Sorting your climate change-related indicators!

Lucie Kuczynski - email

2025-06-18

Contents

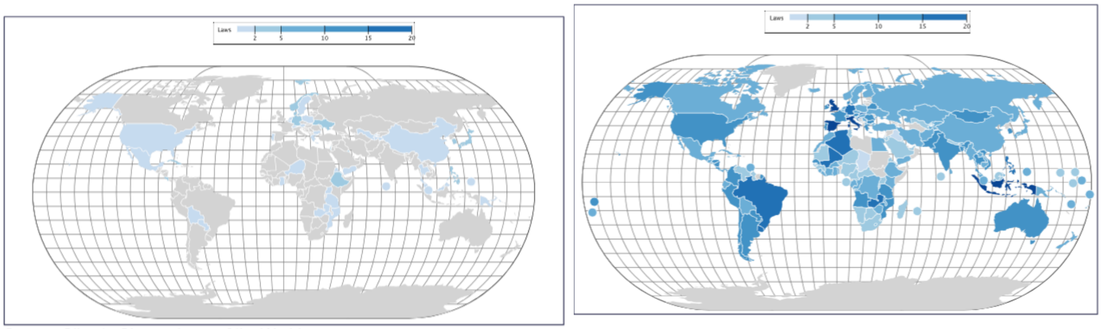
We've created a script that helps identify survey questions that are very similar to each other. It starts by tidying up the text of each question, then converts them into a format that makes comparisons easier. It checks how closely related the questions are, highlights the ones that are nearly identical, and groups similar ones together. This process helps reduce duplicate questions without losing any of the original information.

The script is built on sentence transformers, which offer a major advantage over traditional automated methods like edit-distance, TF–IDF, or topic modeling. While those approaches tend to focus on surface-level similarities or general themes, sentence transformers go deeper by learning to map semantically similar sentences (e.g., rewording, paraphrases) to nearly the same point in a dense vector space. This means that identifying paraphrased indicators becomes as simple as running a cosine similarity check between their embeddings. The result is a powerful, language-model-based comparison - without the need to write any deep learning code yourself.

Warning: Although the automated clustering greatly reduces manual effort (especially with hundreds of indicators) it can still overlook matches. For that reason the user should review the exported .xlsx and, where necessary, merge clusters by editing just two columns: `ClusterID` (to group items) and `Global_Indicator` (to supply the common label). Because the first pass already demanded a quick visual check for false-positives, the user will be familiar with the indicator set by the time they do this tidy-up. Eliminating this manual step altogether is on the roadmap but not urgent. After the user finishes these edits, they should run `cleaning.R`. That script collapses every revised cluster into a single row while preserving and, where appropriate, merging each of the detail, data-need, source and framework fields.

Why is it useful?

To address the challenges of climate change, various international frameworks, treaties, and conventions have been established. These initiatives aim not only to limit the progression of climate change but also to support adaptation efforts and assess its impacts. They typically include sets of indicators (both quantitative and qualitative) to track progress toward specific goals and evaluate the effectiveness of related policies. Over time, environmental frameworks have developed mechanisms that allow them to adapt and respond to emerging needs. For instance, between 2017 and 2019, the number of national policies and measures reported by Member States to curb greenhouse gas emissions rose by approximately 27% (EEA, 2019) while climate legislation globally increased from 1997 to 2017 (Fig. 1).



Source: Climate Change Laws of the World

Source: Climate Change Laws of the World

Fig. 1 - Climate legislation in 164 countries in 1997 (left) and 2017 (right).
Online source, accessed June 2025.

The downside of this otherwise adaptive approach is the growing number of indicators. As new frameworks emerge, they often introduce additional indicators that, over time, tend to overlap with existing ones. These older indicators are still maintained (e.g., they are familiar to stakeholders, their use preserves consistent time series) leading to redundancy across different monitoring systems (Fig. 2).

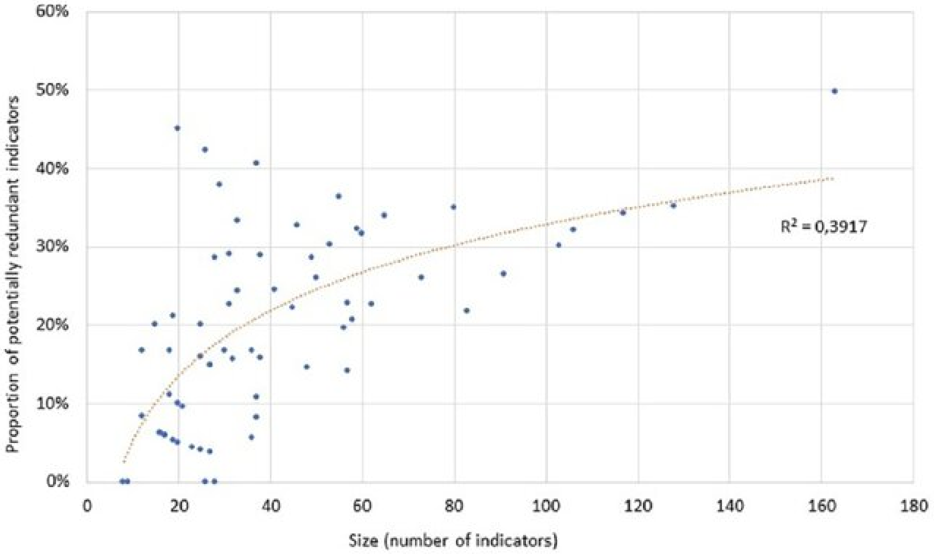


Fig. 2 - Relationship between number of indicators and their redundancy.
Online source, accessed June 2025

A time-consuming approach to managing indicator redundancy is to manually review and classify each one based on perceived similarity. While this method doesn't require technical skills (just familiarity with the indicators) it becomes increasingly inefficient and error-prone as the number of indicators grows.

To streamline this process, we propose a script that performs clustering, grouping similar indicators together. This allows users to review one cluster at a time, rather than comparing all indicators individually. For each cluster, the user can decide whether it makes sense and, if so, assign it a name; otherwise, the cluster can simply be tagged to be manually reviewed.

At the end of the process, a CSV file is generated, which the user can further edit manually. For each cluster, the CSV file contains information on identified related frameworks as well as the overall climate change component (e.g., adaptation, mitigation, drivers, impacts) it relates to.

How does it work?

The script is organized into three sections:

1. User settings, packages
2. Function definitions
3. Main script

User settings and packages

The first step of the script is relatively straightforward. This section sets up the environment for the analysis. It begins by clearing the workspace to ensure no previous data interferes with the current run. The working directory is then set, which is where the script will read input files and write output files. Then, the script specifies the path to the Excel file containing the survey questions, along with the sheet index/name and the column that holds the actual questions. This allows the script to load the relevant data for processing.

Next, the script defines key parameters for deduplication and clustering:

- The `similarity_start` parameter sets the initial threshold for cosine similarity, which determines how closely two indicators must match to be considered duplicates. A higher value means stricter matching, while a lower value allows for more lenient comparisons.
- The `max_rounds` parameter limits the number of deduplication rounds to prevent infinite loops in case of unexpected data issues.
- The `min_cluster_size` parameter specifies the minimum number of indicators required to form a cluster. This is set to 1 by default, meaning even a single indicator can form its own cluster if no close matches are found.
- The `false_positive_tag` is used to mark clusters that are identified as false positives during manual review, allowing for easy reprocessing later.
- A list of `frameworks` is defined, which will be used later to categorize the indicators based on their source.

```r
# Remove all objects from the current R session (start with a clean workspace)
rm(list = ls())

# Set the working directory for file read/write operations
setwd("C:/Users/luciek/My Documents/Knowledge/Indicators/Analysis/")

# File and Data Column Settings
# Path to the Excel file containing your questions
file_path <- "C:/Users/luciek/Documents/Knowledge/Indicators/052025_indicators.xlsx"
# Sheet index or name to load (1 = first sheet)
excel_sheet <- 1
# Column name in the Excel file containing the survey questions
question_col <- "Indicator"

# Deduplication & Clustering Parameters
# Starting threshold for cosine similarity (higher = stricter, lower = more matches)
similarity_start <- 0.90
# Maximum number of deduplication rounds (to avoid infinite loops)
max_rounds <- 10
# Minimum size for a question cluster (rarely changed)
min_cluster_size <- 1

# Manual Review Tag
# Value used to tag clusters as false positives (for later reprocessing)
false_positive_tag <- "FP"
```

```r
# Framework list
frameworks <- c(
  "Blue_pacific", "BRS_convention", "CBD", "FDES", "Global_set",
  "FGS", "HH_survey", "Migratory_fish_convention_Pacific",
  "Minamata_convention", "Montreal_protocol", "Noumea_convention",
  "PIRT", "Ramsar", "Regional_goal", "Rotterdam_convention",
  "Samoa_pathway", "SDG", "Sendai", "SPREP", "Stockholm_convention",
  "UN_fish", "UNCCD", "UNFCCC", "Underwater_cultural_heritage_convention",
  "Waigani_convention"
)
```

The script then loads the necessary R packages, which include `dplyr`, `tidyr`, `stringr`, `data.table`, and `text`. These packages provide functions for data manipulation, string processing, and working with text embeddings.

Warning: This script requires a working installation of Python (3.11 recommended) as well as Java. Several Python packages must be installed for the embedding and clustering steps to work smoothly: `sentence-transformers`, `torch`, `nltk`, `numpy`, `scipy`, and `transformers.` You can install them all at once by running the following command in your terminal or command prompt:

```
pip install sentence-transformers torch nltk numpy scipy transformers
```

Function definitions

This section defines four functions that will be used to clean indicators phrasing, generate embeddings, calculate cosine similarity, and perform clustering.

The function `clean_question` is designed to standardize the phrasing of survey questions. It removes any leading or trailing white space, converts the text to lowercase, and strips out any numbering systems that might be present in the questions. This is important because indicators may be reported with their own numbering systems, which can vary and be inconsistent across different frameworks. The function also removes punctuation and collapses extra spaces, while intentionally retaining stop-words and numbers within the definitions, as the semantic model can handle them effectively.

The function `embed_and_sim` deals with computing embbedings and similarity of indicators. First, the function handles the transformation of each survey question into a numerical format that a computer can work with. The first time the script runs, it downloads the MiniLM model (a compact, license-friendly tool trained on large amounts of text). MiniLM converts each question into a 384-dimensional vector, essentially a long list of numbers that captures the meaning of the sentence. These vectors, known as embeddings, act like unique "fingerprints" for each question. They reflect not just the words used, but also their order, context, and nuances like negation or synonyms. For instance, this approach makes it possible to clearly distinguish between similar-sounding phrases like "proportion of energy from renewable sources" and "proportion of energy from fossil sources". The embedding captures the presence of negation and other subtle differences in meaning, allowing the script to treat these as distinct indicators - embeddings provide a way to represent the meaning of text in a form that algorithms can analyze. In practice, the function sends each question to MiniLM, which returns a vector (e.g., [0.12, -1.07, 0.35, . . . , -0.63]) with hundreds of values. Each row in the resulting `emb_mat` corresponds to one question, now represented in a way that allows for meaningful comparison with others. This step is currently parallelized to speed up computation. It automatically uses the number of available processor cores minus four, ensuring efficient performance while leaving some system resources free for other tasks. Then, the function calculates the cosine similarity between all pairs of question embeddings. This involves first normalizing each question's embedding vector so that all vectors have unit length (i.e., resizing them to make them directly comparable). Once normalized, the script calculates the dot product between every pair of vectors using a fast, optimized function (`tcrossprod`), which avoids explicit loops and leverages BLAS acceleration for efficiency. Cosine similarity measures how closely two vectors "point" in the same direction. If the similarity score is close to 1, it means the questions are likely expressing the same or very similar ideas. Scores near 0 or negative suggest the questions are unrelated or discussing entirely different topics.

The function `get_frameworks` determines which frameworks are associated with each cluster of indicators. For every cluster (i.e., group of merged indicators), it examines both the "Source" and "Link(s) w other frameworks" columns for all indicators in that cluster. Using a predefined list of framework names, it scans these columns to identify any relevant frameworks, regardless of which indicator in the cluster they originated from. This means that if two indicators are merged into a single cluster, any frameworks mentioned in either indicator's source or links will be marked as present for the whole cluster. The result is a logical (TRUE/FALSE) vector for each framework, summarising which frameworks are represented within each cluster.

Finally, the function `deduplicate_interactive` performs interactive clustering and deduplication based on the pairwise similarity of indicators. Using average-linkage hierarchical clustering, which merges groups based on the average distance between all members, it forms coherent and balanced clusters of survey questions that express similar ideas.

The workflow is iterative and user-driven:

- Each round, all survey questions are grouped into clusters using the current similarity threshold (which can be made stricter in subsequent rounds). The process begins by converting the cosine similarity scores into distances (distance = 1 − similarity), making them suitable for clustering. The script then applies average-linkage hierarchical clustering, where each indicator initially forms its own cluster, and clusters are progressively merged based on the average distance between all their members. This average-linkage method balances between overly lenient (single-linkage) and overly strict (complete-linkage) clustering, resulting in more coherent, well-structured groups. The final step involves cutting the clustering tree (dendrogram) at the set threshold, ensuring that all indicators within a cluster are similar to at least one other member by the chosen standard.
- Automatic assignment: If every indicator in a cluster is identical after cleaning, the function assigns them a "global" indicator automatically.
- Manual review: For clusters containing different but similar questions, the user is shown all unique phrasings in that group and can either (a) enter a new "global" label that best represents the group, or (b) simply press Enter to flag the cluster as a false positive (FP).
- Metadata preservation: Throughout, the function preserves each indicator's original text, cleaned version, topic/subtopic, and flags for all relevant frameworks, ensuring traceability and rich output.
- Output: The result is a comprehensive, deduplicated dataset where each indicator has been reviewed, assigned to a meaningful cluster, and, if necessary, manually resolved by the user. The output CSV captures all global indicator assignments, false positive flags, framework memberships, and details for each question.

About the similarity threshold: In this context, false positives refer to pairs of survey questions that the script identifies as being very similar (i.e., potential duplicates or paraphrases), but which are not actually similar in meaning when reviewed by a human. For example, two questions might share a lot of the same words or structure, leading the model to assign them a high similarity score. However, they could be addressing different topics or have subtle but important differences in intent (e.g., "proportion of protected area of total area" and "proportion of forested area of land area"). These mismatches are considered false positives because the model "positively" flagged them as similar, but that classification turns out to be incorrect. Raising the similarity threshold (e.g., from 0.80 to 0.85) can help reduce the number of these false positives by requiring a stronger match before two questions are considered similar. However, setting the threshold too high can have the opposite effect: it may limit clustering to only those indicators that are phrased almost identically. This reduces the script's ability to detect meaningful paraphrases or slightly reworded versions of the same concept, potentially missing out on useful groupings. Finding the right balance is key to capturing both precision and nuance.

Why doing it this way?

When choosing MiniLM with cosine similarity for identifying duplicate or similar survey questions, the decision is based on both performance and practicality. Traditional methods like exact duplicate detection (e.g., using `duplicated()`) are limited to catching only byte-for-byte identical entries, missing any reworded or paraphrased versions. String distance metrics such as Levenshtein or Jaro–Winkler are only effective for very minor edits (e.g., pluralization) and fail when sentence structure changes or synonyms are used.

Other approaches like TF-IDF with cosine similarity or topic modeling techniques (e.g., LDA or LSA) can capture general themes but fall short in understanding sentence-level meaning. They often misinterpret negations or subtle shifts in phrasing, and require extensive tuning, especially for short texts. While heavier models like full BERT embeddings offer slightly better accuracy, they come with significant computational costs. MiniLM, by contrast, offers a strong balance: it's lightweight, fast, and accurate enough for most practical use cases.

In terms of performance, MiniLM can process around 1,000 short questions per second on a standard laptop CPU. The cosine similarity matrix is computed efficiently using BLAS, and even for a few thousand questions, memory usage remains modest (i.e., about 4.4 MB for 3,000 questions). This efficiency allows the script to remain in base R, using only three external CRAN packages, while still delivering results comparable to deep learning models.

For users with compatible hardware, the script can be further optimized. Settings like `batch_size` can be adjusted to maximize GPU usage without exceeding memory limits, and enabling `via_parallel = TRUE` allows sentence tokenization and batching to run in parallel across R workers. Additionally, embeddings can be cached, so previously processed questions don't need to be re-embedded in future runs.

Altogether, this setup provides a powerful, efficient, and flexible solution for identifying and managing duplicate or similar survey indicators - without requiring deep technical expertise or heavy infrastructure.

Issues and ideas for further development

Any feedback is always appreciated as this is very much a work-in-progress!

Issues

As of Wednesday, June 18, 2025, the `Sorting.R` workflow still has a few rough edges:

- `Data_needs` and `Data_sources` are dropped when clusters are merged. Unlike the `Details` field, these two columns are rebuilt from scratch instead of being concatenated. *For now this is low-impact, because most indicators lack content in those columns and adding the missing values by hand is faster than re-running the script.*

Ideas

During the interactive phase of `Sorting.R`, the user currently types a free-text label for each cluster. Possible upgrades:

- Pick by number: allow the user to enter the line-number of one of the composite indicators to adopt its wording as the new cluster label.
- Explicit 'keep separate' flag: a shortcut to mark the entire set as not a true cluster (i.e. a confirmed 'false positive').
- Auto-flag very large clusters: automatically treat any proposed group containing 10+ indicators as suspect and mark it as a false positive until reviewed.

These refinements would speed up the review loop without adding much complexity to the codebase.