Clustering | Hierarchical Clustering | Hybrid Graph-Based Clustering |
| Semantic Accuracy | Moderate | High | Very High |
| Scalability | High | Moderate | High |
| Hierarchy Support | No | Yes | Yes |
| Parameter Dependency | Number of clusters (K) | Depth, distance thresholds | Similarity threshold |
| Computational Cost | Low | High | Moderate |

According to the hierarchical nature of our problem and the expected solution, Hierarchical Clustering seems to be the best choice as it is fairly qualified in terms of Accuracy, Scalability, Hierarchy Support, etc.

**Validation**

To confirm that the solution is correct as an algorithm, we will inspect the final clusters. We check if related topics end up together. For example, if topics about the US President, government policies, and elections appear in the same group, that suggests that our clustering is capturing real relationships.

From an implementation viewpoint, we can validate that our code properly handles the data structure by testing it with a small sample of posts. We see if the co-occurrence matrix has correct counts. Then we run the hierarchical clustering and check if the JSON output has the correct format. A good sign is that every topic is included exactly once at each level of its cluster, and the subclusters match our expectations.

**Strengths and Weaknesses**

The biggest strength of this solution is its simplicity. We use the existing list of topics, generate a straightforward co-occurrence matrix, and apply a known hierarchical clustering method. This is easy to implement, relatively fast for smaller sets of topics, and it produces a tree that matches how we want to show the data.

A known weakness occurs when certain topics are too frequent or too broad. If “Politics” is in almost every post, it might overshadow other finer details. That can lead to unbalanced clusters. Another concern is that a topic can only exist in one cluster at each level. If a topic is strongly related to more than one theme, we have to pick just one place to put it. We also rely on the user-supplied topics, which might not capture hidden ideas or subtopics if the original labeling was incomplete.

Improvements could include filtering out very frequent or very rare topics, or applying a more advanced distance function (like Jaccard similarity or cosine distance) instead of a basic approach. We could also offer an overlap mechanism, but that complicates the requirement of having a clean tree structure.

**Evaluation of the Code**

This code provides a systematic and effective approach for clustering topics based on co-occurrence data and generating a hierarchical JSON output. Below is a detailed evaluation of the code's strengths, weaknesses, and recommendations for optimization, especially considering scalability to multi-gigabyte datasets.

**Strengths**

1. Comprehensive Workflow:
   * The code covers all necessary steps: data loading, co-occurrence matrix computation, hierarchical clustering, and JSON output generation.
   * The recursive function for clustering ensures a hierarchical structure is created, fulfilling the requirement for drill-down exploration.
2. Customizability:
   * Parameters like max\_depth, min\_cluster\_size, and n\_subclusters allow fine-tuning of the clustering process.
   * The use of AgglomerativeClustering with a precomputed distance matrix allows for flexible linkage methods and accurate hierarchical relationships.
3. Efficient Co-occurrence Representation:
   * The co-occurrence matrix captures relationships between topics, enabling the clustering algorithm to group related topics effectively.
4. JSON Output Structure:
   * The build\_output\_tree function provides a well-structured JSON output that includes topic frequencies, making it ready for visualization.
5. Logical Distance Calculation:
   * Converting co-occurrence into a distance matrix (co\_max - sub\_matrix) aligns with hierarchical clustering's requirements for a dissimilarity measure.

**Weaknesses**

1. Scalability Concerns:
   * Co-occurrence Matrix Size: The co-occurrence matrix grows quadratically with the number of topics, which may become infeasible for datasets with tens of thousands of unique topics.
   * Memory Usage: Storing the matrix in memory (np.zeros((num\_topics, num\_topics))) can be problematic for large topic sets.
2. Computational Intensity:
   * Distance Matrix Computation: Generating the dist\_matrix for subsets can become computationally expensive for large clusters.
   * Agglomerative Clustering: The algorithm has a time complexity of O(n3) for a full matrix, making it impractical for very large datasets.
3. Static Hierarchy Parameters:
   * Fixed values for max\_depth, min\_cluster\_size, and n\_subclusters may not adapt well to varying dataset sizes and topic distributions.
4. Frontend Limitations:
   * The JSON output is verbose, which might be challenging for rendering or querying in real-time interactive applications.

**Possible Modifications and Optimizations**

An important improvement is to handle larger data. If we have tens of gigabytes of posts and thousands of topics, we might see memory issues with a full matrix. We can use a *sparse* matrix representation so we do not store zeros unnecessarily. This is what has been implemented in the *solution\_big\_data* file as an improved and optimized version of the solution. The following lines undertake conversion of the co-occurrence matrix to a sparse type:

from scipy.sparse import coo\_matrix

co\_matrix\_sparse = coo\_matrix((data, (row, col)), shape=(num\_topics, num\_topics))

co\_matrix\_sparse = co\_matrix\_sparse + co\_matrix\_sparse.T

Another concept can help with handling massive data is Dimensionality Reduction. It plays a critical role in processing the co-occurrence data efficiently. JSON files often contain high-dimensional, sparse relationships between topics. Dimensionality reduction techniques like *PCA* and *UMAP* transform the high-dimensional co-occurrence matrix into a lower-dimensional representation, preserving the most significant patterns of topic co-occurrence. We have implemented PCA and UMAP dimensionality reduction in the *solution\_big\_data* as follows:

# Dimensionality Reduction with PCA

co\_matrix\_dense = co\_matrix\_sparse.toarray()

pca = PCA(random\_state=42, n\_components=0.99)

reduced\_matrix = pca.fit\_transform(co\_matrix\_sparse)

n\_components parameter is set to 0.99 as the least amount of data variance to meet or cover. It can also be set to an integer as the number of components to be selected.

# Dimensionality Reduction with UMAP

umap = UMAP(n\_neighbors=15, min\_dist=0.1, n\_components=50, random\_state=42)

umap\_embedding = umap.fit\_transform(co\_matrix\_sparse)

UMAP is better at capturing non-linear relationships, keeping clusters intact, and creating clear visualizations.

Another idea is to compute co-occurrences in a distributed or incremental way, so we do not have to keep everything in memory at once.

For a production environment that needs quick responses, most of the clustering and hierarchy creation should happen offline. We would compute the clusters once (perhaps nightly), generate the JSON files, and then serve the final JSON to an interactive UI. This keeps the user experience fast, since the heavy work is done in advance. If data updates happen often, we can do incremental updates or partial recomputations.

Finally, for extremely large sets, we might move from hierarchical clustering to specialized graph-based approaches that scale better, or use advanced libraries optimized for large data. We can also place the final JSON on a content delivery network if it is too big, to reduce load times.

**Folder Structure**

Below is a comparison table evaluating *solution\_small\_data* and *solution\_big\_data* based on various aspects like accuracy, computational efficiency, scalability, ease of implementation, and visualization support.

|  |  |  |
| --- | --- | --- |
| Aspect/code | solution\_small\_data | solution\_big\_data |
| Accuracy | Relies only on co-occurrence distance matrix for clustering, which may be less effective in capturing non-linear relationships. | Utilizes PCA and UMAP for dimensionality reduction, which are robust methods for reducing noise and capturing relationships. |
| Computational Efficiency | More efficient as it directly clusters using the co-occurrence matrix without dimensionality reduction. | Slightly slower due to the inclusion of PCA and UMAP, which add computational overhead, especially for large datasets. |
| Scalability | Scales better as it works directly on a sparse matrix and avoids computationally expensive dimensionality reduction. | Limited scalability due to the dense representation in PCA and UMAP, which can become infeasible for very large datasets. |
| Clustering Method | Focuses solely on hierarchical clustering, making it simpler but potentially less precise for high-dimensional data. | Uses both dimensionality reduction and hierarchical clustering, resulting in potentially better separation but adds complexity. |
| Dimensionality Reduction | No explicit dimensionality reduction; clusters directly on the co-occurrence distance matrix. | Employs PCA and UMAP providing a well-separated embedding for clustering. |
| Visualization Support | Does not include built-in visualization like word clouds; focuses more on data hierarchy generation. | Includes word cloud generation and recursive word clouds for clusters, offering robust visualization options. |
| Ease of Implementation | Simpler and easier to implement as it avoids UMAP, PCA, and advanced visualization techniques. | Slightly more complex due to the integration of PCA, UMAP, and word cloud visualizations, requiring familiarity with these tools. |
| JSON Output Structure | Similar hierarchical JSON output but lacks depth due to the absence of advanced dimensionality reduction. | Outputs hierarchical JSON with topic frequencies and a clear structure, suitable for downstream tasks. |
| Preprocessing | Utilizes dense matrices, which are straightforward but can be less memory-efficient for large topic spaces. | Uses sparse matrix representations, which is efficient but more complex to implement and manage. |
| Flexibility | Less flexible but easier to adapt for basic clustering tasks without dimensionality reduction. | More flexible due to modular steps like dimensionality reduction and custom visualization, allowing easier experimentation. |

**Conclusion**

This approach of co-occurrence-based hierarchical clustering offers a clear, understandable solution for creating a word cloud of Facebook topics. It is easy to implement, relatively lightweight for moderate data, and fits well into any interactive front-end. While it has some weaknesses in handling very frequent topics and extremely large datasets, those can be addressed with filtering, sparse representations, or pre-computation strategies. The final JSON output captures the entire hierarchical structure and can be used by developers to create an engaging topic exploration tool.