Study protocol

2021-10-26

Contents

Research Protocol

This document will guide the reader along the steps to utilise the framework and reproduce our results.

Add installation and setup info here.

Acquisition of the records

Once the framework is loaded, the user defines an initial search query which needs to be as specific as possible while still generating a sufficient number of positive matches:

```
source('R/Setup.R') # Load the framework

# Initial query to be built on domain knowledge. It accepts OR, AND, NOT boolean
# operators and round brackets to group terms.
query <- '((model OR models OR modeling OR network OR networks) AND
(dissemination OR transmission OR spread OR diffusion) AND (nosocomial OR
hospital OR "long-term-care" OR "long term care" OR "longterm care" OR
"long-term care" OR "healthcare associated") AND (infection OR resistance OR
resistant))'

# Year filter. The framework converts it to the API-specific format seamlessly.
# common logical comparators can be used, i.e. <, <=, >, >=, while dashes
# denotes inclusive date intervals. A single year restricts results to one year
# period.
year_filter <- '2010-2020'
```

The query is passed to the `perform_search_session()` function, together with an unique identifier for the **search session** and **search query**. A session identifies a homogeneous group of complementary or alternative queries (e.g. sometimes it is easier to create two different simpler queries than a complex and long one that returns the same results).
In our implementation, we define as sessions the group of actions that includes the utilisation of a search query and the subsequent iterations of manual review and automatic classification of its results.

`perform_search_session()` will use common scientific research database APIs (at the moment Pubmed/Medline, Web of Science and IEEE) to acquire records related to the query. The records will be stored as CSV files in folders with the following structure: `Records/session_id/query_id`, created automatically. The function also creates or updates a **journal** (a CSV or Microsoft Excel file) with all the information about sessions, queries and search results.
Some services (IEEE, WOS) requires an API key to access the data; the API key is not mandatory for Pubmed but can help avoid quota limitations in case of a large number of results. IEEE can also be searched without API: the framework will utilise a web scraping approach if the API key is missing, but the success of this method is not guaranteed. See Supplemental Material S3 for detailed instruction on preliminary steps and tips for each supported search engine. These keys can be stored in the `secrets.R` file in the working directory, with the following example code (the "*baysren*" prefix identifies the package specific options):

```
options(baysren.ieee_api_key = 'your-ieee-api-key')
options(baysren.ncbi_api_key = 'your-ncbi-api-key')
options(baysren.wos_api_key = 'your-wos-api-key')
```

This file will be automatically parsed from the framework.

If previously downloaded results are already available, users can create the `Record` folder manually and put the data there (the file names need to contain the source name as written in the `sources` argument of `perform_search_session()`). The function will acquire and parse them into a standard format, even if split into multiple files due to download limits (e.g. `Pubmed1.nbib`, `Pubmed2.nib`, etc.). In addition to Pubmed, WOS and IEEE, this method permits to import also EMBASE and SCOPUS records, for which an API search was not yet implemented.
Using the `actions` argument, users can choose whether to perform an API search, parse already downloaded results, or both.

Note that for Pubmed, the API search may return slightly fewer records than a manual one due to a different query expansion algorithm (this divergence should be solved in April 2022) between the two services, therefore is advisable to perform also a manual search and put the resulting .nbib files in the `Session/Query` folder. `perform_search_session()` will acquire them seamlessly.

If a user wants to perform the API searches and results parsing manually for each source, the framework exposes the individual functions to search for records using the APIs and parsing bibliography files.

```
# This call will perform an API search on Pubmed, WOS and IEEE services and/or
# (based on the `actions` argument) parse already downloaded data (in our case,
# additional Pubmed results and Embase and Scopus results) manually added to the
# Session1/Query1 folder.

journal <- perform_search_session(
    query = query, year_query = year_filter,
    session_name = 'Session1', query_name = 'Query1',
    records_folder = 'Records',
    journal = 'Session_journal.csv')
```

Once the records are stored, they must be read and merged into an **annotation file**, where the initial manual review will be performed.
A series of functions are available to prepare the data for manual evaluation:

```
# Extract the file paths of records. Arguments can be used to filter by session
# query, source. Only parsed files will be returned, not the raw ones downloaded
# manually.
Annotation_data <- extract_source_file_paths(journal) %>%

    # Read record files. Will parse them if raw data is downloaded manually (not
    # necessary in this case). Return a list of records, one per file.
    read_bib_files() %>%

    # Join a list of records into one dataframe, solving duplicated records
    join_records() %>%

    # Order records by the relative frequency of the query terms in the title +
    # abstract text. Increases the chance of encountering relevant records at the
    # beginning of the manual classification.
    order_by_query_match(query) %>%

    # Add fields for the manual classification
    mutate(
            Rev_manual = NA, # Where the first manual classification will be made
            Rev_prediction = NA, # Column for the evaluation of the predicted classes
            .before = DOI
        )
```

All these steps are joined into the `create_annotation_file()` function. This function can take as input either the path to record files, or the parsed records (one data frame or a list of data frames) or a group of folders where to look for files into./ In addition, it also allows adding the new records to a previous annotation file (arg `prev_records`). This is useful when performing consecutive search sessions with different queries; previous manual classifications can be imported using `prev_classification`.
If the original query is passed to the `reorder_query` argument, the records are reordered according to the frequency of query terms in the title and abstract (**simple query ordering**); this approach lets increase the rate of relevant matches at the beginning of the dataset.

```
# This function extracts the appropriate file paths from a session journal. By
# default it passes all stored files, otherwise it can be filtered by session,
# query, and source (i.e. Pubmed, WOS, IEEE)
record_files <- extract_source_file_paths(journal)

# create_annotation_file() accept a great variety of inputs:

    # either record file paths
input <- record_files

    # or specific record file folders
input <- file.path('Records', 'Session1', 'Query1')

    # or parent folders, since it searches for files recursively
input <- 'Records'

    # or the already parsed files
input <- read_bib_files(record_files)

# We can then call create_annotation_file() with one of the above input
Annotation_data <- create_annotation_file(input, reorder_query = query)
```

Once created, the annotations need to be stored in an **Annotation Session** folder, which will contain all information necessary for the automatic classification. This operation is streamlined by the `create_session()` function, which also helps manage duplicated sessions.

```
# Create the first session
create_session(Annotation_data, session_name = 'Session1')
```

The file annotation file is called `Records_{date}`, situated in the folder `Sessions/{session_name}`; `date` here represents the session creation timestamp, and `session_name` the name passed by the user.

Notice that the framework is "pipe" friendly, and the precedent steps can be all performed in one call:

```
# First, create the Records/Session1/Query1 folder and put the manually
# downloaded records there if there are any. Otherwise, the folders will be
# created automatically during the API search

# perform searches and parse files; save and return the journal
Records <- perform_search_session(
    query = query, year_query = year_filter,
    session_name = 'Session1', query_name = 'Query1',
    journal = 'Session_journal.csv') %>%

    # return the record file paths
```

```
    extract_source_file_paths() %>%

    # read, join, reorder records and save them in the Sessions folder
    create_annotation_file(reorder_query = query) %>%

    # Create a new session with the records
    create_session(session_name = 'Session1')
```

Manual classification and prediction

Now the annotation file is ready for the manual classification records. The annotation file has a column called `Rev_manual`, where records need to be classified as "positive" (y) or "negative" (n) matches by the user. We suggest manually labelling an **initial training set** of 250 records, which should be sufficient if a minimum ~20% positivity rate is present (see the section about model hyperparameter optimisation)./ The function `check_classification_trends()` can help visualise the trend of the positives/negatives as the number of classified records increases./

Once the initial training set is prepared, the function `enrich_annotation_file()` will use a Bayesian Additive Regression Trees (BART) machine learning model to learn the labelling pattern given the record data and predict the positive match probability for the unlabelled records.
`enrich_annotation_file()` comprises two steps. The first step is the creation of a **Document Term Matrix** (DTM):

- The framework extracts **features** from text data (i.e., title and abstract) of the records, authors, keywords and MESH terms (see Methods for more info). Before features extraction, text data terms are lemmatised, stop-words are removed, and only nouns, adjectives, verbs and untagged terms are retained. Authors, keywords, and MESH are kept unmodified. Co-occurrent features (non-consecutive ngrams, **nc-grams**) are identified and **redundant features** (i.e. terms always appearing together) are joined to decrease the feature space noise.
- For each term, a 0/1 label specifies if it is present in each record.
- Positive records are oversampled **10x** to improve sensitivity;
- An extra feature is added for each record field (title, abstract, authors, keywords, MESH terms), reporting the number of terms in each.

The second step is the modelling of the labelling pattern and the predictions of the relevancy/positivity probability:

- The function models the labelling pattern using the manually labelled records (**learning phaseéé);
- After the model is created, it assigns a **posterior predictive distribution** (PPD) of the probability of positivity to each record (**prediction phase**). If the number of records is large, the prediction step is broken into chunks of **5000 records**, to avoid memory saturation; the users can set up this parameter according to their machine capabilities;
- This process is repeated **10 times**, and the predictive distributions are averaged to induce shrinkage of the estimates (i.e., decrease the chance of outliers), creating a prediction **ensemble**;
- The resulting distributions are summarised by the **98% interval** [98% PrI] and the median; the [98% PrI] identifies the **uncertainty zone**;
- The unlabelled records will receive a positive/negative predicted label; the label is positive if its lower [98% PrI] bound is higher than the uncertainty zone upper bound and negative if the upper [98% PrI] bound is lower than the uncertainty zone lower bound. All other records are labelled as **uncertain** (unk);
- If a predicted label does not match an existing one, the record will be labelled as **check**.

The user can define parameters like the positives' oversampling rate, the number of models to average in the ensemble, whether to apply bootstrap resampling before each repetition, the prediction chunk size, and the PPD quantiles used to build the uncertainty zone. To use the function, the only input needed is the session name; the function will automatically identify the relevant file to make predictions on.

```
New_annotations <- enrich_annotation_file('Session1')
```

This function will produce a new file which is stored in the Annotations folder of the session; the file will contain a column called `Rev_prediction_new` where the user will find records that require manual review marked by "*". These are records with uncertain labelling ("unk"), but also positive predictions and"y"/"n" predictions that contradict previous manually input labels (labelled as "check"). This approach increases the accuracy of the predictions since the machine learning system is set up to favour sensitivity at the expense of specificity. If the function finds prelabelled records (`Rev_previous` column) added when the Annotation file was created or passed as a further input to `enrich_annotation_file()` (`prev_classification` argument), they will be used to label the records marked in the `Rev_prediction_new` column.

The function adds a number of diagnostic pages to the annotation file:

- A results summary page reporting the number of records in need of manual review (which includes "unk", "check", and unreviewed "y" records), the changes, and the final distribution of the labels. It also shows other information like the number of features in the DTM, the path to the parent file from which the annotation file was created, the replication number (see later). The content of this summary is also saved as a CSV file in the Results folder.
- A variable importance page which list the used terms in order of the fraction of BART trees in which they were used. If an ensemble of models is used, a more robust score is computed, taking the ratio of the mean of the term tree fraction among models and its standard deviation.
- The arguments passed to the function if different from the default, useful for reproducibility.
- A classification performance summary. The procedure may take some time and requires `cmdstan` installed on the machine. Therefore the `compute_performance` argument is set to `FALSE` by default.

Finally, the PPD ensemble produced by the Bayesian engine is stored on the disk. This feature can be turned off to save disk space, but at least the samples of the last annotation are necessary to build a new search query (see next section).

The process of predicting labels and manually reviewing them is what defines a "classification-review (CR) iteration".

Once the manual review of the generated annotation file is completed, the user can **rerun exactly the same function**. The last reviewed annotation file will be automatically picked up and used for a new CR iteration. If the number of records requiring manual review is excessively high (e.g. > 250), the function needs to be called with the `stop_on_unreviewed` argument set to `FALSE` to allow the classification to proceed even in the presence of unreviewed records in the `Rev_prediction_new` column.

If there are no positive results (after manual review) in a CR iteration, the framework records the next iteration as a **replication**. After four replications with no new positive matches, the function will prevent another iteration and send a warning message, indicating that the session is complete. The user can change the replication limit; furthermore, it is possible to prevent the function from starting if either a specified number (or a fraction) of the total records has been manually reviewed or if a certain number of positive matches have been found. These three parameters are passed to the `limits` argument, which has the following structure and defaults:

```
list(
    stop_after = 4,
    pos_target = NULL,
    labeling_limit = NULL
)
```

The result files in the Results folder are generated after the Classification step but before the Review one, and therefore show preliminary results with still unreviewed records. The function `consolidate_results(session_name)` will update the results files to show the results after the manual review. The original result data is stored anyway in the respective annotations files.

New search query generation

It is not uncommon for the first query used in a systematic review not to be the most effective one, especially in terms of sensitivity. We tried to address the problem by creating an automatic query generation system. The solution is experimental and greatly favours sensitivity at the expense of specificity. This is the general query generation mechanism:

- We fit a partition tree between the DTM and 800 samples from the PPD; if a term is present multiple times in the DTM (e.g., both title and abstract), they are counted just one, and field term count features are removed. This step generates a list of rules composed by AND/NOT "conditions" made of terms/authors/keywords/MESH tokens, which together identify a group of records.
- For each rule, negative conditions (i.e., NOT statements) are added iteratively, starting from the most specific one, until no conditions are found that would not also remove positive records.
- The extended set of rules is sorted by positive-negative record difference in descending order. The cumulative number of unique positive records is computed and used to group the rules. Rules inside each group are ordered by specificity.
- The researcher is then asked to review the rule groups, selecting one or more rules (useful if they convey different meaning) from each, or edit them (in case too specific positive or negative conditions were included). It is possible to exclude a group of rules altogether, especially those with the worse sensitivity/specificity ratio.
- The selected rules are joined together by OR statements, defining a subset of records with a sensibly higher proportion of positive records than the original one.
- Redundant rules (i.e., rules whose positive records are already included in more specific ones) and conditions (i.e., conditions that once removed do not decrease the total number of positive or do not increase the negative records) are removed.
- Finally, the rules are re-elaborated in a query usable on the major scientific databases.

The steps for generating a query are the following:

```
# The last annotation file and posterior sample matrix in the session are found;
# the most predictive terms (based on the importance score) are chosen and
# simple prediction trees are used to map the terms to a random subset of the
# posterior probabilities (extracted from the posterior sample matrix saved
# during the last CR iteration). The score threshold and the number of trees can
# be chosen by the user; less stringent values can greatly increase computation
# time;

candidate_queries <- extract_rules('Session1')

# The 'candidate_queries' contains the rules and the DMT and the labels on which
# the rules were built, which need to be passed to
# generate_rule_selection_set(). This function selects the most effective
# (higher positive/negative matches delta) rules; also adds negative terms to
# increase specificity. A data frame is generated (or automatically saved if
# save_path is set). The output proposes groups of equivalent (in terms of the
# number of positive matches found) groups of query, ordered by specificity; for
# each group, the user needs to select the best rule by setting the
# selected_rule column to TRUE. The groups are in order of growing
```

```
# sensitivity/decreasing specificity, so it may make sense to exclude the last
# groups totally in order to avoid too many search results.

Target <- candidate_queries$DTM$Target
SpecificDTM <- candidate_queries$SpecificDTM

selection_set <- generate_rule_selection_set(
    candidate_queries$rule,
    target_vec = Target,
    target_data = SpecificDTM)

selection_set_file <- file.path('Sessions', 'Session1', 'Selected_rules.xlsx')
openxlsx::write.xlsx(selection_set, selection_set_file)

# Once the manual selection is performed the selected rules can be further
# simplified in order to remove redundant rules and terms. The aim is to provide
# the shortest rule without loss of retrieving accuracy.

simplified_rules <- file.path('Sessions', 'Session1', 'Selected_rules.xlsx') %>%
    import_data() %>%
    simplify_ruleset(ruleset = Target, SpecificDTM) %>%
    pull(rule)

# Finally, the selected rules need to be joined in a query that scientific
# search engines can understand

new_query <- rules_to_query(simplified_rules)
```

One last thing to keep in mind is that the query generation algorithm is only aware of the literature collected inside the annotation file. So it will only be able to pick term combinations that identify positive matches inside the already specific set of documents while not necessarily being discriminant from the rest of the global literature.
Once the final query is generated, it is advisable to add a constraining subquery with an AND logic to restrict the results to the scope of the topic of interest.
In our case, we added "AND ((antimicrobial resistance) OR (healthcare infection))" since, being the global topic of our research, those terms were understandably not selected by the query generation algorithm as discriminant.

The final query was the following:

```
query <- "(((Donker T) NOT (bacterium isolate)) OR ((network patient) AND
(resistant staphylococcus aureus) NOT (monte carlo) NOT isolation) OR
(facility AND (network patient) AND regional NOT hospitals NOT increase NOT
(patient transport) NOT (control infection use)) OR ((patient transport) NOT
(Donker T) NOT worker) OR (hospitals AND (network patient) NOT
(patient transport) NOT regional NOT clinical) OR (facility AND
(network patient) NOT hospitals NOT (patient transport) NOT regional NOT
prevention NOT medical) OR ((healthcare facility) NOT (Donker T) NOT
worker NOT positive) OR (hospitals NOT (network patient) NOT medical NOT
environmental NOT outcome NOT global) OR ((network patient) NOT facility NOT
hospitals NOT (patient transport) NOT therapy NOT global)) AND
((antimicrobial resistance) OR (healthcare infection))"
```

Another advantage of this technique is an at-glance view of relevant concepts for the topic of interest.

New search session

Once the new query was generated, a second search session called `Session2` was created, using the strategy described in the protocol. Due to the lower specificity of the second query, particular care may be needed in managing the far larger amount of records: multiple record files may need to be downloaded from Pubmed, EMBASE, and SCOPUS since they have limits on the number of records one can download at once; also, the machine learning classification processing time increases proportionally with the number of records to label. It is advisable then to consider removing the less specific rules iteratively from the new query until a manageable number of records is obtained, considering that the probability of finding new positive matches not already found by more specific queries drops exponentially.

Once the new records are collected, `enrich_annotation_file('Session2')` (or any other new session name) can be used to perform the new CR iterations. It may be possible that during the first CR iteration, the number of records requiring manual review is a significant number: we suggest evaluating still not more than 250, keeping in mind to set the `stop_on_unreviewed` argument to `FALSE`.

Result analysis and reporting

A number of tools are available to evaluate the classification performance.
As said before, a result file is stored in the session folder for each CR iteration with information on the classification status; remember to run `consolidate_results(session_name)` if you want these results to take into account the manual review of the predicted labels.
`compute_changes()` is a simple function that take an annotation data frame and computes the changes from the last annotation iteration.
The primary function for result analysis is `estimate_performance()`. The function uses a Bayesian model to estimate how the probability of a positive match drops as records' lower boundary of the posterior predictive interval decreases. This model computes the cumulative distribution of possibly missed positive records, which permits extraction of the engine's theoretical sensitivity and efficiency to lower 90% credibility boundaries (i.e., the indicators are better than these values with 95% probability). These boundaries depict worse case scenarios since it is impossible to know the actual number of missed positive records.
The function returns the Bayesian R^2 of the model (values below 85-90% would invalidate the model analysis), the posterior sensitivity, the observed and theoretical number positives, the expected efficiency given the theoretical number of positives (that is 1 - the number of reviewed records above the number of records to review to find the predicted number of positives according to a negative hypergeometrical distribution). Also, the function plots the cumulative distribution of the observed and theoretical number of positive records.
`extract_var_imp()` extract the most relevant terms used by the BART model and summarises their relevancy for the machine learning model and their statistical relevancy in a simple generalised linear model. A discrepancy between these two scores indicates a non-linear effect/interaction between terms.

Finally, several functions are available (check `R/Reporting.R`) to summarise and format the results.
For example, `summarise_by_source()` and `summarise_sources_by_session()`, respectively for one session or a group of sessions, report the scientific databases the records are collected from, while `get_source_distribution()` describes the distribution of the number of common sources per record.
`summarise_annotations()` and `summarise_annotations_by_session()` describe the results of the CR iterations, again for one or multiple sessions, reporting the number of positive and negative matches, the number of terms (features) used by the machine learning model, and how the labels change between iterations. `format_performance()` and `format_var_imp()` format the output of the similarly named analyses. `plot_predictive_densities()` plots how the mixture of posterior predictive distributions of records, divided into positive, negative and to-review records, change between iterations.

Model parameters' optimisation

As with many complex machine learning models, the choice of hyperparameters can significantly influence the framework performance. For a systematic review helper algorithm, we define the performance as the ability to find as many positive matches by manually labelling/reviewing as few records as possible:

$$Sensitivity \times Efficiency = \frac{Pos.\ found}{Tot.\ pos.} \times \left(1 - \frac{Records\ reviewed}{Tot.\ records}\right)$$

We utilise a "grid search" algorithm to find the best set of hyperparameters. We tested these tools on our dataset, manually labelling the first 1200 records, ordered by simple query match.
The first step is to create a folder where to store the file required for the search, for example, *Grid Search*. Then a fully reviewed record file in the format generated by `create_annotation_file()` is necessary; this file can be put in the *Grid Search*, but it is not mandatory.

Once the folder and the file are ready, run:

```
Grid_search <- perform_grid_evaluation(
    records, sessions_folder = 'Grid_Search',
    prev_classification = records,
    ## Model parameters (can be changed by users)
    resample = c(FALSE, TRUE),
    n_init = c(50, 100, 250, 500),
    n_models = c(1, 5, 10, 20, 40, 60),
    pos_mult = c(1, 10, 20),
    pred_quants = list(
        c(.1, .5, .9),
        c(.05, .5, .95),
        c(.01, .5, .99))
)
```

with `records` being the fully annotated data frame or a file path to it. `prev_classification` can be used to import a different classification file with prelabeled records.

The hyperparameters are the following:

- resample: whether to use bootstrapping to increase variability between the ensemble models;
- n_init: number of initially labelled records to train the machine learning model on;
- n_models: the number of models to fit and then average to decrease the variance due to random sampling in the MCMC algorithm. It reduces the likelihood of extreme predictions;
- pos_mult: the oversampling rate of positive records. Useful to improve sensitivity;
- pred_quants: the distribution quantiles of each PPD, used to define the uncertainty zone and identify records to review.

Once the grid search is performed, the function `analyse_grid_search()` can be used to analyse its results. This function uses a partition tree analysis to identify hyperparameter sets (**performance clusters**) with different mean performance scores. As said before, the default score is Sensitivity x Efficiency; alternatives are the Positive Rate (i.e., positive records found over total records) or Positive Rate x Sensitivity.
The function produces a data frame with the score and the cluster of reference of each combination of parameters, the best hyperparameter set (i.e., the most efficient set among the most sensitive in the best cluster) and the best set for each cluster. Also, the function creates a plot to visualise the individual impact on performance of each hyperparameter.

Once chosen, the hyperparameters can be passed to `enrich_annotation_file()`.
Mind that the grid search process may take many days, even on powerful computers, so we suggest sticking to the default set of parameters when using `enrich_annotation_file()`.