FLAGR: A Flexible High-Performance Library for Rank Aggregation

Leonidas Akritidis[a,*], Miltiadis Alamaniotis[b], Panayiotis Bozanis[a]

[a] *School of Science and Technology, International Hellenic University, 14th km Thessaloniki - N. Moudania, Thermi, Thessaloniki, Greece*

[b] *Department of Electrical and Computer Engineering University of Texas at San Antonio San Antonio, USA*

Abstract

The fusion of multiple preference lists into a single aggregate list with improved element ranking is a well-studied research area with numerous applications in bioinformatics, information retrieval, collaborative filtering, and election systems. Despite the existence of a large number of rank aggregation methods, only a small portion of them have publicly available implementations. In this paper we introduce FLAGR, a high performance, modular, open source library for rank aggregation. The library contains efficient implementations of both baseline and state-of-the-art algorithms that receive multiple ranked preference lists and output a single consensus ranking. We also introduce PyFLAGR, a library that links to the FLAGR core and allows the invocation of its implementations from standard Python programs. The package also includes a special tool that can be used to evaluate and compare the performance of the underlying algorithms. In contrast to other solutions, FLAGR has been created with flexibility in mind: third-party researchers and analysts may easily integrate their implementations into the library by developing only a single function. These features render FLAGR and PyFLAGR an attractive research platform for developing, comparing and evaluating rank aggregation algorithms.

Extended descriptions of the library components and useful code examples are provided in the accompanying user manual (provided as supplemental material) and the supporting Web site at https://flagr.site.

Keywords: rank aggregation, library, FLAGR, PyFLAGR, Python, C++

Email address: lakritidis@ihu.gr, Miltos.Alamaniotis@utsa.edu, pbozanis@ihu.gr (Leonidas Akritidis[a,*], Miltiadis Alamaniotis[b], Panayiotis Bozanis[a])

Required Metadata

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current code version	1.0.8
C2	Permanent link to code/repository used for this code version	https://github.com/lakritidis/flagr
C3	Code Ocean compute capsule	
C4	Legal Code License	Apache License, 2.0 (Apache-2.0)
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	C, C++, Python
C7	Compilation requirements, operating environments & dependencies	GCC or MingW. No dependencies.
C8	If available Link to developer documentation/manual	https://flagr.site/
C9	Support email for questions	lakritidis@ihu.gr

Table 1: Code metadata (mandatory)

1. Motivation and significance

Knowledge discovery from the aggregation of multiple ranked lists is a multi-disciplinary problem with significant challenges. The objective of rank aggregation algorithms is to generate an improved output list derived from the processing of a number of given input lists. Such goals are frequently set in numerous scientific applications, including genomic data analysis, information retrieval, recommender systems, collaborative filtering, and so on.

The challenge of creating an improved list from a set of individual input lists has attracted the attention of multiple researchers. Despite the popularity of the problem, very few rank aggregation libraries exist. Most of them implement deprecated, or simplistic methods that ignore important aspects of the problem such as input lists with partial overlap, and/or unequal sizes, and/or different importance. Built by multiple independent vendors in diverse programming languages, these packages do not allow researchers to reliably evaluate and compare the performance of the implemented methods.

As a remedy to these drawbacks, we introduce FLAGR, an open-source, flexible and scalable library for developing and testing rank aggregation algorithms. FLAGR is designed to combine efficiency, portability, extendibility, and platform independence. As of version 1.0.8, the package implements the following algorithms:

- Two linear combination methods: CombSUM and CombMNZ, each accompanied by four weight normalization techniques: Rank, Borda, Score, and Z-Score normalization [1].

- Borda Count, equivalent to CombSUM with Borda normalization [1, 2].

- Three majoritarian methods: Condorcet winners [3], Copeland winners [4] and the Outranking Approach of [5].

- Kemeny optimal aggregation (brute force implementation).

- Four methods based on the principles of Markov Chains [6] and the MCT variant introduced in [7].

- Robust Rank Aggregation [8].

- A weighted method, based on a preference relations graph [9].

- A second weighted method that progressively merges the most proximal lists in a fashion very similar to agglomerative clustering [10].

- A distance-based, weighted method called DIBRA [11]. DIBRA may use the computed weights not only for item scoring, but also for pruning the preference lists of the weakest rankers.

The last 3 algorithms are known as weighted aggregators. They apply exploratory analysis to automatically identify the expert rankers in an unsupervised fashion. Then, they assign higher weights to those who are declared experts, thus boosting the scores of their submitted elements. To the best of our knowledge, no publicly available implementation for these methods exist.

Regarding efficiency, the implementations utilize robust data structures and algorithms that ensure high performance. For instance, the aggregate list is implemented as a hash table with separate chaining to support fast element searches that accelerate input list fusion. Such choices allow FLAGR to be used in demanding applications that involve thousands of very long preference lists.

FLAGR adopts a modular architecture, allowing third-party programmers to easily integrate their own methods into the library. The design of the core facilitates the development of new algorithms by implementing only a single function. This function receives all the necessary information to conduct the aggregation via its arguments.

The library is developed in C++ 11.0, one of the most popular programming languages for both scientific and industrial applications. Moreover, FLAGR exposes a set of C functions that allow it to be built as a shared (SO), or dynamic link library (DLL). This SO/DLL can be subsequently imported into third-party applications developed in other languages, including Python, Java, R, and PHP. PyFLAGR is an example of such case: it links to the aforementioned shared library and enables the execution of the FLAGR implementations from standard Python programs. Since the introduced libraries support the most widespread programming languages presently, we expect their broad adoption by the research community.

The initial versions of FLAGR have been used to support QuadSearch, a Web metasearch engine that drew results from 4 major search engines to respond to user queries [12, 13]. Recently, the library was substantially expanded to implement and evaluate DIBRA [11]. On the other hand, PyFLAGR has been created to popularize the software, support the Python community, and materialize a tool for easily utilizing and comparing the underlying algorithm implementations. Both libraries are licensed under the Apache 2 license that, among others, allows their free usage, modification, and redistribution.

2. Software description

A rank aggregation application involves a set of queries $Q = \{q_1, q_2, ..., q_N\}$ and a set of rankers $R = \{r_1, r_2, ..., r_m\}$. Each query $q \in Q$ is submitted to all rankers in R, who respond by returning a ranked list of preference items sorted in decreasing importance, or relevance order. The goal of a rank aggregation algorithm is to merge all the preference lists for each query, discover the important latent information, and generate a single output list $L^{(q)}$ with improved element ordering.

Table 2 illustrates an example where three rankers submit their ranked answers to the hypothetical query "*Which accessories do you buy for your smartphone?*". Notice that similar queries are frequently posed to users who purchase goods through an e-commerce platform; the answers are extensively utilized in recommender systems.

The objective of a rank aggregation method is to process the submitted preference lists and produce an ordering L of the most popular smartphone

r_1	r_2	r_3	L
MicroSD	headphones	headphones	headphones
PowerBank	MicroSD	PowerBank	MicroSD
headphones	case	case	PowerBank

Table 2: An example of rank aggregation

accessories. In many cases, an algorithm must fulfil additional requirements, such as handling input lists of unequal lengths, lists with missing elements, or rankers of different importance. FLAGR implements the appropriate mechanisms to effectively handle these cases.

2.1. Compilation

The software is accompanied by two building scripts that include:

- a typical makefile for building FLAGR on Linux. The `make` command automatically creates the executable `FLAGR` and the shared library `flagr.so`.
- a batch file for building FLAGR on Windows. Similarly to Linux, `makefile.bat` creates the executable `FLAGR.exe` and the dynamic link library `flagr.dll`.

Both scripts require GCC compiler. All generated binary files are stored in `bin/Release`.

2.2. Software Architecture

2.2.1. FLAGR Architecture

The FLAGR core comprises 14 C++ classes that link to each other according to the architecture of Fig. 1. The input includes one file that stores the preference lists to be aggregated, one file with the relevance judgments for each list element (called the `Rels` file), and several user-defined values that determine algorithm hyper-parameters, execution modes, etc. The `Rels` file is optional: if provided, then a built-in evaluation tool quantifies the aggregate list quality by computing the values of multiple measures.

The input data is managed by two objects: `InputParams` and `InputData`. The first one stores vital execution parameters such as the selected rank aggregation algorithm, its hyper-parameters, input/output file locations, etc. `InputParams` is virtually visible by all FLAGR components as it is frequently required to access its contents. On the other hand, `InputData` accommodates the input preference lists, organized into an array of `Query` objects.

A `Query` object maintains two major components: The first one is the `Aggregator`, a multi-purpose object that: i) stores an array with the input preference lists (`InputList`), ii) applies the selected algorithm, and iii) generates the output aggregate list (called `MergedList`). The second component of `Query` is the `Evaluator`, an object that optionally computes the quality of the generated aggregate list by utilizing the user-defined relevance judgments (`Rels` object).

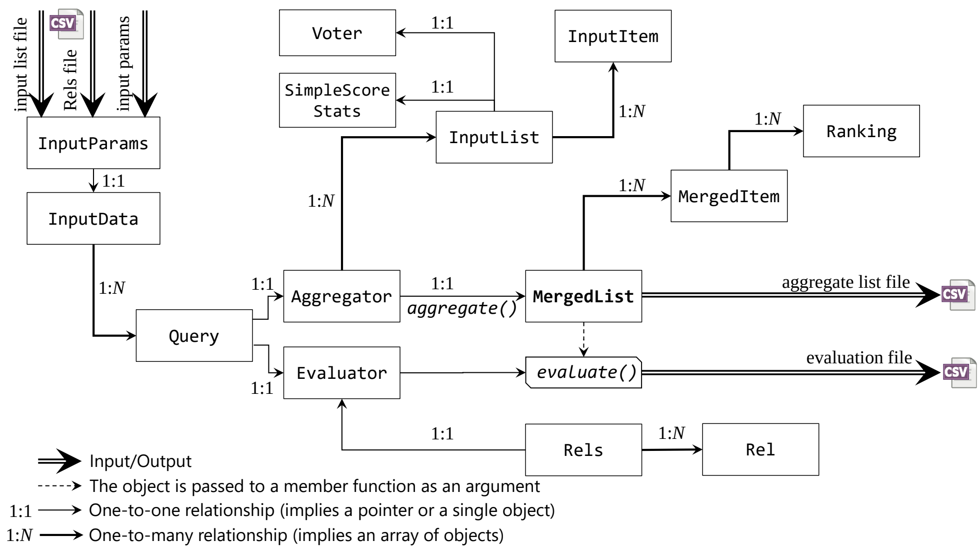


Figure 1: Low-level architecture of FLAGR. The rectangles represent the 14 classes of the library core. The functionality of the four arrow styles is described in the legend.

The output of FLAGR consists of a CSV file that stores the generated aggregate lists. The library creates one aggregate list per input query; so, if there are Q queries, FLAGR generates Q aggregate lists. Each row in the file represents an item of an aggregate list sorted in decreasing score order. In case a valid `Rels` file is provided, the evaluation process takes place automatically. In this case, the `Evaluator` computes multiple evaluation measures for each aggregate list and outputs a second CSV file, where it writes their values. We refer the reader to Subsection 3.1 of the supplemental material for a detailed description of the file organizations.

2.2.2. Shared/Dynamic Link Library Architecture

One of the most powerful features of FLAGR is its ability to be compiled as a shared library, allowing its utilization from third-party applications developed in other programming languages. Despite this type of compilation is OS-specific, there are pre-compiled shared libraries for Windows (DLL) and Linux (SO) in the `bin/Release` folder of the package repository. A pre-compiled DYLIB for MacOS is planned for the future versions of FLAGR.

FLAGR exposes a set of C functions that wrap around the original C++ algorithm implementations and enable their linkage from other programs. They essentially act as dynamic references to which a client code can link to. When an exposed function is called, it accesses the FLAGR core and calls the implementation of the respective algorithm. Then, the procedure that was described previously takes place. Figure 2 depicts this architecture.

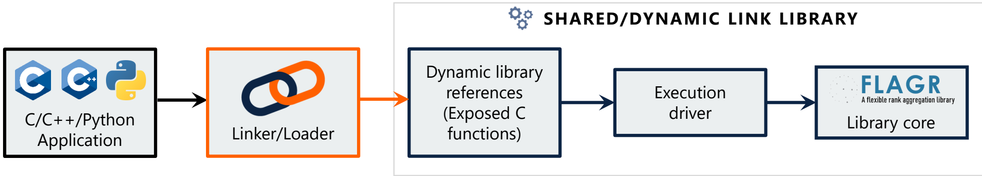


Figure 2: Linking to the Shared/Dynamic Link Library of FLAGR from a third-party application. The application accesses the exposed dynamic references that are subsequently executed by the Execution Driver.

The Execution Driver enables the exposed functions to be executed in a unified manner. Its input comprises a simple structure that stores the user-defined parameters and orchestrates the execution flow. Initially, it copies the parameters to an `InputParams` object and initializes an `InputData` object that reads the input data file. Then, the Driver sequentially invokes the `aggregate()` and `evaluate()` methods of `InputData` to perform rank aggregation and evaluation of the generated list, respectively.

2.2.3. PyFLAGR Architecture

PyFLAGR is a Python package built on top of FLAGR. It constitutes an example of how a third-party application can link to the exposed dynamic references to access the respective algorithm implementations. The required SO/DLL files are bundled within the main package, allowing PyFLAGR to be immediately utilized without any other requirements. PyFLAGR is a member of Python repository index and can be installed with the `pip` package manager: `pip install pyflagr`.

The library comprises a base class called `RAM` (Rank Aggregation Method), and a collection of derived classes whose main role is to call the respective exposed functions of FLAGR. Figure 3 illustrates the class hierarchy and demonstrates how it fits into the global architecture of Figure 2. `RAM` performs a number of vital operations, including I/O handling, linking to the FLAGR shared library, data integrity checks, etc.

The implementations of the aforementioned derived classes reside within 6 modules: `Linear`, `Majoritarian`, `MarkovChains`, `Weighted`, `Kemeny`, and `RRA`. Each class handles a specific rank aggregation method by calling the respective exposed function of the linked shared library. Notice that a module is essentially a group of classes that handle methods belonging into the same category. For instance, the `Majoritarian` module includes the classes `CondorcetWinners`, `CopelandWinners` and `OutrankingApproach`, which in turn handle the respective rank aggregation methods.

Moreover, PyFLAGR offers a special class named `Comparator` that im-

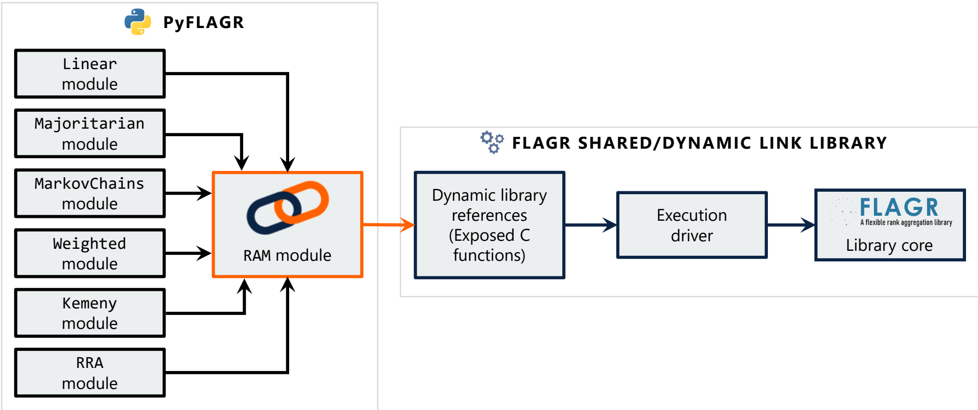


Figure 3: Fitting PyFLAGR into the architecture of Figure 2.

plements several tools for conducting performance comparisons. Its input includes a data file with the input preference lists, another file with the relevance judgments for the involved list elements, and a group of rank aggregation algorithms to be compared. After running the algorithms on the input data, the class produces comparison tables in various formats (e.g., CSV, LaTeX, etc.) and plots of Precision, Recall, Mean Average Precision (MAP) [14], DCG (Discounted Cumulative Gain), and nDCG (normalized DCG) [15].

2.3. Software Functionalities

2.3.1. Rank Aggregation

The introduced libraries implement baseline and state-of-the-art algorithms for aggregating multiple preference lists and generating improved consensus rankings. Such algorithms are presently utilized in a wide variety of applications in the fields of bioinformatics, recommender systems, etc. These applications frequently involve large volumes of data. Therefore, the code has been carefully optimized for execution speed and memory consumption.

2.3.2. Custom Algorithm Implementations

FLAGR adopts a modular architecture, allowing new algorithms to be integrated into the core. A special file named `CustomMethods.cpp` contains empty function bodies where a programmer may implement new algorithms. This file is imported in advance by the appropriate FLAGR components, enabling the library to be compiled without any further actions. Additionally, an exposed function exists for these custom methods, so the FLAGR shared library immediately obtains access to them.

2.3.3. Performance Evaluation and Comparison

FLAGR includes a robust tool for evaluating rank aggregation algorithms by computing the values of multiple well-established measures, including MAP, Precision, Recall, DCG, and nDCG. The results of the evaluation are written into a CSV file, which can be subsequently converted to other formats, or be visualized by using plotting libraries.

2.3.4. Results' Vizualization

Apart from allowing the usage of FLAGR in Python programs, PyFLAGR is also capable of illustrating the generated results. As of version 1.0.8, PyFLAGR produces bar plots for MAP, and comparative diagrams for Precision, Recall, DCG and nDCG. Internally, PyFLAGR constructs a Dataframe that stores the computed evaluation measures. Then, it employs `matplotlib` to illustrate the Dataframe's tabular data.

3. Illustrative Examples

In this section, we present two examples that demonstrate the usefulness of PyFLAGR in conducting comparative studies on rank aggregation methods. Both examples utilize a dataset[1] that contains 1000 ranked preference lists submitted by 50 customers of an electronic store, in response to 20 given queries. Each list contains 30 elements (namely, products). To aggregate the 50 preference lists for each query, and obtain a list of the most popular products, we imported 19 rank aggregation methods into the `Comparator` class of PyFLAGR.

Figure 4 depicts a comparative bar plot of the MAP [14] achieved by the 19 participating methods. The results indicate that MC3 achieved the highest MAP, followed by DIBRA. Moreover, Figure 5 depicts the value of mean Precision at the first 5 positions of the 19 aggregate lists. MC1 and MC2 achieved the highest P@1, whereas DIBRA and DIBRA-prune outperformed all the other methods in terms of P@3, P@4 and P@5.

The complete example, with additional documentation, reside in the `/examples/jupyter/` folder of the package repository.

4. Impact

According to the relevant literature, the weighted algorithms frequently produce aggregate lists of higher quality compared to their non-weighted

[1]Created with RASDaGen, an open-source synthetic dataset generator for rank aggregation problems `https://github.com/lakritidis/RASDaGen`

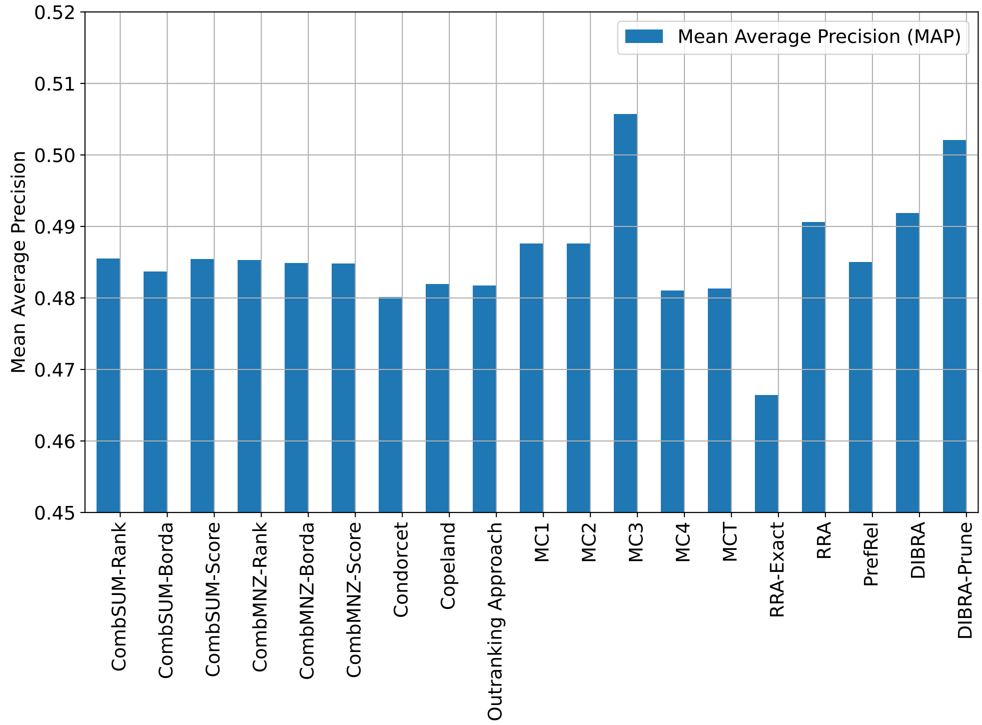


Figure 4: A comparative bar plot of Mean Average Precision achieved by the 19 participating methods.

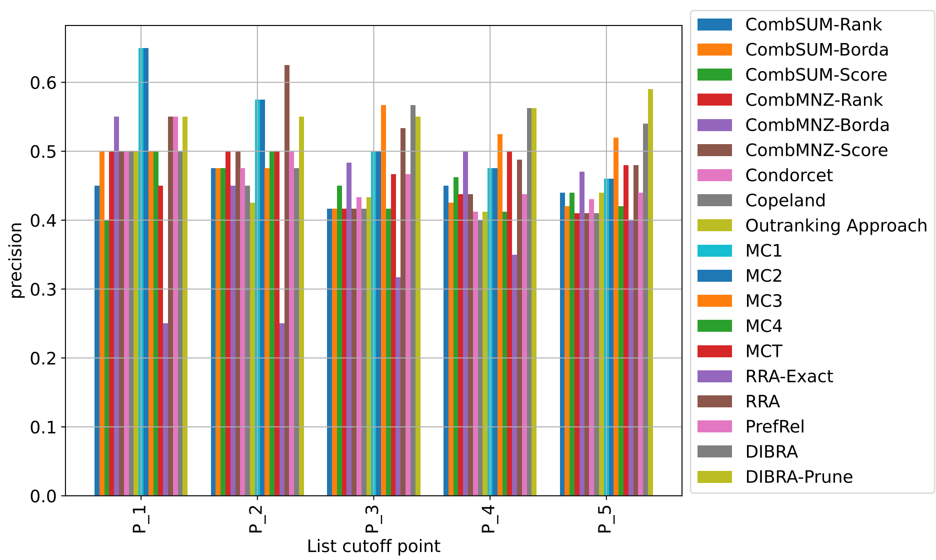


Figure 5: A comparative bar plot of Precision for all top-5 list elements.

counterparts. However, to the best of our knowledge, no publicly available implementations exist for these techniques. In contrast, FLAGR implements 3 such methods [9, 10, 11]. Therefore, its modular design, combined with the embodied evaluation tool, establishes a strong ground for developing and testing novel weighted aggregators.

Additional novel research topics that can be studied with FLAGR including weight-based list pruning and discarding. The former topic examines whether, or how the lowest-ranked elements of the less important input lists can be discarded. The latter examines whether entire low-weighted input lists should be ignored during the aggregation process.

One of the primary objectives of FLAGR is to establish a reliable environment for developing and evaluating rank aggregation methods. This objective is achieved by three key elements : i) its modular design, that facilitates the implementation and integration of new methods, ii) the implementations of numerous competitive algorithms for testing purposes, and iii) its built-in evaluation tool that provides accurate performance measurements for the implemented techniques.

Compiling FLAGR as a shared library is another crucial step towards increasing its impact. Third-party scientific and industrial applications can link to this library and execute the implemented methods without considering complex theoretical and practical details. PyFLAGR constitutes a demonstration of this logic. PyFLAGR also includes several advanced tools for testing and comparing rank aggregation algorithms that it an attractive research platform for this area.

5. Conclusions

In this paper we introduced FLAGR and PyFLAGR, two flexible rank aggregation libraries. The former includes efficient C++ implementations of numerous state-of-the-art algorithms and adopts a modular architecture that facilitates the integration of new methods. Furthermore, an embodied evaluation tool provides quality measurements of the generated aggregate lists by using multiple well-established metrics, like MAP, Precision, Recall, DCG, and nDCG. One of the most powerful characteristics of FLAGR is the exposure of several C functions that allow it to be built as a shared library. In this case, FLAGR can be imported by independent programs to exploit the existing implementations.

On the other hand, PyFLAGR is a library that imports the algorithm implementations of FLAGR into standard Python programs. Specifically, PyFLAGR links to the aforementioned shared library and provides access to its exposed functions. It also includes robust visualization tools that produce various plots of the acquired performance measurements.

6. Conflict of Interest

No conflict of interest exists: We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgements

The authors would like to thank John Burkardt for kindly permitting the usage of the ASA063 and ASA109 algorithm implementations for the computation of the incomplete beta integrals.

References

[1] M. E. Renda, U. Straccia, Web Metasearch: Rank vs. Score Based Rank Aggregation Methods, in: Proceedings of the 2003 ACM Symposium on Applied Computing, 2003, pp. 841–846.

[2] J. d. Borda, Mémoire sur les élections au scrutin, Histoire de l'Academie Royale des Sciences pour 1781 (Paris, 1784).

[3] M. Condorcet, Essai sur r application de Ianalyse à la probabilité des décisions rendues à la pluralité des voix, Paris.

[4] A. H. Copeland, A reasonable social welfare function, Tech. rep., Mimeo, University of Michigan USA (1951).

[5] M. Farah, D. Vanderpooten, An Outranking Approach for Rank Aggregation in Information Retrieval, in: Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 2007, pp. 591–598.

[6] C. Dwork, R. Kumar, M. Naor, D. Sivakumar, Rank Aggregation Methods for the Web, in: Proceedings of the 10th International Conference on World Wide Web, 2001, pp. 613–622.

[7] R. P. DeConde, S. Hawley, S. Falcon, N. Clegg, B. Knudsen, R. Etzioni, Combining Results of Microarray Experiments: A Rank Aggregation Approach, Statistical Applications in Genetics and Molecular Biology 5 (1).

[8] R. Kolde, S. Laur, P. Adler, J. Vilo, Robust Rank Aggregation for Gene List Integration and Meta-Analysis, Bioinformatics 28 (4) (2012) 573–580.

[9] M. S. Desarkar, S. Sarkar, P. Mitra, Preference Relations Based Unsupervised Rank Aggregation for Metasearch, Expert Systems with Applications 49 (2016) 86–98.

[10] S. Chatterjee, A. Mukhopadhyay, M. Bhattacharyya, A Weighted Rank Aggregation Approach Towards Crowd Opinion Analysis, Knowledge-Based Systems 149 (2018) 47–60.

[11] L. Akritidis, A. Fevgas, P. Bozanis, Y. Manolopoulos, An Unsupervised Distance-Based Model for Weighted Rank Aggregation with List Pruning, Expert Systems with Applications 202 (2022) 117435.

[12] L. Akritidis, D. Katsaros, P. Bozanis, Effective ranking fusion methods for personalized metasearch engines, in: Proceedings of the 12th Panhellenic Conference on Informatics, IEEE, 2008, pp. 39–43.

[13] L. Akritidis, D. Katsaros, P. Bozanis, Effective rank aggregation for metasearching, Journal of Systems and Software 84 (1) (2011) 130–143.

[14] C. Manning, P. Raghavan, H. Schtze, Introduction to Information Retrieval, Cambridge University Press, 2008.

[15] K. Järvelin, J. Kekäläinen, Cumulated gain-based evaluation of IR techniques, ACM Transactions on Information Systems (TOIS) 20 (4) (2002) 422–446.

Current executable software version

Nr.	(Executable) software metadata description	Please fill in this column
S1	Current software version	1.0.8
S2	Permanent link to executables of this version	https://github.com/lakritidis/flagr
S3	Legal Software License	Apache License, 2.0 (Apache-2.0)
S4	Computing platforms/Operating Systems	Linux, Microsoft Windows
S5	Installation requirements & dependencies	No dependencies
S6	If available, link to user manual - if formally published include a reference to the publication in the reference list	https://flagr.site and https://github.com/lakritidis/FLAGR/blob/main/docs/FLAGR_manual.pdf
S7	Support email for questions	lakritidis@ihu.gr

Table 3: Software metadata (optional)

FLAGR

A flexible rank aggregation library

User Manual

Version: 1.0.8

Web Site: https://flagr.site/

Github: https://github.com/lakritidis/FLAGR

Python Package Index: https://pypi.org/project/pyflagr/

Table of Contents

1. Getting started

1.1 Introduction

FLAGR is a high performance, modular, open source library for rank aggregation problems. It implements baseline and recent state-of-the-art aggregation algorithms that accept ranked preference lists and generate a single consensus list of elements.

The core project is developed in C++. The source code is available on GitHub and can be compiled as a standard application, or as a shared library. In the second case, the library file can be linked or loaded by other programs in other languages. PyFLAGR is an example of such application. In brief, FLAGR:

- employs efficient data structures and algorithms that ensure high performance,
- is cross-platform supporting Windows and Linux. An extension that supports MacOS users is currently under construction,
- is modular, allowing third-party programmers to easily implement their methods within the core library,
- is open-source.

The current version of FLAGR is 1.0.8. It includes implementations of the following algorithms:

- CombSUM linear combination with 5 different score/rank normalization techniques; namely: Rank, Borda, Simple Borda, Score, and Z-Score normalization (Renda, et al., 2003).
- CombMNZ linear combination with 5 different score/rank normalization techniques; namely: Rank, Borda, Simple Borda, Score, and Z-Score normalization (Renda, et al., 2003).
- Borda Count (equivalent to CombSUM with Borda normalization, (Renda, et al., 2003)).
- Condorcet Winners.
- Copeland Winners.
- Outranking Approach of Farah & Vanderpooten, 2007.
- Distance-based iterative unsupervised algorithm of Akritidis et al., 2022 (all the above methods can be used as the starting non-weighted aggregator).
- Robust Rank Aggregation in two variants: the first one employs the Stuart/Ares method for p-value correction, whereas the other one does not.
- Kemeny optimal aggregation (brute force implementation, not applicable to large, or many input preference lists).
- Markov Chains (MC) methods of Dwork et al., 2001 and DeConde et al., 2006.
- Weighted agglomerative aggregation method of Chatterjee et al., 2018.
- Preference relation unsupervised algorithm of Desarkar et al., 2016.

These methods are also supported by PyFLAGR through a set of modules and classes.

1.2 Download

FLAGR is an open-source project, licensed under the Apache License, version 2.

The library can be downloaded from its GitHub repository at https://github.com/lakritidis/FLAGR. The repository contains:

- The core C++ components (`/src` directory) and rank aggregation algorithm implementations (`/src/ram` directory).
- The dynamic library references (`cflagr.cpp` and `dllflagr.cpp` files), which allow the compilation of FLAGR as a shared or a dynamic link library.
- A precompiled shared library (`pyflagr/pyflagr/flagr.so`, Linux only) and a dynamic link library (`pyflagr/pyflagr/flagr.dll`, Windows only) with its dependencies.
- The PyFLAGR library, which allows the usage of FLAGR in Python applications (`/pyflagr` directory).
- Documentation (`/docs` directory).
- Code examples for C++ and Python (`/examples` directory).

1.3 Compilation and execution

FLAGR can be compiled as a standard console application, or as a shared/dynamic link library in both Linux and Windows systems.

Building FLAGR in Linux

In Linux platforms, the user must navigate to the root directory of the package through the Terminal and execute the makefile that exists there by typing:

```
make
```

The build script compiles the source code and produces two files within the `/bin/Release` directory:

- the binary executable file `/bin/Release/FLAGR`
- the Linux shared library `/bin/Release/flagr.so` that can be linked by third-party applications to obtain access to the FLAGR algorithm implementations.

Building FLAGR in Windows

Similarly to the previous case, in Windows platforms the user must navigate to the root directory of the package through the Command Prompt and execute the batch file that exists there by typing:

```
makefile.bat
```

The build script compiles the source code and produces two files within the `/bin/Release` directory:

- the binary executable file `/bin/Release/FLAGR.exe`
- the Dynamic Link Library library `/bin/Release/flagr.dll` that can be linked by third-party applications to obtain access to the FLAGR algorithm implementations.

Running the FLAGR executable

The FLAGR binary is executed in an identical manner, regardless of the operating system. The application accepts 4 optional arguments in the following fashion:

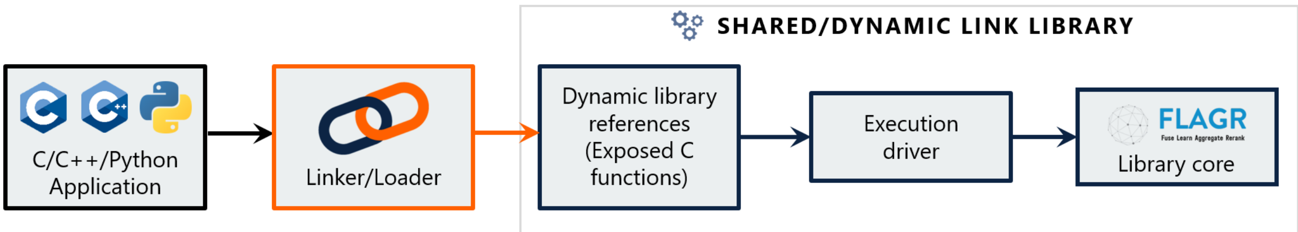
```
FLAGR [cutoff] [input_file] [output_path] [qrels_file]
```

The input arguments are:

- `cutoff`: this is the evaluation cut-off point. That is, the number of items of the aggregate list that will be included in the evaluation process. If nothing is passed, then the value `10` is used.
- `input_file`: The full path to the input file that stores the input lists to be aggregated. This is where the aggregation algorithm/s read data from.
- `output_path`: This is where the program writes the generated aggregate lists and the results of the evaluation process. If nothing is passed, then the default value `output` is used.
- `qrels_file`: This file stores the relevance judgments of the list elements. It is used by FLAGR to evaluate the employed rank aggregation algorithm/s. If nothing is passed, then no evaluation takes place.

1.4 Dynamic library references (Exposed C functions)

FLAGR exposes a set of C functions through an `extern "C"` statement, allowing their linkage from other C programs. These are the dynamic library references to which a client C code can link to. At first, FLAGR must be compiled as a shared/dynamic link library. Then, the client C program can link to that shared library and include a typical C header file that contains just the declaration of these functions. The called function is able to access the C++ FLAGR core through the Execution Driver. The following diagram depicts this scenario:



The FLAGR architecture and the possibility of building it as a shared library allows its usage not only in third party C programs, but also in programs written in other languages. PyFLAGR is an example of such case. PyFLAGR is a Python library that enables the execution of the algorithm C++ implementations of FLAGR from standard Python programs. The Python program must be able to successfully import the PyFLAGR modules.

The exposed C functions of FLAGR exist in two files: `cflagr.cpp` and `dllflagr.cpp`. These files are almost identical: they contain the same functions with exactly the same arguments and bodies. What changes is the usage of several special keywords in the function declarations of `dllflagr.cpp` that enable the building of FLAGR as a DLL for Windows-based systems. These functions are:

- `void Linear()`: It executes one of the supported linear combination methods (CombSUM and CombMNZ, each one with 5 variants).

- `void Condorcet()`: It executes the Condorcet Winners method.
- `void Copeland()`: It executes the Copeland Winners method.
- `void OutrankingApproach()`: It executes the Outranking Approach of Farah & Vanderpooten, 2007.
- `void Kemeny()`: It executes the optimal Kemeny optimal aggregation algorithm (brute force implementation).
- `void RobustRA()`: It executes the Robust Rank Aggregation (RRA) method of Kolde et al., 2012.
- `void DIBRA()`: It executes the distance-based iterative rank aggregation method of Akritidis et al., 2022.
- `void PrefRel()`: It executes the preference relations method of Desarkar et al., 2016.
- `void Agglomerative()`: It executes the agglomerative rank aggregation method of Chatterjee et al., 2018.
- `void MC()`: It executes the Markov Chain-based rank aggregation method of Dwork et al., 2001.

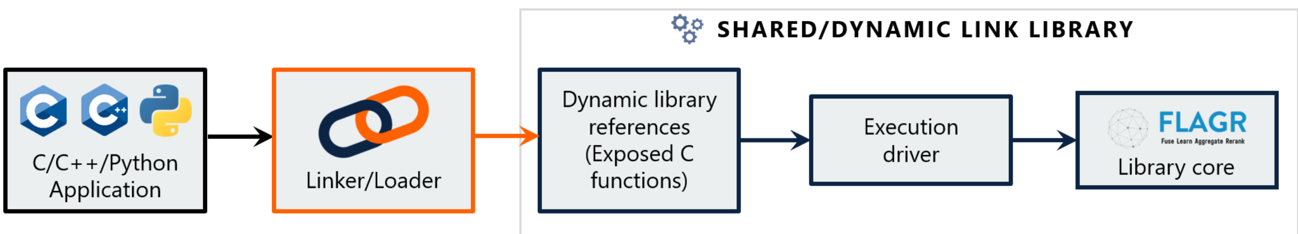
1.5 Execution driver

The Execution Driver is a simple function defined in `driver.cpp`. Its role is to offer a unified manner of executing the exposed C functions. It takes as an argument a simple C structure that stores the user-defined input parameters and algorithm hyper-parameters, and orchestrates the execution flow.

More specifically, the Execution driver initially copies the user-defined input parameters from the aforementioned C structure to an `InputParams` C++ object. In the sequel, it creates an `InputData` object which immediately starts reading the provided input file/s. Next, the `aggregate()` method of `InputData` is called to perform rank aggregation. Notice that the `InputParams` object is propagated to all FLAGR components and carries all the required parameters with it. Therefore, the rank aggregation method to be applied is automatically executed without any further checks, since `InputParams` notifies the `Aggregator` about the aggregation method that was selected by the user.

In case the user provided a valid input file with relevance judgments, the Execution Driver proceeds to the evaluation of the generated aggregate list. This is achieved by calling the `evaluate()` method of `InputData`, which in turn triggers the `Evaluator`.

The following block diagram depicts the role of the execution driver as the connector between the dynamic library references and the C++ FLAGR core.



2. Rank Aggregation methods

This section describes the rank aggregation algorithms implemented in FLAGR. The following articles also contain brief descriptions of the respective exposed C functions that make these algorithms accessible from other programs (in case the FLAGR shared library is used).

2.1 Linear Combination methods – The `Linear()` function

In linear combination methods, the score of each element is computed by summing up the partial scores of that element with respect to its rankings in each input preference list. The `Linear()` function triggers the execution of two such combination methods: CombSUM, and CombMNZ. Both of them are implemented in accordance to the paper of Renda et al., 2003.

Each method is accompanied by an element rank/score normalization technique, as it is described in the aforementioned paper. These techniques are: Rank normalization, Borda normalization, Score normalization, and Z-Score normalization. In FLAGR, there is a fifth normalization technique, called Simple Borda. In contrast to the traditional Borda normalization, Simple Borda assigns zero partial scores in case an element has not been ranked by an input preference list.

In addition, notice that CombSUM with Borda normalization is equivalent to the well-known BordaCount rank aggregation method.

Function Definitions

```
void Linear(const char inf[], const char relf[], const int evpts, const int
ram, const char ranstr[], const char out[])
```

and

```
__declspec(dllexport) void __cdecl Linear(const char inf[], const char relf[],
const int evpts, const int ram, const char ranstr[], const char out[])
```

Implementation Files

- `Linear()` function: `cflagr.cpp` and `dllflagr.cpp`.
- CombSUM: `src/ram/CombSUM.cpp`.
- CombMNZ: `src/ram/CombMNZ.cpp`.

Input arguments

- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const int ram`: The selected rank aggregation method (see the `aggregation_method` parameter in this document for a list of possible values).
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.
- `const char out[]`: The file system location where the output file with the aggregate list will be stored.

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

2.2 Condorcet Winners – The `Condorcet()` function

The `Condorcet()` function executes the Condorcet Winners method. The score of an element r_i is determined by the number of its "victories" against all the other involved elements. A victory for r_i is achieved if the majority of the voters rank r_i higher than another element r_j. Finally, the elements are sorted in decreasing victory order.

Function Definitions

```
void Condorcet(const char inf[], const char relf[], const int evpts, const char ranstr[], const char out[])
```

and

```
__declspec(dllexport) void __cdecl Condorcet(const char inf[], const char relf[], const int evpts, const char ranstr[], const char out[])
```

Implementation Files

- `Condorcet()` function: `cflagr.cpp` and `dllflagr.cpp`.
- Condorcet Winners method: `src/ram/CondorcetWinners.cpp`.

Input arguments

- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.
- `const char out[]`: The file system location where the output file with the aggregate list will be stored.

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

2.3 Copeland Winners – The `Copeland()` function

The `Copeland()` function executes the Copeland Winners method. The score of an element r_i is determined by the number of its "victories" against all the other involved elements. A victory for r_i is achieved if the majority of the voters rank r_i higher than another element r_j. In contrast to the Condorcet method, Copeland Winners assign "half" a victory (i.e., a score of 0.5) to each element of a pair (r_i, r_j) in the case of tie. A tie happens when exactly half of the voters rank r_i higher than r_j and the other half voters rank r_j higher than r_i.

Finally, the elements are sorted in decreasing victory order.

Function Definitions

```
void Copeland(const char inf[], const char relf[], const int evpts, const char
ranstr[], const char out[])
```

and

```
__declspec(dllexport) void __cdecl Copeland(const char inf[], const char relf[],
const int evpts, const char ranstr[], const char out[])
```

Implementation Files

- `Copeland()` function: `cflagr.cpp` and `dllflagr.cpp`.
- Copeland Winners method: `src/ram/CopelandWinners.cpp`.

Input arguments

- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.
- `const char out[]`: The file system location where the output file with the aggregate list will be stored.

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

2.4 Outranking Approach – `OutrankingApproach()`

The Outranking Approach of Farah & Vanderpooten is a majoritarian method that identifies the "winning" elements by performing pairwise comparisons of their individual rankings. The method is implemented in accordance to the following paper:

- Farah, M., Vanderpooten, D., "An outranking approach for rank aggregation in information retrieval", In Proceedings of the 30th ACM Conference on Research and Development in Information Retrieval, pp. 591-598, 2007.

The algorithm is based on four threshold values that introduce different perspectives of the majority criterion. These values are the concordance, discordance, preference, and veto thresholds. The user may pass all of them to FLAGR as hyper-parameters, through the input arguments of the `OutrankingApproach()` function (see below).

Function Definitions

```
void OutrankingApproach(const char inf[], const char relf[], const int evpts,
const char ranstr[], const char out[], const float pref_t, const float veto_t,
const float conc_t, const float disc_t)
```

and

```
__declspec(dllexport) void __cdecl OutrankingApproach(const char inf[], const
char relf[], const int evpts, const char ranstr[], const char out[], const float
pref_t, const float veto_t, const float conc_t, const float disc_t)
```

Implementation Files

- The `OutrankingApproach()` C function: `cflagr.cpp` and `dllflagr.cpp`.
- Algorithm Implementation: `src/ram/OutrankingApproach.cpp`.

Input arguments

- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.
- `const char out[]`: The file system location where the output file with the aggregate list will be stored.
- `const float pref_t`: Algorithm hyper-parameter - The value of the preference threshold.
- `const float veto_t`: Algorithm hyper-parameter - The value of the veto threshold.
- `const float conc_t`: Algorithm hyper-parameter - The value of the concordance threshold.
- `const float disc_t`: Algorithm hyper-parameter - The value of the discordance threshold.

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

2.5 Markov Chains methods – The `MC()` function

The Markov Chains methods constitute a well-established family of rank aggregation methods. Originally proposed by Dwork et al., (2001), they consider an aggregate list as a system that moves from one state to another with respect to a particular criterion. Dwork et al. (2001) introduced four such methods in the following article:

- C. Dwork, R. Kumar, M. Naor, D. Sivakumar, "Rank Aggregation Methods for the Web", In Proceedings of the 10th International Conference on World Wide Web, pp. 613-622, 2001.

In addition, DeConde et al. (2006) introduced MCT, a variant that constructs the transition matrix by considering the proportion of lists which prefer an element over another.

- R.P. DeConde, S. Hawley, S. Falcon, N. Clegg, B. Knudsen, R. Etzioni, "Combining results of microarray experiments: a rank aggregation approach", Statistical Applications in Genetics and Molecular Biology, vol. 5, no. 1, 2006.

The execution of all five methods takes place by passing the appropriate arguments to the MC exposed C function of FLAGR.

Function Definitions

```
void MC(const char inf[], const char relf[], const int evpts, const int ram,
const char ranstr[], const char out[], const float ergodic_number, const float
delta, const int iter)
```

and

```
__declspec(dllexport) void __cdecl MC(const char inf[], const char relf[], const
int evpts, const int ram, const char ranstr[], const char out[], const float
ergodic_number, const float delta, const int iter)
```

Implementation Files

- `MC()` function: `cflagr.cpp` and `dllflagr.cpp`.
- Markov Chains implementation: `src/ram/MC.cpp`.

Input arguments

- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const int ram`: The selected (Markov Chains-based) rank aggregation method (`801`, `802`, `803`, `804`, or `805` - see the `aggregation_method` parameter in this document).
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.
- `const char out[]`: The file system location where the output file with the aggregate list will be stored.
- `const float ergodic_number`: The ergodic number, used during the computation of the ergodic transition matrix from the normalized transition matrix.
- `const float iter`: Controls the maximum number of iterations before convergence.

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

2.6 Kemeny Optimal Aggregation – The `Kemeny()` function

This function executes Kemeny optimal aggregation. This algorithm identifies the optimal aggregate list as the list that minimizes its distance from all the input preference lists.

Kemeny optimal aggregation is an NP-hard problem, with very high computational complexity. It requires the computation of all permutations of the input items and the calculation of the distances of each permutation from all input lists. *The brute force solution becomes infeasible when the number of elements gets greater than 15-20, or the number of input lists is greater than 4*, so caution is advised.

Function Definitions

```
void Kemeny(const char inf[], const char relf[], const int evpts, const char
ranstr[], const char out[])
```

and

```
__declspec(dllexport) void __cdecl Kemeny(const char inf[], const char relf[],
const int evpts, const char ranstr[], const char out[])
```

Implementation Files

- `Kemeny()` function: `cflagr.cpp` and `dllflagr.cpp`.
- Kemeny optimal aggregation method: `src/ram/KemenyOptimal.cpp`.

Input arguments

- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.
- `const char out[]`: The file system location where the output file with the aggregate list will be stored.

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

2.7 Robust Rank Aggregation – The `RobustRA()` function

This function executes the Robust Rank Aggregation (RRA) method of Kolde et al., 2012. The method is implemented in accordance to the following paper:

- R. Kolde, S. Laur, P. Adler, J. Vilo, "Robust rank aggregation for gene list integration and meta-analysis", Bioinformatics, vol. 28, no. 4, pp. 573-580, 2012.

RRA is mostly used in bio-informatics applications to aggregate gene lists. It is based on a probabilistic model (beta distribution) that makes the algorithm parameter free and robust to outliers, noise and errors. The FLAGR C++ implementation of RRA produces the same results as the R implementation of Kolde (see the RobustRankAggreg R package).

The computation of the incomplete beta function is performed with the John Burkardt's implementation of ASA063 algorithm (K.L. Majumder and G. Bhattacharjee):

- K.L. Majumder, G.P. Bhattacharjee, "Algorithm AS 63: The incomplete Beta Integral", Applied Statistics, vol. 22, no. 3, pp. 409-411, 1973.

Furthermore, the computation of the inverse of the incomplete beta function is performed with the John Burkardt's implementation of ASA109 algorithm (GW Cran, KJ Martin and GE Thomas):

- G.W. Cran, M.J. Martin, G.E. Thomas, "Remark AS R19 and Algorithm AS 109: A Remark on Algorithms AS 63: The Incomplete Beta Integral and AS 64: Inverse of the Incomplete Beta Integeral", Applied Statistics, Volume 26, Number 1, 1977, pages 111-114.

Function Definitions

```
void RobustRA(const char inf[], const char relf[], const int evpts, const char ranstr[], const char out[], const bool exact)
```

and

```
__declspec(dllexport) void __cdecl RobustRA(const char inf[], const char relf[], const int evpts, const char ranstr[], const char out[], const bool exact)
```

Implementation Files

- `RobustRA()` function: `cflagr.cpp` and `dllflagr.cpp`.
- RRA implementation: `src/ram/RobustRA.cpp`.
- Incomplete beta function implementation (ASA 063 algorithm), inverse of the incomplete Beta function implementation (ASA 109 algorithm): `src/ram/tools/BetaDistribution.cpp`.

Input arguments

- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.

- `const char out[]`: The file system location where the output file with the aggregate list will be stored.
- `const bool exact`: If true the Stuart algorithm for p-value correction is applied.

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

2.8 Iterative Distance-Based weighted aggregation – The `DIBRA()` function

This function executes the iterative, distance-based method (abbreviated DIBRA) of Akritidis el. al 2022. The method is implemented in accordance to the following paper:

- L. Akritidis, A. Fevgas, P. Bozanis, Y. Manolopoulos, "An Unsupervised Distance-Based Model for Weighted Rank Aggregation with List Pruning", Expert Systems with Applications, vol. 202, pp. 117435, 2022.

DIBRA belongs to the weighted rank aggregation methods. It employs exploratory analysis to automatically identify the expert voters in an unsupervised fashion. Then, it assigns higher weights to the voters who were identified as experts, thus boosting the scores of their submitted elements.

In particular, DIBRA employs a standard non-weighted method to generate an initial aggregate ranking (see `aggregation_method` in this document for a list of the supported methods). Then, it repeatedly assigns higher weights to the input lists that have smaller distances from the aggregate lists. The process terminates when the voter weights converge and a stable aggregate list is obtained.

The algorithm also includes an optional list pruning mechanism that arranges the input list lengths according to the respective voter weights.

Function Definitions

```
void DIBRA(const char inf[], const char relf[], const int evpts, const int agg,
const char ranstr[], const char out[], const int wnorm, const int dist, const
bool prune, const float gamma, const float d1, const float d2, const float tol,
const int iter, const float pref_t, const float veto_t, const float conc_t,
const float disc_t)
```

and

```
__declspec(dllexport) void __cdecl DIBRA(const char inf[], const char relf[],
const int evpts, const int agg, const char ranstr[], const char out[], const
int wnorm, const int dist, const bool prune, const float gamma, const float d1,
const float d2, const float tol, const int iter, const float pref_t, const float
veto_t, const float conc_t, const float disc_t)
```

Implementation Files

- `DIBRA()` function: `cflagr.cpp` and `dllflagr.cpp`.
- DIBRA method implementation: `src/ram/DIBRA.cpp`.

Input arguments

- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const int agg`: The selected non-weighted base rank aggregation method (see the aggregation_method parameter in this document for a list of possible values).
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.
- `const char out[]`: The file system location where the output file with the aggregate list will be stored.
- `const int wnorm`: The voter weights normalization method (see the `weights_normalization` parameter in this document for a list of possible values).
- `const int dist`: The correlation method that is used to measure the distance between an input list and the temporary aggregate list (see the `correlation_method` parameter in this document for a list of possible values).
- `const bool prune`: Triggers a weight-dependent list pruning mechanism.
- `const float gamma`: The γ hyper-parameter.
- `const float d1`: The δ_1 hyper-parameter (applicable when `prune=true`).
- `const float d2`: The δ_2 hyper-parameter (applicable when `prune=true`).
- `const float tol`: Controls the convergence precision. This tolerance threshold represents the minimum precision of the difference between the voter weight in an iteration and the voter weight of the previous iteration.
- `const int iter`: Controls the maximum number of iterations.
- `const float pref_t`: The preference threshold (applicable when the Outranking Approach is selected as the non-weighted base method; namely, `agg=5300`).
- `const float veto_t`: The veto threshold (applicable when the Outranking Approach is selected as the non-weighted base method; namely, `agg=5300`).
- `const float conc_t`: The concordance threshold (applicable when the Outranking Approach is selected as the non-weighted base method; namely, `agg=5300`).
- `const float disc_t`: The discordance threshold (applicable when the Outranking Approach is selected as the non-weighted base method; namely, `agg=5300`).

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

2.9 Preference Relations weighted method

This function executes the Preference Relations weighted rank aggregation method of Desarkar et al., 2016. The method is implemented in accordance to the following paper:

- M.S. Desarkar, S. Sarkar, P. Mitra, "Preference relations based unsupervised rank aggregation for metasearch", Expert Systems with Applications, vol. 49, pp. 86-98, 2016.

The Preference Relations algorithm belongs to the weighted rank aggregation methods. It employs exploratory analysis to automatically identify the expert voters in an unsupervised fashion. Then, it assigns higher weights to the voters who were identified as experts, thus boosting the scores of their submitted elements.

The method constructs a preference relations graph which contains the individual elements as vertices and their weights as edges.

Function Definitions

```
void PrefRel(const char inf[], const char relf[], const int evpts, const char
ranstr[], const char out[], const float alpha, const float beta)
```

and

```
__declspec(dllexport) void __cdecl PrefRel(const char inf[], const char relf[],
const int evpts, const char ranstr[], const char out[], const float alpha, const
float beta)
```

Implementation Files

- `PrefRel()` function: `cflagr.cpp` and `dllflagr.cpp`.
- Preference Relations method implementation: `src/ram/PrefRel.cpp`.
- Helper class `MergedItemPair` implementation: `src/ram/tools/MergedItemPair.cpp`.

Input arguments

- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.
- `const char out[]`: The file system location where the output file with the aggregate list will be stored.
- `const float alpha`: The α hyper-parameter.
- `const float beta`: The β hyper-parameter.

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

2.10 Agglomerative weighted aggregation

This function executes the Agglomerative weighted rank aggregation method of Chatterjee et al., 2018. The method is implemented in accordance to the following paper:

- S. Chatterjee, A. Mukhopadhyay, M. Bhattacharyya, "A weighted rank aggregation approach towards crowd opinion analysis", Knowledge-Based Systems, vol. 149, pp. 47-60, 2018.

The Agglomerative Aggregation algorithm belongs to the weighted rank aggregation methods. It employs exploratory analysis to automatically identify the expert voters in an unsupervised fashion. Then, it assigns higher weights to the voters who were identified as experts, thus boosting the scores of their submitted elements.

This method works very similarly to the well-established agglomerative clustering algorithm. Specifically, it repeatedly merges the two most similar input lists into a temporary aggregate list. During list merging, it modifies the weights of the respective voters, thus affecting the future merges.

Function Definitions

```
void Agglomerative(const char inf[], const char relf[], const int evpts, const
char ranstr[], const char out[], const float c1, const float c2)
```

and

```
__declspec(dllexport) void __cdecl Agglomerative(const char inf[], const char
relf[], const int evpts, const char ranstr[], const char out[], const float c1,
const float c2)
```

Implementation Files

- `Agglomerative()` function: `cflagr.cpp` and `dllflagr.cpp`.
- Agglomerative Aggregation method implementation: `src/ram/Agglomerative.cpp`.

Input arguments

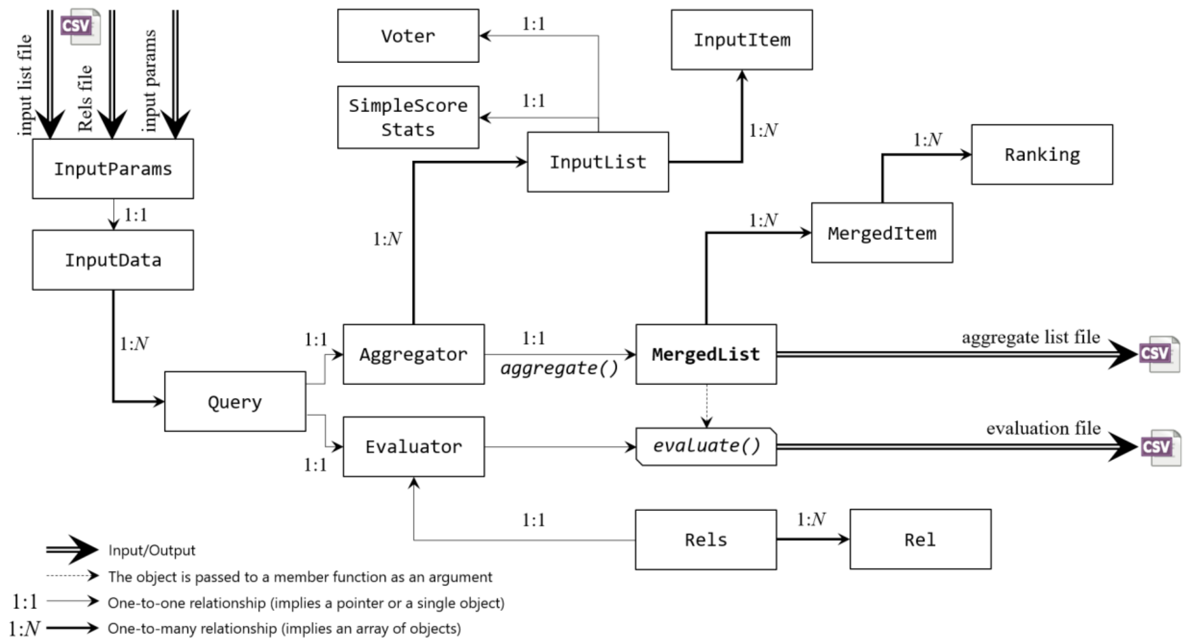
- `const char inf[]`: The path of the input file that stores the preference lists to be aggregated.
- `const char relf[]`: The path of the input file that stores the element relevance judgments.
- `const int evpts`: the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed.
- `const char ranstr[]`: A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.
- `const char out[]`: The file system location where the output file with the aggregate list will be stored.
- `const float c1`: The c_1 hyper-parameter.
- `const float c2`: The c_2 hyper-parameter.

Description

The input parameters are parsed and stored in a special C structure called `UserParams` that is defined in `src/InputParams.h`. Then, `UserParams` is passed to the execution driver and the rank aggregation process starts.

3. FLAGR API Documentation

In this section the FLAGR library core is presented. The following Figure depicts a block diagram of the core's architecture.



In the following subsections, the format of the input and output files is described. Then, the participating classes of FLAGR are presented. The final Subsection contains a detailed guide on how custom algorithm implementations can be integrated to FLAGR.

3.1 Input and Output files

Input 1: The file of the input preference lists

FLAGR requires that the input preference lists to be aggregated are stored in a single CSV file, regardless of the number of the involved topics (queries) or voters (rankers). The columns of this CSV file must be organized in the following manner:

```
Query/Topic String, Voter Name, Item Code, Item Score, Algorithm/Dataset
```

where:

- `Query/Topic`: the query string or the topic for which the preference list is submitted.
- `Voter`: the name of the voter, or the ranker who submits the preference list for the specified Query/Topic.
- `Item Code`: a unique name that identifies a particular element of the preference list. A voter cannot submit the same element for the same query/topic two or more times. This means that each element appears exactly once in each preference list. However, the same element may appear in lists submitted by other voters.

- `Item Score`: the preference score assigned to an item by a specific voter. It reflects the importance (or the relevance, or the weight) of the element. In many cases (e.g., search engine rankings), the preference scores are unknown. In such cases, the scores can be replaced by the (reverse) ranking of an item in such a manner that the top rankings receive higher scores than the ones that have been assigned lower rankings.
- `Algorithm/Dataset`: A user-defined string that usually represents the origin of a particular preference list. It may receive any non-blank value.

You may find an example of an input list CSV file here. This example file contains the preference lists that were submitted by 50 voters for 20 queries. Each input list contains 30 elements. Therefore, the number of rows in this file is equal to 50·20·30=30000.

Output 1: The file of the aggregate list/s

In this file FLAGR stores the result (output) of the selected rank aggregation method. Namely, the final lists that derive after the aggregation of the input preference lists. The library creates one aggregate list per input query/topic. So, if there are Q input queries, FLAGR generates Q aggregate lists and stores them in a CSV file. Each row in the file represents an element of the aggregate list stored in decreasing score order. The columns are organized as follows:

```
Query/Topic String, Voter Name, Item Code, Item Score
```

Input 2: The file of relevant elements (or, the Rels file)

Optionally, the user may provide a second CSV file (we call it Rels file) that contains relevance judgments for the preference list elements of the primary input file for each query. The Rels file is employed by the FLAGR's evaluation module to evaluate each created aggregate list. Its columns must be formatted as follows:

```
Query/Topic String, 0, Item Code, Relevance Score
```

where:

- `Query/Topic`: the query string or the topic for which the list is submitted.
- `0`: unused. This value *must be always* 0.
- `Item Code`: a unique name that identifies a particular element. There cannot be two relevance judgments for the same element for the same query.
- `Relevance Score`: An integer value that represents the relevance of the item with respect to the mentioned Query/Topic. Typically, zero values represent irrelevant and incorrect elements, negative values represent spam elements; and positive values represent relevant, correct and informative elements.

You may find an example of an input Rels file here. This example file contains the relevance judgments for the elements of all preference lists for all queries of the previous input list file. *Notice that in case FLAGR does not find a relevance judgment for an element, then it automatically considers it as irrelevant (that is, it sets its Relevance Score equal to 0).*

Output 2: The evaluation file

As soon as a valid Rels file is provided, the evaluation process takes place automatically. In this case FLAGR evaluates each aggregate list individually and outputs a second CSV file, where it writes the results of the evaluation.

If there are Q input queries, then Q aggregate lists are generated and the evaluation file contains $Q + 1$ rows. The first Q rows store the evaluation metrics for each aggregate list, whereas the last row contains the average values. On the other hand, the columns of the evaluation file depend on the eval_pts parameter that is set by the user. More specifically, the columns are $6 + 4 \cdot \text{eval\_pts}$:

```
q, num_ret, num_rel, num_rel_ret, ap, P@1, ..., P@eval_pts, R@1, ...,
R@eval_pts, D@1, ..., D@eval_pts, N@1, ..., N@eval_pts, ram
```

where:

- q is the query string.
- num_ret is the length (i.e. the number of elements in the) aggregate list.
- num_rel is the total number of relevant elements for this query.
- num_rel_ret is the number of relevant elements included in the aggregate list.
- ap is the Average Precision for a specific aggregate list w.r.t q.
- P@X is the running Precision at the X-th element of the aggregate list.
- R@X is the running Recall at the X-th element of the aggregate list..
- D@X is the running Discounted Cumulative Gain (DCG) at the X-th item of the aggregate list.
- N@X is the running normalized Discounted Cumulative Gain (nDCG) at the X-th item.
- ram is the name of the applied rank aggregation method.

3.2 Aggregator class

The Aggregator class triggers the execution of a rank aggregation algorithm on a collection (array) of InputLists. Typically, the output of the aggregation process is a single MergedList object.

Implementation files

The Aggregator class is defined in the src/Aggregator.h header file; its member functions are implemented in src/Aggregator.cpp.

Technical Details

The input_lists member variable stores the input lists to be aggregated. Technically, it is an array (i.e. double pointer) of InputList objects. The allocated size of this array is equal to num_alloc_lists, whereas the real number of non-null input lists is num_lists. Naturally, it must always hold that num_alloc_lists ≥ num_lists.

The rank aggregation process takes places inside a Query object. For this reason, a Query object contains a pointer to a single Aggregator object (see the respective block diagram).

The aggregate() member function is responsible for executing the rank aggregation procedure. It accepts an InputParams object that stores the selected rank aggregation method, the hyper-parameter values and the execution parameters and returns a single MergedList.

3.3 Evaluator class

The Evaluator class evaluates the quality of an aggregate list with respect to a given set of judgments that determine the relevance of (some, or all) list elements. The set of relevance judgments is stored in a Rels object. The results of the evaluation (i.e. evaluation measures) are written in a CSV file.

Implementation files

The `Evaluator` class is defined in the `src/Evaluator.h` header file; its member functions are implemented in `src/Evaluator.cpp`.

Technical Details

The evaluation process takes place inside a `Query` object, provided that a file of relevance judgments is provided to FLAGR (see more in the article about Input and Output files). For this reason, a `Query` object contains a pointer to a single `Evaluator` object.

The evaluation process is performed by the `evaluate()` member function. The procedure is based on the `Rels` object that contains the aforementioned relevance judgments. Finally, the four following arrays are created:

- `precision`: in its i-th position, it stores the value of the Precision measure at the i-th element of the aggregate list.
- `recall`: in its i-th position, it stores the value of the Recall measure at the i-th element of the aggregate list.
- `dcg`: in its i-th position, it stores the value of the Discounted Cumulative Gain (DCG) measure at the i-th element of the aggregate list.
- `ndcg`: in its i-th position, it stores the value of the normalized Discounted Cumulative Gain (nDCG) measure at the i-th element of the aggregate list.

3.4 `InputItem` class

An `InputItem` is a single element that is read from the input file. It is a part of an `InputList` and possesses three properties:

- A unique string identifier that is used by FLAGR to identify the common elements among all `InputLists`,
- Its (integer) ranking in the input preference list, and
- An optional score value that is assigned by the associated voter and justifies its ranking in the input list. The score data type can be either `float`, or `double`; this is determined by the `score_t` data type definition in `driver.cpp`.

Implementation files

The `InputItem` class is defined in the `src/InputItem.h` header file; its member functions are implemented in `src/InputItem.cpp`.

3.5 `InputList` class

This class is used to store and represent an input preference list submitted by a voter. It consists of an array of `InputItem` objects and it is connected to the voter who submitted it via a pointer that points to a `Voter` object.

The input lists are read from the input file by using an `InputData` object. The entire collection of them is handled by an `Aggregator`.

Implementation files

The `InputList` class is defined in the `src/InputList.h` header file; its member functions are implemented in `src/InputList.cpp`.

Technical Details

The actual number of elements in an `InputList` is `num_items`; the allocated memory is `num_alloc_items`. Naturally it derives that `num_alloc_items` must always be greater than, or equal to `num_items`.

3.6 `InputParams` class

The `InputParams` class stores options and execution parameters that have been passed to FLAGR by the user. These parameters concern input and output file locations, rank aggregation methods, algorithm hyper-parameters, etc. See the table below for a complete list of the supported parameters and their respective valid values.

This object is passed as an argument to multiple functions of FLAGR including the implementations of the rank aggregation methods. This is how these implementations get access to the user-defined hyper-parameters.

Implementation files

The `InputParams` class is defined in the `src/InputParams.h` header file; its member functions are implemented in `src/InputParams.cpp`.

Details

The supported parameters include:

Parameter	Data Type	Description
`input_file`	String (ASCII)	The path of the input CSV file that contains the preference lists to be aggregated.
`rels_file`	String (ASCII)	The path of the optional CSV file that contains the relevance judgments of the input list elements. If set, it automatically triggers the evaluation process of the generated aggregate list. Otherwise, no evaluation takes place.
`output_file`	String (ASCII)	The file system location where the output file with the aggregate list will be stored. If left empty, the default OS temp directory is used.
`eval_file`	String (ASCII)	The file system location where the output file with the evaluation of the aggregate list will be stored. If left empty, the default OS temp directory is used.
`random_string`	String (ASCII)	A string that is embedded in the names of the output files. Used when FLAGR is compiled as a shared library.

`aggregation_method`	Integer	Determines the algorithm that will be used to perform rank aggregation. The valid values are: `100`: for CombSUM with Borda normalization `101`: for CombSUM with Rank normalization `102`: for CombSUM with Score normalization `103`: for CombSUM with Z-Score normalization `104`: for CombSUM with Simple Borda normalization `110`: for CombMNZ with Borda normalization `111`: for CombMNZ with Rank normalization `112`: for CombMNZ with Score normalization `113`: for CombMNZ with Z-Score normalization `114`: for CombMNZ with Simple Borda normalization `200`: for the Condorcet Winners method `201`: for the Copeland Winners method `300`: for the Outranking Approach `400`: for Kemeny optimal aggregation `401`: for Robust Rank Aggregation (RRA) `5100`: for DIBRA with CombSUM and Borda Normalization `5101`: for DIBRA with CombSUM and Rank Normalization `5102`: for DIBRA with CombSUM and Score Normalization `5103`: for DIBRA with CombSUM and Z-Score Normalization `5104`: for DIBRA CombSUM and Simple Borda Normalization `5110`: for DIBRA with CombMNZ and Borda Normalization `5111`: for DIBRA with CombMNZ and Rank Normalization `5112`: for DIBRA with CombMNZ and Score Normalization `5113`: for DIBRA with CombMNZ and Z-Score Normalization `5114`: for DIBRA CombMNZ and Simple Borda Normalization `5200`: for DIBRA with the Condorcet Winners method `5201`: for DIBRA with the Copeland Winners method `5300`: for DIBRA with the Outranking Approach `600`: for the Preference Relations Method `700`: for the Agglomerative Aggregation Method `801`: for Markov Chains 1 (MC1) `802`: for Markov Chains 2 (MC2) `803`: for Markov Chains 3 (MC3) `804`: for Markov Chains 4 (MC4) `805`: for MCT
`correlation_method`	Integer	The correlation method that is used to measure the distance between an input list and the temporary aggregate list. The valid values are: 1: for the Spearman's ρ correlation coefficient. 2: for the scaled variant of Spearman's Footrule distance. 3: for Cosine similarity of the lists' vector representations. 5: for the Kendall's τ correlation coefficient.

`weights_normalization`	Integer	The voter weights normalization method. Used when the DIBRA algorithm is selected. The valid values are: 1: for no voter weight normalization. 2: for normalizing the voter weights with min-max scaling. 3: for z-normalizing the voter weights.
`max_iterations`	Integer	This parameter controls the maximum number of iterations. FLAGR will stop the execution of DIBRA if the requested number of iterations have been performed, even if the voter weights have not fully converged.
`max_list_items`	Integer	Limits the length of the input preference lists.
`eval_points`	Integer	Determines the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed. For example, for `eval_pts=10` FLAGR will compute $P@1, P@2, \dots, P@10$ and $N@1, N@2, \dots, N@10$.
`list_pruning`	Boolean	Triggers a weight-dependent list pruning mechanism. Used in combination with the DIBRA weighted method only.
`convergence_precision`	Float or Double	Controls the convergence precision. This tolerance threshold represents the minimum precision of the difference between the voter weight in an iteration and the voter weight of the previous iteration. Used in combination with the DIBRA weighted method only.
`alpha`	Float or Double	The α hyper-parameter of the Preference Relations method.
`beta`	Float or Double	The β hyper-parameter of the Preference Relations method.
`gamma`	Float or Double	The γ hyper-parameter of DIBRA.
`c1`	Float or Double	The c_1 hyper-parameter of the Agglomerative Aggregation method.
`c2`	Float or Double	The c_2 hyper-parameter of the Agglomerative Aggregation method.
`pref_thr`	Float or Double	The preference threshold of the Outranking Approach. It takes values in the range [0,1].
`veto_thr`	Float or Double	The veto threshold of the Outranking Approach. It takes values in the range [0,1].
`conc_thr`	Float or Double	The concordance threshold of the Outranking Approach. It takes values in the range [0,1].
`disc_thr`	Float or Double	The discordance threshold of the Outranking Approach. It takes values in the range [0,1].

3.7 InputData class

This class is responsible for reading and parsing the input data file/s. For the moment, FLAGR accepts only CSV-formatted input files.

Implementation files

The `InputData` class is defined in the `src/input/InputData.h` header file; its member functions are implemented in `src/input/InputData.cpp` and `src/input/InputDataCSV.cpp`.

Input list files

Detailed information about how the input files must be formatted can be found here.

Also notice that FLAGR is designed to accept data directly from RASDaGen, a synthetic dataset generator for rank aggregation problems.

Technical Details

The role of `InputData` is broader and it is not limited to just reading the input files. More specifically, during the input file parsing process, one or more `Query` objects are constructed. In the sequel, inside each `Query` the corresponding `Aggregator` and `Evaluator` objects are created to initialize and evaluate the aggregation process.

There is also a pointer called `params` that connects `InputData` with `InputParams`. In this manner, `InputData` is able to subsequently pass user-defined algorithm hyper-parameters and several other execution parameters to the rest of the application components.

3.8 MergedItem class

A `MergedItem` is an element of an aggregate `MergedList`. The class derives from `InputItem` and inherits all its member variables and functions.

Implementation files

The MergedItem class is defined in the `src/MergedItem.h` header file; its member functions are implemented in `src/MergedItem.cpp`.

Technical Details

`MergedItem` maintains an array of `Ranking` objects that store the individual rankings (and scores) of the element in each preference `InputList`. Notice that the number of elements of this array (`num_alloc_rankings`) is always equal to the number of the input preference lists. In case the associated `InputItem` has not been ranked by a preference list (i.e. it is not included in the list), then its corresponding ranking in the `Rankings` array is set equal to `NOT_RANKED_ITEM_RANK` (defined in `driver.cpp`). The actual number of the input preference lists that include a particular `MergedItem` is stored in the `num_rankings` member variable. This approach consumes more memory, but enables $O(1)$ constant search times of the ranking of a particular item on a particular `InputList`.

The class also includes a self-class next pointer that points to another `MergedItem` object. This allows the storage of a collection of `MergedItems` in dynamic data structures, e.g. linked lists. Such structures are employed in `MergedList`, where `MergedItems` are stored in a hash table with linked lists as chains.

3.9 `MergedList` class

The result of the rank aggregation process is an aggregate list that is stored in a `MergedList` object. `MergedList` contains a collection of `MergedItems`, typically sorted in decreasing score order. The score is assigned by a rank aggregation method.

Regarding the evaluation, the generated aggregate list is fed to the `evaluate()` function of an `Evaluator`. The results are written in a CSV file according to this document.

Implementation files

The `MergedList` class is defined in the `src/MergedList.h` header file; its member functions are implemented in `src/MergedList.cpp`. The implementations of the supported rank aggregation methods are stored in the `/ram` directory. More specifically:

- `src/ram/Agglomerative.cpp` implements the Agglomerative Aggregation method of Chatterjee et al., 2018.
- `src/ram/CombMNZ.cpp` implements the CombMNZ linear combination methods as they are described in the paper of Renda et al., 2003.
- `src/ram/CombSUM.cpp` implements the CombSUM linear combination methods (including Borda Count) as they are described in the paper of Renda et al., 2003.
- `src/ram/CondorcetWinners.cpp` implements one of the oldest approaches to rank aggregation, the Condorcet criterion method.
- `src/ram/CopelandWinners.cpp` implements the method of Copeland Winners which is a variant of the Condorcet Winners method.
- `src/ram/CustomMethods.cpp` contains two sample functions that facilitate the integration of custom rank aggregation methods.
- `src/ram/DIBRA.cpp` implements the Iterative Distance-Based Weighted method of Akritidis et al., 2022.
- `src/ram/KemenyOptimal.cpp` contains the brute force implementation of Kemeny optimal aggregation.
- `src/ram/MC.cpp` implements the four Markov Chains method of Dwork et al., 2001 and the MCT method of DeConde et al., 2006.
- `src/ram/OutrankingApproach.cpp` implements the Outranking Approach of Farah and Vanderpooten, 2007.
- `src/ram/PrefRel.cpp` implements the Preference Relations Weighted method of Desarkar et al., 2016.
- `src/ram/RobustRA.cpp` implements the Robust Rank Aggregation method (RRA) of Kolde et al., 2012.

For more details, please visit the Publications section, or follow the links in the Introduction section. A guide on how custom rank aggregation implementations can be integrated to FLAGR is given here.

Technical Details

The elements of `MergedList` are organized in two ways. More specifically:

- A hash table (hash_table member variable) with linked lists as chains (for collision resolution) is employed to support fast fusion of the individual InputLists. The contents of this hash table are MergedItem objects; the search keys are the unique identifiers of the associated InputItems (MergedItem inherits the members of InputItem).
- A typical array of MergedItem pointers (item_list member variable) that is used to sort the objects in decreasing score order.

3.10 Query class

A Query represents a topic for which a set of voters or rankers submit their preference lists. For example, a Query may be a simple question like "*who is the best football player for 2022?*", or a more complex structure like a sequence of genes.

Implementation files

The Query class is defined in the src/Query.h header file; its member functions are implemented in src/Query.cpp.

Technical Details

The role of this object is central in the entire rank aggregation process, since it connects the input preference lists and the output aggregate list. In particular, a Query is connected to:

- an Aggregator that triggers the execution of a rank aggregation algorithm on the collection of the InputLists, and
- an Evaluator the evaluates the quality of the generated aggregate list with respect to a set of relevance judgments.

3.11 Ranking class

This is a simple class that stores the ranking information of a MergedItem in a particular input preference list. An array of Ranking objects is maintained inside each MergedItem.

Implementation files

The Ranking class is defined in the src/Ranking.h header file; its member functions are implemented in src/Ranking.cpp.

Technical Details

A Ranking class comprises three members:

- A pointer to the corresponding preference InputList,
- The (integer) ranking in the preference InputList, and
- The score value that is assigned by the voter to this element. The score data type can be either float, or double; this is determined by the score_t data type definition in driver.cpp.

3.12 `Rel` class

A `Rel` object contains a relevance judgement about a list element. It is read from a special input CSV file and it is used by an `Evaluator` to compute several evaluation measures about the generated aggregate list.

A `Rel` object is a member of a `Rels` collection, which in turn is referenced by an `Evaluator`.

Implementation files

The `Rel` class is defined in the `src/Rel.h` header file; its member functions are implemented in `src/Rel.cpp`.

Technical Details

The `Rel` class consists of the following member variables:

- a string that represents the unique identifier of a `MergedItem`,
- an integer relevance judgment. The higher the value of this variable, the more relevant/important the item is considered.
- a `next` pointer to another `Rel` object that allows the creation of dynamic data structures (e.g. linked lists of `Rel` objects, etc.).

3.13 `Rels` class

`Rels` contains the relevance judgements that are required by the `Evaluator` in order to evaluate the quality of the generated aggregate list.

The relevance judgments are provided as an input to the library via a special CSV file.

Implementation files

The `Rels` class is defined in the `src/Rels.h` header file; its member functions are implemented in `src/Rels.cpp`.

Technical Details

Notice that the `Evaluator` contains a pointer to a `Rels` object. In this way, the `Evaluator` can quickly access the required relevance judgments during the evaluation procedure.

The `Rels` object is implemented as a standard hash table with the string item identifiers being its search key. The hash values are computed by the hash function of Daniel J. Bernstein, whereas the collisions are resolved by using chains in the form of linked lists.

3.14 `SimpleScoreStats` class

`SimpleScoreStats` is a very simple class that stores several score statistics. In the current FLAGR implementations, it is used exclusively for storing statistical information about the generated aggregate lists. More specifically, the minimum, maximum, and mean score values are stored there, including their standard deviation.

Implementation files

The SimpleScoreStats class is defined in the src/SimpleScoreStats.h header file; its member functions are implemented in src/SimpleScoreStats.cpp.

3.15 Voter class

A voter (also called ranker, or source), submits preferences for one or more topics (queries) in the form of a ranked preference list. The preference lists of all voters are subsequently aggregated by a rank aggregation method in order to generate a single consensus ranking. In FLAGR, a Voter submits a single InputList for each Query.

A Voter object possesses two properties:

- A unique string identifier that represents the voter's name, and
- A weight value that reflects the importance (degree of expertise) of the voter for a particular query. Non-weighted rank aggregation methods consider that all voters are equivalent. Therefore, their lists are processed in an identical manner. In contrast, the weighted methods apply unsupervised learning techniques and exploratory analysis to automatically determine the significance of each voter. The weight data type can be either float, or double; this is determined by the score_t data type definition in driver.cpp.

Implementation files

The Voter class is defined in the src/Voter.h header file; its member functions are implemented in src/Voter.cpp.

3.16 Integrating custom methods

This step-by-step guide describes how custom rank aggregation methods can be implemented and integrated into FLAGR. If you intend to implement fewer than three methods, then several of the steps below are already implemented in FLAGR. For three or more custom methods, additional actions must be performed.

Implementing your own method/s

Custom methods must be implemented as C++ functions in the src/ram/CustomMethods.cpp file. This file already contains two such functions: CustomMethod1() and CustomMethod2(). In case you desire to implement more methods (e.g., CustomMethod3(), etc.), then you must implement them similarly in that file.

Typically, a rank aggregation method assigns scores to the list elements and then, it sorts the elements in either increasing, or decreasing score order. For this reason, the last step in your implementation must be the sorting of the elements of MergedList. Observe that CustomMethod1() and CustomMethod2() contain a call to qsort (QuickSort) in order to perform the required sorting.

Each algorithm implementation (including the built-in ones) takes three arguments:

- An array of the input preference lists (`class InputList ** inlists`). Most rank aggregation methods do not require access to this array, since when the function is called, the input lists have already been merged in the `MergedList` object. However, several methods require the computation of list distances (e.g. DIBRA, Agglomerative) and the `inlists` pointer provides access to this array.
- A pointer to a `SimpleScoreStats` object in case you desire to store score statistics (max, min, mean, etc.).
- A pointer to the `InputParams` object that contains the user-defined input parameters.

The following code example demonstrates an iteration through the elements of the aggregate list. For each element q, an iteration through its individual rankings in each input preference list is performed:

```
void MergedList::CustomMethod1 (class InputList ** inlists, class
SimpleScoreStats * s, class InputParams * prms) {
	class MergedItem * q;
	class Ranking * r;

	for (rank_t i = 0; i < this->num_nodes; i++) {
		/// q stores an element of the aggregate list
		q = this->item_list[i];

		/// Iterate through the individual rankings of q
		for (uint32_t j = 0; j < q->get_num_alloc_rankings(); j++) {
			r = q->get_ranking(j);
			/// Do something with q and r

		}
	}
	/// Sort the list elements in decreasing score order
	qsort(this->item_list, this->num_nodes, sizeof(class MergedItem *),
		&MergedList::cmp_score_desc);
}
```

Calling the new method/s

The implementation of the new methods can be immediately used, provided that you have not changed the names of the functions `CustomMethod1()` and `CustomMethod2()`. In this case, the following piece of code inside the `main()` function in `main.cpp` the new implementations:

```
/// Execution of CustomMethod1
Custom1(input_file, qrels_file, 20, "Custom1", output_dir);

/// Execution of CustomMethod2
Custom2(input_file, qrels_file, 20, "Custom2", output_dir);
```

If you change the aforementioned function names, or you create a new function for a new algorithm implementation, then a sequence of actions must be taken so that it becomes available for usage. More specifically:

- The user must determine an integer identifier for the algorithm. To avoid conflicts with the built-in methods of FLAGR, it is advised that the leading digit of the identifier is `9` (e.g., `900`).
- The user must update the `aggregate()` function of the `Aggregator` to allow the execution of the custom method. More specifically, the `if` statement must be appropriately extended.
- An exposed C function must be written in both `cflagr.cpp` and `dllflagr.cpp` with the aim of including the custom implementation in a shared/dynamic link library. The function can be called in the `main()` function of `main.cpp`.

Additional details for custom method implementations

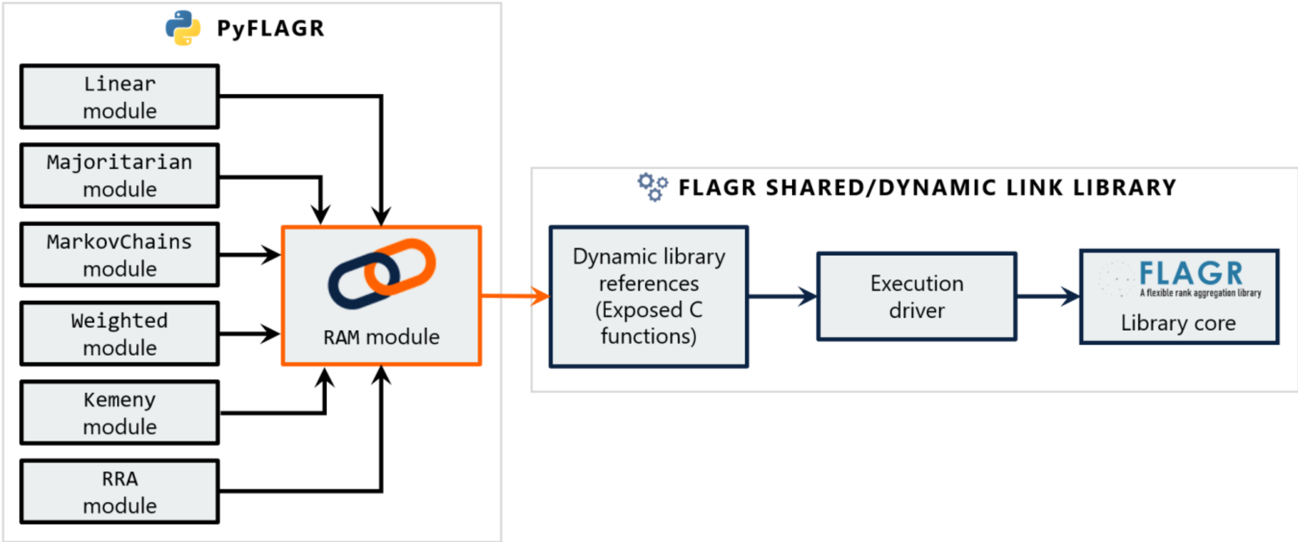
1. A custom rank aggregation method must be declared as a `public` member function of the `MergedList` class. This is performed in the class descriptor, in the `src/MergedList.h` header file. `CustomMethod1()` and `CustomMethod2()` are already public members of `MergedList`.

2. The required `cpp` file that contains the implementation of the member function must be stored in the `src/ram` directory.

3. Then, the `cpp` file is imported in the project with an include statement in `src/MergedList.cpp`.

4. PyFLAGR API Documentation

PyFLAGR is a Python library built on top of FLAGR. It includes a driver module called `RAM` that links to the FLAGR shared library and provides access to the algorithm implementations. Then, a set of classes inherit from RAM and allow the end user to execute the selected algorithm.



4.1 Introduction

PyFLAGR is a Python library built on top of FLAGR. It provides easy access to the algorithm implementations of FLAGR from standard Python programs. PyFLAGR has been designed with simplicity in mind: with only a few lines of code the programmer may efficiently execute complex rank aggregation methods and get the results in a Pandas Dataframe.

From a technical perspective, PyFLAGR links to the FLAGR shared/dynamic link library and makes use of its reference functions (namely, its exposed C functions) to pass the user-defined parameters and perform rank aggregation. In the sequel, it simply reads the output files that are produced by FLAGR and stores their contents in Pandas Dataframes. The Dataframes are then returned to the user.

PyFLAGR consists of the following modules:

- `RAM`: It implements the `RAM` base class that is responsible for several functional procedures like the library I/O, the linkage and loading of the FLAGR shared library, Dataframe handling, and so on. All the other classes of PyFLAGR derive from this class.
- `Linear`: It includes several classes that execute linear combination rank aggregation methods (CombSUM, CombMNZ, Borda Count).
- `Majoritarian`: It includes several classes that execute majoritarian rank aggregation methods (Condorcet Winners, Copeland Winners, Outranking Approach of Farah and Vanderpooten, 2007).
- `MarkovChains`: It executes the algorithms which are based on Markov Chains (MC1, MC2, MC3, MC4, see Dwork et al., 2001).

- `Weighted`: It executes the rank aggregation methods that automatically determine the voter weights in an unsupervised learning fashion (Preference Relations Method of Desarkar et al., 2016, Agglomerative Aggregation method of Chatterjee et al., 2018, Iterative Distance-based method of Akritidis et al., 2022).
- `Kemeny`: It executes Kemeny optimal aggregation (brute force, NP-Hard implementation).
- `RRA`: It executes the Robust Rank Aggregation (RRA) method of Kolde et al., 2012.

Please refer to the Publications section for more information about the relevant papers.

4.2 Installation

PyFLAGR can be installed directly by using pip:

```
pip install pyflagr
```

Alternatively, PyFLAGR can be installed from the sources by navigating to the directory where `setup.py` resides:

```
pip install /path/to/setup.py
```

4.3 RAM module

The `RAM` module implements a driver base class also named `RAM`. The `RAM` base class performs several important functional procedures, including PyFLAGR I/O, linkage and loading of the FLAGR shared library, I/O Dataframe handling, and so on. The majority of the other classes of PyFLAGR derive from this class and inherit its members and properties.

The class constructor takes as argument the `eval_pts` parameter that determines the elements in the aggregate list on which the evaluation measures (i.e., Precision, and nDCG) will be computed. The most important operation of the constructor is the loading of the FLAGR shared library, according to the underlying operating system. Therefore, if PyFLAGR is executed on a Linux-based system, then `pyflagr/pyflagr/flagr.so` is loaded. Similarly, if PyFLAGR is executed on a Windows-based system, then `pyflagr/pyflagr/flagr.dll` is loaded. For the time being, FLAGR has not been tested on MacOS-based systems and no pre-compiled shared libraries exist for this platform.

The successful loading of the FLAGR shared/dynamic link library creates the `flagr_lib` connection handler. `flagr_lib` acts as a connector between PyFLAGR and FLAGR, making the exposed C functions of FLAGR accessible from `RAM` and its derived classes.

Other member functions include `check_get_input()` and `check_get_rels_input()`. These two functions perform several sanity checks on the provided input files. On the other hand, the role of `get_output()` is to read the output files created by FLAGR and load their content into two Pandas Dataframes. These two Dataframes are eventually returned to the user.

Implementation file

`pyflagr/pyflagr/RAM.py`

4.4 Linear module

The Linear module provides access to the implementations of the linear combination methods of FLAGR. In these methods, the score of each element is computed by summing up the partial scores of that element with respect to its rankings in each input preference list. The module includes four classes which are described below: CombSUM, CombMNZ, BordaCount and SimpleBordaCount.

Implementation file

pyflagr/pyflagr/Linear.py

The CombSUM and CombMNZ Python classes

Both classes derive from RAM, a base class defined in the RAM module. They inherit the flagr_lib connector from RAM, and through it, they obtain access to the FLAGR shared library. Their constructors are identical and determine the data types of the input arguments and the return type of the Linear() exposed function. Observe the similarity between the members of self.flagr_lib.Linear.argtypes and the input arguments of the Linear() exposed function.

The arguments of the constructors of CombSUM and CombMNZ include:

Parameter	Type	Default	Description
eval_pts	Integer, Optional. Considered only if rels_file or rels_df is set.	10	Determines the elements in the aggregate list on which the evaluation measures (i.e., Precision, and nDCG) will be computed. For example, for eval_pts=10 FLAGR will compute Average Precision, $P@1, P@2, \ldots P@10$ and $N@1, N@2, \ldots N@10$.
norm	String, Optional.	borda	Rank or score normalization methods. • borda: The aggregation is performed by normalizing the element rankings according to the Borda normalization method. • rank: The aggregation is performed by normalizing the element rankings according to the Rank normalization method. • score: The aggregation is performed by normalizing the element scores according to the Score normalization method. • z-score: The aggregation is performed by normalizing the element scores according to the Z-Score normalization method. • simple-borda: Similar to borda normalization but no partial score is assigned to an element if it is not ranked by a voter.

CombSUM and CombMNZ also include an aggregate() function that receives the user-defined input parameters and passes them to the Linear() exposed C function, that subsequently performs the aggregation of the ranked input preference lists. The arguments of the aggregate() function include the following:

Parameter	Type	Default	Description
`input_file`	String Required, unless `input_df` is set.	Empty String	A CSV file that contains the input lists to be aggregated.
`input_df`	Pandas DataFrame - Required, unless `input_file` is set.	None	A Pandas DataFrame that contains the input lists to be aggregated. **Note:** If both `input_file` and `input_df` are set, only the former is used; the latter is ignored.
`rels_file`	String, Optional.	Empty String	A CSV file that contains the relevance judgements of the involved list elements. If such a file is passed, FLAGR will evaluate the generated aggregate list(s) by computing several retrieval effectiveness evaluation measures. The results of the evaluation will be stored in the `eval_df` DataFrame. Otherwise, no evaluation will take place and `eval_df` will be empty.
`rels_df`	Pandas DataFrame, Optional.	None	A Pandas DataFrame that contains the relevance judgements of the involved list elements. If such a dataframe is passed, FLAGR will evaluate the generated aggregate list(s) by computing several retrieval effectiveness evaluation measures. The results of the evaluation will be stored in the `eval_df` DataFrame. Otherwise, no evaluation will take place and `eval_df` will be empty. **Note:** If both `rels_file` and `rels_df` are set, only the former is used; the latter is ignored.
`output_dir`	String, Optional.	Temporary directory (OS-specific)	The directory where the output files (aggregate lists and evaluation) will be stored. If it is not set, the default location will be used.

The `BordaCount` and `SimpleBordaCount` Python classes

These two classes have been included in PyFLAGR for historical reasons. They are equivalent to the CombSUM linear combination method with `norm='borda'` and `norm='simple-borda'` normalization methods, respectively.

`BordaCount` and `SimpleBordaCount` derive from `CombSUM` (which in turn derives from `RAM`). They do not have an `aggregate()` method, so they both call the same `aggregate()` function of `CombSUM`. Their only difference lies in their constructors: The former initializes `CombSUM` with `norm='borda'`, whereas the latter with `norm='simple-borda'`.

4.5 `Majoritarian` module

The Majoritarian module provides access to the implementations of the majoritarian rank aggregation methods of FLAGR. These methods are based on the majority criterion that, under several circumstances, identify the "winning" elements. The module includes three classes which are described below: `CondorcetWinners`, `CopelandWinners` and `OutrankingApproach`. Each method implements different scenarios for the majority criterion.

Implementation file

pyflagr/pyflagr/Majoritarian.py

The `CondorcetWinners` and `CopelandWinners` Python classes

Both classes derive from `RAM`, a base class defined in the `RAM` module. They inherit the `flagr_lib` connector from `RAM`, and through it, they obtain access to the respective exposed functions of FLAGR.

The constructor of `CondorcetWinners` determines the data types of the input arguments and the return type of the `Condorcet()` exposed function. Similarly, the constructor of `CopelandWinners` determines the data types of the input arguments and the return type of the `Copeland()` exposed function. Both constructors accept just one argument:

Parameter	Type	Default	Description
`eval_pts`	Integer, Optional. Considered only if `rels_file` or `rels_df` is set.	`10`	Determines the elements in the aggregate list on which the evaluation measures (i.e., Precision, and nDCG) will be computed. For example, for `eval_pts=10` FLAGR will compute Average Precision, $P@1, P@2, \ldots P@10$ and $N@1, N@2, \ldots N@10$.

`CondorcetWinners` and `CopelandWinners` also include an `aggregate()` function that receives the user-defined parameters and passes them to the `Condorcet()` and `Copeland()` exposed functions respectively. The arguments of the `aggregate()` function are identical to those of the `CombSUM` and `CombMNZ` classes (refer to the respective table of Subsection 4.4).

The `OutrankingApproach` Python class

This class can be used to execute the Outranking Approach of Farah and Vanderpooten, 2007. Similarly to the other majoritarian methods, `OutrankingApproach` derives from `RAM`, a base class defined in the `RAM` module. It also inherits the `flagr_lib` connector from `RAM` and obtains access to the `OutrankingApproach()` exposed function of FLAGR.

The constructor of `OutrankingApproach` determines the data types of the input arguments and the return type of the `OutrankingApproach()` exposed function. Observe the similarity between the members of `self.flagr_lib.OutrankingApproach.argtypes` and the input arguments of the `OutrankingApproach()` exposed function.

Parameter	Type	Default	Description
eval_pts	Integer, Optional. Considered only if rels_file or rels_df is set.	10	Determines the elements in the aggregate list on which the evaluation measures (i.e., Precision, and nDCG) will be computed. For example, for eval_pts=10 FLAGR will compute Average Precision, $P@1, P@2, \ldots P@10$ and $N@1, N@2, \ldots N@10$.
preference	Hyper-parameter, Float, Optional.	0.00	The value of the preference threshold.
veto	Hyper-parameter, Float, Optional.	0.75	The value of the veto threshold.
concordance	Hyper-parameter, Float, Optional.	0.00	The value of the concordance threshold.
discordance	Hyper-parameter, Float, Optional.	0.25	The value of the discordance threshold.

OutrankingApproach includes an aggregate() function that receives the user-defined parameters and passes them to the OutrankingApproach() exposed function. The arguments of the aggregate() function are identical to those of the CombSUM and CombMNZ classes (refer to the respective table of Subsection 4.4).

4.6 MarkovChains module

The MarkovChains module provides access to the implementations of the Markov Chains methods of Dwork et al., 2001 and DeConde et al., 2006. It includes the MC driver class, and five small derived classes (namely, MC1, MC2, MC3, MC4 and MCT) that can be employed by the user to execute the corresponding methods.

Implementation file

pyflagr/pyflagr/MarkovChains.py

The MC base class

This is the driver class of the module. Its direct usage is weakly discouraged. The users should prefer employing the 5 classes that derive from MC (see below) and indirectly trigger the corresponding algorithm implementations of FLAGR.

Similarly to the other PyFLAGR classes, MC derives from RAM, another base class that is defined in the RAM module. It inherits the flagr_lib connector from RAM, and through it, it obtains access to the FLAGR shared library. Its constructor determines the data types of the input arguments and the return type of the FLAGR's MC() exposed function. Observe the similarity between the members of self.flagr_lib.MC.argtypes and the input arguments of the MC() exposed function.

The constructor of MC takes the following arguments:

Parameter	Type	Default	Description
`eval_pts`	Integer, Optional. Considered only if `rels_file` or `rels_df` is set.	`10`	Determines the elements in the aggregate list on which the evaluation measures (i.e., Precision, and nDCG) will be computed. For example, for `eval_pts=10` FLAGR will compute Average Precision, $P@1, P@2, \ldots P@10$ and $N@1, N@2, \ldots N@10$.
`ergodic_number`	Float, Optional.	`0.15`	The ergodic number, used during the computation of the ergodic transition matrix from the normalized transition matrix.
`max_iterations`	Integer, Optional.	`100`	The maximum number of iterations for the computation of the state matrix.
`chain`	Integer, Optional.	`804`	The Markov Chain method to execute. The possible values include: • `801`: Markov Chains Method 1 (MC1). • `802`: Markov Chains Method 2 (MC2). • `803`: Markov Chains Method 3 (MC3). • `804`: Markov Chains Method 4 (MC4). • `805`: Markov Chains Thurstone Method (MCT).

`MC` includes an `aggregate()` function that receives the user-defined parameters and passes them to the `MC()` exposed function, that subsequently performs the aggregation of the ranked input preference lists. The arguments of the `aggregate()` function are identical to those of the `CombSUM` and `CombMNZ` classes (refer to the respective table of Subsection 4.4).

The `MC1`, `MC2`, `MC3`, `MC4` and `MCT` derived classes

These classes derive from the aforementioned `MC` base class; as children of `MC`, they also inherit from the `RAM` class. Each of these classes triggers the execution of a different Markov Chain algorithm, simply by passing different parameters to the constructor `MC`. Hence, `MC1` sets `chain=801`, `MC2` sets `chain=802`, and so on.

Also notice that notice that none of these classes have an `aggregate()` function. Consequently, the `aggregate()` of the base class (i.e. `MC`) is executed when the end user invokes that function.

4.7 `Weighted` module

The `Weighted` module provides access to the implementations of the weighted methods of FLAGR. Most rank aggregation approaches treat all input preference lists equally. In contrast, the weighted methods employ exploratory analysis techniques to automatically identify the expert voters in an unsupervised fashion. In the sequel, they assign higher weights to those who were identified as experts, thus boosting the scores of their submitted elements. The module in question includes three classes: `DIBRA`, `Agglomerative` and `PreferenceRelationsGraph`.

Implementation file

`pyflagr/pyflagr/Weighted.py`

Distance-Based Iterative Rank Aggregation: The `DIBRA` Python class

This class links to the FLAGR implementation of the weighted method of Akritidis et al., 2022. It derives from `RAM`, a base class defined in the `RAM` module. It inherits the `flagr_lib` connector from `RAM` and through it, it obtains access to the FLAGR shared library. Its constructor determines the data types of the input arguments and the return type of the `DIBRA()` exposed function. Observe the similarity between the members of `self.flagr_lib.DIBRA.argtypes` and the input arguments of the `DIBRA()` exposed function.

The algorithm initially employs a standard non-weighted method to generate a starting consensus list. Then, it iteratively assigns converging weights to the voters according to the distances of their submitted lists with this consensus list. Therefore, the `DIBRA` constructor takes as arguments all the possible hyper-parameters of the supported non-weighted methods. Specifically:

Parameter	Type	Default	Description
`eval_pts`	Integer, Optional. Considered only if `rels_file` or `rels_df` is set.	`10`	Determines the elements in the aggregate list on which the evaluation measures (i.e., Precision, and nDCG) will be computed. For example, for `eval_pts=10` FLAGR will compute Average Precision, $P@1, P@2, \ldots P@10$ and $N@1, N@2, \ldots N@10$.
`aggregator`	Hyper-parameter, String, Optional.	`combsum:borda`	The selected non-weighted method that does the initial aggregation. Possible values include: • `combsum:borda`: CombSUM with Borda normalization. • `combsum:rank`: CombSUM with Rank normalization. • `combsum:score`: CombSUM with minmax score normalization. • `combsum:z-score`: CombSUM with Z-score normalization. • `combsum:simple-borda`: CombSUM with simple Borda normalization. • `combmnz:borda`: CombMNZ with Borda normalization. • `combmnz:rank`: CombMNZ with Rank normalization. • `combmnz:score`: CombMNZ with minmax score normalization. • `combmnz:z-score`: CombMNZ with Z-score normalization.

			• `combmnz:simple-borda`: CombMNZ with simple Borda normalization. • `condorcet`: Condorcet Winners. • `copeland`: Copeland Winners. • `outrank`: Outranking Approach.
`w_norm`	Hyper-parameter, String, Optional.	`minmax`	The voter weights normalization method. Possible values include: • none: no weight normalization takes place. • `minmax`: minmax weight normalization. • `z`: z weight normalization.
`dist`	Hyper-parameter, String, Optional.	`cosine`	The correlation/distance metric that measures the distance between an input list and the temporary aggregate list. Possible values include: • `rho`: Spearman's ρ. • `cosine`: a metric based on cosine similarity (see Akritidis et al., 2022). • `footrule`: Spearman's Footrule distance. • `tau`: Kendall's τ
`prune`	Hyper-parameter, Boolean, Optional.	`False`	Triggers a weight-dependent list pruning mechanism.
`gamma`	Hyper-parameter, Float, Optional.	`1.5`	The γ hyper-parameter that determines the steplength of weight learning.
`d1`	Hyper-parameter, Float, Optional.	`0.4`	The δ_1 hyper-parameter of the list pruning mechanism. Applies only if `prune=True`.
`d2`	Hyper-parameter, Float, Optional.	`0.1`	The δ_2 hyper-parameter of the list pruning mechanism. Applies only if `prune=True`.
`tol`	Hyper-parameter, Float, Optional.	`0.01`	Controls the convergence precision. This tolerance threshold represents the minimum precision of the difference between the voter weight in an iteration and the voter weight of the previous iteration.
`max_iter`	Hyper-parameter, Integer, Optional.	`50`	Controls the maximum number of iterations before the voter weights converge.
`pref`	Hyper-parameter, Float, Optional.	`0.0`	The preference threshold. Applies only if `aggregator=outrank`.
`veto`	Hyper-parameter, Float, Optional.	`0.75`	The veto threshold. Applies only if `aggregator=outrank`.

conc	Hyper-parameter, Float, Optional.	0.0	The concordance threshold. Applies only if aggregator=outrank.
disc	Hyper-parameter, Float, Optional.	0.25	The discordance threshold. Applies only if aggregator=outrank.

DIBRA also includes an aggregate() function that receives the user-defined parameters and passes them to the DIBRA() exposed function, that subsequently performs the aggregation of the ranked input preference lists. The arguments of the aggregate() function are identical to those of the CombSUM and CombMNZ classes (refer to the respective table of Subsection 4.4).

Agglomerative weighted aggregation: The Agglomerative Python class

This class employs the implementation of the Agglomerative weighted method of Chatterjee et al., 2018. Similarly to all weighted methods, it derives from RAM, a base class defined in the RAM module. It inherits the flagr_lib connector from RAM, and, through it, they obtain access to the FLAGR shared library. Its constructor determines the data types of the input arguments and the return type of the Agglomerative() exposed function. Observe the similarity between the members of self.flagr_lib.Agglomerative.argtypes and the input arguments of the Agglomerative() exposed function.

The constructor's arguments include:

Parameter	Type	Default	Description
eval_pts	Integer, Optional. Considered only if rels_file or rels_df is set.	10	Determines the elements in the aggregate list on which the evaluation measures (i.e., Precision, and nDCG) will be computed. For example, for eval_pts=10 FLAGR will compute Average Precision, $P@1, P@2, \dots P@10$ and $N@1, N@2, \dots N@10$.
c1	Hyper-parameter, Float, Optional.	0.1	The c_1 hyper-parameter of the algorithm.
c2	Hyper-parameter, Float, Optional.	0.5	The c_2 hyper-parameter of the algorithm.

Agglomerative includes an aggregate() function that receives the user-defined parameters and passes them to the Agglomerative() exposed function, that subsequently performs the aggregation of the ranked input preference lists. The arguments of the aggregate() function are identical to those of the CombSUM and CombMNZ classes (refer to the respective table of Subsection 4.4).

Preference relations weighted method: The PreferenceRelationsGraph Python class

This class links to the FLAGR implementation of the weighted method of Desarkar et al., 2016. Similarly to all weighted methods, it derives from RAM, a base class defined in the RAM module. It inherits the flagr_lib connector from RAM, and through it, they obtain access to the FLAGR shared library. Its constructor determines the data types of the input arguments and the return type of the PrefRel()

exposed function. Observe the similarity between the members of `self.flagr_lib.PrefRel.argtypes` and the input arguments of the `PrefRel()` exposed function.

The constructor's arguments include:

Parameter	Type	Default	Description
`eval_pts`	Integer, Optional. Considered only if `rels_file` or `rels_df` is set.	`10`	Determines the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed. For example, for `eval_pts=10` FLAGR will compute Average Precision, $P@1, P@2, \dots P@10$ and $N@1, N@2, \dots N@10$.
`alpha`	Hyper-parameter, Float, Optional.	`0.1`	The α hyper-parameter of the algorithm.
`beta`	Hyper-parameter, Float, Optional.	`0.5`	The β hyper-parameter of the algorithm.

`PreferenceRelationsGraph` also includes an `aggregate()` function that receives the user-defined parameters and passes them to the `PrefRel()` exposed function, that subsequently performs the aggregation of the ranked input preference lists. The arguments of the `aggregate()` function are identical to those of the `CombSUM` and `CombMNZ` classes (refer to the respective table of Subsection 4.4).

4.8 Kemeny module

The `Kemeny` module provides access to the implementation of Kemeny Optimal Aggregation of FLAGR. It is stressed out that due to the method's high complexity, the brute force implementation becomes infeasible even when the number of elements, or the number of input lists receive moderate values.

Implementation file

`pyflagr/pyflagr/Kemeny.py`

The `KemenyOptimal` class

The `KemenyOptimal` class derives from `RAM`, a base class defined in the `RAM` module. It inherits the `flagr_lib` connector from `RAM`, and through it, it obtains access to the FLAGR shared library. Its constructor determines the data types of the input arguments and the return type of the FLAGR's `Kemeny()` exposed function. Observe the similarity between the members of `self.flagr_lib.Kemeny.argtypes` and the input arguments of the `Kemeny()` exposed function.

The arguments of the constructor includes one parameter:

Parameter	Type	Default	Description
eval_pts	Integer, Optional. Considered only if rels_file or rels_df is set.	10	Determines the elements in the aggregate list on which the evaluation measures (i.e., Precision, and nDCG) will be computed. For example, for eval_pts=10 FLAGR will compute Average Precision, $P@1, P@2, \dots P@10$ and $N@1, N@2, \dots N@10$.

KemenyOptimal also includes an aggregate() function which receives the user-defined parameters and passes them to the Kemeny() exposed function, that subsequently performs the aggregation of the ranked input preference lists. The arguments of the aggregate() function are identical to those of the CombSUM and CombMNZ classes (refer to the respective table of Subsection 4.4).

4.9 RRA module

The RRA module provides access to the implementation of Robust Rank Aggregation of Kolde et al., 2012.

Implementation file

pyflagr/pyflagr/RRA.py

The RRA class

The RRA class derives from RAM, a base class defined in the RAM module. It inherits the flagr_lib connector from RAM, and through it, it obtains access to the FLAGR shared library. Its constructor determines the data types of the input arguments and the return type of the FLAGR's RobustRA() exposed function. Observe the similarity between the members of self.flagr_lib.RobustRA.argtypes and the input arguments of RobustRA() exposed function.

The arguments of the constructor includes two parameters:

Parameter	Type	Default	Description
eval_pts	Integer, Optional. Considered only if rels_file or rels_df is set.	10	Determines the elements in the aggregate list on which the evaluation measures (i.e. Precision, and nDCG) will be computed. For example, for eval_pts=10 FLAGR will compute Average Precision, $P@1, P@2, \dots P@10$ and $N@1, N@2, \dots N@10$.
exact	Boolean, Optional.	False	Determines whether the computed p-Values of the list elements will be corrected with the Stuart-Ares method.

RRA also includes an aggregate() function that receives the user-defined parameters and passes them to the RobustRA() exposed function, that subsequently performs the aggregation of the ranked input preference lists. The arguments of the aggregate() function are identical to those of the CombSUM and CombMNZ classes (refer to the respective table of Subsection 4.4).

4.10 Comparator module

The Comparator module includes a class that implements several tools for conducting performance comparisons of rank aggregation algorithms. The input includes a data file with the input preference lists, another file that contains the relevance judgments for the involved list elements, and a group of rank aggregation algorithms to be compared. After running the selected algorithms on the input data, Comparator produces comparison tables in various formats (e.g. CSV, LaTeX, etc.) and plots of multiple evaluation measures. Extended code examples of usage are presented in this notebook.

Implementation file

pyflagr/pyflagr/Comparator.py

Member variables

The class maintains three member variables:

- aggregators: A Python list that contains the objects that handle rank aggregation algorithms, along with a user-defined description.
- results: A Pandas Dataframe that stores the results of the evaluation (namely, the values of various evaluation measures).
- ev_pts: An integer that represents the cutoff point at which the evaluation measures will be computed.

Member methods

add_aggregator(): This function appends new records into the aggregators list. Each record represents a rank aggregation method that will participate in the comparison tests.

Parameter	Type	Default	Description
name	String, Required.	-	The name of the rank aggregation algorithm that is inserted.
obj	Object, Required.	-	An object that handles the corresponding rank aggregation method.

Here is a quick example that initializes a Comparator object and appends three rank aggregation methods:

```
import pyflagr.Linear as Linear
import pyflagr.Majoritarian as Majoritarian
import pyflagr.Weighted as Weighted
import pyflagr.Comparator as Comparator

EV_PTS = 10
cmp = Comparator.Comparator(EV_PTS)
cmp.add_aggregator("CombSUM-Borda",Linear.CombSUM(norm='borda',
eval_pts=EV_PTS))
cmp.add_aggregator("Copeland", Majoritarian.CopelandWinners(eval_pts=EV_PTS))
cmp.add_aggregator("DIBRA-Prune",   Weighted.DIBRA(aggregator='combsum:borda',
gamma=1.2, prune=True, w_norm='minmax', d1=0.3, d2=0.05, eval_pts=EV_PTS))
```

aggregate(): Sequentially invokes the aggregate() method of each algorithm included in the aggregators array.

This method also requires a file (or a Dataframe) that contains relevance judgments for the individual list elements. The generated aggregate lists of each algorithm are automatically evaluated (by FLAGR) by using these relevance judgments. The class computes the values of multiple well-established evaluation measures including Mean Average Precision (MAP), Precision, Recall, DCG (Discounted Cumulative Gain), and nDCG (normalized DCG). The computed values are written into the self.results Dataframe.

aggregate() takes four arguments:

Parameter	**Type**	**Default**	**Description**
input_file	String, Required, unless input_df is set.	Empty String	A CSV file that contains the input lists to be aggregated.
input_df	Pandas DataFrame, Required, unless input_file is set.	None	A Pandas DataFrame that contains the input lists to be aggregated. **Note:** If both input_file and input_df are set, only the former is used; the latter is ignored.
rels_file	String, Required, unless rels_df is set.	Empty String	A CSV file that contains the relevance judgements of the involved list elements. FLAGR will evaluate the generated aggregate list/s by computing the values of multiple performance evaluation measures. The results of the evaluation will be stored in the self.results Dataframe.
rels_df	Pandas DataFrame, Required, unless rels_file is set.	None	A Pandas DataFrame that contains the relevance judgements of the involved list elements. FLAGR will evaluate the generated aggregate list/s by computing the values of multiple performance evaluation measures. The results of the evaluation will be stored in the self.results Dataframe. **Note:** If both rels_file and rels_df are set, only the former is used; the latter is ignored.

Example:

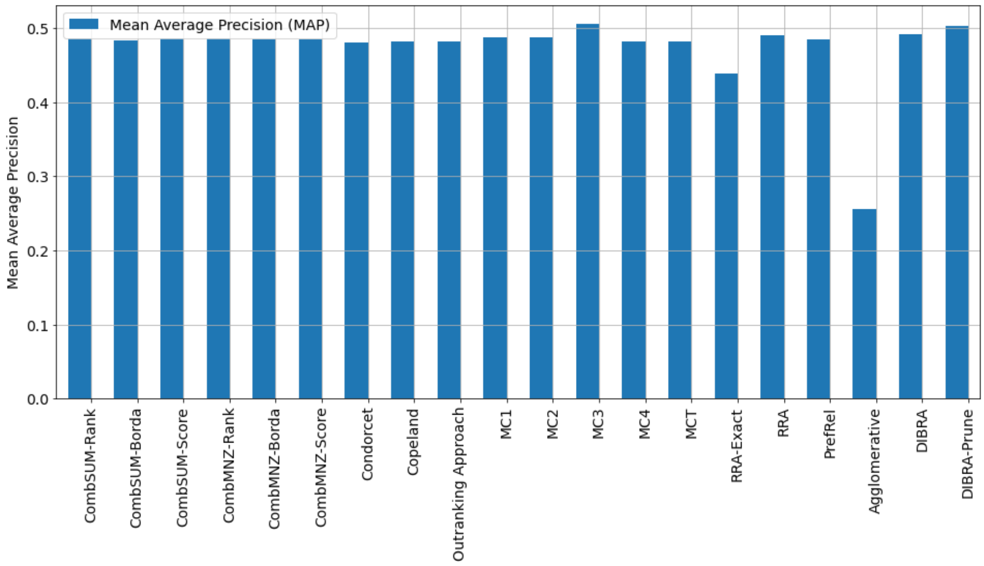
```
# The input data file with the input lists to be aggregated.
lists = 'testdata.csv'

# The input data file with the relevance judgements.
qrels = 'testdata_qrels.csv'

cmp.aggregate(input_file=lists, rels_file=qrels)
```

`plot_average_precision()`: Creates a comparative bar plot of Mean Average Precision (MAP). The arguments include:

Parameter	**Type**	**Default**	**Description**
`dimensions`	(x,y) tuple - Optional.	`(10.24,7.68)`	The plot dimensions `(width, height)`.
`show_grid`	Boolean - Optional.	`True`	Determines whether the plot will include grid lines.
`query`	String - Optional.	`'all'`	In case the input data file contains preference lists for multiple queries, this parameter determines which query to plot. Notice that `'all'` does not mean that all queries will be plotted; instead, it dictates the plotting of the average MAP for all queries.

Example:

```
cmp.plot_average_precision((16, 7), True, query='all')
```



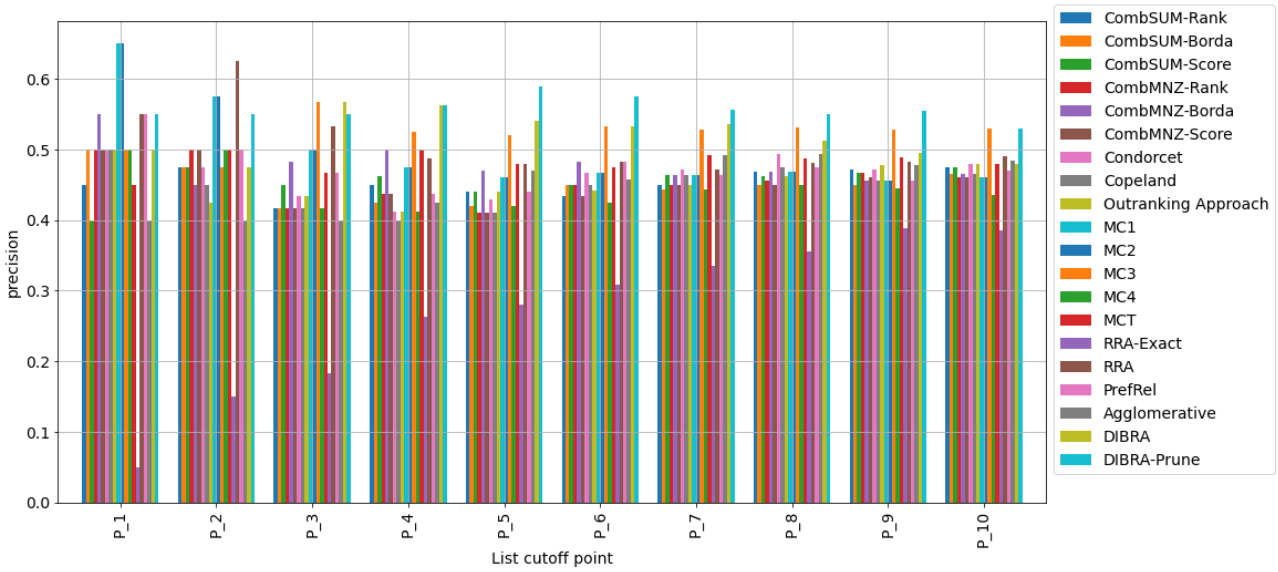
`plot_metric()`: Creates a plot for a metric at a given cutoff point. The input arguments include:

Parameter	**Type**	**Default**	**Description**
`cutoff`	Integer - Required.	-	The cutoff point in the aggregate list. The cutoff point must be lower than `self.ev_pts`.
`metric`	String - Required.	-	Determines the evaluation measure to be plotted. Acceptable values are `'precision'`, `'recall'`, `'dcg'`, and `'ndcg'`.
`plot_type`	String - Optional.	`'bar'`	Determines the plot type. Acceptable values are `'bar'` and `'lines'`.

`dimensions`	(x,y) tuple - Optional.	`(10.24, 7.68)`	The plot dimensions (width, height).
`show_grid`	Boolean - Optional.	`True`	Determines whether the plot will include grid lines.
`query`	String - Optional.	`'all'`	In case the input data file contains preference lists for multiple queries, this parameter determines which query to plot. Notice that `'all'` does not mean that all queries will be plotted; instead, it dictates the plotting of the average MAP for all queries.

Example:

```
cmp.plot_metric (EV_PTS, metric='precision', plot_type='bar', dimensions=(16,8), show_grid=True, query='all')
```



`get_results()`: Returns slices of `self.results` by setting specific evaluation measures (columns) and queries (rows). The input arguments include:

Parameter	**Type**	**Default**	**Description**
`cutoff`	Integer - Required.	-	The cutoff point in the aggregate list. The cutoff point must be lower than `self.ev_pts`.
`metric`	String - Required.	-	Determines the evaluation measure to be plotted. Acceptable values are `'precision'`, `'recall'`, `'dcg'`, and `'ndcg'`.
`query`	String - Optional.	`'all'`	In case the input data file contains preference lists for multiple queries, this parameter determines which query to retrieve. Notice that `'all'` does not mean that all queries will be plotted; instead, it dictates the plotting of the average MAP for all queries.

`convert_to_latex()`: Returns the LaTeX code of slices of `self.results`. The input arguments are:

Parameter	Type	Default	Description
`cutoff`	Integer - Required.	-	The cutoff point in the aggregate list. The cutoff point must be lower than `self.ev_pts`.
`metric`	String - Required.	-	Determines the evaluation measure to be plotted. Acceptable values are `'precision'`, `'recall'`, `'dcg'`, and `'ndcg'`.
`query`	String - Optional.	`'all'`	In case the input data file contains preference lists for multiple queries, this parameter determines which query to retrieve. Notice that `'all'` does not mean that all queries will be plotted; instead, it dictates the plotting of the average MAP for all queries.
dec_pts	Integer - Optional - Maximum value is 6.	6	Sets the precision (i.e. the number of decimal points) of the values of the returned evaluation measures.

5. Appendix

5.1 Evaluation measures

This section provides some brief descriptions of the most popular performance evaluation measures for rank aggregation algorithms. These measures are computed by the evaluation tool of FLAGR in case a valid qrels file is provided as input.

Precision@k

Precision measures the ability of an algorithm to precisely detect the relevant elements. It is defined as the ratio of the number of relevant elements at the k-th element of a list, divided by the number of retrieved elements (i.e. k)}:

$$\text{Precision}@k = \frac{\text{true positives}@k}{(\text{true positives}@k) + (\text{false positives}@k)}$$

Recall@k

Recall measures the ability of an algorithm to detect the relevant elements early. It is defined as the ratio of the number of relevant elements at the k-th element of a list, divided by the number of all relevant elements:

$$\text{Recall}@k = \frac{\text{true positives}@k}{(\text{true positives}@k) + (\text{false negatives}@k)}$$

$F1@k$

$F1$ is a well-established measure that combines Precision and Recall into a single scoring formula:

$$F1@k = \frac{2 \cdot \text{Precision}@k \cdot \text{Recall}@k}{\text{Precision}@k + \text{Recall}@k}$$

Discounted Cumulative Gain $DCG@k$

DCG is another measure for evaluating the performance of an algorithm. In contrast to the previous measures, this one can handle non-binary relevance judgments. In other words, the relevance score of an item may be a real value, and not just a relevant/non-relevant label. It is defined by the following formula:

$$DCG@k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

where rel_i is the relevance score of the element at index i. For binary problems, we set $rel_i = 1$ if the i-th list element is relevant and $rel_i = 0$ otherwise.

Normalized Discounted Cumulative Gain ($nDCG@k$)

One disadvantage of DCG is that it is non-decreasing; it either stays the same (if the current element is non-relevant), or it increases (if the current element is relevant). This means that queries that return larger result sets will probably always have higher DCG scores than queries that return small result sets. The Normalized Discounted Cumulative Gain ($nDCG$) confronts this problem by dividing DCG with the maximum possible DCG at each threshold k:

$$nDCG@k = \frac{DCG@k}{IDCG@k}$$

where $IDCG@k$ is the Ideal $DCG@k$. To compute it, we first create an ideal ranking, where the elements are ranked in decreasing relevance order. Then, $IDCG@k$ is simply equal to $DCG@k$ in that ideal ranking, namely:

$$IDCG@k = \sum_{i=1}^{\text{relevant items }@k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

Average Precision (AP)

AP is another evaluation metric that quantifies the ability of an algorithm to rank the relevant elements in the highest list positions. It is defined by the following equation:

$$AP = \sum_k (\text{Recall}@k - \text{Recall}@k - 1) \cdot \text{Precision}@k$$

Mean Average Precision (MAP)

Average Precision quantifies the quality of a single ranked list compared with the ground truth. In other words, AP examines a single ranking that is generated in response to a single query. In contrast, Mean Average Precision (MAP) evaluates a ranking model for a set of queries Q. MAP is simply defined as the mean of the Average Precisions over all queries. Consequently, its computation is performed by firstly summing up the AP_q value for each query q in the dataset and then, the sum is divided by the number of queries:

$$MAP = \frac{1}{|Q|} \sum_{q=1}^{|Q|} AP_q$$

Example

Consider a ranked list including 8 elements that has been submitted as a response to a query. From these elements, the 1st, 3rd, 4th, and 6th are relevant to the query. The rest of the elements are considered as not relevant. The first two columns of the following table show the list and the relevance of its elements. The rest of the columns contain the running values of Precision, Recall, $F1$, DCG, $IDCG$, $nDCG$ and Average Precision (AP) at each list element.

Rank	Relevant	$\mathbf{Precision@k}$	$\mathbf{Recall@k}$	$\boldsymbol{F1@k}$	$\boldsymbol{DCG@k}$	$\boldsymbol{DCG@k}$	$\boldsymbol{DCG@k}$	$\boldsymbol{AP}$
1	Yes	1.00	0.25	0.40	1.00	1.00	1.00	1.00
2	No	0.50	0.25	0.33	1.00	1.63	0.61	1.00
3	Yes	0.67	0.50	0.62	1.50	2.13	0.70	0.83
4	Yes	0.75	0.75	0.75	1.93	2.56	0.75	0.81
5	No	0.60	0.75	0.66	1.93	2.56	0.75	0.81
6	Yes	0.67	1.00	0.80	2.29	2.56	0.89	0.77
7	No	0.57	1.00	0.73	2.29	2.56	0.89	0.77
8	No	0.50	1.00	0.66	2.29	2.56	0.89	0.77

Example calculations at the 5th element of the list (@5)

According to the aforementioned definitions, the following calculations are performed:

$$\text{Precision@5} = \frac{\text{relevant elements found up to position 5}}{\text{retrieved elements up to position 5}} = \frac{3}{5} = 0.60$$

$$\text{Recall@5} = \frac{\text{relevant elements found up to position 5}}{\text{all relevant elements}} = \frac{3}{4} = 0.75$$

$$F1@5 = \frac{2 \cdot \text{Precision@5} \cdot \text{Recall@5}}{\text{Precision@5} + \text{Recall@5}} = \frac{2 \cdot 0.60 \cdot 0.75}{0.60 + 0.75} = 0.67$$

$$DCG@5 = \sum_{i=1}^{5} \frac{2^{rel_i} - 1}{\log_2(i+1)} = \frac{2^1 - 1}{\log_2(1+1)} + \frac{2^0 - 1}{\log_2(2+1)} + \frac{2^1 - 1}{\log_2(3+1)} + \frac{2^1 - 1}{\log_2(4+1)} + \frac{2^0 - 1}{\log_2(5+1)} = 1.9$$

$$IDCG@5 = \sum_{i=1}^{4} \frac{2^{rel_i} - 1}{\log_2(i+1)} = \frac{2^1 - 1}{\log_2(1+1)} + \frac{2^1 - 1}{\log_2(2+1)} + \frac{2^1 - 1}{\log_2(3+1)} + \frac{2^1 - 1}{\log_2(4+1)} = 2.56$$

$$nDCG@5 = \frac{DCG@5}{IDCG@5} = \frac{1.93}{2.56} = 0.75$$

$$AP = \frac{1 + 0.67 + 0.75}{3} = 0.81$$

5.2 References

FLAGR utilizes algorithms, methods, and techniques from the following bibliography:

- E. Renda, U. Straccia, "Web metasearch: rank vs. score based rank aggregation methods", In Proceedings of the 2003 ACM symposium on Applied computing, pp. 841-846, 2003.
- M. Farah, D. Vanderpooten, "An outranking approach for rank aggregation in information retrieval", In Proceedings of the 30th ACM Conference on Research and Development in Information Retrieval, pp. 591-598, 2007.
- M.S. Desarkar, S. Sarkar, P. Mitra, "Preference relations based unsupervised rank aggregation for metasearch", Expert Systems with Applications, vol. 49, pp. 86-98, 2016.
- S. Chatterjee, A. Mukhopadhyay, M. Bhattacharyya, "A weighted rank aggregation approach towards crowd opinion analysis", Knowledge-Based Systems, vol. 149, pp. 47-60, 2018.
- L. Akritidis, A. Fevgas, P. Bozanis, Y. Manolopoulos, "An Unsupervised Distance-Based Model for Weighted Rank Aggregation with List Pruning", Expert Systems with Applications, vol. 202, pp. 117435, 2022.
- C. Dwork, R. Kumar, M. Naor, D. Sivakumar, "Rank Aggregation Methods for the Web", In Proceedings of the 10th International Conference on World Wide Web, pp. 613-622, 2001.
- R. Kolde, S. Laur, P. Adler, J. Vilo, "Robust rank aggregation for gene list integration and meta-analysis", Bioinformatics, vol. 28, no. 4, pp. 573-580, 2012.

- K.L. Majumder, G.P. Bhattacharjee, "Algorithm AS 63: The incomplete Beta Integral", Applied Statistics, vol. 22, no. 3, pp. 409-411, 1973.
- R.P. DeConde, S. Hawley, S. Falcon, N. Clegg, B. Knudsen, R. Etzioni, "Combining results of microarray experiments: a rank aggregation approach", Statistical Applications in Genetics and Molecular Biology, vol. 5, no. 1, 2006.
- J. d. Borda, M´emoire sur les ´elections au scrutin, Histoire de l'Academie Royale des Sciences pour 1781 (Paris, 1784).
- M. Condorcet, Essai sur r application de lanalyse `a la probabilit´e des d´ecisions rendues a` la pluralit´e des voix, Paris.
- A. H. Copeland, "A reasonable social welfare function", Technical Report., Mimeo, University of Michigan USA, 1951.
- L. Akritidis, D. Katsaros, P. Bozanis, "Effective ranking fusion methods for personalized metasearch engines", In Proceedings of the 12th Panhellenic Conference on Informatics, IEEE, 2008, pp. 39–43.
- L. Akritidis, D. Katsaros, P. Bozanis, "Effective rank aggregation for metasearching, Journal of Systems and Software, vol. 84, no. 1, pp. 130–143, 2011.
- C. Manning, P. Raghavan, H. Schtze, "Introduction to Information Retrieval", Cambridge University Press, 2008.
- K. Jarvelin, J. Kekäläinen, "Cumulated gain-based evaluation of IR techniques", ACM Transactions on Information Systems (TOIS), vol. 20, no. 4, pp. 422–446, 2002.

5.3 Acknowledgments

FLAGR, PyFLAGR, the supporting Web site https://flagr.site and this user manual have been created by Leonidas Akritidis. You may contact the author at lakritidis@ihu.gr.

Deep thanks to John Burkardt for kindly permitting the usage of the ASA063 and ASA109 algorithm implementations about the computation of the incomplete beta integrals.

Bugs should be reported through the FLAGR's GitHub repository.

Third-party researchers are strongly encouraged to submit their own algorithm implementations, regardless of the programming language they are using.