Comprior Documentation

Cindy Perscheid

Jul 12, 2021

Contents:

CHAPTER 1

Installation and Usage

For working with Comprior, you have two options:

- *Run Comprior in a Docker Container* - for just executing Comprior with a custom data set and configuration
- *Install and Run Comprior from Source* - for developing new approaches and extending Comprior

1.1 Install and Run Comprior from Source

- **Prerequisites:** R 3.5+, Python 3.5+, Java with JDK, and Maven. See in *Installing Java JDK and Maven on Ubuntu* how to install JDK and Maven.

1.1.1 1. Installation

- check out the repository on your machine (or download the sources from https://github.com/CPerscheid/Comprior/archive/master.zip):

```
git clone https://github.com/CPerscheid/Comprior.git
```

- Run the installation bash script *install.sh* (if your are on MacOS, use *install_macos.sh*) and let it write its output into a file (the tee command prints the output to both command line and a file) - installation execution requires root access rights:

```
sudo ./install.sh 2>&1 | tee installout.out
```

- go grab some healthy snacks, for this might take a while depending on what is already installed on your machine (a couple of hours in the worst case) :-)
- check *code/configs/config.ini* if the variables *homePath* (path to Comprior's root directory), *RscriptLocation* (path to your Rscript), and *JavaLocation* (path to your Java location) point to the right locations.

1.1.2 2. Usage

- **Prior information:** In order to enable a flexible pipeline design for users, Comprior makes use of config files. All config files are to be stored in *code/configs* directory. **The main config file is located at code/configs/config.ini.** It is recommended not to be changed, as it specifies all parameters that Comprior needs for functioning properly, including access points to knowledge base web services and output folder structure. Instead, users can specify an own config file that contains only those parameters they want to overwrite from *config.ini*, e.g. where the input data is located or what feature selectors to apply. **Store your custom config file in the code/configs/ directory.** For a complete overview of the input parameters, see *Configuration Parameters*. If you write an own config file, make sure to provide it as input parameter for the framework (*config.ini* will always be loaded by default)
- **To start Comprior**
  - navigate to *code/Python/comprior*:

    ```
    cd code/Python/comprior
    ```

  - start Comprior (optionally provide a custom config file):

    ```
    python3 pipeline.py --config ../../configs/exampleconfig.ini
    ```

- Check your results in *data/results/example* - see *Folder Structure - Where to find what Files (In- and Output)* for where to find what results and *Output Generated by Comprior* for a more detailed explanation on the generated plots.

1.2 Run Comprior in a Docker Container

- **Prerequisites:** Docker
- check out the repository on your machine (or download the sources from https://github.com/CPerscheid/Comprior/archive/master.zip):

```
git clone https://github.com/CPerscheid/Comprior.git
```

- Via command line, navigate to *Comprior/* where the Dockerfile is located and create the image **using root privileges** (this might take a while):

```
cd Comprior
sudo docker build -t comprior .
```

- provide the **absolute path** of your *Comprior/comprior_docker* directory as mounting directory (it contains the config file and input data sets), and the config file as parameter (note the two dashes there) to Comprior and run it with **root privileges**:

```
sudo docker run -it --rm -v /your/absolute/path/to/Comprior/comprior_docker:/home/
↪app/data comprior ---config /home/app/data/dockerexampleconfig.ini
```

1.3 Installing Java JDK and Maven on Ubuntu

- install a JDK distribution for your Ubuntu version, e.g.:

```
sudo apt-get install openjdk-8-jdk
```

or:

```
apt install default-jdk
```

- let your JAVA_HOME variable point to your installed JDK. Typically, Ubuntu installs it in /usr/lib/jvm, so find it there and provide the correct path, e.g.:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

- also update your PATH variable to point to your JDK:

```
export PATH=$PATH:$JAVA_HOME/bin
```

- check your variables by typing:

```
echo $PATH
echo $JAVA_HOME
```

- you can store the above variables permanently by just adding the above commands to */etc/profile.d/myenvvars.sh* (or similar name):

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$PATH:$JAVA_HOME/bin
```

- install Maven:

```
wget https://ftp.fau.de/apache/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-
↪bin.tar.gz -P /your/path/to/
tar xzvf /your/path/to/apache-maven-3.6.3-bin.tar.gz -C /your/path/to/
export PATH=/your/path/to/apache-maven-3.6.3/bin:$PATH
```

1.4 Troubleshooting

```
mvn: not found
```

- if the install script keeps saying this although you are sure maven is installed on your device, check if your PATH variable (*echo $PATH*) points to maven's bin directory. Alternatively, you can add the absolute path to your maven installation to the script: *export PATH="$PATH:your/mvn/path"*

```
Cannot find xml2-config
ERROR: configuration failed for package 'XML'
```

- install libxml2-dev, e.g. when on Ubuntu *apt install libxml2-dev* or similar (the available package name depends on your Ubuntu distribution, which you can find out with the help of this tutorial https://itsfoss.com/unable-to-locate-package-error-ubuntu/)

```
FileNotFoundError: [Errno 2] No such file or directory: 'curl-config': 'curl-config
```

- this error comes from pycurl - install *libcurl4-openssl-dev* and *libssl-dev* packages, e.g. when on Ubuntu *apt install libcurl4-openssl-dev libssl-dev* or similar (the available package name depends on your Ubuntu distribution, which you can find out with the help of this tutorial https://itsfoss.com/unable-to-locate-package-error-ubuntu/)

```
WARNING: Failed to load implementation from: com.github.fommil.netlib.Native***␣
↪(SystemBLAS, RefBLAS, SystemLAPACK, RefLAPACK, SystemARPACK, RefARPACK)
```

- this can happen when running on Ubuntu and is related packages internally used by WEKA
- install *libgfortran-6-dev* package, e.g. when on Ubuntu *apt-get install libgfortran-6-dev* or similar (the available package name depends on your Ubuntu distribution, which you can find out with the help of this tutorial https://itsfoss.com/unable-to-locate-package-error-ubuntu/)

```
[ERROR] No compiler is provided in this environment. Perhaps you are running on a JRE␣
↪rather than a JDK?
```

- you either do not have a JDK installed or your variables point to the wrong location. Follow *Installing Java JDK and Maven on Ubuntu* for installing JDK and setting the environment variables correctly.

```
Configuration failed because libxml-2.0 was not found. Try installing:
* deb: libxml2-dev (Debian, Ubuntu, etc)
* rpm: libxml2-devel (Fedora, CentOS, RHEL)
* csw: libxml2_dev (Solaris)
If libxml-2.0 is already installed, check that 'pkg-config' is in your
PATH and PKG_CONFIG_PATH contains a libxml-2.0.pc file. If pkg-config
is unavailable you can set INCLUDE_DIR and LIB_DIR manually via:
R CMD INSTALL --configure-vars='INCLUDE_DIR=... LIB_DIR=...'
```

- lixml-2.0 is not installed. Follow the recommendations stated there and install it, e.g. by *apt-get install libxml2-dev* or similar (the available package name depends on your Ubuntu distribution, which you can find out with the help of this tutorial https://itsfoss.com/unable-to-locate-package-error-ubuntu/)

```
Configuration failed because openssl was not found. Try installing:
* deb: libssl-dev (Debian, Ubuntu, etc)
* rpm: openssl-devel (Fedora, CentOS, RHEL)
* csw: libssl_dev (Solaris)
* brew: openssl@1.1 (Mac OSX)
If openssl is already installed, check that 'pkg-config' is in your
PATH and PKG_CONFIG_PATH contains a openssl.pc file. If pkg-config
is unavailable you can set INCLUDE_DIR and LIB_DIR manually via:
R CMD INSTALL --configure-vars='INCLUDE_DIR=... LIB_DIR=...'
```

- openssl is not installed. Follow the recommendations stated there and install it, e.g. by *apt-get install libssl-dev* or similar (the available package name depends on your Ubuntu distribution, which you can find out with the help of this tutorial https://itsfoss.com/unable-to-locate-package-error-ubuntu/)

```
** package ‘xml2’ successfully unpacked and MD5 sums checked
Found pkg-config cflags and libs!
Using PKG_CFLAGS=-I/usr/include/libxml2
Using PKG_LIBS=-lxml2 -lz -llzma -licui18n -licuuc -licudata -lm -ldl
** libs
g++ -I/usr/share/R/include -DNDEBUG -I../inst/include -I/usr/include/libxml2 -DUCHAR_
↪TYPE=wchar_t    -fvisibility=hidden -fpic  -g -O2 -fstack-protector-strong -Wformat␣
↪-Werror=format-security -Wdate-time -D_FORTIFY_SOURCE=2 -g  -c connection.cpp -o␣
↪connection.o
In file included from /usr/include/unicode/uenum.h:23:0,
         from /usr/include/unicode/ucnv.h:53,
         from /usr/include/libxml2/libxml/encoding.h:31,
         from /usr/include/libxml2/libxml/parser.h:810,
         from /usr/include/libxml2/libxml/globals.h:18,
         from /usr/include/libxml2/libxml/threads.h:35,
```

(continues on next page)

(continued from previous page)

```
         from /usr/include/libxml2/libxml/xmlmemory.h:218,
         from /usr/include/libxml2/libxml/tree.h:1307,
         from xml2_utils.h:5,
         from connection.cpp:3:
/usr/include/unicode/localpointer.h:224:34: error: expected ‘,’ or ‘...’ before ‘&&’␣
↪token
LocalPointer(LocalPointer<T> &&src) U_NOEXCEPT : LocalPointerBase<T>(src.ptr) {
...
make: *** [connection.o] Error 1
ERROR: compilation failed for package ‘xml2’
* removing ‘/usr/local/lib/R/site-library/xml2’

The downloaded source packages are in
    ‘/tmp/Rtmpashma8/downloaded_packages’
Warning message:
In install.packages("xml2") :
installation of package ‘xml2’ had non-zero exit status
```

- There seems to be a different compiler required than what is currently provided in your *~/.R/Makevars* file. Add *CXX=g++ -std=c++11* (or whatever is stated at the very beginning of the error) to your *~/.R/Makevars* file. The problem and solution are also described here: https://github.com/r-lib/xml2/issues/294

```
Error: package or namespace load failed for ‘glmnet’ in dyn.load(file, DLLpath =␣
↪DLLpath, ...):
unable to load shared object '/usr/local/lib/R/site-library/00LOCK-glmnet/00new/
↪glmnet/libs/glmnet.so':
/usr/lib/x86_64-linux-gnu/libgfortran.so.5: version `GFORTRAN_1.0' not found␣
↪(required by /usr/local/lib/R/site-library/00LOCK-glmnet/00new/glmnet/libs/glmnet.
↪so)
```

- the R package glmnet (used by xtune package) needs a Fortran interpreter. If you have not installed it already, install it. If you have installed it already, adapt *~/.R/Makevars* and add *CC=gcc*

```
ImportError: pycurl: libcurl link-time ssl backends (secure-transport, openssl) do␣
↪not include compile-time ssl backend (none/other)
```

- Looks like something went wrong with pycurl/openssl. Try this:

  ```
  pip3 uninstall pycurl
  pip3 install --compile --install-option="--with-openssl" pycurl
  ```

  - if it still fails, try this as well:

    ```
    brew reinstall openssl
    ```

```
configparser.DuplicateOptionError: While reading from '../../configs/config.ini'␣
↪[line 17]: option 'rscriptlocation' in section 'R' already exists
```

- **If this error occurs, then you probably have adapted *config.ini* before installing Comprior, e.g. by removing or adding a li**

  - If you are executing **from source**:You can now either update *config.ini* directly or (more sustainable for the future) you can adapt *install.sh* and *install_macos.sh* scripts as they replace the values of parameters *RscriptLocation* and *JavaLocation* and *homePath* based on their line numbers. Update the script to contain the correct line number and then rerun the installation script.

- If you are executing **in a Docker container**: You need to adapt the *Dockerfile* and update the line numbers that are used to replace parameters *homePath*, *RscriptLocation*, *JavaLocation*, and *code*. Check if these parameters are still located in the correct line of *config.ini*. If not, update the line numbers given in the sed command that is executed there.

```
FileNotFoundError: [Errno 2] No such file or directory: '/my abs path/Comprior/data/
↪input/TCGA-SCANB/BRCA_TP_expressions_normalized.csv'
```

- If this error occurs when executing Comprior in a Docker container, you likely specified the absolute path to your input file in the config file. Make sure that you provide a path relative from your input directory: *input = ${General:inputDir}TCGA-SCANB/BRCA_TP_expressions_normalized.csv*. Docker containers have their own file structure, and the mounted directory you provide when running Comprior in a Docker container will be resolved to */home/app/data/* automatically. As such, if you provide an absolute path to your input files, Comprior will not be able to find it as this path does not exist (but instead it will exist at: */home/app/data/input/TCGA-SCANB/BRCA_TP_expressions_normalized.csv*).

CHAPTER 2

Example Use Cases

2.1 Breast Cancer

The prepared breast cancer use case tests selected feature selection approaches (both prior knowledge and traditional) to retrieve genes that best separate the samples into their PAM50 breast cancer subtypes (basal, normal-like, luminal A, luminal B, HER2). The repository contains two breast cancer data sets that are ready for execution with Comprior (links to R-based preprocessing scripts will be added later and are also available upon request):

- *TCGA.zip*: normalized BRCA samples (no normals) from TCGA including metadata. Contains 1090 samples for 20950 genes. This data must be downloaded by following the instructions in *TCGA_README.txt*.
- *SCANB_labeled.csv.zip*: labeled data set for cross-validation. Contains 378 SCANB samples for 15011 genes from the training cohort (GSE81538)
- unzip the files:

```
cd data/input/TCGA-SCANB
unzip SCANB_labeled.csv.zip
unzip TCGA.zip
```

- **make sure the file paths in *Comprior/code/configs/TCGA_SCANBconfig.ini* are still correct**
  - check *input* and *metadata* parameters in *Dataset* section and *crossEvaluationData* parameter in *Evaluation* section)
  - the resolved *inputDir* parameter is provided in the main config.ini file and should point to your local Comprior folder
  - adapt further configuration parameter as you like, e.g. add "Random" to *traditional_methods*

2.1.1 Execution via Source Installation

- make sure you installed Comprior (see *Install and Run Comprior from Source*) beforehand and *homePath*, *RscriptLocation*, and *JavaLocation* parameters are correctly set in the main *config.ini* file

- navigate to *code/Python/comprior* (assuming you are still located at *Comprior/data/input/TCGA-SCANB*):

```
cd ../../../code/Python/comprior
```

- start Comprior and provide the config file for this use case:

```
python3 pipeline.py --config ../../configs/TCGA_SCANBconfig.ini
```

- Check your results in *data/results/TCGA-SCANB_UseCase* - see *Folder Structure - Where to find what Files (In- and Output)* for an explanation on the folder structure.

2.1.2 Execution via Docker Image

- make sure you have built the docker container as described in *Run Comprior in a Docker Container*
- copy the folder containing the input files (*BRCA_TP_expressions_normalized.csv*, *BRCA_TP_metadata.csv*, and *SCANB_labeled.csv*) to *Comprior/comprior_docker/input* (assuming you are in Comprior's main directory):

```
cp -r data/input/TCGA-SCANB comprior_docker/input
```

- copy the config file *TCGA_SCANBconfig.ini* to Comprior/comprior_docker:

```
cp code/configs/TCGA_SCANBconfig.ini comprior_docker
```

- make sure the *input*, *metadata*, and *crossEvaluationData* parameters point to the right location. If your data is located within a subfolder of the *Comprior/comprior_docker/input* directory, add this to the parameter. **Do not provide absolute paths** and keep the *${General:inputDir}*, as it is internally resolved to point towards the input directory:

```
[Dataset]
input = ${General:inputDir}TCGA-SCANB/BRCA_TP_expressions_normalized.csv
metadata = ${General:inputDir}TCGA-SCANB/BRCA_TP_metadata.csv

[Evaluation]
crossEvaluationData = ${General:inputDir}TCGA-SCANB/SCANB_labeled.csv
```

- run the Docker container as root and provide the absolute path to *Comprior/comprior_docker* as mount directory (only change */your/absolute/path/to/* AND retype the double hyphen for *–config*; for some reason the config parameter will not be recognized if the statement is just copied from here):

```
sudo docker run -it --rm -v /your/absolute/path/to/Comprior/comprior_docker:/home/
↪app/data comprior --config /home/app/data/TCGA_SCANBconfig.ini
```

- Check your results in *Comprior/comprior_docker/results/TCGA-SCANB_UseCase* - see *Folder Structure - Where to find what Files (In- and Output)* for an explanation on the folder structure and *Output Generated by Comprior* for a more detailed description on the generated plots.

2.2 Glioma

The prepared glioma use case tests selected feature selection approaches (both prior knowledge and traditional) to retrieve genes that best separate the samples into their glioma subtypes astrocytoma, glioblastoma, and oligodendroglioma. The repository contains two glioma data sets that are ready for execution with Comprior(links to R-based preprocessing scripts will be added later and are also available upon request):

- *TCGA.zip*: normalized GBM and LGG samples (no normals) from TCGA including metadata. Contains 496 samples for 19301 genes.
- *REMBRANDT_labeled.csv.zip*: labeled data set for cross-validation. Contains 436 samples for 31442 probes from the REMBRANDT study (microarray data, GSE108474)
- unzip the files:

```
cd data/input/GBM-LGG
unzip REMBRANDT_labeled.csv.zip
unzip TCGA.zip
```

- **make sure the file paths in *Comprior/code/configs/GBMLGGconfig.ini* are still correct**
  - check *input* and *metadata* parameters in *Dataset* section and *crossEvaluationData* parameter in *Evaluation* section)
  - the resolved *inputDir* parameter is provided in the main config.ini file and should point to your local Comprior folder
  - adapt further configuration parameter as you like, e.g. add "Random" to *traditional_methods*

2.2.1 Execution via Source Installation

- make sure you installed Comprior (see *Install and Run Comprior from Source*) beforehand and *homePath*, *RscriptLocation*, and *JavaLocation* parameters are correctly set in the main *config.ini* file
- navigate to *Comprior/code/Python/comprior*(assuming you are still located at *Comprior/data/input/GBM-LGG*):

```
cd ../../../code/Python/comprior
```

- start Comprior and provide the config file for this use case:

```
python3 pipeline.py --config ../../configs/GBMLGGconfig.ini
```

- Check your results in *data/results/GBMLGG_UseCase* - see *Folder Structure - Where to find what Files (In- and Output)* for an explanation on the folder structure.

2.2.2 Execution via Docker Image

- make sure you have built the docker container as described in *Run Comprior in a Docker Container*
- copy the folder containing the input files (*GBM-LGG_TP_expressions_normalized.csv*, *GBM-LGG_TP_metadata.csv*, and *REMBRANDT_labeled.csv*) to *Comprior/comprior_docker/input* (assuming you are in Comprior's main directory):

```
cp -r data/input/GBM-LGG comprior_docker/input
```

- copy the config file *GBMLGGconfig.ini* to Comprior/comprior_docker:

```
cp code/configs/GBMLGGconfig.ini comprior_docker
```

- make sure the *input*, *metadata*, and *crossEvaluationData* parameters point to the right location. If your data is located within a subfolder of the *Comprior/comprior_docker/input* directory, add this to the parameter. **Do not provide absolute paths** and keep the *${General:inputDir}*, as it is internally resolved to point towards the input directory:

```
[Dataset]
input = ${General:inputDir}GBM-LGG/GBM-LGG_TP_expressions_normalized.csv
metadata = ${General:inputDir}GBM-LGG/GBM-LGG_TP_metadata.csv

[Evaluation]
crossEvaluationData = ${General:inputDir}GBM-LGG/SCANB_labeled.csv
```

- run the Docker container as root and provide the absolute path to *Comprior/comprior_docker* as mount directory (only change */your/absolute/path/to/* AND retype the double hyphen for *–config*; for some reason the config parameter will not be recognized if the statement is just copied from here):

```
sudo docker run -it --rm -v /your/absolute/path/to/Comprior/comprior_docker:/home/
↪app/data comprior --config /home/app/data/GBMLGGconfig.ini
```

- Check your results in *Comprior/comprior_docker/results/GBMLGG_UseCase* - see *Folder Structure - Where to find what Files (In- and Output)* for an explanation on the folder structure and *Output Generated by Comprior* for more a more detailed explanation on the generated plots.

2.3 Output Generated by Comprior

2.3.1 Detailed Processing Outputs

- *Comprior.log* contains more detailed outputs compared to the command line output. If you find that there is a figure or other output missing that you expected to be there, look at the log file as it is likely that a corresponding warning was posted there (e.g. Comprior does not automatically stop when a knowledge base does not return any results but just continues with the empty set of prior knowledge).

2.3.2 Plots on Datasets

If *preanalysis_plots* (Evaluation section) was provided with one or more keywords in your config file, Comprior creates corresponding plots for the input data sets (both the main and - if available - the labeled one for cross-validation). Colors for class labels are the same across all plots created. You can find the plots in *Comprior/data/results/YourExperimentName/preanalysis*

2.3.2.1 MDS plot

- keyword: mds
- multidimensional scaling plot showing dis-/similarities between samples
- output file name: *mds_InputDatasetName.pdf*

2.3.2.2 Density Plot

- keyword: density
- shows the average density distribution of expression levels per class label
- output file name: *density_InputDatasetName.pdf*

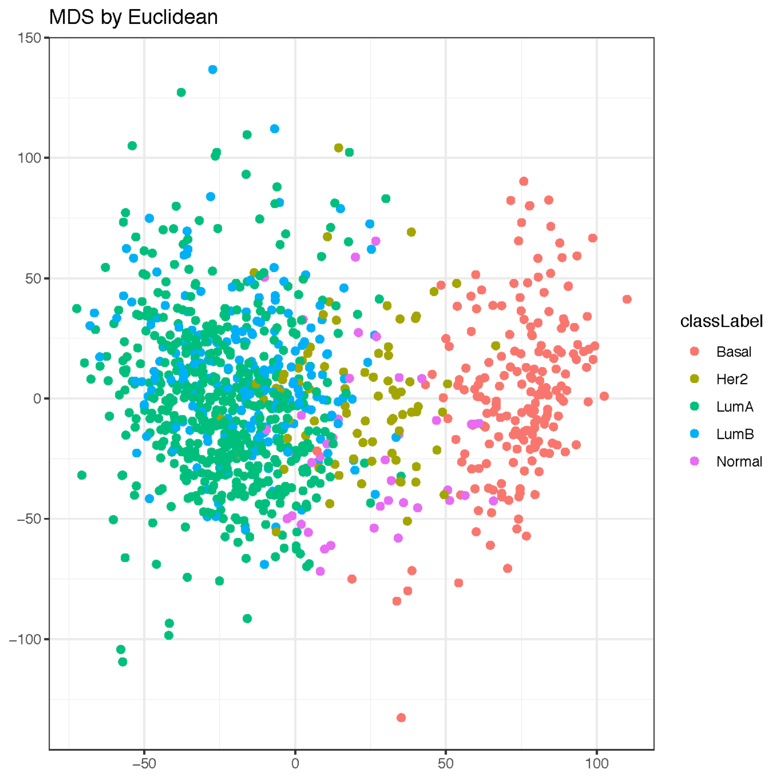


Fig. 1: Multidimensional scaling (MDS) plot for the input data set (TCGA) of the BRCA use case, one line per class label.

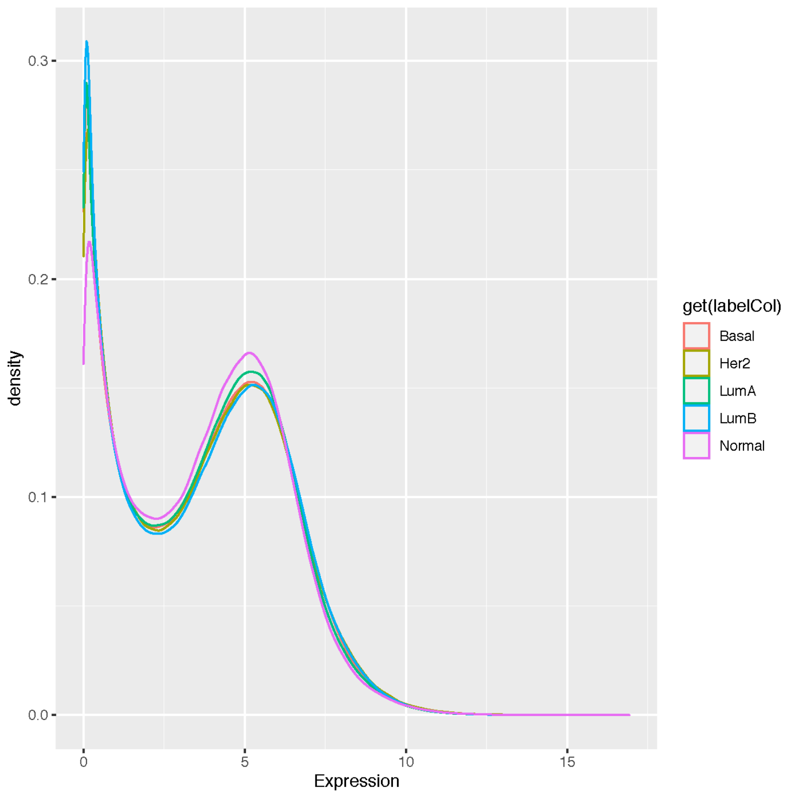


Fig. 2: Density plot for the input data set (TCGA) of the BRCA use case, one line per class label.

2.3.2.3 Box Plot

- keyword: box
- shows the expression levels with one box per class label
- output file name: *distribution_InputDatasetName.pdf*

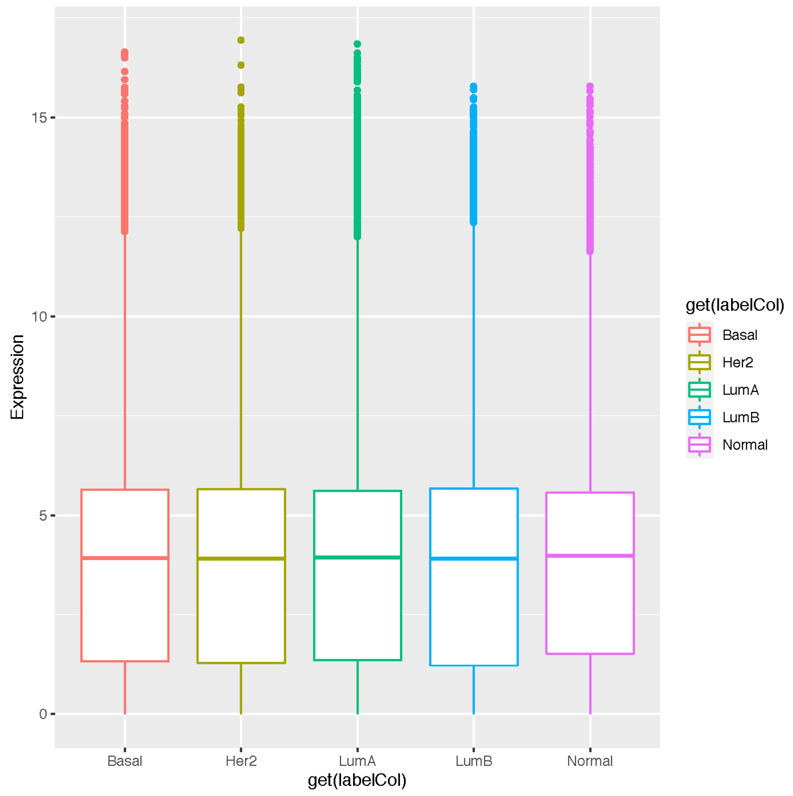


Fig. 3: Box plot for the input data set (TCGA) of the BRCA use case, one box per class label.

2.3.3 Knowledge Base Coverage

If *evaluateKBcoverage* (Evaluation section) is set to true in your config file, Comprior examines how much the search terms provided in the config file are covered by the knowledge bases that are used in this experiment. For example, if you use Postfilter_Variance_DisGeNET, then Comprior will check the coverage of DisGeNET only. You can find the plots in *Comprior/data/results/YourExperimentName/preanalysis*.

If a search term has more than 15 signs, Comprior will map the search terms to a shorter ID and use this instead for the plots. The mapping from search term to ID is then provided in *Comprior/data/results/YourExperimentName/preanalysis/searchterm_IDs.txt*.

2.3.3.1 Gene Coverage Plots

- created for every knowledge base (see *Gene Association Score Computation from Network Information* for how an individual gene association score is computed for interaction knowledge bases that only retrieve network information)
- file name: *KnowledgeBaseName_GeneCoverage.pdf* (source file: *KnowledgeBaseName_GeneStats.csv*)

- box plot shows the distribution of association scores (left-hand y axis) that are returned for a given search term, the underlying bar plot shows how many genes (right-hand y axis) were returned per search term (attention: gene sets are not disjunct, i.e. results for search terms can show a high overlap if the search terms are similar)

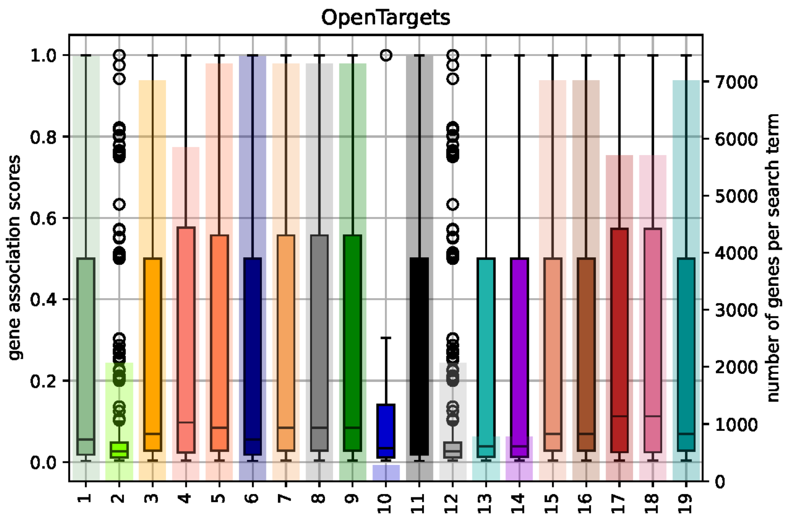


Fig. 4: Prior knowledge coverage in OpenTargets for the search terms used in the GBM-LGG use case. Boxes show the association scores returned for the genes related to the search term, bars showing the overall number of genes returned for a search term.

2.3.3.2 Pathway Coverage Plots

- created only for knowledge bases providing network information, e.g. KEGG or PathwayCommons
- file name: *KnowledgeBaseName_PathwayCoverage.pdf* (source file: *KnowledgeBaseName_PathwayStats.csv*)
- box plot shows the distribution of network sizes (i.e., number of member genes, left-hand y axis) that are returned for a given search term, the underlying bar plot shows how many networks (right-hand y axis) were returned per search term (attention: pathway sets are not disjunct, i.e. results for search terms can show a high overlap if the search terms are similar)

2.3.4 Feature Rankings

Every feature selection approach creates a corresponding feature ranking of all the input features. The corresponding rankings (*ApproachName.csv*) are located at *Comprior/data/results/YourExperimentName/GeneRankings/* and contains an ordered feature list, thus the top k features in the list will be used for classification later on. Based on what keywords are provided for the *metrics* (Rankings section) parameter in your config file, Comprior creates corresponding plots in *Comprior/data/results/YourExperimentName/evaluation/rankings* and uses a consistent color scheme.

2.3.4.1 Ranking Overlaps

- file name: *metrics/geneSignatureOverlaps.pdf*
- <= 3 feature rankings: Venn diagram
- > 3 feature rankings: Upset plot
- shows overlaps of the top k feature sets for all feature selection approaches

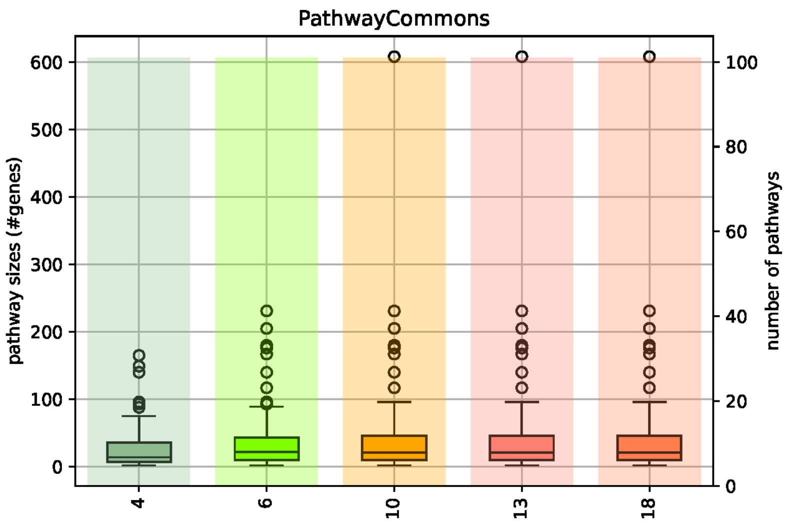


Fig. 5: Prior knowledge coverage in PathwayCommons for the search terms used in the GBM-LGG use case. Boxes show the pathway sizes (=number of genes) of the retrieved pathways, bars showing the overall number of pathways returned for a search term.

- hint: *NetworkActivity* and *CorgsNetworkActivity* will not have any overlap with other approaches, as they have pathways as features and not genes as the others

2.3.5 Feature Annotations and Enrichments

Comprior uses Enrichr to a) annotate the top k features and b) enrich these feature sets with terms. You can select the library to be used for that by providing its corresponding name as stated by Enrichr (see here) in the *geneSetLibrary* parameter (Enrichr section) in your config file. The plots and their source files are located at *Comprior/data/results/YourExperimentName/evaluation/rankings/*.

2.3.5.1 Annotation Overlaps

- file name: *annotation/overlaps_annotatedGenes.pdf* (based on source files with annotations: *topk_ApproachName_annotatedGenes.csv*)
- <= 3 approaches used: Venn diagram, > 3 approaches used: Upset plot
- shows overlaps of annotated terms for the top k annotated terms per feature set (=how many annotations feature sets share)
- for example, two feature sets might not actually have an overlap in their features, but some features might be annotated with the same terms

2.3.5.2 Enrichment Overlaps

- file name: *annotation/overlaps_enrichedTerms.pdf* (based on source files with annotations: *topk_ApproachName_enrichedTerms.csv*)
- <= 3 approaches used: Venn diagram, > 3 approaches used: Upset plot
- shows overlaps of the top k enriched terms per feature set (=how much the enriched terms identified for the feature sets overlap)

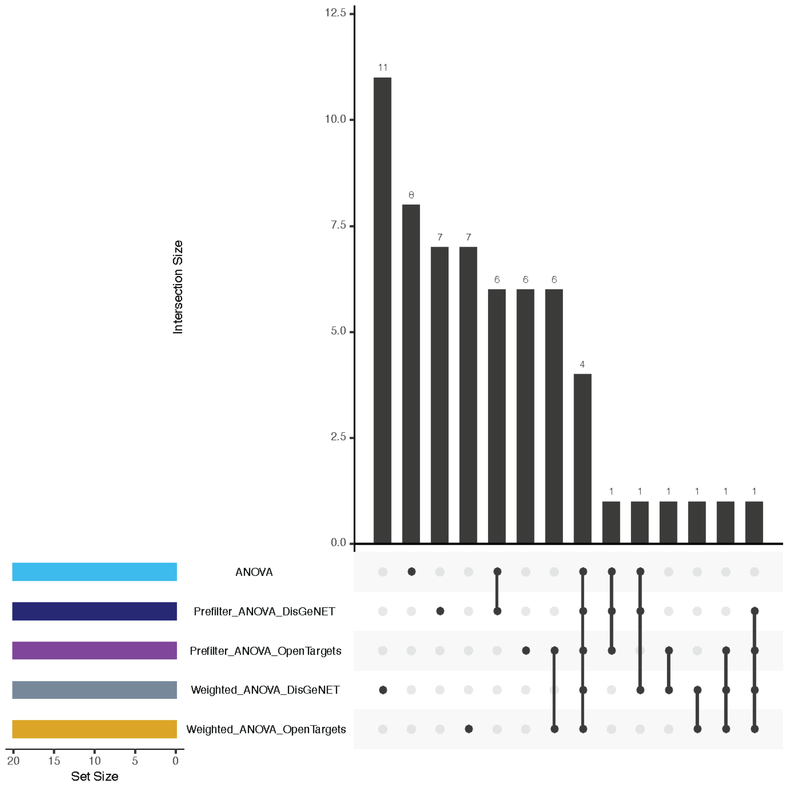


Fig. 6: Feature set overlaps for the top 20 features selected by the approaches used in the BRCA use case.

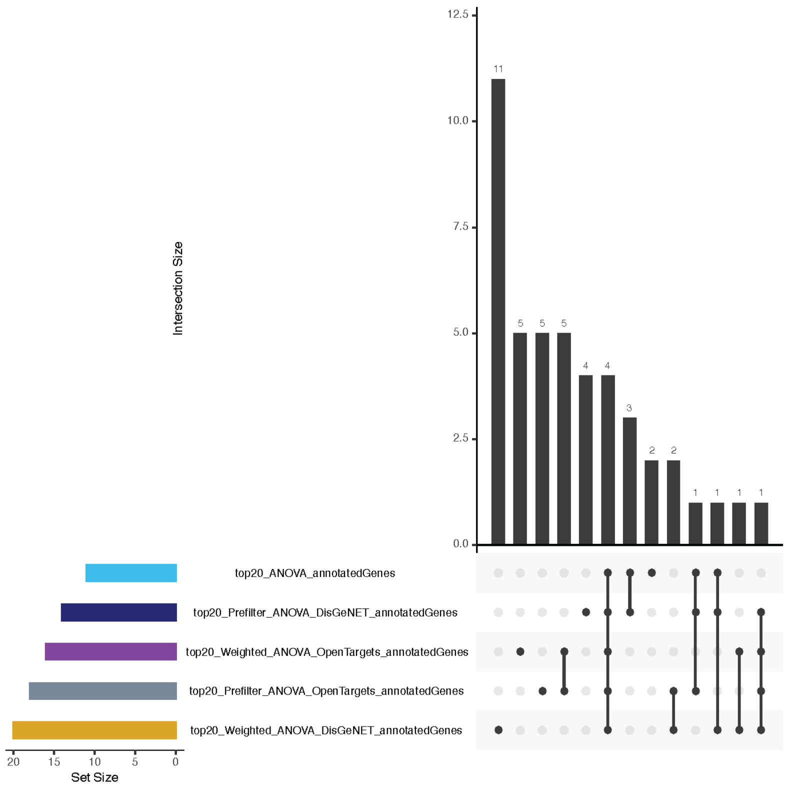


Fig. 7: Overlaps of top 20 annotations made to the feature sets (top 20) selected by the approaches used in the BRCA use case.

- for example, two feature sets might not actually have an overlap in their features, but in both feature sets the same terms are enriched

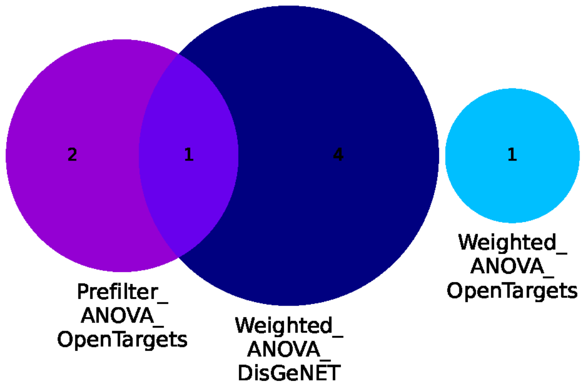


Fig. 8: Overlaps of top 20 enrichments for feature sets (top 20) selected by the approaches used in the BRCA use case.

2.3.6 Classification Performance

Comprior uses the top k features from every approach and classifies the original data set (and, if provided, the second labeled data set) with these features according to the parameters (which classifiers to use, k-fold cross-validation, which metrics to apply, etc.) specified in the config file. For every metric selected, Comprior creates line plots that show the average classification metric, e.g. accuracy, for all feature selection approaches. The plots and their source files for metrics (of type *ApproachName_MetricName.csv*) on the original data set are located at *Comprior/data/results/YourExperimentName/evaluation/classification/metrics*. The plots and their source files for metrics (of type *ApproachName_MetricName.csv*) on the cross-validation data set are located at *Comprior/data/results/YourExperimentName/evaluation/classification/crossEvaluation/classification*. The colors assigned to the approaches are consistent across all plots generated by Comprior.

2.3.7 Feature Selection Runtimes

Comprior also logs runtimes of feature selection approaches (though no plots are created currently). Runtime statistics are located at *Comprior/data/results/YourExperimentName/timeLogs*, with one file per feature selection approach. The last line of each file always contains the overall runtime for feature selection. The remaining lines trace runtimes of single parts of the feature selection process, e.g. of prior knowledge retrieval or a traditional feature selection strategy.

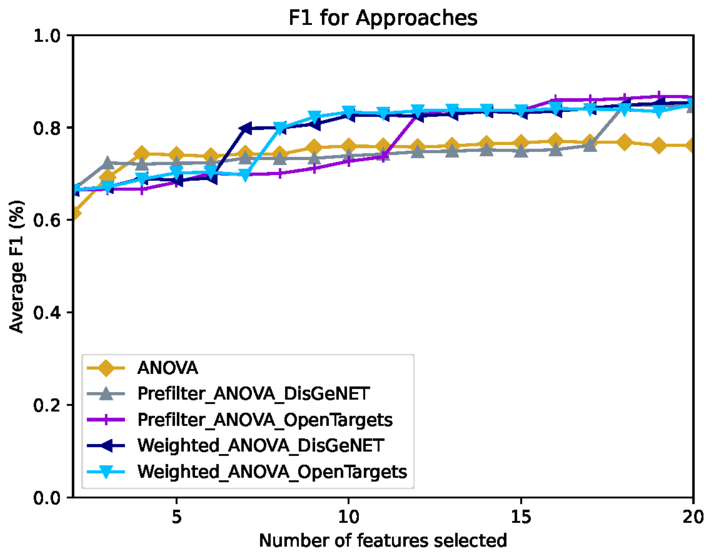


Fig. 9: Average F1 score for increasing feature set sizes selected by the approaches used in the GBM-LGG use case (original TCGA data set was classified).

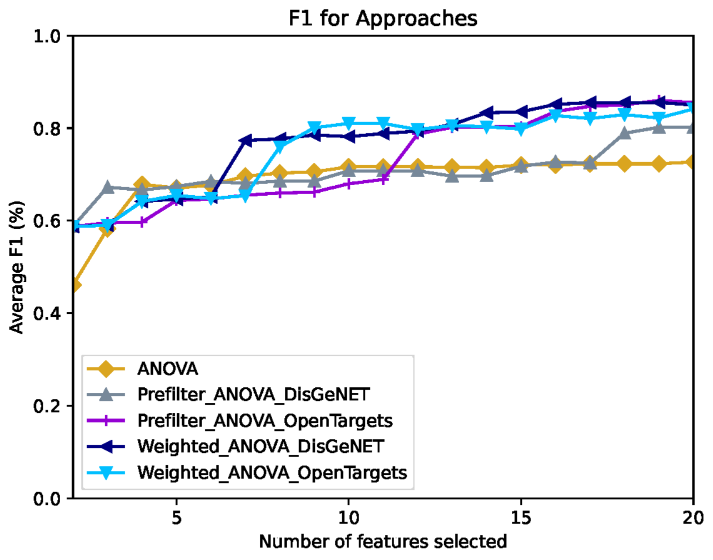


Fig. 10: Average F1 score for increasing feature set sizes selected by the approaches used in the BRCA use case (labeled SCANB data set was classified, features were originally selected on the TCGA data set).

CHAPTER 3

Input Data Sets

3.1 Gene Expression Data

The gene expression dataset must be provided in table format. Genes/features can be located either in columns or rows, however make sure to set the *genesInColumns* parameter accordingly in your config file. Data separators can be chosen arbitrarily, however make sure to set the *dataSeparator* parameter accordingly. **Make sure to always leave the first column of the header empty as shown here**:

,ERBB2,TP53,DVL1,BRCA1,BRCA2
Sample1,12.02,4.12,11.25,9.87,10.02
Sample2,10.32,4.76,10.73,10.94,8.72

3.2 Metadata

Metadata must always be provided for an input data set. It must have the same separator as the gene expression data (pay attention: here you do not have to leave a blank column). Whether samples or metadata type is located in the columns can be specified with the *metadataIDsInColumns* parameter. Specify the class labels by selecting a corresponding metadata type with the *classLabelName* parameter (e.g. if we want to have subtypes as class labels, specify *subtype*). If the metadata provides further information on the disease that you want to provide as search terms for the knowledge bases, specify it with the *diseaseLabelName* parameter (e.g. by setting it to *primary_diagnosis*).

Sample1,Sample2
project_id,BRCA,BRCA
subtype,LumA,LumB
gender,female,female
primary_diagnosis,"Infiltrating duct carcinoma, NOS","Lobular carcinoma, NOS"

3.3 Data for Cross-Validation

Preprocessing for data sets for cross-validation is currently not supported (except for identifier mapping), so the data must be provided with a) genes in columns and b) the corresponding class labels. Make sure to set the name of the class label column in the config file with the *crossEvaluationClassLabel* parameter (in the example, it would be *diseaseCode*).

diseaseCode,SampleName,ERBB2,TP53,DVL1,BRCA1,BRCA2
LumA,Sample1,12.02,4.12,11.25,9.87,10.02
LumB,Sample2,10.32,4.76,10.73,10.94,8.72

CHAPTER 4

Configuration Parameters

Config parameters are grouped into sections and needed by the framework to function properly. In the following, we provide brief explanations for the existing parameters and their values

4.1 General

- **name** - framework name, mainly used for logging
- **numCores** (*int*) - number of available cores that can be used for parallel running of gene selectors
- **homePath** - contains the path to the framework
- **inputDir** - the path to the directory where all input data is located
- **intermediateDir** - where to put intermediate results (recommended not to changed)
- **externalKbDir** - where to put intermediate query results from knowledge bases (recommended not to changed)
- **resultsDir** - where to put the final results (recommended not to changed)
- **preprocessing** - where to put preprocessed data sets (recommended not to changed)
- **crossVal_preprocessing** - where to put preprocessed data sets for cross-validation (recommended not to changed)
- **outputDir_name** - name of the overall output directory. If that directory already exists, Comprior adds numbering.
- **log_filename** - name of the log file to which processing information, warnings, etc. are written

4.2 R

- **code** - path to R source files (recommended not to changed)
- **RscriptLocation** - path to Rscript

4.3 Java

- **code** - path to Java jar files (recommended not to changed)
- **JavaLocation** - path to Java

4.4 Dataset

- **input** - path to expression dataset
- **metadata** - path to metadata file for expression dataset
- **classLabelName** - specify the metadata column name of the label to use for classification (=keywords that will also be used for search in knowledge bases)
- **alternativeSearchTerms** - specify alternative search terms that will be used by knowledge bases. Separate search terms by spaces and replace spaces within a search term with _, e.g. "Breast Cancer Kidney_Cancer" will be parsed to the following search terms:"Breast", "Cancer", "Kidney Cancer"
- **genesInColumns** (*true/false*) - specify if genes are in the columns so that the data can be transformed automatically
- **metadataIDsInColumns** (*true/false*) - specify if sample IDs in the metadata file are in the columns
- **dataSeparator** (*sep*) - specify file separator (must be the same for metadata and gene expression), e.g. ,, *t*
- **currentGeneIDFormat** - specify the current gene ID format with g:Convert IDs, e.g. *ENTREZGENE, ENSG, AFFY_HG_U133_PLUS_2, HGNC* (see https://biit.cs.ut.ee/gprofiler/convert)
- **finalGeneIDFormat** - specify the gene ID format with g:Convert IDs you want to have in your gene rankings, e.g. *HGNC* (see https://biit.cs.ut.ee/gprofiler/convert)

4.5 Preprocessing

- **filterMissingsInGenes** (*true/false*) - filter genes that have a higher percentage of missing fields than specified in treshold parameter
- **filterMissingsInSamples** (*true/false*) - filter samples that have a higher percentage of missing fields than specified in the treshold parameter
- **threshold** (*[0..100]*) - percentage used for filtering

4.6 Gene Selection - General

- **outputDirectory** - where the final gene rankings are stored (recommended not to changed)
- **selectKgenes** (*int*) - maximum number of genes to select (to reduce runtimes). Must be >= topKmax param in *Evaluation*.

4.7 Gene Selection - Methods

- **traditional_methods** - select multiple traditional gene selection methods (separated by spaces):

  - filter: *Random*, *InfoGain*, *ReliefF*, *VB-FS* (R-based variance selection), *ANOVA*, *mRMR*, *Variance* (Python-based variance selection)
  - wrapper: *SVMpRFE* (SVM-RFE with polykernel), *x-SFS*, *x-RFE* (x = add the desired classifier: K-nearest Neighbor (KNN3, KNN5), Naive Bayes (NB), Linear Regression (LR), Support Vector Machines with linear kernel (SVMl))
  - embedded: *RandomForest*, *Lasso*
- **modifying_methods** - combine knowledge base with traditional approach from above. Currently implemented:
  - *Postfilter_trad_kb*: Filter the features selected by traditional approach (*trad*) by the features/genes retrieved from knowledge base (*kb*)
  - *Prefilter_trad_kb*: Filter the input features by the genes retrieved from the knowledge base (*kb*) first and forward this reduced input set to the traditional approach (*trad*) for feature selection
  - *Extension_trad_kb*: Extends features selected by traditional approach (*trad*) by features/genes retrieved from knowledge base (*kb*). Traditionally selected features and external features/genes make up 50% of top k features, respectively (so if topKmax param is 2, 1 feature from traditional approach, 1 feature from external will be selected)
  - *KBonly_kb*: Only use the scores from the knowledge base to rank features, setting a default score of 0.000001.
- **combining_methods** - select combining methods to apply. Combining methods should combine a traditional approach (*trad*) with a knowledge base (*kb*), the prefix indicates the actual combining strategy. Currently implemented:
  - *Weighted_trad_kb*: weights the score from traditional approach *trad* by the score retrieved from knowledge base *kb*)
  - *LassoPenalty_kb*: includes external knowledge via Lasso penalty as described by Zeng et al.: "Incorporating prior knowledge into regularized regression"
- **network_methods** - select network approaches to apply. Currently implemented:
  - *NetworkActivity_kb*: retrieves pathway from knowledge base *kb*, ranks them via average of (ANOVA of gene expression value/sample class) for every gene in pathway, and creates an activity score as new feature value for the pathway for every sample (= average of (gene expression value x variance x average of (Pearson correlation with network neighbors))) for all genes in the pathway)
  - *CorgsNetworkActivity_kb*: retrieves pathway from knowledge base *kb*, ranks them via average of (ANOVA of gene expression value/sample class) for every gene in pathway, and creates an activity score as new feature value for the pathway for every sample as described by Lee et al.: "Inferring Pathway Activity toward Precise Disease Classification"

4.8 Evaluation

- **topKmin** (*int*) - minimum number of features to select
- **topKmax** (*int*) - maximum number of features to select
- **kfold** (*int*) - k parameter for k-fold cross-validation during classification
- **results** - where to put the results (recommended not to changed)
- **reducedDataset** - where to put the reduced data sets (with k features) (recommended not to changed)
- **preanalysis** - where to put plots created during preanalysis
- **preanalysis_plots** - create plots on input data before any analysis. Currently implemented:

- *density*: density plot showing the average density distribution of expression values (per class)
- *box*: box plot showing the average gene expression (per class)
- *mds*: multidimensional scaling plot showing dis-/similarities between samples

- **evaluateKBcoverage** (*true/false*) - create diagrams showing coverage of search terms in the used knowledge bases
- **robustnessResults** - where to put the cross-validation results (recommended not to changed)
- **enableCrossEvaluation** (*true/false*) - whether to use a second data set for cross-validation
- **crossEvaluationData** - path to second data set for cross-validation (must have genes in columns and be already labeled)
- **crossEvaluationClassLabel** - column name of second data set for cross-validation containing the class label
- **crossEvaluationGeneIDFormat** - current g:Convert gene ID format of the second data set for cross-validation (see https://biit.cs.ut.ee/gprofiler/convert)
- **enableClassification** (*true/false*) - use classification algorithms for evaluation
- **enablePrediction** (*true/false*) - use predictive algorithms for evaluation (functionality not implemented yet)

4.9 Rankings

- **results** - where to put the actual rankings (recommended not to change)
- **metricsDir** - where to put the metric results (recommended not to change)
- **annotationsDir** - where to put the annotation information(recommended not to change)
- **metrics** - specify which evaluation metrics to apply/compare on feature rankings. Currently provided:
  - *top_k_overlap*: overlap of top k features of rankings
  - *kendall_w*: Kendall's W ranking comparison
  - *fleiss_kappa*: Fleiss' Kappa ranking comparison
  - *annotation_overlap*: shows gene annotation overlap in rankings (e.g. which annotated genes where found jointly by all rankings)
  - *enrichment_overlap*: shows enrichment term overlap of rankings (e.g. which enrichment terms where found jointly by all rankings)
  - *annotation_percentage*: compares average p
  - *average_foldchange*: compares average fold change of genes in rankings

4.10 Classification

- **classifiers** - specify classifiers to use for classsification task. Currently provided:
  - *KNN(int)*: K-nearest Neighbor, e.g. *KNN3*
  - *NB*: Naive Bayes
  - *LR*: Linear Regression
  - *SMO*: Support Vector Machines

   - *RF*: Random Forest
- **metrics** - specify which evaluation metrics to apply on classification results. Currently provided:
   - *accuracy*
   - *sensitivity*
   - *specificity*
   - *F1*
   - *kappa*
   - *AUROC*
   - *precision*
   - *matthewcoef*: Matthews Correlation Coefficient
- **results** - where to put the classification results (recommended not to change)
- **crossEvaluationDir** - where to put classification results from cross-validation (recommended not to change)
- **metricsDir** - where to put the evaluation results (recommended not to change)

4.11 Prediction (not implemented yet)

- **predictors** - specify predictors to use for prediction task (still under construction)
- **metrics** - specify which evaluation metrics to apply on prediction results (still under construction)
- **results** - where to put the classification results (recommended not to change)
- **crossEvaluationDir** - where to put classification results from cross-validation (recommended not to change)
- **metricsDir** - where to put the evaluation results (recommended not to change)

4.12 Enrichr

- **webservice_uri** - URL of Enrichr web service (recommended not to change)
- **outputDir** - output directory for intermediate results from web service (recommended not to change)
- **geneSetLibrary** - gene set library to use for annotation/enrichment. Choose any available at https://maayanlab.cloud/Enrichr/#stats

4.13 OpenTargets

- **outputDir** - output directory for intermediate results from web service (recommended not to change)

4.14 KEGG

- **outputDir** - output directory for intermediate results from web service (recommended not to change)
- **maxNumPathways** (*int*) - specify the maximum number of pathways to retrieve per search term (for performance reasons)

4.15 UMLS (needed to transform search terms into CUIs for using DisGeNET)

- **login_uri** - URL of login web service (recommended not to change)
- **loginservice_uri** - URL of login web service (recommended not to change)
- **auth_endpoint** - authentication endpoint for API (recommended not to change)
- **apikey** - API key for accessing UMLS (recommended not to change unless own API key available, register for free at UMLS to create an own key)
- **webservice_uri** - URL of UMLS web service (recommended not to change)

4.16 DisGeNET

- **associationScore** (*score/gene_dsi/gene_dpi*) - which association score to use for knowledge retrieval: score (overall score), gene_dsi (disease specificity), gene_dpi (disease pleiotropy)
- **webservice_url** - URL of DisGeNET web service (recommended not to change)
- **outputDir** - output directory for intermediate results from web service (recommended not to change)
- **apikey** - API key for accessing DisGeNET (recommended not to change unless own API key is available, register at DisGeNET to create your own key)

4.17 PathwayCommons

- **webservice_url** - URL of PathwayCommons web service (recommended not to change)
- **outputDir** - output directory for intermediate results from web service (recommended not to change)
- **maxNumPathways** (*int*) - specify the maximum number of pathways to retrieve per search term (for performance reasons)

4.18 BiomaRt

- **outputDir** - output directory for intermediate results from web service (recommended not to change)

4.19 gConvert

- **webservice_url** - URL of g:Convert web service (recommended not to change)
- **outputDir** - output directory for intermediate results from web service (recommended not to change)

CHAPTER 5

Folder Structure - Where to find what Files (In- and Output)

Unless the paths are not adapted/overwritten in *config.ini*, Comprior builds up the following folder infrastructure during processing:

```
data
 ├── input
 │   ├── dataset
 │   └── example
 ├── intermediate
 │   ├── dataset
 │   ├── crossvalidation
 │   └── externalKnowledge
 └── results
     └── XXX
         ├── timeLogs
         ├── preanalysis
         ├── geneRankings
         └── evaluation
             ├── rankings
             │   ├── annotation
             │   └── metrics
             ├── reducedData
             └── classification
                 ├── metrics
                 └── crossEvaluation
                     ├── reducedData
                     └── classification
```

5.1 input/

- **dataset/**: put your input dataset here
- **example/**: folder with example files for trying out Comprior

5.2 intermediate/

- **dataset/**: preprocessed input data (currently metadata added to one file)
- **crossvalidation/**: contains preprocessed dataset for cross-validation (e.g. mapped to the right identifier or pathway features)
- **externalKnowledge/**: one sub-folder per knowledge base that is queried with query results

5.3 results/

- **XXX/**: output folder for the current run, whose name is specified by the *outputDir_name* parameter in *config.ini* (if there already exists a folder with such a name, Comprior adds a number to the name)
  - **timeLogs/**: one file for every selected approach, containing logs with time durations of different selection activities, e.g. external knowledge retrieval or statistical feature selection
  - **preanalysis/**: contains - if selected via *preanalysis_plots* and *evaluateKBcoverage* parameters in *config.ini* - plots on data set characteristics and knowledge base coverage
  - **geneRankings/**: contains the actual feature rankings, one CSV file for every selected approach
  - **evaluation/**: contains all evaluation results
    - **rankings/**: contains evaluation results from analyzing the feature rankings
      - **annotation/**: contains annotation/enrichment files for every ranking
      - **metrics/**: contains the actual metrics results to compare the rankings
    - **reducedData/**: one sub-folder per selection approach containing input data for the top k features; these files are used for the actual classification/prediction
    - **classification/**:
      - **metrics/**: contains the actual classification metrics results, one CSV file for every selected metric, also contains pdfs for visualizations
      - **crossEvaluation/**: contains evaluation data from the second data set for cross-validation
      - **reducedData/**: one sub-folder per selection approach containing input data (second data set) for the top k features; these files are used for the actual classification/prediction
      - **classification/**: contains the actual classification metrics results, one CSV file for every selected metric, also contains pdfs for visualizations

CHAPTER 6

Knowledge Bases

- Comprior queries a knowledge base to retrieve relevant genes, gene association scores, or pathway information, which is then integrated into a prior knowledge feature selection approach
- for the search, Comprior uses a) the class labels and b) alternative search terms that are provided in the config file (see *alternative search terms* parameter in configuration specification)

6.1 DisGeNET

- aggregates biological information from multiple sources (meta knowledge base, see sources description for original sources)
- provides gene-disease, variant-disease, and variant-gene association scores for genes, disease, and variants (no pathway information) –> used for retrieving relevant genes and gene association scores
- users can choose in the config whether to use DisGeNET's gene-disease association (GDA) score, disease pleiotropy index, or disease specificity index (see their description for more information)
- DisGeNET uses UMLS identifiers for the search, so Comprior internally first maps the search terms to their corresponding UMLS CUI (via the UMLS terminology web service) and forwards these to DisGeNET

6.2 OpenTargets

- provides biological information from multiple sources (meta knowledge base, see their documentation for original sources)
- provides association gene-disease association scores (no pathway information) –> used for retrieving relevant genes and gene association scores

6.3 KEGG

- provides manually curated pathway information –> used for retrieving relevant genes, gene association scores, and pathway information
- pathways are parsed into pypath.Network format

6.3.1 Retrieving Relevant Genes from Pathway Information

- Comprior retrieves relevant genes from a set of pathways by selecting all their member genes and removing duplications

6.3.2 Gene Association Score Computation from Network Information

- Comprior computes a gene association score s_i for a gene i from the sum of its degree percentile rank $pr_{n,i}$ (= pathway genes are ranked by their number of in- and outgoing edges) for every pathway n, normalized by P_i (=the overall number of pathways containing gene i): $\frac{\sum_{n=1}^{|P_i|} pr_{n,i} if i \in p_n}{|P_i|}$
- this way, hub genes with a lot of interactions receive a higher score than genes at the outside rim of a pathway, becoming even more important if they have many interactions across multiple pathways

6.4 PathwayCommons

- provides pathway information from multiple sources (meta knowledge base, see their sources for original sources)
- relevant genes and gene association scores are currently retrieved the same way as KEGG
- pathways are parsed into pypath.Network format

CHAPTER 7

Prior Knowledge Approaches

- Comprior provides multiple prior knowledge approaches of types *Modifying Prior Knowledge Approaches*, *Combining Approaches*, *Network/Pathway Approaches* as defined by Perscheid: "Integrative biomarker detection on high-dimensional gene expression data sets: a survey on prior knowledge approaches"
- all of them can be flexibly combined with any of the available knowledge bases (see the configuration parameter description on how to do that)

7.1 Modifying Prior Knowledge Approaches

- type of prior knowledge used: list of relevant genes (no association scores)
- traditional feature selection and prior knowledge retrieval are carried out independently
- Comprior allows to design flexible modifying prior knowledge approaches that can be combined with any knowledge base and any traditional approach
- kind of two-level approaches that introduce an additional filtering or extension step before or after a traditional feature selection approach

7.1.1 Filtering

- *Prefilter*: prior knowledge is retrieved first and the input data set is filtered for those genes that were retrieved from the knowledge base; traditional feature selection is carried out afterwards
- *Postfilter*: Traditional feature selection is carried out first, and the resulting features are then filtered to keep only those that were also retrieved by the knowledge base
- prefilter and postfilter approaches have the same results for univariate feature selection approaches, e.g. Variance

7.1.2 Extension

- Comprior retrieves relevant genes and interleaves the gene ranking retrieved by a traditional approach with the set of relevant genes from the knowledge base
- this way, a feature set always not only contains traditionally selected genes, but also nearly as much genes that were retrieved from a knowledge base so that the feature set can contain genes that have a high statistical relevance but no (so far identified) biological relevance according to the knowledge base and vice versa

7.2 Combining Approaches

- type of prior knowledge used: relevant genes and their association scores (for the search terms)
- traditional feature selection and prior knowledge retrieval are carried out in parallel and integrated more thoroughly
- if a gene has multiple association scores (because it is associated to multiple search terms), Comprior will always keep the highest association score and remove the duplicate entries
- potentially, network information can also be retrieved via Comprior and then be mapped to some kind of relevance score, e.g. by incorporating topological information of a gene

7.2.1 LassoPenalty

- gene association scores are used as individual penalty term per feature applied to Lasso
- Comprior uses the xtune R package implementation by Zeng et al.: "Incorporating prior knowledge into regularized regression"

7.2.2 WeightedScore

- the final relevance score s_i for a gene i is made up of two parts: the association score from the knowledge base $s_{i,kb}$, and the statistical relevance score $s_{i,trad}$ from a traditional approach
- both scores are equally weighted to compute the final relevance score for a gene: $s_i = s_{i,kb} \times s_{i,trad}$

7.3 Network/Pathway Approaches

- network/pathway approaches use network information to identify (sub-) networks or pathways as new features and map the feature space from the original genes to the (sub-)networks
- network/pathway approaches thus always have a) a feature, i.e. pathway/subnetwork, selection step and b) a mapping step where new feature values must be computed

7.3.1 NetworkActivity

- feature selection as described by Tian et al.: "Discovering statistically significant pathways in expression profiling studies"
  - a pathway/subnetwork is considered relevant if the gene expression profiles of its member genes correlate with the data set classes

  - average ANOVA score from all pathway member genes and class labels
  - rank pathways (= new features) by their ANOVA scores
- feature mapping is based on Vert and Kanehisa's definition of pathway relevance and smoothness: "Graph-driven feature extraction from microarray data using diffusion kernels and kernel CCA"
  - omputes pathway activity scores for every sample and pathway as new feature values.
  - the feature value $v_{p,s}$ for a pathway $p`andsample: math: `s$ is computed by taking the expression levels of all member genes i ($expr_i$) and weighting these by the variance var_i of gene i and the average correlation score $corr_{i,neighbors_i}$ of its neighbor genes in pathway p: $average(expr_i \times var_i \times corr_{i,neighbors_i})$

7.3.2 CorgsNetworkActivity

- feature selection as described by *NetworkActivity*
- **feature mapping as described by Lee et al: "Inferring pathway activity toward precise disease classification"**
  - the feature value $v_{p,s}$ for a pathway p and sample s is computed in the following way:
    * find the subset of genes (=CORGs) for which the score $S(CORGs)$ is maximal (via greedy search)
    * $S(CORGs)$ comes from a t-test between an activity vector $a = (a_1, ..., a_n)$ and class vector $c = (c_1, ...c_n)$ with $n = \#samples$, i.e. every sample i gets an activity score a_i for the particular set of genes, c_i is the class label of that sample
    * a_i is computed from $\frac{average(expr_{i,CORGs})}{\sqrt{k}}$, with $k = \#CORGs$ and $expr_{i,CORGs}$ being the expression values of all CORGs genes for sample i

CHAPTER 8

Extending Comprior - How-Tos

8.1 Add New Preprocessing Functionality

8.1.1 1. Implement a new Preprocessor

Every preprocessing functionality must be implemented in its own class. To achieve that, create a new class that inherits from *preprocessing.Preprocessor* and implements *preprocessing.Preprocessor.preprocess()* method, e.g.

```
class ExamplePreprocessor(Preprocessor):
  """Does some example preprocessing

    :param input: absolute path to the input file to preprocess.
    :type input: str
    :param output: absolute path to the output directory where to store the␣
↪preprocessing results.
    :type output: str
    :param whatever_you_need: whatever other parameters your preprocessor needs.
    """

  def __init__(self, input, output, whatever_you_need):
    self.whatever_you_need = whatever_you_need
    super().__init__(input, None, output)

  def preprocess(self):
    """Does some example preprocessing

       :return: absolute path to the new file location.
       :rtype: str
       """
    #implement your preprocessing here...

    return self.output
```

8.1.2 2. Update the Config File (optional)

If your preprocessing functionality requires that the user sets any specific parameters, adapt the preprocessing section of *config.ini* and include these parameters. Do not forget to provide this config to your preprocessor class then.

8.1.3 3. Include the Preprocessor in the Execution Pipeline

Add your preprocessor to `pipeline.Pipeline.preprocessData()` method. Make sure that a) `preprocessing.DataTransformationPreprocessor` is always the first preprocessor (it transforms the file to use pandas' default separator so that we do not need to handle different separators anymore), b) all preprocessores write their outputs to the same intermediate directory, and c) set the correct inputs: your preprocessor's input is the output of the preceding preprocessor, your preprocessor's output will be the input of the subsequent preprocessor:

```
dataFormatter = preprocessing.DataTransformationPreprocessor(input, input_metadata,␣
↪intermediate_output, sep)
transposed_input = dataFormatter.preprocess()

#here comes your new preprocessor
examplePreprocessor = preprocessing.ExamplePreprocessor(transposed_input,␣
↪intermediate_output, whatever_you_need)
exampled_input = examplePreprocessor.preprocess()

mappingPreprocessor = preprocessing.MappingPreprocessor(exampled_input, intermediate_
↪output, currentIDFormat, desiredIDFormat, False)
mapped_input = mappingPreprocessor.preprocess()
```

8.2 Add a New Knowledge Base

Every knowledge base needs to classes: One class inheriting from `knowledgebases.KnowledgeBase` and implementing the interface methods and a second class inheriting from bioservice's REST class for web service access.

8.2.1 1 Implement KnowledgeBase Class

The class inheriting from `knowledgebase.KnowledgeBase` is accessed from any other class within Comprior that wants to retrieve prior knowledge. It must implement the methods `knowledgebases.KnowledgeBase.getRelevantGenes()`, `knowledgebases.KnowledgeBase.getGeneScores()`, and `knowledgebases.KnowledgeBase.getRelevantPathways()` (depending on the type of knowledge base, raise a NotImplementedError).

```
class ExampleKB(KnowledgeBase):
  def __init__(self):
    #pass the knowledge base name, its config, the web service accessing class, and␣
↪booleans indicating what type of knowledge is provided by that knowledge base
    geneInfo = True
    pathwayInfo = False
    super().__init__("ExampleKB", util.getConfig("ExampleKB"), ExampleKBWS(),␣
↪geneInfo, pathwayInfo)

  def getRelevantGenes(self, labels):
    """Get all genes that are somehow associated to the given labels, e.g. disease␣
↪names.

```

(continues on next page)

(continued from previous page)

```
            :param labels: list of identifiers, e.g. disease names, for which to find␣
↪associated genes.
            :type labels: list of str
            :return: list of associated genes.
            :rtype: list of str
            """

        #implement the gene retrieval strategy here...
        return genes


    def getGeneScores(self, labels):
        """Get all genes and their association scores that are related to the given labels,␣
↪e.g. disease names.

            :param labels: list of identifiers, e.g. disease names, for which to find␣
↪associated genes.
            :type labels: list of str
            :return: DataFrame of associated genes and their association scores, in␣
↪descending order.
            :rtype: :class:`pandas.DataFrame`
            """

        #implement the gene score retrieval strategy here...
        #if your knowledge base provides pathways only, you can implement an own strategy␣
↪or raise a NotImplementedError

        return geneScores

    def getRelevantPathways(self, labels):
        """As this knowledge base currently does not provide pathway information, this␣
↪feature is not implemented.

            :param labels: list of labels for which to find related pathways.
            :type labels: list of str
            :return: :class:`NotImplementedError` as this knowledge base is not intended␣
↪to be used for such analyses.
            :rtype: :class:`NotImplementedError`
            """

        #if the knowledge base does not provide pathway information, raise a␣
↪NotImplementedError

        raise NotImplementedError()
```

8.2.2 2 Implement a Pathway Parser (optional)

If the knowledge base provides pathway or network information, parse the pathway information into a pypath `pypath.Network` (for an example, see *knowledgebases.KEGGPathwayParser*). Inherit from *knowledgebases.PathwayParser* to do that and implement your own *knowledgebases.PathwayParser.parsePathway()* method:

```
class ExamplePathwayParser(PathwayParser):
    """Parses a pathway from a custom format to :class:`pypath.Network`.
        """
```

(continues on next page)

(continued from previous page)

```
  def parsePathway(self, pathway, pathwayID):
    """Parse pathway to the internally used format of :class:`pypath.Network`.

       :param pathway: pathway string to parse
       :type pathway: str
       :param pathwayID: name of the pathway
       :type pathwayID: str
       :return: pathway in the internally used format..
       :rtype: :class:`pypath.Network`
       """

    #implement your pathway parsing strategy here...

    return pathway
```

8.2.3 3 Implement Web Service Accessing Class

First of all, check if bioservices already provides a class for accessing the knowledge base web service: https://bioservices.readthedocs.io/en/master/references.html#services If your knowledge base is not available, implement your own class by inheriting from bioservice's REST or WSDL classes. Optimally, your REST class provides the API endpoints as methods, so if there is an endpoint *search*, implement a corresponding method named *search*. To send the actual request, construct a query string that specifies our endpoint (without the general API url) and provide that string to *self.http_get(your_string)*.

```
class ExampleKBWS(REST):
"""Queries the web service of ExampleKB for a given set of labels and retrieves␣
↪association scores for all genes related to the query labels.
   """
def __init__(self):
    super().__init__("ExampleKBWS", url=util.config["ExampleKB"]["webservice_url"])

def getVersion(self):
    """Get the current version of the API endpoint.

       :return: web service version infos.
       :rtype: json dict
       """
    ret = self.http_get("/version")
    return ret
```

8.2.4 4 Update the Config File

Create a new section in *config.ini* for your knowledge base. It must contain an output directory (where intermediate results are written to), optionally the web service API URL (if you had to implement your own REST class), and any other parameter you need for your knowledge base class to function.

```
[ExampleKB]
webservice_url = https://www.myexamplekbwebservice.org/api
outputDir = ${General:externalKbDir}ExampleKB/
```

8.2.5 5 Register the Knowledge Base at KnowledgeBaseFactory

Register your new knowledge base at `knowledgebases.KnowledgeBaseFactory.createKnowledgeBase()`. If your knowledge base is o not forget to provide the pathway parser to your knowledge base when creating

```
def createKnowledgeBase(self, knowledgebase):
"""Creates knowledge base based on a given name.

 :param knowledgebase: name of the knowledge base to be created.
 :type knowledgebase: str
 :return: knowledge base object.
 :rtype: :class:`KnowledgeBase` or inheriting classes
 """

 if knowledgebase == "ExampleKB":
    return ExampleKB()

  if knowledgebase == "ExamplePathwayKB":
    #create a pathway parser if your knowledge base requires that
    pathwayparser = ExamplePathwayParser()
    return ExamplePathwayKB(pathwayparser)
```

8.3 Add a new Feature Selector Approach

8.3.1 1a. Implement a Feature Selector

Feature selection approaches are implemented in separate classes for each approach. Inherit from one (or multiple) of the many abstract classes that are available for feature selectors, e.g. *`featureselection.RSelector`* when invoking R code or *`featureselection.PriorKnowledgeSelector`* when implementing a prior knowledge approach that uses a knowledge base. See all the class hierarchy HERE.

```
class ExampleSelector(PriorKnowledgeSelector):
  def __init__(self, knowledgebase, whatever_you_need):
    self.whatever_you_need = whatever_you_need
    super().__init__("YourSelectorName", knowledgebase)

  def selectFeatures(self):
  """Your feature selection strategy.

    :return: absolute path to the output ranking file.
    :rtype: str
    """

    #define the name of the output file name (must follow this schema!)
    outputFilename = self.output + self.getName() + ".csv"

    #implement your feature selection procedure here...

    return outputFilename
```

8.3.2 1b. Implement a Network Selector

If you want to implement a network approach that maps the original feature space of the data to (sub-)networks, e.g. pathways, inherit from *featureselection.NetworkSelector* and implement its *featureselection.NetworkSelector.selectPathways()* instead:

```
class ExampleNetworkSelector(NetworkSelector):

  def __init__(self, knowledgebase, featuremapper):
    super().__init__("YourNetworkSelector", knowledgebase, featuremapper)

  def selectPathways(self, pathways):
  """Your pathway selection strategy

    :param pathways: selector name
    :type pathways: str
    :returns: pathway ranking with pathway scores
    :rtype: :class:`pandas.DataFrame`
    """
    #implement your pathway/subnetwork selection strategy here...

    return pathwayRanking
```

Classes inheriting from *featureselection.NetworkSelector* additionally require a feature mapper that, once the (sub-) networks were selected as new features by your new network selector, creates new feature values for every selected (sub-) network. To do that, either use an existing feature mapper or implement a new one that inherits from *featureselection.FeatureMapper* and implements *featureselection.FeatureMapper.mapFeatures()*:

```
class ExampleFeatureMapper(FeatureMapper):

        def __init__(self, ):
            super().__init__()

        def mapFeatures(self, original_data, subnetworkNames, subnetworks):
        """Your feature mapping strategy

          :param original_data: the original data set of which to map the feature␣
↪space.
          :type original_data: :class:`pandas.DataFrame`
          :param pathways: dict of pathway names as keys and corresponding pathway␣
↪:class:`pypath.Network` objects as values
          :type pathways: dict
          :returns: the transformed data set with new feature values
          :rtype: :class:`pandas.DataFrame`
          """
          #implement your feature mapping strategy here...

          return mappedDataset
```

8.3.3 2. Update the Config File

List the new feature selection approach in the comments of the *config.ini* file and preferably, this Wiki ;). When providing a name to your feature selection approach, follow this naming schema (*YourSelectorName* MUST be the same as the name provided in the selector's init method):

- *YourSelectorName* for a traditional approach without any knowledge base

- *YourSelectorName_kbName* for a selector that uses a knowledge base
- *YourSelectorName_tradName_kbName* for a selector that uses a knowledge base and another selector.

8.3.4 3. Register Feature/Network Selection Approach to the FeatureSelectorFactory

Register your new feature selector class at *featureselection.FeatureSelectorFactory* in one of the following methods, depending on the type of selector you implemented (see the sources as the methods are encapsulated in a singleton construct):

- `featureselection.FeatureSelectorFactory.createTraditionalSelector()` if it is a traditional selector not using a knowledge base
- `featureselection.FeatureSelectorFactory.createCombinedSelector()` if it uses a traditional approach and a knowledge base in a combined manner
- `featureselection.FeatureSelectorFactory.createIntegrativeSelector()` if it is a selector that only uses a knowledge base

When registering, you need to specify the first part (=YourSelectorName) of the overall name as an if-statement

```
def createIntegrativeSelector(self, selectorName, kb):
"""Creates a feature selector using a knowledge base from the given selector and␣
↪knowledge base names.
  Register new implementations of a prior knowledge selector here that does not␣
↪requires a (traditional) selector.
  Stops processing if the selector could not be found.

  :param selectorName: selector name
  :type selectorName: str
  :param kb: knowledge base name
  :type kb: str
  :return: instance of a feature selector implementation.
  :rtype: :class:`FeatureSelector` or inheriting class
  """
  kbfactory = knowledgebases.KnowledgeBaseFactory()
  knowledgebase = kbfactory.createKnowledgeBase(kb)

  if selectorName == "YourSelectorName":
    return ExampleSelector(knowledgebase)

  if selectorName == "YourNetworkSelectorName":
      featureMapper = ExampleFeatureMapper()
      return ExampleNetworkSelector(knowledgebase, featureMapper)
```

8.4 Add Custom Code from R/Java/another Programming Languages

8.4.1 Invoking R or Java Code

The benchutils package provides methods for invoking R or Java code (*benchutils.runRCommand()* and *benchutils.runJavaCommand()*, respectively). These methods are already used, e.g. by *featureselection.RSelector.selectFeatures()* and *featureselection.JavaSelector.selectFeatures()*. If you have R or Java code that you want to invoke, use these methods and provide them with the R/Java config parameters, the name of the script/jar to execute, and a list of parameters. The example below runs an R script called "FS_LassoPenalty.R" that expects three parameters providing file names to the input, output, and external score files.

```
params = [input_filename, output_filename, externalscores_filename]
benchutils.runRCommand(self.rConfig, "FS_LassoPenalty.R" , params)
```

8.4.2 Invoking Code from Another Programming Language than R or Java

Currently, Comprior supports to run Python, R, and Java code. If you want to integrate custom code from another programming language, you can implement a corresponding function like `benchutils.runRCommand()` and `benchutils.runJavaCommand()`. Such a function constructs an execution string that is then forwarded to the command line. To do that,

- create a new folder for your programming language in Comprior's *code* directory (next to the *Python*, *R*, and *Java* directories), e.g. *Cpp* for adding C++ code
- place your executable files or script(s) into the new directory
- adapt the *config.ini* file and add a new segment for the programming language that contains the path to your executable source code (e.g. the *compiled* files of your C++ code) and to the programming language interpreter (for C++, however, code is invoked by just typing *./filename* on the command line)

```
[C++]
code = ${General:homePath}code/Cpp
CppLocation=./
```

- adapt *benchutils.py* and implement an additional function (do not forget to add code documentation!)

```
def runCppCommand(cppConfig, filename, params):
"""Run external C++ code.

  :param cppConfig: C++ config parameters (store paths to C++ and␣
↪the C++ code).
  :type cppConfig: dict
  :param filename: name of the C++ file to be executed.
  :type filename: str
  :param params: list of parameters that will be forwarded to the␣
↪C++ file.
  :type params: list of str
  """
  args = [cppConfig["C++"], filename]
  args.extend(params)
  print(args)
  p = subprocess.Popen(args, cwd=cppConfig["code"])
  p.wait()
```

- invoke your code from within Python as described above in *Invoking R or Java Code*.

CHAPTER 9

Python Code Documentation

9.1 pipeline module

The framework module is responsible for orchestrating the complete benchmarking process. It is the starting point that is invoked when running Comprior. It tidies up and prepares working directories, creates and coordinates the execution order of preprocessing modules, feature selectors, and evaluation procedures.

class pipeline.**Pipeline**(*userConfig*)

Bases: `object`

Class that executes the complete benchmarking pipeline.

Parameters **outputRootPath** (*str*) – absolute path to the overall output directory (will be extended by own folders by every `evaluation.Evaluator`).

prepareExecution(*userConfig*)

Prepares the pipeline execution by loading the configuration file, clearing intermediate directories, and creating output directories.

Parameters **userConfig** (*str*) – absolute path to an additional user configuration file (config.ini will always be used by default) to overwrite default configuration.

evaluateInputData(*inputfile*)

Run `evaluation.DatasetEvaluator` to create plots as specified by the config's Evaluation-preanalysis parameter.

Parameters **inputfile** (*str*) – absolute path to the input data set to be analyzed.

evaluateKnowledgeBases(*labeledInputDataPath*)

Evaluates knowledge base coverage for all knowledge bases that are used in the specified feature selection methods. Uses the class labels and alternativeSearchTerms from the config, queries the knowledge bases and creates corresponding plots regarding coverage of theses search terms.

Parameters **labeledInputDataPath** (*str*) – absolute path to the labeled input data set.

runFeatureSelector(*selector*, *datasetLocation*, *outputDir*, *loggingDir*)

Runs a given feature selector.

Parameters

- **selector** (*featureselection.FeatureSelector*) – Any feature selector that inherits from *featureselection.FeatureSelector*.
- **datasetLocation** (*str*) – absolute path to the input data set (from which features should be selected).
- **outputDir** (*str*) – absolute path to the selector's output directory (where ranking will be written to).

selectFeatures (*datasetLocation*)

Creates and runs all feature selectors that are listed in the config file. Applies parallelization by running as much feature selectors in parallel as stated in the config's General–>numCores attribute.

Parameters

- **datasetLocation** (*str*) – absolute path to the input data set (from which features should be selected).
- **outputRootPath** (*str*) – absolute path to the selector's output directory (where ranking will be written to).

Returns absolute path to directory that contains generated feature rankings.

Return type str

assignColors (*methods*)

Assigns each (feature selection) method a unique color. Will be delivered later on to every *evaluation.Evaluator* instance to create visualizations with consistent coloring for evaluated approaches.

Parameters **methods** (List of str) – List of method names.

Returns Dictionary containing hex color codes for every method

Return type dict

assignMarkers (*approaches*)

Assigns each (feature selection) method a unique color. Will be delivered later on to every *evaluation.Evaluator* instance to create visualizations with consistent coloring for evaluated approaches.

Parameters **methods** (List of str) – List of method names.

Returns Dictionary containing hex color codes for every method

Return type dict

evaluateBiomarkers (*inputDir*, *dataset*, *rankingsDir*)

Covers the evaluation phase. Processes input data to only contain the top k selected features per feature selection approach via the *evaluation.AttributeRemover*. Runs all selected evaluation strategies that cover assessment of rankings (*evaluation.RankingsEvaluator*), annotations(*evaluation.AnnotationEvaluator*), and classification performance (*evaluation.ClassificationEvaluator*). If selected, also conducts cross-validation across data sets with *evaluation.CrossEvaluator*.

Parameters

- **inputDir** (*str*) – absolute path to the directory where input data sets are located (for *evaluation.AttributeRemover*).
- **dataset** (*str*) – absolute file path to the input data set (from which features should be selected).

- **`rankingsDir`** (*`str`*) – absolute path to the directory that contains all rankings.

`preprocessData``()`

Preprocesses the input data set specified in the config file. Preprocessing consists of a) transposing the data so that features are in the columns (if necessary), b) mapping the features to the right format (if necessary), c) labeling the data with the user-specified metadata attribute, d) filtering features or samples that have too few information (optional, specified via config), and finally e) putting the analysis-ready data set to the right location for further processing.

Returns A tuple consisting of the absolute path to the analysis-ready data set and the absolute path to the mapped input final_filename and mapped_input

Return type tuple(str,str)

`loadConfig``(`*userConfig*`)`

Loads the config files. config.ini will always be loaded as default config file, all other config files provided by userConfig overwrite corresponding values.

Parameters **`userConfig`** (str or `List` of str, optional) – absolute path(s) to user-defined config files that should be used. If config files specify the same parameter, the value specified by the last config file in the list will be used.

`prepareDirectories``()`

Prepares directory structure for benchmarking run. Creates all necessary directories in the output folder. Also cleans up intermediate directory so that no old data is accidentially used.

Returns absolute path to the directory where all results from this run will be stored.

Return type str

`executePipeline``()`

The entry point for the overall benchmarking process. This method is invoked when running the framework, and from here all other steps of the benchmarking process are encapsulated in own methods.

Parameters **`userConfig`** (*`str`*) – absolute path to an additional user configuration file (config.ini will always be used by default) to overwrite default configuration.

9.2 benchutils module

Utility module that provides functionality that is repeatedly used across the system, e.g. directory handling and file loading, identifier mapping, logging, and running external code from R or Java. It also loads and stores the configuration parameters.

`benchutils.`**`loadConfig`**`(`*path*`)`

Loads the config files.

Parameters **`path`** (*`str or list of str`*) – absolute path or list of absolute paths to the config files. For multiple config files specifying the same parameters, the ones from the last config file in the list will be used.

`benchutils.`**`getConfig`**`(`*category*`)`

Get the config entries for a particular category.

Parameters **`category`** (*`str`*) – category name.

Returns all parameters for that config category

Return type dict

`benchutils.`**`getConfigValue`**`(`*category*, *identifier*`)`

Get the value for a given config parameter.

Parameters

- **`category`** (*`str`*) – the parameter's category name.
- **`identifier`** (*`str`*) – the parameter name.

Returns the parameter value.

Return type str

`benchutils.`**`getConfigBoolean`**`(`*category*, *identifier*`)`

Get the boolean value for a given config parameter.

Parameters

- **`category`** (*`str`*) – the parameter's category name.
- **`identifier`** (*`str`*) – the parameter name.

Returns the parameter boolean value.

Return type bool

`benchutils.`**`loadRanking`**`(`*rankingFile*`)`

Load a feature ranking from a file.

Parameters **`rankingFile`** (*`str`*) – absolute path to the file containing a feature ranking.

Returns the feature ranking as a DataFrame.

Return type `pandas.DataFrame`

`benchutils.`**`createOrClearDirectory`**`(`*directoryLocation*`)`

If the provided directory location is already existing, remove all files in that directory. Create a new directory otherwise.

Parameters **`directoryLocation`** (*`str`*) – absolute path to the directory that must be cleared or created.

`benchutils.`**`createDirectory`**`(`*directoryLocation*`)`

Creates a directory.

Parameters **`directoryLocation`** (*`str`*) – absolute path to the directory to be created.

`benchutils.`**`removeDirectoryContent`**`(`*directoryLocation*`)`

Remove the files inside a directory.

Parameters **`directoryLocation`** (*`str`*) – absolute path to the directory that must be cleared.

`benchutils.`**`removeFile`**`(`*file*`)`

Delete a file.

Parameters **`file`** (*`str`*) – absolute path to the file that must be deleted.

`benchutils.`**`cleanupResults`**`()`

Remove all intermediate files from former runs, e.g. generated during preprocessing or mapping.

`benchutils.`**`createLogger`**`(`*outputPath*`)`

Create a logger for Comprior. Creates two handlers for this logger: one for console output that only contains high-level status update logs and error messages. Warnings and other tracing information is written to an extra log file.

Parameters **`outputPath`** (*`String`*) – absoulte path to where the log file will be stored.

`benchutils.`**`logDebug`**`(`*message*`)`

Write a log at debug level.

Parameters **`message`** (*`String`*) – the log message to print.

`benchutils.`**`logInfo`**`(`*message*`)`

Write a log at info level.

Parameters **`message`** (*`String`*) – the log message to print.

`benchutils.`**`logWarning`**`(`*message*`)`

Write a log at warning level.

Parameters **`message`** (*`String`*) – the log message to print.

`benchutils.`**`logError`**`(`*message*`)`

Write a log at error level.

Parameters **`message`** (*`String`*) – the log message to print.

`benchutils.`**`createTimeLog`**`()`

Create the data structure for tracing runtimes of feature selection approaches.

Returns the logging data structure.

Return type `pandas.DataFrame`

`benchutils.`**`flushTimeLog`**`(`*timeLogs*, *outputFilePath*`)`

Write the whole log (of runtimes) to a file.

Parameters

- **`timeLogs`** (`pandas.DataFrame`) – the logs in a DataFrame.
- **`outputFilePath`** (*`str`*) – absolute path to the log file.

`benchutils.`**`logRuntime`**`(`*timeLogs*, *start*, *end*, *message*`)`

Write a runtime log entry and add it to the runtime log data structure.

Parameters

- **`timeLogs`** (`pandas.DataFrame`) – logs to which the new entry should be added
- **`start`** (*`str`*) – starting time.
- **`end`** (*`str`*) – ending time.
- **`message`** (*`str`*) – description of that entry.

Returns updated logs.

Return type `pandas.DataFrame`

`benchutils.`**`runRCommand`**`(`*rConfig*, *scriptName*, *params*`)`

Run external R code.

Parameters

- **`rConfig`** (*`dict`*) – R config parameters (store paths to Rscript and the R code).
- **`scriptName`** (*`str`*) – name of the R script to be executed.
- **`params`** (*`list of str`*) – list of parameters that will be forwarded to the R script.

`benchutils.`**`runJavaCommand`**`(`*javaConfig*, *scriptName*, *params*`)`

Run external Java code.

Parameters

- **javaConfig** (*dict*) – java config parameters (store paths to java and the java code).
- **scriptName** (*str*) – name of the jar to be executed.
- **params** (*list of str*) – list of parameters that will be forwarded to the jar.

benchutils.**mapIdentifiers**(*itemList*, *originalFormat*, *desiredFormat*)

Write a log entry and add it to the log data structure.

Parameters

- **itemList** (*list of str*) – list of identifiers, e.g. gene names, to be mapped
- **originalFormat** (*str*) – current format of the identifiers.
- **desiredFormat** (*str*) – desired format to which the identifiers should be mapped.

Returns mapping table where every item from itemList is now mapped to desiredFormat.

Return type pandas.DataFrame

benchutils.**mapGeneList**(*genes*, *originalFormat*, *desiredFormat*, *outputFile*)

Map a list of genes to the desired format.

Parameters

- **genes** (*list of str*) – list of gene names to be mapped
- **originalFormat** (*str*) – current format of the gene names.
- **desiredFormat** (*str*) – desired format to which the gene names should be mapped.
- **outputFile** (*str*) – absolute path to the output file in which the mapping should be stored.

Returns list of mapped gene names.

Return type list of str

benchutils.**mapRanking**(*ranking*, *originalFormat*, *desiredFormat*, *outputFile*)

Map the feature names of a ranking to the desired format.

Parameters

- **ranking** (pandas.DataFrame) – DataFrame of the ranking.
- **originalFormat** (*str*) – current format of the feature names in the ranking.
- **desiredFormat** (*str*) – desired format to which the feature names should be mapped.
- **outputFile** (*str*) – absolute path to the output file in which the mapped feature ranking should be stored.

Returns mapped feature ranking.

Return type pandas.DataFrame

benchutils.**retrieveMappings**(*itemList*, *originalFormat*, *desiredFormat*)

Query the knowledge base to map the identifiers. We have mapping via BiomaRt and gConvert available. gConvert is currently used because BiomaRt is unstable and blocks when parallel queries are sent.

Parameters

- **itemList** (*list of str*) – list of identifier names to be mapped
- **originalFormat** (*str*) – current format of the identifiers.
- **desiredFormat** (*str*) – desired format to which the identifiers should be mapped.

Returns mapping table for all identifiers.

Return type `pandas.DataFrame`

`benchutils.`**`mapDataMatrix`**(*inputMatrix*, *genesInColumns*, *originalFormat*, *desiredFormat*, *outputFile*, *labeled*)

Map the features of a data set to the desired format.

Parameters

- **`inputMatrix`** (`pandas.DataFrame`) – DataFrame of the ranking.
- **`genesInColumns`** (*`bool`*) – if the genes/features are located in the columns.
- **`originalFormat`** (*`str`*) – current format of the feature names in the data set.
- **`desiredFormat`** (*`str`*) – desired format to which the feature names should be mapped.
- **`outputFile`** (*`str`*) – absolute path to the output file in which the mapped data set should be stored.
- **`labeled`** (*`bool`*) – if the data matrix is additionally labeled.

Returns mapped data set.

Return type `pandas.DataFrame`

9.3 preprocessing module

Contains all classes related to preprocessing. All classes providing preprocessing functionality have to inherit from the abstract class *`preprocessing.Preprocessor`* and implement its *`preprocessing.Preprocessor.preprocess()`* method. For a detailed look at the class architecture, have a look at ADD CLASS ARCHITECTURE LINK HERE.

`class` `preprocessing.`**`Preprocessor`**(*input*, *metadata*, *output*)

Bases: `object`

Super class of all preprocessor implementations. Inherit from this class and implement *`preprocessing.Preprocessor.preprocess()`* if you want to add a new preprocessor class.

Parameters

- **`input`** (*`str`*) – absolute path to the input file.
- **`metadata`** (*`str`*) – absolute path to the metadata file.
- **`output`** (*`str`*) – absolute path to the output directory.

`preprocess`()

Abstract method. Interface method that is invoked externally to trigger preprocessing.

Returns absolute path to the preprocessed output file.

Return type str

`class` `preprocessing.`**`MappingPreprocessor`**(*input*, *output*, *currentFormat*, *desiredFormat*, *labeled*)

Bases: *`preprocessing.Preprocessor`*

Maps the input data set to a desired format.

Parameters

- **`input`** (*`str`*) – absolute path to the input file.

- **output** (*str*) – absolute path to the output directory.
- **currentFormat** (*str*) – current identifier format.
- **desiredFormat** (*str*) – desired identifier format.
- **labeled** (*bool*) – boolean value if the input data is labeled.

preprocess()

Maps the identifiers in the input dataset to the desired format that was specified when constructing the preprocessor.

Returns absolute path to the mapped file.

Return type str

class preprocessing.**FilterPreprocessor**(*input*, *metadata*, *output*)

Bases: `preprocessing.Preprocessor`

Filters features or samples above a user-defined threshold of missing values.

Parameters

- **input** (*str*) – absolute path to the input file.
- **metadata** (*str*) – absolute path to the metadata file.
- **output** (*str*) – absolute path to the output directory.
- **config** (*str*) – configuration parameter for preprocessing as specified in the config file.

preprocess()

Depending on what is specified in the config file, filter samples and/or features. Remove all samples/features that have missing values above the threshold specified in the config.

Returns absolute path to the filtered output file.

Return type str

filterMissings(*threshold*, *data*)

Filter the data for entries that have missing information above the given threshold.

Parameters

- **threshold** (*str*) – maximum percentage of allowed missing items as string.
- **data** (`pandas.DataFrame`) – a DataFrame to be filtered

Returns filtered DataFrame.

Return type `pandas.DataFrame`

class preprocessing.**DataTransformationPreprocessor**(*input*, *metadata*, *output*, *dataSeparator*)

Bases: `preprocessing.Preprocessor`

Transform the input data to have features in the columns for subsequent processing.

Parameters

- **input** (*str*) – absolute path to the input file.
- **metadata** (*str*) – absolute path to the metadata file.
- **output** (*str*) – absolute path to the output directory.
- **dataSeparator** (*str*) – delimiter to use when parsing the input file.

preprocess()

If not already so, transpose the input data to have the features in the columns.

Returns absolute path to the correctly formatted output file.

Return type str

class preprocessing.**MetaDataPreprocessor**(*input*, *metadata*, *output*, *separator*)

Bases: *preprocessing.Preprocessor*

Add labels to input data. Get labels from meta data attribute that was specified in the user config.

Parameters

- **input** (*str*) – absolute path to the input file.
- **metadata** (*str*) – absolute path to the metadata file.
- **output** (*str*) – absolute path to the output directory.
- **dataSeparator** (*str*) – delimiter to use when parsing the input and metadata file.
- **diseaseColumn** (*str*) – column name of the class labels.
- **transposeMetadataMatrix** (*bool*) – boolean value if the identifier names are located in the columns, as specified in the config file.

preprocess()

Labels all samples of a data set. Labels are taken from the corresponding metadata file and the metadata attribute that was specified in the config file. Samples without metadata information well be assigned to class "NotAvailable".

Returns absolute path to the labeled data set.

Return type str

class preprocessing.**DataMovePreprocessor**(*input*, *output*)

Bases: *preprocessing.Preprocessor*

Moves the input data set to the specified location.

Parameters

- **input** (*str*) – absolute path to the input file.
- **output** (*str*) – absolute path to the output directory.

preprocess()

Moves a file (self.input) to another location (self.output). Typically used at the end of preprocessing, when the final data set is moved to a new location for the actual analysis.

Returns absolute path to the new file location.

Return type str

9.4 featureselection module

Contains all classes related to feature selection. Each feature selection approach must be implemented in its own class inheriting from the abstract super class *featureselection.FeatureSelector* or one of its abstract subclasses, e.g. for including R or Java code. Each feature selection class must implement setParams() and selectFeatures(), as input or output parameters are just set at runtime.

Feature extraction methods are implemented in the same structure, except that they need to have an instance of a class inheriting from `featureselection.PathwayMapper` assigned to them so that the feature space can be transformed from the original to the new, e.g. pathways.

The creation of feature selectors is encapsulated by the class *`featureselection.FeatureSelectorFactory`* that takes care that every selector is equipped correspondingly, e.g. with a knowledge base or another feature selector. For a detailed look at the class architecture and the inheritance structure, have a look at ADD CLASS ARCHITECTURE LINK HERE.

class `featureselection.`**`FeatureSelectorFactory`**

Bases: `object`

Singleton class. Python code encapsulates it in a way that is not shown in Sphinx, so have a look at the descriptions in the source code.

Creates feature selector object based on a given name. New feature selection approaches must be registered here. Names for feature selectors must follow to a particular scheme, with keywords separated by _: - first keyword is the actual selector name - if needed, second keyword is the knowledge base - if needed, third keyword is the (traditional) approach to be combined Examples: - Traditional Approaches have only one keyword, e.g. InfoGain or ANOVA - LassoPenalty_KEGG provides KEGG information to the LassoPenalty feature selection approach - Weighted_KEGG_InfoGain –> Factory creates an instance of KBweightedSelector which uses KEGG as knowledge base and InfoGain as traditional selector. While the focus here lies on the combination of traditional approaches with prior biological knowledge, it is theoretically possible to use ANY selector object for combination that inherits from *`FeatureSelector`*.

Parameters **`config`** (*`dict`*) – configuration parameters for UMLS web service as specified in config file.

`instance = None`

class `featureselection.`**`FeatureSelector`**(*name*)

Bases: `object`

Abstract super class for feature selection functionality. Every feature selection class has to inherit from this class and implement its *`FeatureSelector.selectFeatures()`* method and - if necessary - its *`FeatureSelector.setParams()`* method. Once created, feature selection can be triggered by first setting parameters (input, output, etc) as needed with *`FeatureSelector.setParams()`*. The actual feature selection is triggered by invoking *`FeatureSelector.selectFeatures()`*.

Parameters

- **`input`** (*`str`*) – absolute path to input dataset.
- **`output`** (*`str`*) – absolute path to output directory (where the ranking will be stored).
- **`dataset`** (`pandas.DataFrame`) – the dataset for which to select features. Will be loaded dynamically based on self.input at first usage.
- **`dataConfig`** (*`dict`*) – config parameters for input data set.
- **`name`** (*`str`*) – selector name

`selectFeatures`()

Abstract. Invoke feature selection functionality in this method when implementing a new selector

Returns absolute path to the output ranking file.

Return type str

`getTimeLogs`()

Gets all logs for this selector.

Returns dataframe of logged events containing start/end time, duration, and a short description.

Return type `pandas.DataFrame`

`setTimeLogs`(*newTimeLogs*)
Overwrites the current logs with new ones.

Parameters **`newTimeLogs`** (`pandas.DataFrame`) – new dataframe of logged events containing start/end time, duration, and a short description.

`disableLogFlush`()
Disables log flushing (i.e., writing the log to a separate file) of the selector at the end of feature selection. This is needed when a *`CombiningSelector`* uses a second selector and wants to avoid that its log messages are written, potentially overwriting logs from another selector of the same name.

`enableLogFlush`()
Enables log flushing, i.e. writing the logs to a separate file at the end of feature selection.

`getName`()
Gets the selector's name.

Returns selector name.

Return type str

`getData`()
Gets the labeled dataset from which to select features.

Returns dataframe containing the dataset with class labels.

Return type `pandas.DataFrame`

`getUnlabeledData`()
Gets the dataset without labels.

Returns dataframe containing the dataset without class labels.

Return type `pandas.DataFrame`

`getFeatures`()
Gets features from the dataset.

Returns list of features.

Return type list of str

`getUniqueLabels`()
Gets the unique class labels available in the dataset.

Returns list of distinct class labels.

Return type list of str

`getLabels`()
Gets the labels in the data set.

Returns all labels from the dataset.

Return type list of str

`setParams`(*inputPath*, *outputDir*, *loggingDir*)
Sets parameters for the feature selection run: path to the input datast and path to the output directory.

Parameters

- **inputPath** (`str`) – absolute path to the input file containing the dataset for analysis.
- **outputDir** (`str`) – absolute path to the output directory (where to store the ranking)
- **loggingDir** (`str`) – absolute path to the logging directory (where to store log files)

writeRankingToFile(*ranking*, *outputFile*, *index=False*)
Writes a given ranking to a specified file.

Parameters

- **ranking** (`pandas.DataFrame`) – dataframe with the ranking.
- **outputFile** (`str`) – absolute path of the file where ranking will be stored.
- **index** (`bool, default False`) – whether to write the dataframe's index or not.

class `featureselection.`**PythonSelector**(*name*)
Bases: `featureselection.FeatureSelector`

Abstract. Inherit from this class when implementing a feature selector using any of scikit-learn's functionality. As functionality invocation, input preprocessing and output postprocessing are typically very similar/the same for such implementations, this class already encapsulates it. Instead of implementing `PythonSelector.selectFeatures()`, implement `PythonSelector.runSelector()`.

runSelector(*data*, *labels*)
Abstract - implement this method when inheriting from this class. Runs the actual feature selector of scikit-learn. Is invoked by `PythonSelector.selectFeatures()`.

Parameters

- **data** (`pandas.DataFrame`) – dataframe containing the unlabeled dataset.
- **labels** (`list of int`) – numerically encoded class labels.

Returns sklearn/mlxtend selector that ran the selection (containing coefficients etc.).

selectFeatures()
Executes the feature selection procedure. Prepares the input data set to match scikit-learn's expected formats and postprocesses the output to create a ranking.

Returns absolute path to the output ranking file.

Return type str

prepareInput()
Prepares the input data set before running any of scikit-learn's selectors. Removes the labels from the input data set and encodes the labels in numbers.

Returns dataset (without labels) and labels encoded in numbers.

Return type `pandas.DataFrame` and list of int

prepareOutput(*outputFile*, *data*, *selector*)
Transforms the selector output to a valid ranking and stores it into the specified file.

Parameters

- **outputFile** (`str`) – absolute path of the file to which to write the ranking.
- **data** (`pandas.DataFrame`) – input dataset.

- **`selector`** – selector object from scikit-learn.

class `featureselection.`**`RSelector`**(*name*)

Bases: *`featureselection.FeatureSelector`*

Selector class for invoking R code for feature selection. Inherit from this class if you want to use R code, implement *`RSelector.createParams()`* with what your script requires, and set self.scriptName accordingly.

Parameters **`rConfig`** (*`dict`*) – config parameters to execute R code.

`createParams`(*filename*)

Abstract. Implement this method to set the parameters your R script requires.

Parameters **`filename`** (*`str`*) – absolute path of the output file.

Returns list of parameters to use for R code execution, e.g. input and output filenames.

Return type list of str

`selectFeatures`()

Triggers the feature selection. Actually a wrapper method that invokes external R code.

Returns absolute path to the result file containing the ranking.

Return type str

class `featureselection.`**`JavaSelector`**(*name*)

Bases: *`featureselection.FeatureSelector`*

Selector class for invoking R code for feature selection. Inherit from this class if you want to use R code, implement *`RSelector.createParams()`* with what your script requires, and set self.scriptName accordingly.

Parameters **`javaConfig`** (*`dict`*) – config parameters to execute java code.

`createParams`()

Abstract. Implement this method to set the parameters your java code requires.

Returns list of parameters to use for java code execution, e.g. input and output filenames.

Return type list of str

`selectFeatures`()

Triggers the feature selection. Actually a wrapper method that invokes external java code.

Returns absolute path to the result file containing the ranking.

Return type str

class `featureselection.`**`PriorKnowledgeSelector`**(*name*, *knowledgebase*)

Bases: *`featureselection.FeatureSelector`*

Super class for all prior knowledge approaches. If you want to implement an own prior knowledge approach that uses a knowledge base (but not a second selector and no network approaches), inherit from this class.

Parameters

- **`knowledgebase`** (*`knowledgebases.KnowledgeBase`* or inheriting class) – instance of a knowledge base.
- **`alternativeSearchTerms`** (*`list of str`*) – list of alternative search terms to use for querying the knowledge base.

`selectFeatures`()

Abstract. Implement this method when inheriting from this class.

Returns absolute path to the output ranking file.

Return type str

collectAlternativeSearchTerms()
Gets all alternative search terms that were specified in the config file and put them into a list.

Returns list of alternative search terms to use for querying the knowledge base.

Return type list of str

getSearchTerms()
Gets all search terms to use for querying a knowledge base. Search terms that will be used are a) the class labels in the dataset, and b) the alternative search terms that were specified in the config file.

Returns list of search terms to use for querying the knowledge base.

Return type list of str

getName()
Returns the full name (including applied knowledge base) of this selector.

Returns selector name.

Return type str

class featureselection.**CombiningSelector**(*name*, *knowledgebase*, *tradApproach*)
Bases: *featureselection.PriorKnowledgeSelector*

Super class for prior knoweldge approaches that use a knowledge base AND combine it with any kind of selector, e.g. a traditional approach. Inherit from this class if you want to implement a feature selector that requires both a knowledge base and another selector, e.g. because it combines information from both.

Parameters

- **knowledgebase** (*knowledgebases.KnowledgeBase* or inheriting class) – instance of a knowledge base.
- **tradApproach** (*FeatureSelector*) – any feature selector implementation to use internally, e.g. a traditional approach like ANOVA

selectFeatures()
Abstract. Implement this method as desired when inheriting from this class.

Returns absolute path to the output ranking file.

Return type str

getName()
Returns the full name (including applied knowledge base and feature selector) of this selector.

Returns selector name.

Return type str

getExternalGenes()
Gets all genes related to the provided search terms from the knowledge base.

Returns list of gene names.

Return type list of str

class featureselection.**NetworkSelector**(*name*, *knowledgebase*, *featuremapper*)
Bases: *featureselection.PriorKnowledgeSelector*

Abstract. Inherit from this method if you want to implement a new network approach that actually conducts feature EXTRACTION, i.e. maps the original data set to have pathway/subnetworks. Instead of *`FeatureSelector.selectFeatures()`* implement *`NetworkSelector.selectPathways()`* when inheriting from this class.

Instances of *`NetworkSelector`* and inheriting classes also require a `PathwayMapper` object that transfers the dataset to the new feature space. Custom implementations thus need to implement a) a selection strategy to select pathways and b) a mapping strategy to compute new feature values for the selected pathways.

Parameters **`featureMapper`** (*`FeatureMapper`* or inheriting class) – feature mapping object that transfers the feature space.

`selectPathways`(*pathways*)

Selects the pathways that will become the new features of the data set. Implement this method (instead of *`FeatureSelector.selectFeatures()`* when inheriting from this class.

Parameters **`pathways`** (*`dict`*) – dict of pathways (pathway names as keys) to select from.

Returns pathway ranking as dataframe

Return type `pandas.DataFrame`

`writeMappedFile`(*mapped_data*, *fileprefix*)

Writes the mapped dataset with new feature values to the same directory as the original file is located (it will be automatically processed then).

Parameters

- **`mapped_data`** (`pandas.DataFrame`) – dataframe containing the dataset with mapped feature space.
- **`fileprefix`** (*`str`*) – prefix of the file name, e.g. the directory path

Returns absolute path of the file name to store the mapped data set.

Return type str

`getName`()

Gets the selector name (including the knowledge base).

Returns selector name.

Return type str

`filterPathways`(*pathways*)

`selectFeatures`()

Instead of selecting existing features, instances of *`NetworkSelector`* select pathways or submodules as features. For that, it first queries its knowledge base for pathways. It then selects the top k pathways (strategy to be implemented in *`NetworkSelector.selectPathways()`*) and subsequently maps the dataset to its new feature space. The mapping will be conducted by an object of `PathwayMapper` or inheriting classes. If a second dataset for cross-validation is available, the feature space of this dataset will also be transformed.

Returns absolute path to the pathway ranking.

Return type str

class `featureselection.`**`RandomSelector`**

Bases: *`featureselection.FeatureSelector`*

Baseline Selector: Randomly selects any features.

`selectFeatures`()

Randomly select any features from the feature space. Assigns a score of 0.0 to every feature

Returns absolute path to the ranking file.

Return type str

class featureselection.**AnovaSelector**

Bases: *featureselection.PythonSelector*

Runs ANOVA feature selection using scikit-learn implementation

runSelector(*data*, *labels*)

Runs the ANOVA feature selector of scikit-learn. Is invoked by *PythonSelector.selectFeatures()*.

Parameters

- **data** (pandas.DataFrame) – dataframe containing the unlabeled dataset.
- **labels** (*list of int*) – numerically encoded class labels.

Returns sklearn/mlxtend selector that ran the selection (containing coefficients etc.).

class featureselection.**Variance2Selector**

Bases: *featureselection.PythonSelector*

Runs variance-based feature selection using scikit-learn.

prepareOutput(*outputFile*, *data*, *selector*)

Transforms the selector output to a valid ranking and stores it into the specified file. We need to override this method because variance selector has no attribute scores but variances.

Parameters

- **outputFile** (*str*) – absolute path of the file to which to write the ranking.
- **data** (pandas.DataFrame) – input dataset.
- **selector** – selector object from scikit-learn.

runSelector(*data*, *labels*)

Runs the actual variance-based feature selector of scikit-learn. Is invoked by *PythonSelector.selectFeatures()*.

Parameters

- **data** (pandas.DataFrame) – dataframe containing the unlabeled dataset.
- **labels** (*list of int*) – numerically encoded class labels.

Returns sklearn/mlxtend selector that ran the selection (containing coefficients etc.).

class featureselection.**MRMRSelector**

Bases: *featureselection.RSelector*

Runs maximum Relevance minimum Redundancy (mRMR) feature selection using the mRMRe R implementation: https://cran.r-project.org/web/packages/mRMRe/index.html Actually a wrapper class for invoking the R code.

Parameters

- **scriptName** (*str*) – name of the R script to invoke.
- **maxFeatures** (*int*) – maximum number of features to select. Currently all features (=0) are ranked..

createParams(*outputFile*)

Sets the parameters the R script requires (input file, output file, maximum number of features).

Returns list of parameters to use for mRMR execution in R.

Return type list of str

class featureselection.**VarianceSelector**

Bases: *featureselection.RSelector*

Runs variance-based feature selection using R genefilter library. Actually a wrapper class for invoking the R code.

Parameters **scriptName** (*str*) – name of the R script to invoke.

createParams (*outputFile*)

Sets the parameters the R script requires (input file, output file).

Parameters **outputFile** (*str*) – absolute path to the output file that will contain the ranking.

Returns list of parameters to use for mRMR execution in R.

Return type list of str

class featureselection.**InfoGainSelector**

Bases: *featureselection.JavaSelector*

Runs InfoGain feature selection as provided by WEKA: https://www.cs.waikato.ac.nz/ml/weka/ Actually a wrapper class for invoking java code.

createParams ()

Sets the parameters the java program requires (input file, output file, selector name).

Returns list of parameters to use for InfoGain execution in java.

Return type list of str

class featureselection.**ReliefFSelector**

Bases: *featureselection.JavaSelector*

Runs ReliefF feature selection as provided by WEKA: https://www.cs.waikato.ac.nz/ml/weka/ Actually a wrapper class for invoking java code.

createParams ()

Sets the parameters the java program requires (input file, output file, selector name).

Returns list of parameters to use for InfoGain execution in java.

Return type list of str

class featureselection.**KbSelector** (*knowledgebase*)

Bases: *featureselection.PriorKnowledgeSelector*

Knowledge base selector. Selects features exclusively based the information retrieved from a knowledge base.

Parameters **knowledgebase** (*knowledgebases.KnowledgeBase*) – instance of a knowledge base.

updateScores (*entry*, *newGeneScores*)

Updates a score entry with the new score retrieved from the knowledge base. Used by apply function.

Parameters

- **entry** (pandas.Series) – a gene score entry consisting of the gene name and its score
- **newGeneScores** (pandas.DataFrame) – dataframe containing gene scores retrieved from the knowledge base.

Returns updated series element.

Return type `pandas.Series`

selectFeatures()

Does the actual feature selection. Retrieves association scores for genes from the knowledge base based on the given search terms.

Returns absolute path to the resulting ranking file.

Return type str

class featureselection.**KBweightedSelector**(*knowledgebase*, *tradApproach*)

Bases: *featureselection.CombiningSelector*

Selects features based on association scores retrieved from the knowledge base and the relevance score retrieved by the (traditional) approach. Computes the final score via tradScore * assocScore.

Parameters

- **knowledgebase** (*knowledgebases.KnowledgeBase* or inheriting class) – instance of a knowledge base.
- **tradApproach** (*FeatureSelector*) – any feature selector implementation to use internally, e.g. a traditional approach like ANOVA

updateScores(*entry*, *newGeneScores*)

Updates a score entry with the new score retrieved from the knowledge base. Used by apply function.

Parameters

- **entry** (`pandas.Series`) – a gene score entry consisting of the gene name and its score
- **newGeneScores** (`pandas.DataFrame`) – dataframe containing gene scores retrieved from the knowledge base.

Returns updated series element.

Return type `pandas.Series`

getName()

Gets the selector name (including the knowledge base and (traditional) selector).

Returns selector name.

Return type str

computeStatisticalRankings(*intermediateDir*)

Computes the statistical relevance score of all features using the (traditional) selector.

Parameters **intermediateDir** (*str*) – absolute path to output directory for (traditional) selector (where to write the statistical rankings).

Returns dataframe with statistical ranking.

Return type `pandas.DataFrame`

computeExternalRankings()

Computes the association scores for every gene using the knowledge base. Genes for which no entry could be found receive a default score of 0.000001.

Returns dataframe with statistical ranking.

Return type `pandas.DataFrame`

`combineRankings`(*externalRankings*, *statisticalRankings*)

Combines score rankings from both the knowledge base and the (traditional) selector (kb_score * trad_score) to retrieve a final score for every gene.

Parameters

- **`externalRankings`** (`pandas.DataFrame`) – dataframe with ranking from knowledge base.
- **`statisticalRankings`** (`pandas.DataFrame`) – dataframe with statistical ranking.

Returns dataframe with final combined ranking.

Return type `pandas.DataFrame`

`selectFeatures`()

Runs the feature selection process. Retrieves scores from knowledge base and (traditional) selector and combines these to a single score.

Returns absolute path to final output file containing the ranking.

Return type str

`class` `featureselection.`**`LassoPenalty`**(*knowledgebase*)

Bases: *`featureselection.PriorKnowledgeSelector`*, *`featureselection.RSelector`*

Runs feature selection by invoking xtune R package: https://cran.r-project.org/web/packages/xtune/index.html

xtune is a Lasso selector that uses feature-individual penalty scores. These penalty scores are retrieved from the knowledge base.

`selectFeatures`()

Triggers the feature selection. Actually a wrapper method that invokes external R code.

Returns absolute path to the result file containing the ranking.

Return type str

`getName`()

Returns the full name (including applied knowledge base) of this selector.

Returns selector name.

Return type str

`createParams`(*outputFile*)

Sets the parameters the xtune R script requires (input file, output file, filename containing rankings from knowledge base).

Returns list of parameters to use for xtune execution in R.

Return type list of str

`computeExternalRankings`()

Computes the association scores for each feature based on the scores retrieved from the knowledge base. Features that could not be found in the knowledge base receive a default score of 0.000001.

Returns absolute path to the file containing the external rankings.

Return type str

`class` `featureselection.`**`WrapperSelector`**(*name*)

Bases: *`featureselection.PythonSelector`*

Selector implementation for wrapper selectors using scikit-learn. Currently implements recursive feature eliminatin (RFE) and sequential forward selection (SFS) strategies, which can be combined with nearly any classifier offered by scikit-learn, e.g. SVM.

Parameters

- **`selector`** – scikit-learn selector strategy (currently RFE and SFS)
- **`classifier`** – scikit-learn classifier to use for wrapper selection.

`createClassifier``()`

Creates a classifier instance (from scikit-learn) to be used during the selection process. To enable the framework to use a new classifier, extend this method accordingly.

Returns scikit-learn classifier instance.

`createSelector``()`

Creates a selector instance that leads the selection process. Currently, sequential forward selection (SFS) and recursive feature elimination (RFE) are implemented. Extend this method if you want to add another selection strategy.

Returns scikit-learn selector instance.

`prepareOutput`(*outputFile*, *data*, *selector*)

Overwrites the inherited prepareOutput method because we need to access the particular selector's coefficients. The coefficients are extracted as feature scores and will be written to the rankings file.

Parameters

- **`outputFile`** (*`str`*) – selector name
- **`data`** (`pandas.DataFrame`) – input dataset to get the feature names.
- **`selector`** – selector instance that is used during feature selection.

`runSelector`(*data*, *labels*)

Runs the actual feature selector of scikit-learn. Is invoked by *`PythonSelector.selectFeatures()`*.

Parameters

- **`data`** (`pandas.DataFrame`) – dataframe containing the unlabeled dataset.
- **`labels`** (*`list of int`*) – numerically encoded class labels.

Returns sklearn/mlxtend selector that ran the selection (containing coefficients etc.).

`class` `featureselection.`**`SVMRFESelector`**

Bases: *`featureselection.JavaSelector`*

Executes SVM-RFE with poly-kernel. Uses an efficient java implementation from WEKA and is thus just a wrapper class to invoke the corresponding jars.

`createParams``()`

Sets the parameters the java program requires (input file, output file, selector name).

Returns list of parameters to use for InfoGain execution in java.

Return type list of str

`class` `featureselection.`**`RandomForestSelector`**

Bases: *`featureselection.PythonSelector`*

Selector class that implements RandomForest as provided by scikit-learn.

prepareOutput (*outputFile*, *data*, *selector*)
Overwrites the inherited prepareOutput method because we need to access the RandomForest selector's feature importances. These feature importances are extracted as feature scores and will be written to the rankings file.

Parameters

- **outputFile** (`str`) – selector name
- **data** (`pandas.DataFrame`) – input dataset to get the feature names.
- **selector** – RandomForest selector instance that is used during feature selection.

runSelector (*data*, *labels*)
Runs the actual feature selection using scikit-learn's RandomForest classifier. Is invoked by *`PythonSelector.selectFeatures()`*.

Parameters

- **data** (`pandas.DataFrame`) – dataframe containing the unlabeled dataset.
- **labels** (*`list of int`*) – numerically encoded class labels.

Returns scikit-learn RandomForestClassifier that ran the selection.

class `featureselection.`**LassoSelector**
Bases: *`featureselection.PythonSelector`*

Selector class that implements Lasso feature selection using scikit-learn.

prepareOutput (*outputFile*, *data*, *selector*)
Overwrites the inherited prepareOutput method because we need to access Lasso's coefficients. These coefficients are extracted as feature scores and will be written to the rankings file.

Parameters

- **outputFile** (`str`) – selector name
- **data** (`pandas.DataFrame`) – input dataset to get the feature names.
- **selector** – RandomForest selector instance that is used during feature selection.

runSelector (*data*, *labels*)
Runs the actual Lasso feature selector using scikit-learn. Is invoked by *`PythonSelector.selectFeatures()`*.

Parameters

- **data** (`pandas.DataFrame`) – dataframe containing the unlabeled dataset.
- **labels** (*`list of int`*) – numerically encoded class labels.

Returns Lasso selector that ran the selection.

class `featureselection.`**PreFilterSelector** (*knowledgebase*, *tradApproach*)
Bases: *`featureselection.CombiningSelector`*

Applies a two-level prefiltering strategy for feature selection. Filters all features that were not retrieved by a knowledge base based on the search terms provided in the config file. Applies a (traditional) feature selector on the remaining features afterwards.

For traditional univariate filter approaches, the results retrieved by this class and *`PostFilterSelector`* will be the same.

selectFeatures ()
Carries out feature selection. First queries the assigned knowledge base to get genes that are associated to

the given search terms. Filter feature set of input data set to contain only features that are in the retrieved gene set. Apply (traditional) selector on the filtered data set.

Returns absolute path to rankings file.

Return type str

`class featureselection.`**`PostFilterSelector`**`(`*knowledgebase*, *tradApproach*`)`

Bases: *`featureselection.CombiningSelector`*

Applies a two-level postfiltering strategy for feature selection. Applies (traditional) feature selection to the input data set. Afterwards, removes all genes for which no information in the corresponding knowledge base was found based on the search terms provided in the config file. For traditional univariate filter approaches, the results retrieved by this class and *`PreFilterSelector`* will be the same.

`selectFeatures``()`

Carries out feature selection. First executes (traditional) selector. Then queries the assigned knowledge base to get genes that are associated to the given search terms. Finally filters feature set to contain only features that are in the retrieved gene set.

Returns absolute path to rankings file.

Return type str

`class featureselection.`**`ExtensionSelector`**`(`*knowledgebase*, *tradApproach*`)`

Bases: *`featureselection.CombiningSelector`*

Selector implementation inspired by SOFOCLES: "SoFoCles: Feature filtering for microarray classification based on Gene Ontology", Papachristoudis et al., Journal of Biomedical Informatics, 2010

This selector carries out (traditional) feature selection and in parallel retrieves relevant genes from a knowledge base based on the provided search terms. The ranking is then adapted by alternating the feature ranking retrieved by the (traditiona) selection approach and the externally retrieved genes. This is kind of related to an extension approach, where a feature ranking that was retrieved by a traditional approach is extended by such external genes.

`selectFeatures``()`

Carries out feature selection. Executes (traditional) selector and separately retrieves genes from the assigned knowledge base based on the search terms specified in the config. Finally merges the two feature lists alternating to form an "extended" feature ranking.

Returns absolute path to rankings file.

Return type str

`class featureselection.`**`NetworkActivitySelector`**`(`*knowledgebase*, *featuremapper*`)`

Bases: *`featureselection.NetworkSelector`*

Selector implementation that selects a set of pathways from the knowledge base and maps the feature space to the pathways. Pathway ranking scores are computed based on the average ANOVA p-value of its member genes and the sample classes. This method is also used by Chuang et al. and Tian et al. (Discovering statistically significant pathways in expression profiling studies) Pathway feature values are computed with an instance of *`FeatureMapper`* or inheriting classes, whose mapping strategies can vary. If pathways should be selected according to another strategy, use this class as an example implementation to implement a new class that inherits from *`NetworkSelector`*.

`selectPathways``(`*pathways*`)`

Computes a pathway ranking for the input pathways. Computes a pathway score based on the average ANOVA's f-test p-values of a pathway's member genes and the sample classes.

Parameters **`pathways`** (*`str`*) – selector name

Returns pathway ranking with pathway scores

Return type `pandas.DataFrame`

class `featureselection.`**`FeatureMapper`**

Bases: `object`

Abstract. Inherit from this class and implement *`FeatureMapper.mapFeatures()`* to implement a new mapping strategy. Maps the feature space of the given input data to a given set of pathways. Computes a new feature value for every feature and sample based on the implemented strategy.

`mapFeatures`(*original_data*, *pathways*)

Abstract method. Implement this method when inheriting from this class. Carries out the actual feature mapping.

Parameters

- **`original_data`** (`pandas.DataFrame`) – the original data set of which to map the feature space.
- **`pathways`** (*`dict`*) – dict of pathway names as keys and corresponding pathway `pypath.Network` objects as values

Returns the transformed data set with new feature values

Return type `pandas.DataFrame`

`getUnlabeledData`(*dataset*)

Removes the labels from the data set.

Parameters **`dataset`** (`pandas.DataFrame`) – data set from which to remove the labels.

Returns data set without labels.

Return type `pandas.DataFrame`

`getLabels`(*dataset*)

Gets the dataset labels.

Parameters **`dataset`** (`pandas.DataFrame`) – data set from which to extract the labels.

Returns label vector of the data set.

Return type `pandas.Series`

`getFeatures`(*dataset*)

Gets the features of a data set.

Parameters **`dataset`** (`pandas.DataFrame`) – data set from which to extract the features.

Returns feature vector of the data set.

Return type `pandas.Series`

`getSamples`(*dataset*)

Gets all samples in a data set.

Parameters **`dataset`** (`pandas.DataFrame`) – data set from which to extract the samples.

Returns list of samples from the data set.

Return type list

`getPathwayGenes`(*pathway*, *genes*)

Returns the intersection of a given set of genes and the genes contained in a given pathway.

Parameters

- **`pathway`** (`pypath.Network`) – pathway object from which to get the genes.
- **`genes`** (*`list of str`*) – list of gene names.

Returns list of genes that are contained in both the pathway and the gene list.

Return type list of str

`class` `featureselection.`**`CORGSActivityMapper`**

Bases: *`featureselection.FeatureMapper`*

Pathway mapper that implements the strategy described by Lee et al.: "Inferring Pathway Activity toward Precise Disease Classification" Identifies CORGS genes for every pathway: uses random search to find the minimal set of genes for which the pathway activity score is maximal. First, every sample receives an activity score, which is the average expression level of the (CORGS) genes / number of genes. The computed activity scores are then used for f-testing with the class labels, and the p-values are the new pathway feature values. These steps are executed again and again until the p-values are not decreasing anymore.

`getANOVAscores`(*data*, *labels*)

Applies ANOVA f-test to test the association/correlation of a feature (pathway) with a given label. The feature has activity scores (computed from CORGS genes) for every sample, which are to be tested for the labels.

Parameters

- **`data`** (`pandas.DataFrame`) – the data set which to test for correlation with the labels (typically feature scores of a pathway for samples).
- **`labels`** (`pandas.Series`) – class labels to use for f-test.

Returns series of p-values for every sample.

Return type `pandas.Series`

`computeActivityScore`(*sampleExpressionLevels*)

Computes the activity score of a given set of genes for a specific sample. The activity score of a sample is the mean expression value of the given genes divided by the overall number of given genes.

Parameters **`sampleExpressionLevels`** (`pandas.DataFrame`) – data set containing expression levels from a given set of genes for samples.

Returns activity scores for the given samples.

Return type `pandas.Series`

`computeActivityVector`(*expressionLevels*)

Computes the activity score of a given set of genes for a all samples.

Parameters **`expressionLevels`** (`pandas.DataFrame`) – input data set of expression levels for a given set of (CORGS) genes.

Returns instance of a feature selector implementation.

Return type `pandas.DataFrame` or inheriting class

`mapFeatures`(*original_data*, *pathways*)

Carries out the actual feature mapping. Follows the strategy described by Lee et al.: "Inferring Pathway Activity toward Precise Disease Classification" Identifies CORGS genes for every pathway: uses random search to find the minimal set of genes for which the pathway activity score is maximal. First, every sample receives an activity score, which is the average expression level of the (CORGS) genes / number of genes. The computed activity scores are then used for f-testing with the class labels, and the p-values are the new pathway feature values. These steps are executed again and again until the p-values are not decreasing anymore.

Parameters

- **original_data** (`pandas.DataFrame`) – the original data set of which to map the feature space.
- **pathways** (*dict*) – dict of pathway names as keys and corresponding pathway `pypath.Network` objects as values

Returns the transformed data set with new feature values

Return type `pandas.DataFrame`

class featureselection.**PathwayActivityMapper**

Bases: *featureselection.FeatureMapper*

Pathway mapper that implements a strategy that is related to Vert and Kanehisa's strategy: Vert, Jean-Philippe, and Minoru Kanehisa. "Graph-driven feature extraction from microarray data using diffusion kernels and kernel CCA." NIPS. 2002. Computes pathway activity scores for every sample and pathway as new feature values. The feature value is the average of: expression level weighted by gene variance and neighbor correlation score)

getAverageCorrelation(*correlations*, *gene*, *neighbors*)

Computes the average correlation from the correlations of a given gene and its neighbors.

Parameters

- **correlations** (`pandas.DataFrame`) – correlation matrix of all genes.
- **gene** (*str*) – gene name whose average neighbor correlation to compute.
- **neighbors** (*list of str*) – list of gene names that are neighbors of the given gene.

Returns average correlation value.

Return type float

computeGeneVariances(*data*)

Computes the variances for every gene across all samples.

Parameters **data** (`pandas.DataFrame`) – data set with expression values.

Returns variance for every gene.

Return type `pandas.Series`

mapFeatures(*original_data*, *pathways*)

Executes the actual feature mapping procedure. A feature value is the average of (for every gene in a pathway): (expression level weighted by gene variance and neighbor correlation score)

Parameters

- **original_data** (`pandas.DataFrame`) – the original data set of which to map the feature space.
- **pathways** (*dict*) – dict of pathway names as keys and corresponding pathway `pypath.Network` objects as values

Returns the transformed data set with new feature values

Return type `pandas.DataFrame`

9.5 knowledgebases module

Contains all classes related to knowledge bases. A knowledge base is realized with two classes: * A class inheriting from *`knowledgebases.KnowledgeBase`* and implementing the three interface methods `knowledgebases.KnowledgeBase:getRelevantGenes()`, `knowledgebases.KnowledgeBase:getGeneScores()`, and `knowledgebases.KnowledgeBase:getRelevantPathways()`. * A class that is responsible for querying the corresponding web service and inherits from Bioservice's REST class. Those knowledge bases that retrieve pathway information also need an additional PathwayMapper class, which transforms the original pathway results from the knowledge base (which can range from SIF to any other pathway specification format) into the pathway representation that is used throughout Comprior. For Comprior's internal pathway representation, we use pypath.

The creation of knowledge bases is encapsulated by the class `knowledgebase.KnowledgeBaseFactory` that takes care that every knowledge base is equipped with a web service querying class and, if needed, the right type of `knowledgebase.PathwayMapper`. For a detailed look at the class architecture, have a look at ADD CLASS ARCHITECTURE LINK HERE.

`knowledgebases.`**`suppress_stdout`**(*suppress=True*)

`class` `knowledgebases.`**`ENRICHR`**

Bases: `bioservices.services.REST`

Queries some of the API endpoints of the EnrichR web service (https://maayanlab.cloud/Enrichr/help#api).

Parameters **`config`** (*`dict`*) – configuration parameters for EnrichR web service (as specified in config file).

`addlist`(*geneList*)

Queries EnrichR to annotate a given list of genes. Returns a userListID, which can be used to retrieve the actual results in a second query.

Parameters **`geneList`** – list of genes to annotate

Returns json response containing a userListID.

Return type dict of str

`export`(*params*)

Download file of enrichment results. Requires a userListId that was retrieved from a prior query.

Parameters **`params`** (*`list of str`*) – list of parameters to use for that query (userListId: Identifier returned from addList endpoint, filename: Name of text file download, backgroundType: Gene set library for which to download results)

Returns text file containing enrichment results.

Return type str

`genemap`(*params*)

Finds all terms, their descriptions, and optional categorizations, for a given gene identifier.

Parameters **`params`** (*`list of str`*) – list of parameters to be used for the query (gene Gene to use in search for terms, json (optional): Set "true" to return JSON rather plaintext, setup (optional): Set "true" to category information for the libraries)

Returns json object of all terms containing the specified gene and their descriptions.

Return type dict of str

`enrich`(*params*)

Returns all that are terms available in library (specified by backgroundType param) and enriched in the given set of genes (specified by userListId param).

Parameters **`params`** (*`list of str`*) – list of parameters to be used for the query (userListId: Identifier returned from addList endpoint; backgroundType: Gene set library to enrich against)

Returns dataframe object of all enriched terms (unsorted, unfiltered.

Return type dataframe

`class` `knowledgebases.`**`UMLS_AUTH`**

Bases: `bioservices.services.REST`

Singleton class. Python code encapsulates it in a way that is not shown in Sphinx, so have a look at the descriptions in the source code.

Authentication service to get access to the UMLS database UMLS database (which we need for retrieving CUI disease codes for querying DisGeNET). You first have to get a ticket-granting ticket (tgt, valid for 8 hours) with the help of an API key. With the tgt, you can then request a service ticket for every new query to the UMLS database. The service ticket must then be used for the query. The task of this class is to generate a valid tgt and subsequent service ticket. Documentation on the authentication process: https://documentation.uts.nlm.nih.gov/rest/authentication.html

Parameters

- **`config`** (*`dict`*) – configuration parameters for UMLS web service as specified in config file.
- **`tgt_timestamp`** (*`str`*) – timestamp of the tgt. If it is older than 8 hours, we need to request a new tgt.
- **`tgt`** (*`list of str`*) – id of the ticket-granting ticket (valid for 8 hours). With this ticket, we can then query the actual UMLS web service.
- **`service`** (*`str`*) – uri for the service login

`instance = None`

`class` `knowledgebases.`**`UMLS`**

Bases: `bioservices.services.REST`

Retrieves UMLS CUI codes for labels, which can then be used for querying DisGeNET.

Parameters

- **`config`** (*`dict`*) – configuration parameters for UMLS web service (as specified in configuration file).
- **`auth`** (*`UMLS_AUTH`*) – authentication component to generate a valid service ticket (required for every query).

`getCUIs`(*labels*)

Get CUIs for the given labels.

Parameters **`labels`** (*`list of str`*) – list of identifiers for which to retrieve CUIs, e.g. disease names.

Returns list of CUIs.

Return type list of str

`class` `knowledgebases.`**`DISGENET`**

Bases: `bioservices.services.REST`

Queries the DisGeNET web service for a given set of labels and retrieves association scores for all genes related to the query labels. DisGeNET API documentation: https://www.disgenet.org/api/

Parameters **`umls`** (*UMLS* for transforming disease names to CUIs (required for query)) – list of gene names to be mapped

`getVersion`()

Get the current version of the DisGeNET API endpoint.

Returns web service version infos.

Return type json dict

`query`(*labels*)

Conducts the actual query to retrive gene-disease association scores for a given list of disease labels. Transforms the disease labels into CUIs before with the UMLS web service.

Parameters **`labels`** (*`list of str`*) – list of disease labels for which to retrieve gene-disease associations.

Returns DataFrame with gene-disease association scores.

Return type `pandas.DataFrame`

class `knowledgebases.`**`GCONVERT`**

Bases: `object`

Queries the g:Convert web service to map a list of identifiers to a desired format. g:Convert makes use of the Ensembl build. g:Convert API documentation: https://biit.cs.ut.ee/gprofiler/page/apis

Parameters **`url`** (*`str`*) – API url as specified in the configuration file.

`query`(*items*, *originalFormat*, *desiredFormat*)

Map a list of itendifiers to the desired format.

Parameters

- **`items`** (*`list of str`*) – list of identifiers to be mapped
- **`originalFormat`** (*`str`*) – current format of the identifiers
- **`desiredFormat`** (*`str`*) – desired identifier format

Returns DataFrame containing the identifier mapping.

Return type `pandas.DataFrame`

class `knowledgebases.`**`PATHWAYCOMMONSWS`**

Bases: `bioservices.services.REST`

Queries the PathwayCommons web service. Bioservices' existing implementation to query PathwayCommons was not used because it contained outdated values for _valid_formats for pathway retrieval, so we used the original code and adapted it to work correctly.

`getVersion`()

Map a list of genes to the desired format.

Parameters **`genes`** (*`list of str`*) – list of gene names to be mapped

Returns list of mapped gene names.

Return type list of str

`default_extension`

set extension of the requests (default is json). Can be 'json' or 'xml'

`search`(*q*, *page=0*, *datasource=None*, *organism=None*, *type=None*)

Text search in PathwayCommons using Lucene query syntax

Some of the parameters are BioPAX properties, others are composite relationships.

All index fields are (case-sensitive): comment, ecnumber, keyword, name, pathway, term, xrefdb, xrefid, dataSource, and organism.

The pathway field maps to all participants of pathways that contain the keyword(s) in any of its text fields.

Finally, keyword is a transitive aggregate field that includes all searchable keywords of that element and its child elements.

All searches can also be filtered by data source and organism.

It is also possible to restrict the domain class using the 'type' parameter.

This query can be used standalone or to retrieve starting points for graph searches.

Parameters

- **q** (*str*) – requires a keyword , name, external identifier, or a Lucene query string.
- **page** (*int*) – (N>=0, default is 0), search result page number.
- **datasource** (*str*) – filter by data source (use names or URIs of pathway data sources or of any existing Provenance object). If multiple data source values are specified, a union of hits from specified sources is returned. datasource=[reactome,pid] returns hits associated with Reactome or PID.
- **organism** (*str*) – The organism can be specified either by official name, e.g. "homo sapiens" or by NCBI taxonomy id, e.g. "9606". Similar to data sources, if multiple organisms are declared a union of all hits from specified organisms is returned. For example organism=[9606, 10016] returns results for both human and mice.
- **type** (*str*) – BioPAX class filter

get (*uri*, *frmt='BIOPAX'*)
Retrieves full pathway information for a set of elements

elements can be for example pathway, interaction or physical entity given the RDF IDs. Get commands only retrieve the BioPAX elements that are directly mapped to the ID. Use the `traverse()` query to traverse BioPAX graph and obtain child/owner elements.

Parameters

- **uri** (*str*) – valid/existing BioPAX element's URI (RDF ID; for utility classes that were "normalized", such as entity refereneces and controlled vocabularies, it is usually a Identifiers.org URL. Multiple IDs can be provided using list uri=[http://identifiers.org/uniprot/Q06609, http://identifiers.org/uniprot/Q549Z0'] See also about MIRIAM and Identifiers.org.
- **format** (*str*) – output format (values)

Returns a complete BioPAX representation for the record pointed to by the given URI is returned. Other output formats are produced by converting the BioPAX record on demand and can be specified by the optional format parameter. Please be advised that with some output formats it might return "no result found" error if the conversion is not applicable for the BioPAX result. For example, BINARY_SIF output usually works if there are some interactions, complexes, or pathways in the retrieved set and not only physical entities.

class knowledgebases.**KnowledgeBaseFactory**
Bases: `object`

Singleton class. Python code encapsulates it in a way that is not shown in Sphinx, so have a look at the descriptions in the source code.

Creates knowledge bases based on the provided name and creates all corresponding objects, e.g. web service endpoints. Every knowledge base implementation must be registered here, otherwise it will not be accessible.

`instance = None`

class `knowledgebases.`**`KnowledgeBase`**(*name*, *kb_config*, *webservice*, *geneInfo*, *pathwayInfo*)

Bases: `object`

Super class for every knowledge base implementation. If a new knowledge base is implemented, it must inherit from this class and implement methods *`KnowledgeBase.getRelevantGenes()`*, *`KnowledgeBase.getGeneScores()`*, and *`KnowledgeBase.getRelevantPathways()`*.

Parameters

- **`name`** (*`str`*) – name of the knowledge base
- **`config`** (*`dict`*) – configuration parameter of the knowledge base as specified in the config file.
- **`webservice`** (`bioservices.REST` or inheriting classes.) – web service querying object
- **`hasGeneInformation`** (*`bool`*) – true if the knowledge base provides gene association information, false otherwise
- **`hasPathwayInformation`** (*`bool`*) – true if the knowledge base also provides pathway information, false otherwise

`getRelevantGenes`(*labels*)

Abstract. Get all genes that are associated to a list of labels, e.g. disease names.

Parameters **`labels`** (*`list of str`*) – list of labels for which to retrieve the genes.

Returns list of associated genes.

Return type list of str

`getGeneScores`(*labels*)

Abstract. Get all genes and their association scores for a given list of disease names.

Parameters **`labels`** (*`list of str`*) – list of disease names for which to get gene-disease-association scores.

Returns DataFrame of genes and their association scores.

Return type `pandas.DataFrame`

`getRelevantPathways`(*labels*)

Get all pathways related to a set of labels, e.g disease names.

Parameters **`labels`** (*`list of str`*) – list of labels for which to find related pathways.

Returns dict of pathway names and pathway representations.

Return type dict with `pypath.Network` as values

`getName`()

Returns the name of the knowledge base.

Returns knowledge base name.

Return type str

`hasPathways`()

Returns if knowledge base retrieves pathway information, i.e. if *`KnowledgeBase.getRelevantPathways()`* is implemented..

Returns true if knowledge base provides pathway information, false otherwise.

Return type bool

hasGenes()

Returns if knowledge base retrieves gene information, i.e. if `KnowledgeBase.getRelevantGenes()` `KnowledgeBase.getGeneScores()` are implemented.

Returns true if knowledge base provides gene information, false otherwise.

Return type bool

class knowledgebases.**Enrichr**

Bases: `knowledgebases.KnowledgeBase`

Special knowledge base not intended to be used by feature selection approaches. Instead, it is used for evaluation purposes to annotate and enrich rankings.

Parameters

- **name** (*str*) – name of the knowledge base
- **config** (*dict*) – configuration parameter of the knowledge base as specified in the config file.
- **webservice** (bioservices.REST or inheriting classes.) – web service querying object
- **hasGeneInformation** (*bool*) – true if the knowledge base provides gene association information, false otherwise
- **hasPathwayInformation** (*bool*) – true if the knowledge base also provides pathway information, false otherwise

downloadEnrichedTerms(*userIdList*, *filePrefix*)

Downloads enriched terms from a former query into a file. Filters these terms for those with an adjusted p-value > 0.05, then sorts by combined score in descending order.

Parameters

- **userIdList** (*str*) – userIdList to retrieve enrichment/annotation results from the original query.
- **filePrefix** (*str*) – prefix to use in filename.

getRelevantGenes(*labels*)

Is not implemented for EnrichR.

Parameters **labels** (*list of str*) – list of gene names to be mapped

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

getGeneScores(*labels*)

Is not implemented for EnrichR.

Parameters **labels** (*list of str*) – list of gene names to be mapped

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

getRelevantPathways(*labels*)
Is not implemented for EnrichR.

Parameters **labels** (*list of str*) – list of labels for which to find related pathways.

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

enrichGeneset(*geneList*, *filePrefix*)
Sends a list of identifies (here, genes) to EnrichR web service and stores all term enrichments in a file.

Parameters

- **geneList** (*list of str*) – list of gene names for which to retrieve enrichments.
- **filePrefix** (*str*) – prefix to use in file name (to store enrichments).

annotateGene(*gene*)
Annotates a gene with terms.

Parameters **gene** (*str*) – gene name.

Returns list of all annotations to the provided gene.

Return type list of str

annotateGenes(*geneList*, *filePrefix*)
Annotates a list of genes with relevant terms.

Parameters

- **geneList** (*list of str*) – list of gene names to annotate.
- **filePrefix** (*str*) – prefix to use when storing results in a file.

Returns dict of gene names and lists of their annotations.

Return type dict

class knowledgebases.**BioMART**
Bases: `object`

Maps a identifiers or data sets with identifiers to the desired format by using BiomaRt. Wrapper class that internally invokes BiomaRt's R code. Very unstable, so currently not used. However, it can be exchanged in *benchutils.retrieveMappings()* function.

mapItems(*itemList*, *originalFormat*, *desiredFormat*)
Map a list of identifiers to the desired format. Internally invokes external R code that uses the BiomaRt package.

Parameters

- **itemList** (*list of str*) – list of identifiers to be mapped
- **originalFormat** (*str*) – original identifier format.
- **desiredFormat** (*str*) – format to which to map identifiers.

Returns mapping data frame of identifiers (with original and desired format)

Return type `pandas.DataFrame`

getRelevantGenes(*labels*)
Is not implemented for BiomaRt.

Parameters **labels** (*list of str*) – list of gene names to be mapped

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

`getGeneScores`(*labels*)
Is not implemented for BiomaRt.

Parameters **`labels`** (*`list of str`*) – list of gene names to be mapped

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

`getRelevantPathways`(*labels*)
Is not implemented for BiomaRt.

Parameters **`labels`** (*`list of str`*) – list of labels for which to find related pathways.

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

class `knowledgebases.`**`Gconvert`**
Bases: *`knowledgebases.KnowledgeBase`*

Maps identifiers or data sets containing identifiers to the desired format by using the g:Convert web service.

Parameters

- **`name`** (*`str`*) – name of the knowledge base
- **`config`** (*`dict`*) – configuration parameter of the knowledge base as specified in the config file.
- **`webservice`** (`bioservices.REST` or inheriting classes) – web service querying object.
- **`hasGeneInformation`** (*`bool`*) – true if the knowledge base provides gene association information, false otherwise
- **`hasPathwayInformation`** (*`bool`*) – true if the knowledge base also provides pathway information, false otherwise

`mapItems`(*itemList*, *originalFormat*, *desiredFormat*)
Map a list of identifiers to the desired format.

Parameters

- **`itemList`** (*`list of str`*) – list of identifiers to be mapped.
- **`originalFormat`** (*`str`*) – current format of the identifiers.
- **`desiredFormat`** (*`str`*) – desired format to which to map identifiers.

Returns DataFrame table containing mappings of the identifiers from original to desired format.

Return type `pandas.DataFrame`

`getRelevantGenes`(*labels*)
Is not implemented for g:Convert.

Parameters **`labels`** (*`list of str`*) – list of gene names to be mapped

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

`getGeneScores`(*labels*)
Is not implemented for g:Convert.

Parameters **`labels`** (*`list of str`*) – list of gene names to be mapped

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

`getRelevantPathways`(*labels*)
Is not implemented for g:Convert.

Parameters **`labels`** (*`list of str`*) – list of labels for which to find related pathways.

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

`class` `knowledgebases.`**`OpenTargets`**
Bases: *`knowledgebases.KnowledgeBase`*

Knowledge base implementation of OpenTargets. Uses the OpenTargetsClient Python implementation provided by OpenTargets to query the web service API.

Parameters

- **`name`** (*`str`*) – name of the knowledge base
- **`config`** (*`dict`*) – configuration parameter of the knowledge base as specified in the config file.
- **`webservice`** (`opentargets.OpenTargetsClient`) – web service querying implementation.
- **`hasGeneInformation`** (*`bool`*) – true if the knowledge base provides gene association information, false otherwise
- **`hasPathwayInformation`** (*`bool`*) – true if the knowledge base also provides pathway information, false otherwise

`getAssociations`(*labels*)
Get all relevant information for a given set of labels, sorted by their association scores in descending order. Writes web service results into an intermediate file and maps the identifiers to have the correct format for further processing.

Parameters **`labels`** (*`list of str`*) – list of labels, e.g. disease names.

Returns DataFrame containing all related genes and their association scores.

Return type `pandas.DataFrame`

`getRelevantGenes`(*labels*)
Get all genes that are somehow associated to the given labels, e.g. disease names.

Parameters **`labels`** (*`list of str`*) – list of identifiers, e.g. disease names, for which to find associated genes.

Returns list of associated genes.

Return type list of str

getGeneScores(*labels*)
Get all genes and their association scores that are related to the given labels, e.g. disease names.

Parameters **labels** (*list of str*) – list of identifiers, e.g. disease names, for which to find associated genes.

Returns DataFrame of associated genes and their association scores, in descending order.

Return type `pandas.DataFrame`

getRelevantPathways(*labels*)
As OpenTargets currently does not provide pathway information, this feature is not implemented for OpenTargets.

Parameters **labels** (*list of str*) – list of labels for which to find related pathways.

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

class knowledgebases.**Kegg**(*pathwayparser*)
Bases: *knowledgebases.KnowledgeBase*

Knowledge base implementation for KEGG. Uses the KEGG web service implementation provided by bioservices. Requires an instance of *KEGGPathwayParser* to be able to map retrieved pathways into the internal pathway format.

Parameters

- **name** (*str*) – name of the knowledge base
- **config** (*dict*) – configuration parameter of the knowledge base as specified in the config file.
- **webservice** (`bioservices.KEGG`) – web service querying implementation.
- **hasGeneInformation** (*bool*) – true if the knowledge base provides gene association information, false otherwise
- **hasPathwayInformation** (*bool*) – true if the knowledge base also provides pathway information, false otherwise
- **pathwayparser** (*KEGGPathwayParser*) – pathway mapping class that transforms KEGG pathways in SIF format into the internally used pathway format.

getPathwayNames(*labels*)
Retrieve all pathway names related to the given labels, e.g. disease names.

Parameters **labels** (*list of str*) – list labels, e.g. disease names, for which to find pathways.

Returns list of pathway names.

Return type list of str

getRelevantGenes(*labels*)
Get all genes that are related to a set of labels, e.g. disease names. For KEGG, this means we retrieve all genes that are contained in pathways associated to these labels.

Parameters **labels** (*list of str*) – list of identifiers, e.g. disease names, for which to find associated genes.

Returns list of associated genes.

Return type list of str

getGeneScores(*labels*)
Get association scores for all genes that are related to the provided labels, e.g. disease names. For KEGG, the association score for a gene is the sum of its degree percentile rank for every pathway, normalized by the overall number of pathways retrieved. This favors hub genes/genes having many interactions with other genes.

Parameters **labels** (*list of str*) – list of identifiers, e.g. disease names, for which to find associated genes.

Returns DataFrame of associated genes and their association scores, in descending order.

Return type `pandas.DataFrame`

getRelevantPathways(*labels*)
Get all pathways related to a set of labels, e.g. disease names. Uses the *KEGGPathwayParser* to map KEGG's pathways from SIF to `pypath.Network`.

Parameters **labels** (*list of str*) – list of gene names to be mapped

Returns dict of pathway names and their internal representation as `pypath.Network`.

Return type dict

class knowledgebases.**Disgenet**
Bases: *knowledgebases.KnowledgeBase*

Knowledge base implementation for DisGeNET.

Parameters

- **name** (*str*) – name of the knowledge base
- **config** (*dict*) – configuration parameter of the knowledge base as specified in the config file.
- **webservice** (*DISGENET*) – web service querying implementation.
- **hasGeneInformation** (*bool*) – true if the knowledge base provides gene association information, false otherwise
- **hasPathwayInformation** (*bool*) – true if the knowledge base also provides pathway information, false otherwise

getRelevantGenes(*labels*)
Get all genes that are related to a set of labels, e.g. disease names.

Parameters **labels** (*list of str*) – list of identifiers, e.g. disease names, for which to find associated genes.

Returns list of associated genes.

Return type list of str

getGeneScores(*labels*)
Get association scores for all genes that are related to the provided labels, e.g. disease names. DisGeNET provides a couple of association scores to its genes (https://www.disgenet.org/dbinfo). Which score to use can be defined by the user in the config file.

Parameters **labels** (*list of str*) – list of identifiers, e.g. disease names, for which to find associated genes.

Returns DataFrame of associated genes and their association scores, in descending order.

Return type `pandas.DataFrame`

getRelevantPathways(*labels*)

As DisGeNET currently does not provide pathway information, this feature is not implemented.

Parameters **labels** (*list of str*) – list of labels for which to find related pathways.

Returns `NotImplementedError` as this knowledge base is not intended to be used for such analyses.

Return type `NotImplementedError`

class knowledgebases.**Pathwaycommons**

Bases: *knowledgebases.KnowledgeBase*

Knowledge base implementation for PathwayCommons.

Parameters

- **name** (*str*) – name of the knowledge base
- **config** (*dict*) – configuration parameter of the knowledge base as specified in the config file.
- **webservice** (`opentargets.OpenTargetsClient`) – web service querying implementation.
- **hasGeneInformation** (*bool*) – true if the knowledge base provides gene association information, false otherwise
- **hasPathwayInformation** (*bool*) – true if the knowledge base also provides pathway information, false otherwise

getGeneScores(*labels*)

Get association scores for all genes that are related to the provided labels, e.g. disease names. For PathwayCommons, the association score for a gene is the sum of its degree percentile rank for every pathway, normalized by the overall number of pathways retrieved. This favors hub genes/genes having many interactions with other genes.

Parameters **labels** (*list of str*) – list of identifiers, e.g. disease names, for which to find associated genes.

Returns DataFrame of associated genes and their association scores, in descending order.

Return type `pandas.DataFrame`

getRelevantGenes(*labels*)

Get all genes that are related to a set of labels, e.g. disease names. For PathwayCommons, this means we retrieve all genes that are contained in pathways associated to these labels.

Parameters **labels** (*list of str*) – list of identifiers, e.g. disease names, for which to find associated genes.

Returns list of associated genes.

Return type list of str

readPathway(*pathway*)

Reads a pathway to create `pypath.Network`.

Parameters **pathway** (*str*) – pathway string to parse

getRelevantPathways(*labels*)

Get all pathways related to a set of labels, e.g. disease names as `pypath.Network`.

Parameters **labels** (*list of str*) – list of gene names to be mapped

Returns dict of pathway names and their internal representation as `pypath.Network`.

Return type dict

class knowledgebases.**PathwayParser**

Bases: `object`

Super class that maps a pathway from its original format (provided by a knowledge base) to the internally used `pypath.Network`. When having to map pathways from a knowledge base, implement a new class that inherits from this one and implements *`PathwayParser.parsePathway()`*.

parsePathway (*pathway*, *pathwayID*)

Abstract method. Parse a pathway to the internally used format of `pypath.Network`.

Parameters

- **pathway** (*`str`*) – pathway string to parse
- **pathwayID** (*`str`*) – name of the pathway

Returns pathway in the internally used format..

Return type `pypath.Network`

class knowledgebases.**KEGGPathwayParser**

Bases: *`knowledgebases.PathwayParser`*

Parse KEGG pathways, which are returned in KGML format.

readInteractions (*interactions*, *geneIds*)

Parses interactions for a set of genes.

Parameters

- **interactions** (*`list`*) – interactions to parse
- **geneIds** (*`list of str`*) – gene ids whose interactions to add

parsePathway (*kgml_pathway*, *pathwayID*)

Parse KEGG pathway to the internally used format of `pypath.Network`.

Parameters

- **pathway** (*`str`*) – pathway string to parse
- **pathwayID** (*`str`*) – name of the pathway

Returns pathway in the internally used format..

Return type `pypath.Network`

9.6 evaluation module

Contains all classes related to the evaluation part. There are distinct classes for the following evaluation aspects: * review of knowledge base coverage for the provided search terms (see *`evaluation.KnowledgeBaseEvaluator`*) * inspection of data set quality, e.g. via mds or density plots (see *`evaluation.DatasetEvaluator`*) * comparison and assessment of feature rankings, e.g. overlap (see *`evaluation.RankingsEvaluator`*) * annotation of feature rankings and enrichment via EnrichR (see *`evaluation.AnnotationEvaluator`*) * classification and subsequent visualization of standard metrics (see *`evaluation.ClassificationEvaluator`*) * cross-classification across a second data set and visualization of standard metrics (see *`evaluation.CrossEvaluator`*)

Every one of these classes inherits from the abstract *`evaluation.Evaluator`* and implements the evaluate() method. *`evaluation.AttributeRemover`* is used in the overall benchmarking process to prepare the input data to contain only the selected features. For a detailed look at the class architecture, have a look at ADD CLASS ARCHITECTURE LINK HERE.

class `evaluation.`**`AttributeRemover`**(*dataDir*, *rankingsDir*, *topK*, *outputDir*)

Bases: `object`

Prepares the input data set for subsequent classification by removing lowly-ranked features and only keeping the top k features. Creates one "reduced" file for every ranking and from one to k (so if k is 50, we will end up with 50 files having one and up to 50 features.

Parameters

- **`dataDir`** (*`str`*) – absolute path to the directory that contains the input data set whose features to reduce.
- **`rankingsDir`** (*`str`*) – absolute path to the directory that contains the rankings.
- **`topK`** (*`str`*) – maximum numbers of features to select.
- **`outputDir`** (*`str`*) – absolute path to the directory where the reduced files will be stored.

`loadTopKRankings`()

Loads all available rankings from files.

Returns Dictionary with selection methods as keys and a ranked list of the (column) names of the top k features.

Return type dict

`removeAttributesFromDataset`(*method*, *ranking*, *dataset*)

Creates reduced data sets from dataset for the given method's ranking that only contain the top x features. Creates multiple reduced data sets from topKmin to topKmax specified in the config.

Parameters

- **`method`** (*`str`*) – selection method applied for the ranking.
- **`ranking`** (`List` of str) – (ranked) list of feature names from the top k features.
- **`dataset`** (`pandas.DataFrame`) – original input data set

`removeUnusedAttributes`()

For every method and its corresponding ranking, create reduced files with only the top x features.

class `evaluation.`**`Evaluator`**(*input*, *output*, *methodColors*)

Bases: `object`

Abstract super class. Every evaluation class has to inherit from this class and implement its *`evaluate()`* method.

Parameters

- **`input`** (*`str`*) – absolute path to the directory where the input data is located.
- **`output`** (*`str`*) – absolute path to the directory to which to save results.
- **`methodColors`** (*`dict of str`*) – dictionary containing a color string for every selection method.
- **`javaConfig`** (*`str`*) – configuration parameters for java code (as specified in the config file).
- **`rConfig`** (*`str`*) – configuration parameters for R code (as specified in the config file).

- **evalConfig** (*str*) – configuration parameters for evaluation, e.g. how many features to select (as specified in the config file).
- **classificationConfig** (*str*) – configuration parameters for classification, e.g. which classifiers to use (as specified in the config file).

evaluate()

Abstract. Must be implemented by inheriting class as this method is invoked by `framework.Framework` to run the evaluation.

loadRankings(*inputDir*, *maxRank*, *keepOrder*)

Loads all rankings from a specified input directory. If only the top k features shall be in the ranking, set maxRank accordingly, set it to 0 if otherwise (so to load all features). If feature order is important in the returned rankings, set keepOrder to true; if you are only interested in what features are among the top maxRank, set it to false.

Parameters

- **inputDir** (*str*) – absolute path to directory where all rankings are located.
- **maxRank** (*int*) – maximum number of features to have in ranking.
- **keepOrder** (*bool*) – whether the order of the features in the ranking is important or not.

Returns Dictionary of rankings per method, either as ordered list or set (depending on keepOrder attribute)

Return type dict

computeKendallsW(*rankings*)

Computes Kendall's W from two rankings. Note: measure does not make much sense if the two rankings are highly disjunct, which can happen especially for traditional approaches.

Parameters rankings (*matrix*) – matrix containing two rankings for which to compute Kendall's W.

Returns Kendall's W score.

Return type float

class evaluation.**ClassificationEvaluator**(*inputDir*, *rankingsDir*, *intermediateDir*, *outputDir*, *methodColors*, *methodMarkers*)

Bases: *evaluation.Evaluator*

Evaluates selection methods via classification by using only the selected features and computing multiple standard metrics. Uses *AttributeRemover* to create reduced datasets containing only the top k features, which are then used for subsequent classification. Currently, classification and subsequent evaluation is wrapped here and is actually carried out by java jars using WEKA.

Parameters

- **input** (*str*) – absolute path to the directory where the input data for classification is located.
- **rankingsDir** (*str*) – absolute path to the directory where the rankings are located.
- **intermediateDir** (*str*) – absolute path to the directory where the reduced datasets (containing only the top k features) are written to.
- **output** (*str*) – absolute path to the directory to which to save results.
- **methodColors** (*dict of str*) – dictionary containing a color string for every selection method.

- **javaConfig** (*str*) – configuration parameters for java code (as specified in the config file).
- **rConfig** (*str*) – configuration parameters for R code (as specified in the config file).
- **evalConfig** (*str*) – configuration parameters for evaluation, e.g. how many features to select (as specified in the config file).
- **classificationConfig** (*str*) – configuration parameters for classification, e.g. which classifiers to use (as specified in the config file).

drawLinePlot (*inputDir*, *outputDir*, *topK*, *metric*)
Draws a line plot for a given metric, using all files containing evaluation results for that metric in inputDir. In the end, the plot will have one line per feature selection approach for which classification results are available.

Parameters

- **inputDir** (*str*) – absolute path to directory containing all input files (from which to draw the graph).
- **outputDir** (*str*) – absolute path to the output directory where the the graph will be saved.
- **topK** (*int*) – maximum x axis value
- **metric** (*str*) – metric name for which to draw the graph.

evaluate ()
Triggers classification and evaluation in Java and creates corresponding plots for every metric that was selected in the config.

class evaluation.**RankingsEvaluator** (*input*, *dataset*, *outputPath*, *methodColors*)
Bases: *evaluation.Evaluator*

Evaluates the rankings themselves by generating overlaps and comparing fold change differences.

Parameters

- **input** (*str*) – absolute path to the directory where the input data is located.
- **output** (*str*) – absolute path to the directory to which to save results.
- **methodColors** (*dict of str*) – dictionary containing a color string for every selection method.
- **javaConfig** (*str*) – configuration parameters for java code (as specified in the config file).
- **rConfig** (*str*) – configuration parameters for R code (as specified in the config file).
- **evalConfig** (*str*) – configuration parameters for evaluation, e.g. how many features to select (as specified in the config file).
- **classificationConfig** (*str*) – configuration parameters for classification, e.g. which classifiers to use (as specified in the config file).
- **dataset** (*str*) – absolute file path to the input data set (from which features were selected).
- **metrics** (List of str) – list of metrics to apply to ranking evaluation (as specified in the config file).

generateOverlaps ()
Creates overlap plots for the available rankings (set during creating to self.input). For up to two rankings,

use Python's matplotlib to create Venn diagrams. For three rankings and above, create UpsetR (https://github.com/hms-dbmi/UpSetR) diagrams via R.

`loadGeneRanks`(*inputDir*, *topK*)
Used for computing Kendall's W. Loads rankings and creates a table (approach x features) containing individual ranks per feature per approach, e.g. #approach G1 G2 G3 #Ranker1 1 2 3 #Ranker 2 3 1 2

Parameters

- **`inputDir`** (*`str`*) – absolute path to the directory containing all ranking files.
- **`topK`** (*`int`*) – maximum number of features to use (=length of the rankings).

Returns Ranking table containing every assigned rank for every feature per ranking approach.

Return type `numpy.array`

`computePValue`(*W*, *m*, *n*)
Computes the p-value for a given Kendall's W score via a simple permutation (1000 times) test.

Parameters

- **`W`** (*`float`*) – Kendall's W score.
- **`m`** (*`int`*) – number of approaches/rankings to compare.
- **`n`** (*`int`*) – number of features in each ranking.

Returns p-value of Kendall's W score.

Return type float

`computeKendallsWScores`()
Computes Kendall's correlation coefficients (W) and its corresponding p-value for the top 50, 500, 5,000 and all (code: 0) ranked features of existing rankings. Conducts a permutation test for all scores to receive p-value. Writes output to a file containing the correlation coefficients and their corresponding p-value for different length of rankings.

`drawBoxPlot`(*data*, *labels*, *prefix*)
Draws a box plot from the given data with the given labels on the x axis and the given prefix in the headlines.

Parameters

- **`data`** (`List` of lists of floats) – Data to plot; a list containing lists of values.
- **`labels`** (`List` of str) – List of method names.
- **`prefix`** (*`str`*) – Prefix to use for file name and title.

`computeFoldChangeDiffs`()
Computes median and mean fold changes for all selected features per approach. Writes fold changes to file and creates corresponding box plots.

`evaluate`()
Runs evaluations on feature rankings based on what is specified in the config file. Currently, can compute feature overlaps, Kendall's correlation coefficient (W), and box plots for mean and median fold changes of selected features.

`class` `evaluation.`**`CrossEvaluator`**(*input*, *rankingsDir*, *output*, *methodColors*)
Bases: *`evaluation.Evaluator`*

Runs the evaluation across a second data set. Takes the top k ranked features, removes all other features from that second data set. Runs a *`ClassificationEvaluator`* on that second data set with the selected features.

Parameters

- **input** (*str*) – absolute path to the directory where the second data set for cross-validation is located.
- **rankingsDir** (*str*) – absolute path to the directory containing all rankings.
- **output** (*str*) – absolute path to the directory to which to write all classification results.
- **methodColors** (*dict*) – Dictionary that assigns every (ranking) method a unique color (used for drawing subsequent plots).
- **javaConfig** (*str*) – configuration parameters for java code (as specified in the config file).
- **rConfig** (*str*) – configuration parameters for R code (as specified in the config file).
- **evalConfig** (*str*) – configuration parameters for evaluation, e.g. how many features to select (as specified in the config file).
- **classificationConfig** (*str*) – configuration parameters for classification, e.g. which classifiers to use (as specified in the config file).

evaluate()

Runs crossClassification = takes the features selected on the original data set and uses them to classify a second (cross-validation) data set.

class evaluation.**AnnotationEvaluator**(*input*, *output*, *methodColors*)

Bases: *evaluation.Evaluator*

Annotates and enriches feature rankings with EnrichR (https://maayanlab.cloud/Enrichr/). What library to be used for annotation must be specified in the config file. Can compute annotation and enrichment overlaps for different feature rankings. Annotation = annotate features with terms/information Enrichment = check what terms are enriched in a feature ranking and are related to multiple features. Overlaps then can show a) if feature rankings represent the same underlying processes via annotation (maybe although having selected different features), or b) if the underlying processes are equally strongly represented by checking the enrichment (maybe altough having seleced different features).

Parameters

- **input** (*str*) – absolute path to the directory where the second data set for cross-validation is located.
- **output** (*str*) – absolute path to the directory to which to write all classification results.
- **methodColors** (*dict*) – dictionary that assigns every (ranking) method a unique color (used for drawing subsequent plots).
- **metrics** (List of str) – list of metrics to compute, to be configured in config.
- **dataConfig** (*dict*) – config parameters for the input dataset.
- **javaConfig** (*str*) – configuration parameters for java code (as specified in the config file).
- **rConfig** (*str*) – configuration parameters for R code (as specified in the config file).
- **evalConfig** (*str*) – configuration parameters for evaluation, e.g. how many features to select (as specified in the config file).
- **classificationConfig** (*str*) – configuration parameters for classification, e.g. which classifiers to use (as specified in the config file).

countAnnotationPercentages(*featureLists*, *inputDir*)

Count the number of features (=number of lines in annotation file) that have been annotated and compute percentages. Write output to a file "annotationsPercentages.csv" in self.outputDir.

Parameters

- **`featureLists`** (*`dict`*) – dictionary of lists of features per selection method.
- **`inputDir`** (*`str`*) – absolute path to directory containing annotation files.

`loadAnnotationFiles`(*inputDir*, *inputFiles*)
Loads files with feature annotations.

Parameters

- **`inputDir`** (*`str`*) – absolute path to directory containing annotation files.
- **`inputFiles`** (`List` of str) – list of annotation file names to load.

Returns dictionary of annotation sets per selection method.

Return type dict

`computeOverlap`(*inputDir*, *fileSuffix*)
Creates overlap plots for the available annotations/enrichments. For up to two rankings, use Python's matplotlib to create Venn diagrams. For three rankings and above, create UpsetR (https://github.com/hms-dbmi/UpSetR) diagrams via R.

Parameters

- **`inputDir`** (*`str`*) – absolute path to directory containing files for which to compute overlap.
- **`fileSuffix`** (*`str`*) – suffix in filename to recognize the right files.

`evaluate`()
Runs the annotation/enrichment evaluation on the rankings. Depending on what was specified in the config file, annotate and/or enrich feature rankings and compute overlaps or percentages. Overlaps then can show a) if feature rankings represent the same underlying processes via annotation (maybe although having selected different features), or b) if the underlying processes are equally strongly represented by checking the enrichment (maybe altough having seleced different features).

`class` `evaluation.`**`DatasetEvaluator`**(*input*, *output*, *separator*, *options*)
Bases: *`evaluation.Evaluator`*

Creates plots regarding data set quality, currently: MDS, density, and box plots. Wrapper class because the actual evaluation and plot creation is done in an R script.

Parameters

- **`input`** (*`str`*) – absolute path to the directory where the input data set is located (for which to create the plots).
- **`output`** (*`str`*) – absolute path to the directory to which to save plots.
- **`separator`** (*`str`*) – separator character in data set to read it correctly.
- **`options`** (*`list of str`*) – what plots to create, a list of method names that must be specified in the config file.
- **`javaConfig`** (*`str`*) – configuration parameters for java code (as specified in the config file).
- **`rConfig`** (*`str`*) – configuration parameters for R code (as specified in the config file).
- **`evalConfig`** (*`str`*) – configuration parameters for evaluation, e.g. how many features to select (as specified in the config file).
- **`classificationConfig`** (*`str`*) – configuration parameters for classification, e.g. which classifiers to use (as specified in the config file).

evaluate()

Triggers the actual evaluation/plot generation in R. If a second data set for cross-validation was provided, also run the corresponding R script on that data set.

class evaluation.**KnowledgeBaseEvaluator**(*output*, *knowledgebases*, *searchterms*)

Bases: *evaluation.Evaluator*

Creates plots to evaluate knowledge base coverage. Queries the knowledge bases with the given search terms and checks how many genes or pathways are found.

Parameters

- **output** (*str*) – absolute path to the directory to which to save plots.
- **knowledgebases** (*list of str*) – a list of knowledgebases to test.
- **searchterms** (*list of str*) – list of search terms for which to check knowledge base coverage.
- **javaConfig** (*str*) – configuration parameters for java code (as specified in the config file).
- **rConfig** (*str*) – configuration parameters for R code (as specified in the config file).
- **evalConfig** (*str*) – configuration parameters for evaluation, e.g. how many features to select (as specified in the config file).
- **classificationConfig** (*str*) – configuration parameters for classification, e.g. which classifiers to use (as specified in the config file).

drawCombinedPlot(*stats*, *colIndex*, *filename*, *title*, *ylabel1*, *ylabel2*, *colors*)

Creates combined plot of box and bar plot from a data set.

Parameters

- **stats** (pandas.DataFrame) – statistics to plot.
- **colIndex** (*int*) – column index to use as column.
- **filename** (*str*) – filename for the plot.
- **title** (*str*) – title for the plot.
- **ylabel1** (*str*) – label of y axis (left side/box plot).
- **ylabel2** (*str*) – label of y axis (right side/bar plot).
- **colors** (List of str) – List of colors to use for the different search terms.

createKnowledgeBases(*knowledgebaseList*)

Creates knowledge base objects from a given list.

Parameters **knowledgeBaseList** (List of str.) – List of knowledge base names to create.

Returns List of knowledge base objects

Return type List of KnowledgeBase or inheriting classes

checkCoverage(*kb*, *colors*, *useIDs*)

Checks the coverage for a given knowledge base and creates corresponding plots.

Parameters

- **kb** (*knowledgebases.KnowledgeBase* or inheriting class) – knowledge base object for which to check coverage.

- **colors** (List of str) – List of colors to use for plots.

checkPathwayCoverage(*kb*, *colors*, *useIDs*)

Checks the pathway coverage for a given knowledge base and creates corresponding plots.

Parameters

- **kb** (*knowledgebases.KnowledgeBase* or inheriting class) – knowledge base object for which to check pathway coverage.
- **colors** (List of str) – List of colors to use for plots.

evaluate()

Evaluates every given knowledge base and checks how many genes and pathways (and how large they are) are in there for the given search terms. Creates corresponding plots.

CHAPTER 10

R Code Documentation

10.1 Feature Selection

.. **FS_mRMR.R::**

```
#Runs mRMR feature selection in parallel as implemented in the mRMRe package: De␣
↪Jay, N. et al. "mRMRe: an R package for parallelized mRMR ensemble feature␣
↪selection." Bioinformatics (2013).
#Although run in parallel, the performance is still not very good for high-
↪dimensional data sets (>20.000 features).
#The resulting scores sometimes seem to not be sorted.
#However, these scores are the individual features' scores and feature␣
↪combinations can result in a different overall ranking.
#Invoked by python's featureselection.MRMRSelector class.
#
#@param args the input parameters parsed from the command line, consisting of
#         - absolute path to the input data set file.
#         - absolute path to the output file where the ranking will be stored.
#         - maximum number of features to select.

library(mRMRe)

args = commandArgs(trailingOnly=TRUE)

inputFile <- args[[1]]
outputLocation <- args[[2]]
maxFeatures <- args[[3]]


if (length(args)<3) {
  stop("Please supply three arguments: inputFile (data set), outputLocation␣
↪(feature ranking), and maxFeatures (number of features to select)", call.=FALSE)
}
```

(continues on next page)

(continued from previous page)

```
rawData <- read.csv(inputFile, check.names = FALSE, stringsAsFactors = TRUE)
#check.names= FALSE necessary because R introducing X for column names beginning␣
↪with numbers
geneExpressionMatrix <- rawData[-c(1)]
geneExpressionMatrix[,1] <- as.numeric(geneExpressionMatrix[,1])

#do feature selection here
dd <- mRMR.data(data = geneExpressionMatrix)
features <- mRMR.classic(data = dd, target_indices = c(1), feature_count =␣
↪maxFeatures)

ranking <- solutions(features)
scores <- features@scores[[1]]

colNames <- names(geneExpressionMatrix)
featureNames <- list(colNames[unlist(ranking)])

outputData <- list(featureNames,scores)

fileOutput <- as.data.frame(outputData)

colnames(fileOutput) <- c("attributeName", "score")

write.csv(fileOutput, file = outputLocation, row.names = FALSE, sep = "\t")
```

Runs mRMR feature selection in parallel as implemented in the mRMRe package: De Jay, N. et al. “mRMRe: an R package for parallelized mRMR ensemble feature selection.” Bioinformatics (2013). Although run in parallel, the performance is still not very good for high-dimensional data sets (>20.000 features). The resulting scores sometimes seem to not be sorted. However, these scores are the individual features’ scores and feature combinations can result in a different overall ranking. Invoked by `featureselection.MRMRSelector`.

Param args the input parameters parsed from the command line, consisting of a) the absolute path to the input data set file, b) the absolute path to the output file where the ranking will be stored, and c)the maximum number of features to select.

.. FS_LassoPenalty.R::

```
#Runs Lasso feature selection with individual penalty scores for each feature as␣
↪implemented in the xtune package:
#Zeng, C. et al.: "Incorporating prior knowledge into regularized regression",␣
↪Bioinformatics (2020), https://doi.org/10.1093/bioinformatics/btaa776
#Invoked by python's featureselection.LassoPenaltySelector class.
# #
#@param args the input parameters parsed from the command line, consisting of
#         - absolute path to the input data set file.
#         - absolute path to the output file where the ranking will be stored.
#         - absolute path to the input ranking file (where the external rankings␣
↪that will serve as penalty scores are stored).

library(xtune)

args = commandArgs(trailingOnly=TRUE)

input.filename <- args[[1]]
output.filename <- args[[2]]
```

(continues on next page)

(continued from previous page)

```
external.filename <- args[[3]]

if (length(args)<3) {
  stop("Please supply three arguments: inputFile (gene expression data),␣
↪outputLocation (feature ranking), and filename for external scores", call.
↪=FALSE)
}

expression.matrix <- read.csv(input.filename, check.names = FALSE, row.names = 1)
external.scores <- read.csv(external.filename, check.names = FALSE, row.names = 1)
#check.names= FALSE necessary because R introducing X for column names beginning␣
↪with numbers

expression.levels  <- expression.matrix[-c(1,1)]

#we want to predict the labels = classes
labels <- expression.matrix[1]
#make labels numeric
label.types = unique(labels[,1])

labels <- as.numeric(factor(labels[,1], label.types, labels = 1:length(label.
↪types) ))

#train the model
prior.knowledge.model <- xtune(as.matrix(expression.levels), labels, external.
↪scores)

coefs <- coef(prior.knowledge.model)
final.scores <- as.data.frame(as.matrix(coefs))
#rename score column
colnames(final.scores) <- c("score")
#remove intercept element (drop param keeps row names)
final.scores <- final.scores[-c(1),, drop=FALSE]
feature.indexes <- order(final.scores$score,decreasing = TRUE)
final.scores <- final.scores[feature.indexes,, drop = FALSE] #drop retains row␣
↪names
final.scores <- cbind(attributeName = rownames(final.scores), final.scores)

#write dataframe
write.table(final.scores, output.filename, row.names = FALSE, sep = "\t")
```

Runs Lasso feature selection with individual penalty scores for each feature as implemented in the xtune package: Zeng, C. et al.: “Incorporating prior knowledge into regularized regression”, Bioinformatics (2020), https://doi.org/10.1093/bioinformatics/btaa776 Invoked by `featureselection.LassoPenaltySelector`.

Param args the input parameters parsed from the command line, consisting of a) the absolute path to the input data set file, b) the absolute path to the output file where the ranking will be stored, and c) the absolute path to the input ranking file (where the external rankings that will serve as penalty scores are stored).

.. FS_Variance.R::

```
#Runs variance-based feature selection as implemented in the genefilter package.
#For every feature, computes its variance across all samples.
#Features are then ranked in descending order - highly variant features are more␣
↪likely to separate samples into classes and seem to be the most interesting␣
↪ones.
```

(continues on next page)

(continued from previous page)

```
#Invoked by python's featureselection.VarianceSelector class.
#
#@param args the input parameters parsed from the command line, consisting of
#          - absolute path to the input data set file.
#          - absolute path to the output file where the ranking will be stored.

library(genefilter)

args = commandArgs(trailingOnly=TRUE)

if (length(args)<2) {
  stop("Please supply two arguments: inputFile (gene expression data) and␣
↪outputLocation (feature ranking)", call.=FALSE)
}

rawData <- read.csv(args[1], check.names = FALSE)
#check.names= FALSE necessary because R introducing X for column names beginning␣
↪with numbers

geneExpressionMatrix <- rawData[-c(1,2)]

rV <- rowVars(t(geneExpressionMatrix))

ordered <- rV[order(-rV) , drop = FALSE]

orderedNameList <-  names(ordered)

orderedNameValueList <- c(orderedNameList,data.frame(ordered)[,1])

orderedNameValueMatrix <- matrix(orderedNameValueList,ncol=2)

colnames(orderedNameValueMatrix) <- c("attributeName", "score")

write.table(orderedNameValueMatrix, file = args[2], row.names = FALSE, sep = "\t")
```

Runs variance-based feature selection as implemented in the genefilter package. For every feature, computes its variance across all samples. Features are then ranked in descending order - highly variant features are more likely to separate samples into classes and seem to be the most interesting ones. Invoked by `featureselection.VarianceSelector`.

Param args the input parameters parsed from the command line, consisting of ab) the absolute path to the input data set file and b) the absolute path to the output file where the ranking will be stored.

10.2 Utility

.. DataCharacteristicsPlotting.R::

```
#Creates plots to show some characteristics of a given expression data set and␣
↪its class labels.
#Currently supported plots that can be selected:
# - density plots (density)
# - box plot (box)
```

(continues on next page)

(continued from previous page)

```
# - mds plot (mds)
#Invoked by python's evaluation.DatasetEvaluator class.
#
#@param args the input parameters parsed from the command line, consisting of
#          - absolute path to the input expression file.
#          - absolute path to the output directory where the plots will be stored.
#          - separator to use for reading the input expression file.
#          - a boolean value whether the input expression data is labeled or not.
#          - a list of option names that define what plots are created. Currently␣
↪supported: density (density plot), box (boxplot of expression values), mds).
library(ggplot2)
library(dplyr)
library(tidyr)
library(tools)

args = commandArgs(trailingOnly=TRUE)

input.filename <- args[[1]]
output.dir <- args[[2]]
separator <- args[[3]]
labeledData <- args[[4]]
options <- args[5:length(args)]

if (length(options) == 0){
  stop("No data characteristics were selected in config file", call.=FALSE)

}
rownamescol <- 1

data <- read.table(input.filename, sep= separator, header=TRUE, row.names =␣
↪rownamescol, check.names = FALSE, stringsAsFactors = FALSE)
if (labeledData == "TRUE"){
  unlabeleddata <- data[-1]   #remove label column
  labelCol <- colnames(data)[1]
}
filename = file_path_sans_ext(basename(input.filename))

#################### PLOT DENSITIES ####################
if ("density" %in% options){
  output.filename = paste0(output.dir, "density_", filename, ".pdf")
  pl2 <- data %>% gather(key = "Gene", value = "Expression", -1) %>% ggplot(.,␣
↪aes(x=Expression, color = get(labelCol))) + geom_density()
  pdf(output.filename)
  print(pl2)
  dev.off()
}

#################### PLOT DISTRIBUTION ####################
if ("box" %in% options){
  output.filename = paste0(output.dir, "distribution_", filename, ".pdf")
  boxpl <- data %>% gather(key = "Gene", value = "Expression", -1) %>% ggplot(.,␣
↪aes(get(labelCol), Expression, color = get(labelCol))) + geom_boxplot()
  pdf(output.filename)
  print(boxpl)
  dev.off()
}
```

(continues on next page)

(continued from previous page)

```
#################### PLOT MDS ###################
if ("mds" %in% options){
  output.filename = paste0(output.dir, "mds_", filename, ".pdf")
  #compute euclidean distance matrix (euclidean distance is used in the limma␣
↪package)
  dist.eu <- as.matrix(dist(unlabeleddata, method = "euclidean"))
  mds.eu <- as.data.frame(cmdscale(dist.eu))
  mds.eu <- merge(mds.eu, data[1], by="row.names", all=TRUE)

  mdspl <- ggplot(mds.eu, aes(V1, V2, label = classLabel, color = classLabel)) +
    geom_point(size=2) +
    labs(x="", y="", title="MDS by Euclidean") + theme_bw()

  pdf(output.filename)
  print(mdspl)
  dev.off()
}
```

Creates plots to show some characteristics of a given expression data set and its class labels. Currently supported plots that can be selected: density plots (density), box plot (box), and mds plot (mds). Invoked by `evaluation.DatasetEvaluator`.

Param args the input parameters parsed from the command line, consisting of a) the absolute path to the input expression file, b) the absolute path to the output directory where the plots will be stored, c) the separator to use for reading the input expression file, d) a boolean value whether the input expression data is labeled or not, and e) a list of option names that define what plots are created. Currently supported: density (density plot), box (boxplot of expression values), mds).

.. UpsetDiagramCreation.R::

```
#Uses the UpSetR package to create an upset diagram for a given set of features.
#Invoked by python's evaluation.RankingsEvaluator and evaluation.
↪AnnotationEvaluator classes.
#
#@param args the input parameters parsed from the command line, consisting of
#          - absolute path to the output file where to store the created plot.
#          - number of top k features for which to compute the feature set␣
↪overlaps.
#          - absolute path to the input directory containing the input files.
#          - string of color ids separated by "_", used for giving every feature␣
↪ranking a unique color.
#          - list of input files containing the rankings. their order corresponds␣
↪to the order of colors.
library(UpSetR)
library(stringr)

args = commandArgs(trailingOnly=TRUE)

outputFile = args[[1]]

topK = strtoi(args[[2]])

inputPath = args[[3]]

#remove training _
```

(continues on next page)

(continued from previous page)

```
colorstring = substring(args[[4]], 2)

#split by _
colors = str_split(colorstring, "_")
rankings = list()
setCount = 0

for (filename in args[5:length(args)]) {
  approach.topK = topK
  topGenes = list()
  #find fileending
  parts = gregexpr(pattern ='\\.', filename)[[1]]
  method = substr(filename, 1, parts[length(parts)] - 1)
  file = paste(inputPath, filename, sep="")
  ranking <- read.csv(file, sep = "\t", stringsAsFactors = FALSE)
  #get top n genes for the overlap and adapt topK if we have fewer genes
  numRows = nrow(ranking)
  if (numRows > 0){
    if (numRows < approach.topK){
      approach.topK = numRows
    }
    genes = head(ranking, n = approach.topK)[[1]]
    #as the genes in the columns are by default detected as factors, we just get␣
↪the levels (=distinct names)
    topGenes = genes

  }

  #only add approach results if they are not empty
  if (length(topGenes) > 0){
    #make a list out of topGenes otherwise we have a format issue
    rankings[method] = list(topGenes)
    setCount = setCount + 1
  }
}
print(paste0("UPSETR SETCOUNT ", toString(setCount)))
print(paste0("UPSETR COLORCOUNT ", toString(length(colors))))
pdf(file=outputFile, onefile=FALSE)
upset(fromList(rankings), nsets = setCount, order.by = "freq", sets.bar.color =␣
↪unlist(colors, use.names=FALSE))

dev.off()
```

.. **IdentifierMapping.R::**

```
#Maps a set of identifiers to a desired format by using the biomaRt package.
#Currently, usage of this functionality is not enabled as biomaRt showed to be␣
↪very unstable for returning queries in parallel
#(the server is not reachable, the connection is blocked, ...).
#The current implementation sends the identifiers for mapping in chunks of 10.000␣
↪identifiers
#(that was one desperate try to improve biomaRt stability, but it probably did␣
↪not help...).
# Invoked by python's knowledgebases.BioMART class.
#
```

(continues on next page)

(continued from previous page)

```
#@param args the input parameters parsed from the command line, consisting of
#          - original ID format (corresponding to biomaRt identifiers), e.g.␣
↪ensembl_gene_id
#          - desired ID format (corresponding to biomaRt identifiers), e.g. hgnc_
↪symbol
#          - absolute path to the input file, which contains one identifier per␣
↪line
#          - absolute path to the output file where the mapping will be stored.

library(biomaRt)
library(httr)

args = commandArgs(trailingOnly=TRUE)
originalIDFormat <- args[[1]]
requiredIDFormat <- args[[2]]
inputFile <- args[[3]]
outputFile <- args[[4]]
httr::set_config(httr::config(ssl_verifypeer = FALSE))
mart<-useEnsembl(biomart = "ENSEMBL_MART_ENSEMBL",
                 dataset = "hsapiens_gene_ensembl", mirror = "useast")
print(inputFile)
#load data set from file
data <- read.csv(inputFile, header = FALSE, stringsAsFactors = FALSE)
#fetch gene mapping
#attributes: what your results shall include
#filters: what platform your IDs are currently from
#values: the concrete IDs as input
#mapping <- getBM(attributes = c(originalIDFormat, requiredIDFormat), filters =␣
↪originalIDFormat, values=data, mart = mart)
chunksize = 10000

final_mapping = NULL
chunks = as.integer(nrow(data)/chunksize)
if (chunks > 0){
  for (i in 1:chunks){
    start = ((i-1) * chunksize) + 1
    end = i * chunksize
    genechunk = data[start:end,1]
    mapping <- getBM(attributes = c(originalIDFormat, requiredIDFormat), filters␣
↪= originalIDFormat, values=(genechunk), mart = mart)
    if (is.null(final_mapping)){
      final_mapping = mapping
    } else{
      final_mapping = rbind(final_mapping, mapping)
    }
  }
}

leftover = as.integer(nrow(data) - (chunks * chunksize))
if (leftover > 0){
  start = (chunks * chunksize) + 1
  end = nrow(data)
  last_genechunk = data[start:end,1]
  mapping <- getBM(attributes = c(originalIDFormat, requiredIDFormat), filters =␣
↪originalIDFormat, values=(last_genechunk), mart = mart)
  if (is.null(final_mapping)){
    final_mapping = mapping
```

(continues on next page)

(continued from previous page)

```
  } else{
    final_mapping = rbind(final_mapping, mapping)
  }
}
write.csv(final_mapping, outputFile, row.names = FALSE)
```

The current implementation sends the identifiers for mapping in chunks of 10.000 identifiers (that was one desperate try to improve biomaRt stability, but it probably did not help. . .). Invoked by *knowledgebases.BioMART*.

Param args the input parameters parsed from the command line, consisting of a) the original ID format (corresponding to biomaRt identifiers), e.g. ensembl_gene_id, b) the desired ID format (corresponding to biomaRt identifiers), e.g. hgnc_symbol, c) the absolute path to the input file, which contains one identifier per line, and d) the absolute path to the output file where the mapping will be stored.

CHAPTER 11

Java Code Documentation

11.1 Feature Selection

public class **WEKA_FeatureSelector**

```
package de.hpi.bmg;

import weka.core.Instances;

import java.util.ArrayList;
import java.util.List;

/**
 * Entry point class for running a feature selector on a data set.
 * Invoke the jar of this java file to carry out feature selection␣
↪procedure (see an example in :class:`featureselection.
↪InfoGainSelector` how to do that).
 */
public class WEKA_FeatureSelector {

    /**
     * Loads the input data set and creates selector objects based on␣
↪the provided list of feature selector names.
     * Invokes feature selection procedures for all selectors and writes␣
↪the results to the output directory, one file per selector.
     *
     * @param args the parameters provided when invoking the jar.␣
↪Provide the following parameters:
     *             - absolute path to the input data set.
     *             - absolute path to the output directory (where to␣
↪write the feature rankings).
     *             - a string of feature selectors, separated by a comma␣
↪(e.g. "InfoGain,ReliefF").
     */
```

(continues on next page)

(continued from previous page)

```
    public static void main(String[] args) {

        DataLoader dl = new DataLoader(args[0], ",");

        Instances data = dl.getData();

        //delete sample column
        data.deleteAttributeAt(0);
        //System.out.print(data);
        //set classLabel column to classIndex column
        data.setClassIndex(0);
        //System.out.print(data.classAttribute());

        List<String> attributeSelectionMethods = new ArrayList<String>();

        for (int i=2; i < args.length; i++) {
            attributeSelectionMethods.add(args[i]);
        }

        for (String asMethod : attributeSelectionMethods) {

            AttributeSelector as = new AttributeSelector(data, asMethod);

            as.selectAttributes();
            //System.out.print(asMethod);

            as.saveSelectedAttributes(args[1]);
        }

    }

}
```

Entry point class for running a feature selector on a data set. Invoke the jar of this java file to carry out feature selection procedure. Is invoked by during feature selection by `featureselection.InfoGainSelector`.

public static void **`main`** (String[] *args*)

Loads the input data set and creates selector objects based on the provided list of feature selector names. Invokes feature selection procedures for all selectors and writes the results to the output directory, one file per selector.

Parameters

- **`args`** – The parameters provided when invoking the jar. Provide the following parameters: a) the absolute path to the input data set, b) the absolute path to the output directory (where to write the feature rankings), and c) a string of feature selectors to run, separated by a comma (e.g. “InfoGain,ReliefF”).

public class **`DataLoader`**

```
package de.hpi.bmg;

import weka.core.Instances;
import weka.core.converters.CSVLoader;
```

(continues on next page)

(continued from previous page)

```
import java.io.File;
import java.io.IOException;

/**
 * Class for loading a data set from a file.
 * Used by classes WEKA_Evaluator and WEKA_FeatureSelector.
 */
public class DataLoader {


    String sourceFile;

    Instances data;

    /**
     * Constructor method.
     * Loads the data from the specified source file and stores it in the data␣
↪class attribute.
     *
     * @param sourceFile absolute path of the input file from which to load the␣
↪data.
     * @param separator  separator to use for file reading, e.g a comma.
     */
    public DataLoader(String sourceFile, String separator) {
        this.sourceFile = sourceFile;
        loadData(separator);
    }

    /**
     * Returns the loaded data set.
     *
     * @return the data set.
     */
    public Instances getData() {
        return data;
    }

    /**
     * Carries out the actual data loading.
     * Stores the loaded data set in the data class attribute.
     *
     * @param separator separator to use for file reading, e.g. a comma.
     */
    private void loadData(String separator) {

        CSVLoader loader = new CSVLoader();
        loader.setFieldSeparator(separator);
        try {
            loader.setSource(new File(this.sourceFile));
            this.data = loader.getDataSet();
        } catch (IOException e) {
            e.printStackTrace();
            // see https://opensource.apple.com/source/Libc/Libc-320/include/
↪sysexits.h
            System.exit(66);
        }
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Class for loading a data set from a file. Used by classes *WEKA_FeatureSelector* and *WEKA_Evaluator*.

Instances **data**

String **sourceFile**

public **DataLoader**(String *sourceFile*, String *separator*)

Constructor method. Loads the data from the specified source file and stores it in the data class attribute.

Parameters

- **sourceFile** – absolute path of the input file from which to load the data.
- **separator** – separator to use for file reading, e.g a comma.

public Instances **getData**()

Returns the loaded data set.

Returns the data set.

private void **loadData**(String *separator*)

Carries out the actual data loading. Stores the loaded data set in the data class attribute.

Parameters

- **separator** – separator to use for file reading, e.g. a comma.

public class **AttributeSelector**

```
package de.hpi.bmg;

import com.opencsv.CSVWriter;
import weka.attributeSelection.*;
import weka.core.Instances;

import java.io.*;
import java.util.logging.Logger;

/**
 * Selector class that carries out the actual feature selection procedure.
 * Invoked by
 */
public class AttributeSelector {

    private final static Logger LOGGER = Logger.getLogger(AttributeSelector.class.
↪getName());

    private String selectionMethod;
    private Instances data;
    private AttributeSelection attributeSelection;

    /**
     * Constructor method.
     *
     * @param data the input data set from which to select the features.
```

(continues on next page)

(continued from previous page)

```
     * @param selectionMethod name of the feature selector to apply.
     */
    public AttributeSelector(Instances data, String selectionMethod){

        this.data = data;
        this.selectionMethod = selectionMethod;

    }

    /**
     * Do the actual feature selection.
     * Based on the selector name, create corresponding instances of classes␣
↪provided by WEKA and generate a feature ranking.
     */
    public void selectAttributes() {
        ASEvaluation eval;

        switch (this.selectionMethod) {
            case "SVMpRFE":
                //the default kernel for WEKAs SVMAttributeEval is a poly kernel␣
↪(as defined in class SMO)
                eval = new SVMAttributeEval();

                ((SVMAttributeEval) eval).setPercentThreshold(10);

                ((SVMAttributeEval) eval).setPercentToEliminatePerIteration(10);

                break;

            case "GainRatio":

                eval = new GainRatioAttributeEval();

                break;

            case "ReliefF":

                eval = new ReliefFAttributeEval();

                break;

            default:

                eval = new InfoGainAttributeEval();

        }

        Ranker ranker = new Ranker();

        this.attributeSelection = new AttributeSelection();

        this.attributeSelection.setEvaluator(eval);
        this.attributeSelection.setSearch(ranker);

        // perform attribute selection

        long begin = System.currentTimeMillis();
```

(continues on next page)

(continued from previous page)

```
        try {
            this.attributeSelection.SelectAttributes(data);
        } catch (Exception e) {
            e.printStackTrace();
        }

        long end = System.currentTimeMillis();

        long dt = end - begin;

        LOGGER.info("" + dt + "," + this.selectionMethod);
        System.out.println("" + dt + "," + this.selectionMethod);
    }

    /**
     * Creates a feature ranking list and stores it in the specified file.
     *
     * @param saveLocation absolute path to the output file in which to store the␣
↪ranking.
     */
    public void saveSelectedAttributes(String saveLocation) {

        try {


            CSVWriter writer = new CSVWriter(new FileWriter(saveLocation + "/" +␣
↪this.selectionMethod + ".csv"), '\t');

            String[] header = {"attributeName","score"};

            writer.writeNext(header);

            double[][] rankedAttributes = this.attributeSelection.
↪rankedAttributes();

            for (int i = 0; i < rankedAttributes.length; i++) {

                String attributeName = data.attribute((int)␣
↪rankedAttributes[i][0]).name();

                String score = "" + rankedAttributes[i][1];

                String[] entry = {attributeName, score};

                writer.writeNext(entry);
            }

            writer.close();

        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

Selector class that carries out the actual feature selection procedure. Used by *WEKA_FeatureSelector*.

public **AttributeSelector** (Instances *data*, String *selectionMethod*)
Constructor method.

Parameters

- **data** – the input data set from which to select the features.
- **selectionMethod** – name of the feature selector to apply.

public void **saveSelectedAttributes** (String *saveLocation*)
Create a feature ranking list and stores it in the specified file.

Parameters

- **saveLocation** – absolute path to the output file in which to store the ranking

public void **selectAttributes** ()
Do the actual feature selection. Based on the selector name, create corresponding instances of classes provided by WEKA and generate a feature ranking.

11.2 Evaluation

public class **WEKA_Evaluator**

```
package de.hpi.bmg;

import com.opencsv.CSVWriter;
import org.apache.commons.lang3.ArrayUtils;
import weka.classifiers.AbstractClassifier;
import weka.classifiers.bayes.NaiveBayes;
import weka.classifiers.functions.Logistic;
import weka.classifiers.functions.SMO;
import weka.classifiers.lazy.IBk;
import weka.classifiers.trees.J48;
import weka.classifiers.trees.RandomForest;
import weka.core.Instances;

import java.lang.reflect.Array;
import java.util.Arrays;
import java.util.HashMap;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.logging.Logger;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

/**
 * Entry point class for running classification on a data set using only the top␣
↪1 up to k features (one classification round per k).
 * Invoke the jar of this java file to start the classification procedure (see an␣
↪example in :class:`evaluation.ClassificationEvaluator` how to do that).
 * Uses class:`DataLoader` to load input data set and class:`Analyzer` to run the␣
↪actual classification procedure (and compute evaluation metrics).
```

(continues on next page)

(continued from previous page)

```
 * Summarizes results from all classifiers and all input data sets (depending on␣
↪how many features were used) and writes them into output files.
 */
public class WEKA_Evaluator {

    private final static Logger LOGGER = Logger.getLogger(WEKA_Evaluator.class.
↪getName());

    /**
     * Process some of the command line parameters (for classifiers, metrics, and␣
↪input data set locations).
     * Invokes classification procedure for every subdirectory (=selection␣
↪method) that is contained in the input directory.
     *
     * @param args the parameters provided when invoking the jar. Provide the␣
↪following parameters:
     *              - absolute path of the directory containing the reduced input␣
↪data set files (one subdirectory per selection approach).
     *              - absolute path of the output directory (where to write all␣
↪evaluation results).
     *              - minimum number of features to use for classification.
     *              - maximum number of features to use for classification.
     *              - k param for k-fold cross validation.
     *              - a string of classifiers, separated by a comma (e.g. "SVM,
↪KNN3,KNN5").
     *              - a string of metrics to compute, separated by a comma (e.g.
↪"accuracy,specificity,precision").
     */
    public static void main(String[] args) {

        //get reduced data set locations
        File folder = new File(args[0]);
        //the input path should only contain directories - one for each method
        File[] listOfDirs = folder.listFiles();

        //get classifiers from input
        String classifierParams = args[5];
        String[] classifiers = classifierParams.split(",");
        //get metrics to use from input
        String metricParams = args[6];
        String[] metrics = metricParams.split(",");

        for (File methodDir : listOfDirs) {

            classifyAndEvaluate(methodDir.getName(), methodDir.getAbsolutePath(),␣
↪new File(args[1], methodDir.getName()).getAbsolutePath(),
                    Integer.parseInt(args[2]), Integer.parseInt(args[3]), ␣
↪Integer.parseInt(args[4]), classifiers, metrics);
        }
    }

    /**
     * Runs the overall classification procedure for all feature set sizes of a␣
↪particular selection approach.
     * Creates the specified classifier objects and filewriters for the results.
     * For every feature set size from topKmin to topKmax, invoke an instance of␣
↪:class:`Analyzer`to carry out the actual classification and compute the metrics.
```

(continues on next page)

(continued from previous page)

```
     *
     * @param selectionMethod the name of the feature selection method that␣
↪generated the feature sets to evaluate.
     * @param reducedDatasetLocation absolute path to the directory containing␣
↪the reduced input files (with increasing feature set sizes) for classification.
     * @param resultLocation absolute path to the output file to which to write␣
↪the classification results.
     * @param topKmin minimum number of features to use.
     * @param topKmax maximum number of features to use.
     * @param numFolds k parameter for k-fold cross validation.
     * @param classifiers a list of classifier names to use for classification.
     * @param evalMetrics a list of metric names compute for the classification␣
↪results.
     */
    private static void classifyAndEvaluate(String selectionMethod, String␣
↪reducedDatasetLocation, String resultLocation, int topKmin, int topKmax, int␣
↪numFolds, String[] classifiers, String[] evalMetrics) {
        System.out.println(Integer.toString(Array.getLength(evalMetrics)));
        System.out.println(reducedDatasetLocation);
        System.out.println(selectionMethod);
        System.out.println(reducedDatasetLocation);
        System.out.println(resultLocation);

        //LOGGER.info(Integer.toString(Array.getLength(evalMetrics)));
        //LOGGER.info(reducedDatasetLocation);
        //LOGGER.info(selectionMethod);
        //LOGGER.info(reducedDatasetLocation);
        //LOGGER.info(resultLocation);

        HashMap<String, CSVWriter> writers = new HashMap<String, CSVWriter>();
        try {
            AbstractClassifier[] classifierObjects = null;
            AbstractClassifier analyzer = null;
            String[] classifierNames = null;
            //create desired classifiers
            for (String method : classifiers) {
                switch (method) {
                    case "SMO":
                        analyzer = new SMO();
                        classifierObjects = (AbstractClassifier[]) ArrayUtils.
↪addAll(classifierObjects, analyzer);
                        classifierNames = (String[]) ArrayUtils.
↪addAll(classifierNames, "SMO");
                        break;
                    case "LR":
                        analyzer = new Logistic();
                        classifierObjects = (AbstractClassifier[]) ArrayUtils.
↪addAll(classifierObjects, analyzer);
                        classifierNames = (String[]) ArrayUtils.
↪addAll(classifierNames, "LR");
                        break;
                    case "KNN3":
                        analyzer = new IBk();
                        ((IBk) analyzer).setKNN(3);
                        classifierObjects = (AbstractClassifier[]) ArrayUtils.
↪addAll(classifierObjects, analyzer);
                        classifierNames = (String[]) ArrayUtils.
↪addAll(classifierNames, "KNN3");
```

(continues on next page)

(continued from previous page)

```
                        break;
                    case "KNN5":
                        analyzer = new IBk();
                        ((IBk) analyzer).setKNN(5);
                        classifierObjects = (AbstractClassifier[]) ArrayUtils.
↪addAll(classifierObjects, analyzer);
                        classifierNames = (String[]) ArrayUtils.
↪addAll(classifierNames, "KNN5");
                        break;
                    case "NB":
                        analyzer = new NaiveBayes();
                        classifierObjects = (AbstractClassifier[]) ArrayUtils.
↪addAll(classifierObjects, analyzer);
                        classifierNames = (String[]) ArrayUtils.
↪addAll(classifierNames, "NB");
                        break;
                    case "C4.5":
                        analyzer = new J48();
                        classifierObjects = (AbstractClassifier[]) ArrayUtils.
↪addAll(classifierObjects, analyzer);
                        classifierNames = (String[]) ArrayUtils.
↪addAll(classifierNames, "C4.5");
                        break;
                    case "RF":
                        analyzer = new RandomForest();
                        classifierObjects = (AbstractClassifier[]) ArrayUtils.
↪addAll(classifierObjects, analyzer);
                        classifierNames = (String[]) ArrayUtils.
↪addAll(classifierNames, "RF");
                        break;
                    default:
                        System.out.println(method + " is no valid classifier/
↪analysis module. Do nothing.");
                        //LOGGER.info(method + " is no valid classifier/analysis␣
↪module. Do nothing.");
                        continue;
                }

            }

            for (String metric : evalMetrics){
                String filePath = resultLocation + "_" + metric + ".csv";
                writers.put(metric, new CSVWriter(new FileWriter(filePath), '\t',␣
↪CSVWriter.NO_QUOTE_CHARACTER,
                                    CSVWriter.DEFAULT_ESCAPE_CHARACTER,
                                    CSVWriter.DEFAULT_LINE_END));

                String [] attributes = {"#ofAttributes"};
                String [] average = {"average"};
                String[] headerstart = (String[]) ArrayUtils.addAll(attributes,␣
↪classifierNames);
                String[] header = (String[]) ArrayUtils.addAll(headerstart,␣
↪average);
                writers.get(metric).writeNext(header);
            }

            File folder = new File(reducedDatasetLocation);
```

(continues on next page)

(continued from previous page)

```
            File[] listOfDirs = folder.listFiles();

            for (int k = topKmin; k <= topKmax; k++) {
                String datasetFile = "";
                //get the file with k in its name and load its content
                datasetFile = reducedDatasetLocation + "/top" + String.valueOf(k)␣
↪+ "features_" + selectionMethod + ".csv";
                System.out.println("###################################");
                System.out.println(datasetFile);
                //LOGGER.info("###################################");
                //LOGGER.info(datasetFile);
                if (new File(datasetFile).isFile()){
                    DataLoader dl = new DataLoader(datasetFile, "\t");
                    Instances data = dl.getData();
                    data.deleteAttributeAt(0);
                    data.setClassIndex(0);
                    Analyzer ce = new Analyzer(data);

                    System.out.println(": Starting classification evaluation with␣
↪models " + Arrays.toString(classifierNames) + " with k of " + k + " [" +␣
↪datasetFile + "]");

                    //LOGGER.info(": Starting classification evaluation with␣
↪models " + Arrays.toString(classifierNames) + " with k of " + k + " [" +␣
↪datasetFile + "]");

                    HashMap<String, String> results = ce.
↪trainAndEvaluateWithTopKAttributes(k, numFolds, classifierObjects, evalMetrics);
                    for (String metric : writers.keySet()) {
                        CSVWriter writer = writers.get(metric);
                        String resultLine = results.get(metric);
                        String[] line = resultLine.split("\t");
                        writer.writeNext(line);
                        //writer.flush();
                    }
                }
                else {
                    System.out.println("No rankings found for k = " + Integer.
↪toString(k) + ". Stop classification for " + selectionMethod + ".");
                    //LOGGER.info("No rankings found for k = " + Integer.
↪toString(k) + ". Stop classification for " + selectionMethod + ".");
                    break;
                }
                System.out.println(": Finished classification evaluation with␣
↪models " +  Arrays.toString(classifiers) + " with k of " + k + " [" +␣
↪datasetFile + "]");
                //LOGGER.info(": Finished classification evaluation with models "␣
↪+  Arrays.toString(classifiers) + " with k of " + k + " [" + datasetFile + "]");
            }

            //close all open file writers
            for (String metric : evalMetrics) {
                writers.get(metric).close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Entry point class for running classification on a data set using only the top 1 up to k features (one classification round per k). Is invoked by `evaluation.ClassificationEvaluator` to start the classification procedure. Uses `DataLoader` to load input data set and `Analyzer` to run the actual classification procedure (and compute evaluation metrics). Summarizes results from all classifiers and all input data sets (depending on how many features were used) and writes them into output files.

public static void **main**(String[] *args*)

Process some of the command line parameters (for classifiers, metrics, and input data set locations). Invokes classification procedure for every subdirectory (=selection method) that is contained in the input directory.

Parameters

- **args** – the parameters provided when invoking the jar. Provide the following parameters: a) the absolute path of the directory containing the reduced input data set files (one subdirectory per selection approach), b) the absolute path of the output directory (where to write all evaluation results), c) the minimum number of features to use for classification, d) the maximum number of features to use for classification, e) number of folds for cross validation, f) a string of classifiers, separated by a comma (e.g. "SVM,KNN3,KNN5"), and g) a string of metrics to compute, separated by a comma (e.g. "accuracy,specificity,precision").

private static void **classifyAndEvaluate**(String *selectionMethod*, String *reducedDatasetLocation*, String *resultLocation*, int *topKmin*, int *topKmax*, int *numFolds*, String[] *classifiers*, String[] *evalMetrics*)

Runs the overall classification procedure for all feature set sizes of a particular selection approach. Creates the specified classifier objects and filewriters for the results. For every feature set size from topKmin to topKmax, invoke an instance of `Analyzer` to carry out the actual classification and compute the metrics.

Parameters

- **selectionMethod** – the name of the feature selection method that generated the feature sets to evaluate.
- **reducedDatasetLocation** – absolute path to the directory containing the reduced input files (with increasing feature set sizes) for classification.
- **resultLocation** – absolute path to the output file to which to write the classification results.
- **topKmin** – minimum number of features to use.
- **topKmax** – maximum number of features to use.
- **numFolds** – k parameter for k-fold cross validation.
- **classifiers** – a list of classifier names to use for classification.
- **evalMetrics** – a list of metric names compute for the classification results.

public class **Analyzer**

```
package de.hpi.bmg;

import weka.classifiers.Evaluation;
```

(continues on next page)

(continued from previous page)

```
import weka.classifiers.trees.J48;
import weka.classifiers.bayes.NaiveBayes;
import weka.classifiers.functions.Logistic;
import weka.classifiers.functions.SMO;
import weka.classifiers.lazy.IBk;
import weka.classifiers.trees.RandomForest;
import weka.core.Instances;
import java.io.IOException;
import java.util.logging.Logger;
import weka.classifiers.AbstractClassifier;
import java.util.HashMap;

import java.util.Random;

/**
 * Carries out the actual k-fold cross validation on the specified classifiers.
 * Computes the desired evaluation metrics.
 * Uses WEKA.
 */
public class Analyzer {
    private Instances data;

    private final static Logger LOGGER = Logger.getLogger(Analyzer.class.
↪getName());

    /**
     * Constructor method.
     *
     * @param data the data set to use for classification.
     * @return An instance of Analyzer.
     */
    public Analyzer(Instances data) {
        this.data = data;
    }

    /**
     * Runs the actual classification procedure.
     * Uses WEKA to run multiple classifiers (originally specified in config␣
↪file) in a k-fold cross validation manner.
     * Computes standard evaluation metrics as required afterwards.
     *
     * @param numberOfAttributesRetained the data set to use for classification.
     * @param numFolds number of folds for cross validation.
     * @param classifiers a list of classifier objects to use for classification.
     * @param metrics a list of names of evaluation metrics to compute for the␣
↪results.
     * @return the evaluation results as class:HashMap with the metric name as␣
↪identifier and metric results (across classifiers and average) as values.
     */
    public  HashMap<String, String> trainAndEvaluateWithTopKAttributes(int␣
↪numberOfAttributesRetained, int numFolds, AbstractClassifier[] classifiers,␣
↪String[] metrics) {


        HashMap<String, String> returnStrings = new HashMap<String, String>();
        HashMap<String, Double> sums = new HashMap<String, Double>();
        Evaluation eval = null;
```

(continues on next page)

(continued from previous page)

```
        //initialize maps for return strings and average computations
        for (String metric : metrics){
            String startString = Integer.toString(numberOfAttributesRetained);
            returnStrings.put(metric, startString);
            sums.put(metric, 0.0);
        }

        try {
            eval = new Evaluation(this.data);

            double sum = 0.0d;

            //AbstractClassifier analyzer = null;

            for (AbstractClassifier analyzer : classifiers){

                //run the analysis
                eval.crossValidateModel(analyzer, this.data, numFolds, new␣
↪Random(1));
                for (String metric : metrics) {
                    String returnString = returnStrings.get(metric);
                    double metricVal = 0.0;
                    switch (metric) {
                        case "accuracy":
                            metricVal = eval.pctCorrect();
                            break;
                        case "kappa":
                            metricVal = eval.kappa();
                            break;
                        case "AUROC":
                            metricVal = eval.weightedAreaUnderROC();
                            break;
                        case "sensitivity":
                            metricVal = eval.weightedTruePositiveRate();
                            break;
                        case "specificity":
                            metricVal = eval.weightedTrueNegativeRate();
                            break;
                        case "F1":
                            metricVal = eval.weightedFMeasure();
                            break;
                        case "matthewcoef":
                            metricVal = eval.weightedMatthewsCorrelation();
                            break;
                        case "precision":
                            metricVal = eval.weightedPrecision();
                            break;
                    }
                    returnString += "\t" + String.valueOf(metricVal);
                    returnStrings.put(metric, returnString);
                    //update overall sum for average computation
                    sum = sums.get(metric);
                    sums.put(metric, sum + metricVal);
                }
            }
```

(continues on next page)

(continued from previous page)

```
                for (String metric : metrics){
                    String returnString = returnStrings.get(metric);
                    returnString += "\t" + (sums.get(metric) / classifiers.length);
                    returnStrings.put(metric, returnString);
                }

                //System.out.println(eval.toSummaryString(true));

            } catch (Exception e) {
                e.printStackTrace();
            }
            return returnStrings;
        }
}
```

Carries out the actual k-fold cross validation on the specified classifiers. Computes the desired evaluation metrics. Uses WEKA. Invoked by *WEKA_Evaluator*.

public **Analyzer** (Instances *data*)

Constructor method.

Parameters

- **data** – the data set to use for classification.

Return An instance of *Analyzer*.

public HashMap<String, String> **trainAndEvaluateWithTopKAttributes** (int *numberOfAttributesRetained*, int *numFolds*, AbstractClassifier[] *classifiers*, String[] *metrics*)

Runs the actual classification procedure. Uses WEKA to run multiple classifiers (originally specified in config file) in a k-fold cross validation manner. Computes standard evaluation metrics as required afterwards.

Parameters

- **numberOfAttributesRetained** – the data set to use for classification.
- **numFolds** – number of folds for cross validation.
- **classifiers** – a list of classifier objects to use for classification.
- **metrics** – a list of names of evaluation metrics to compute for the results.

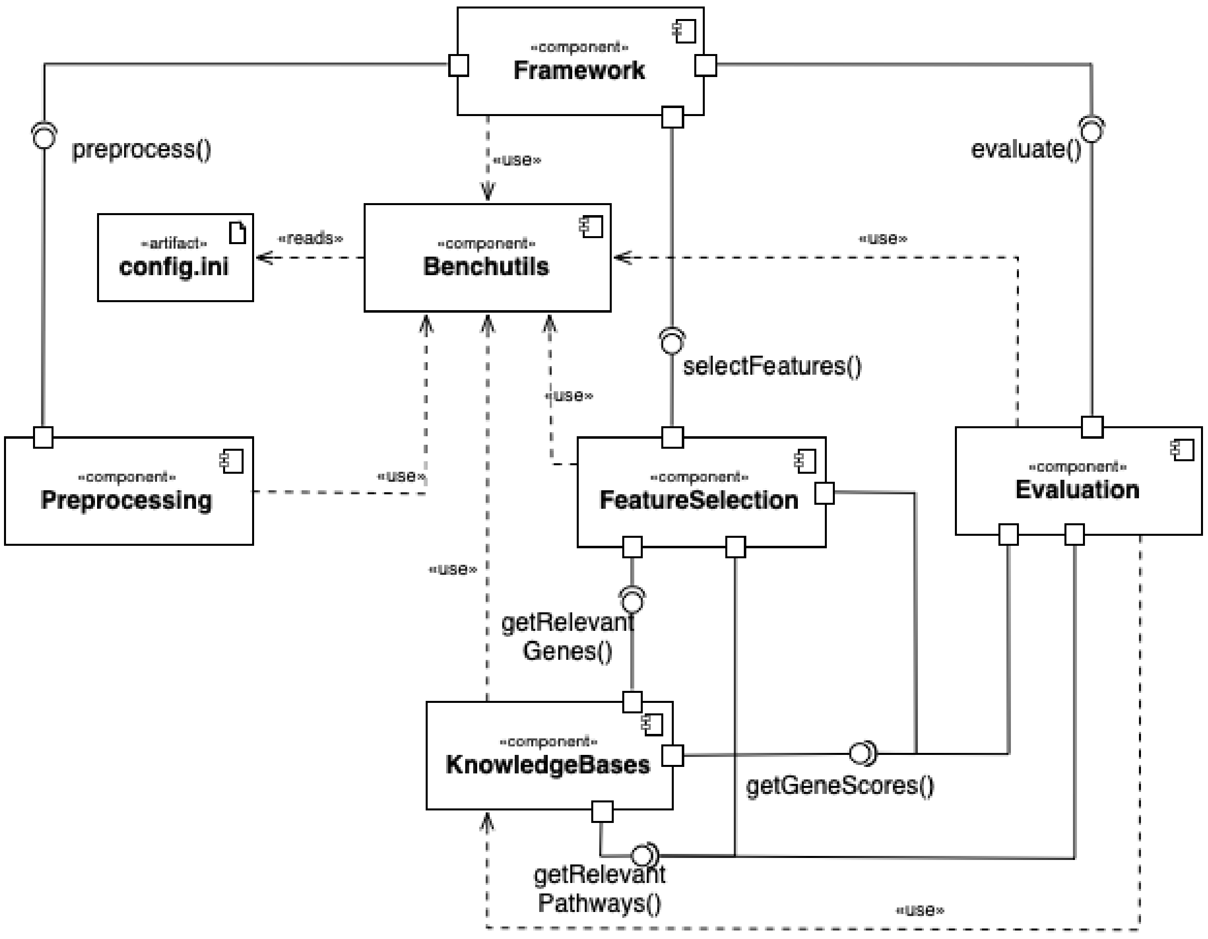
Returns the evaluation results as HashMap with the metric name as identifier and metric results (across classifiers and average) as values.

CHAPTER 12

System Architecture

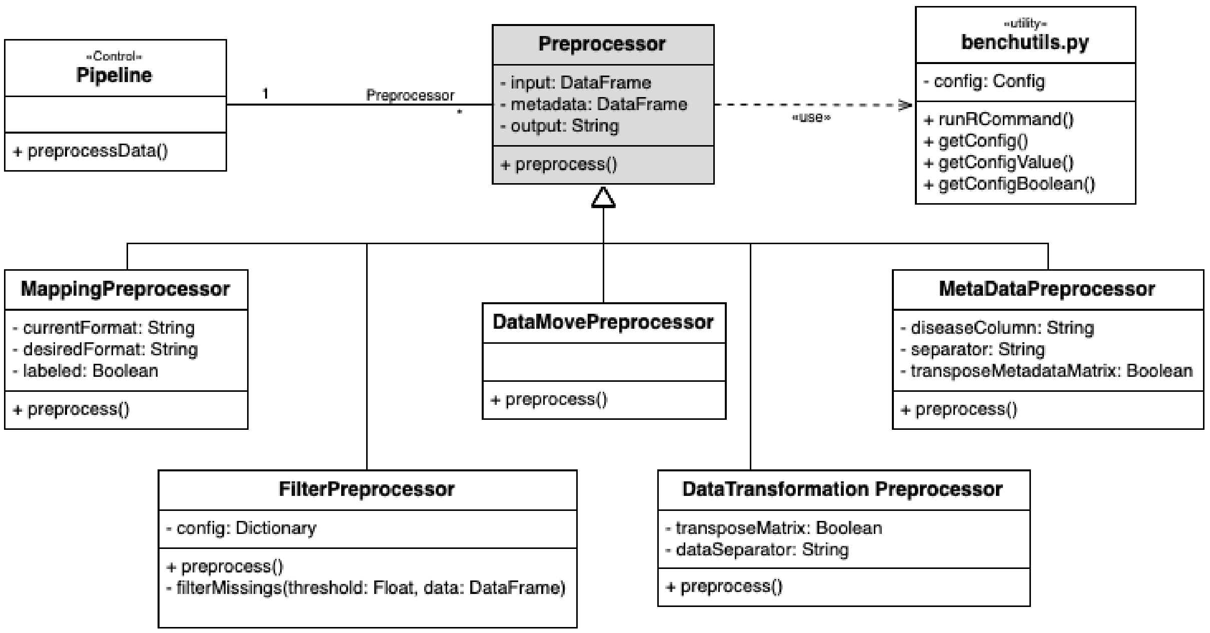
12.1 Components Architecture

The image below describes the system components and their interaction points. The interfaces correspond to concrete methods (see the actual components' class diagrams).



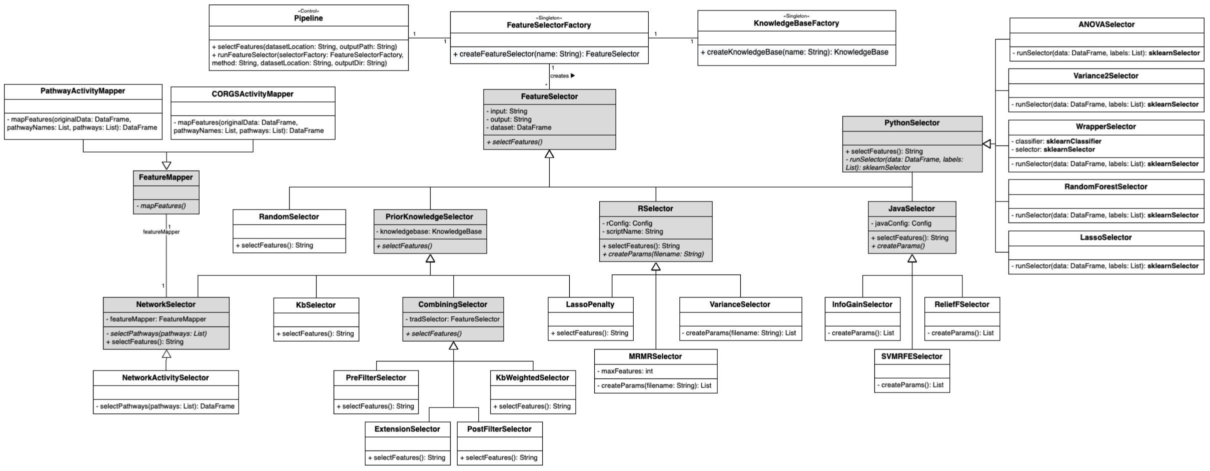
12.2 preprocessing Class Diagram

This module contains all classes related to preprocessing. Every preprocessing functionality is encapsulated in its own class, which must inherit from the abstract *preprocessing.Preprocessor* class (here marked in grey) and implement its *preprocessing.Preprocessor.preprocess()* method



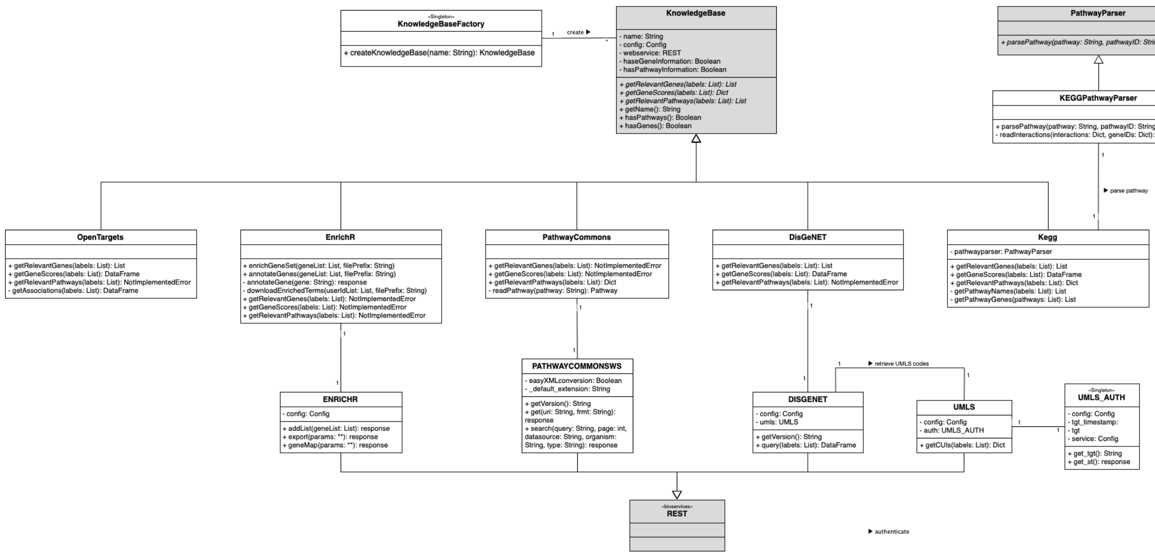
12.3 featureselection Class Diagram

This module contains all classes related to feature selection. Every feature selector is encapsulated in its own class, which must inherit from the abstract `featureselection.FeatureSelector` class or similar (abstract classes are marked in grey) and implement its `featureselection.FeatureSelector.selectFeatures()` method.


«Control»
Pipeline
+ selectFeatures(datasetLocation: String, outputPath: String)
+ runFeatureSelector(selectorFactory: FeatureSelectorFactory, method: String, datasetLocation: String, outputDir: String)
1
1
«Singleton»
FeatureSelectorFactory
+ createFeatureSelector(name: String): FeatureSelector
1
1
«Singleton»
KnowledgeBaseFactory
+ createKnowledgeBase(name: String): KnowledgeBase
1
creates ▶
*
FeatureSelector
- input: String
- output: String
- dataset: DataFrame
+ selectFeatures()
PathwayActivityMapper
- mapFeatures(originalData: DataFrame, pathwayNames: List, pathways: List): DataFrame
CORGSActivityMapper
- mapFeatures(originalData: DataFrame, pathwayNames: List, pathways: List): DataFrame
FeatureMapper
- mapFeatures()
1
featureMapper
1
RandomSelector
+ selectFeatures(): String
PriorKnowledgeSelector
- knowledgebase: KnowledgeBase
+ selectFeatures()
RSelector
- rConfig: Config
- scriptName: String
+ selectFeatures(): String
+ createParams(filename: String)
PythonSelector
+ selectFeatures(): String
- runSelector(data: DataFrame, labels: List): sklearnSelector
JavaSelector
- javaConfig: Config
+ selectFeatures(): String
+ createParams()
ANOVASelector
- runSelector(data: DataFrame, labels: List): sklearnSelector
Variance2Selector
- runSelector(data: DataFrame, labels: List): sklearnSelector
WrapperSelector
- classifier: sklearnClassifier
- selector: sklearnSelector
- runSelector(data: DataFrame, labels: List): sklearnSelector
RandomForestSelector
- runSelector(data: DataFrame, labels: List): sklearnSelector
LassoSelector
- runSelector(data: DataFrame, labels: List): sklearnSelector
NetworkSelector
- featureMapper: FeatureMapper
- selectPathways(pathways: List)
+ selectFeatures(): String
KbSelector
+ selectFeatures(): String
CombiningSelector
- tradSelector: FeatureSelector
+ selectFeatures()
LassoPenalty
+ selectFeatures(): String
VarianceSelector
- createParams(filename: String): List
InfoGainSelector
- createParams(): List
ReliefFSelector
- createParams(): List
MRMRSelector
- maxFeatures: int
- createParams(filename: String): List
SVMRFESelector
- createParams(): List
NetworkActivitySelector
- selectPathways(pathways: List): DataFrame
PreFilterSelector
+ selectFeatures(): String
KbWeightedSelector
+ selectFeatures(): String
ExtensionSelector
+ selectFeatures(): String
PostFilterSelector
+ selectFeatures(): String

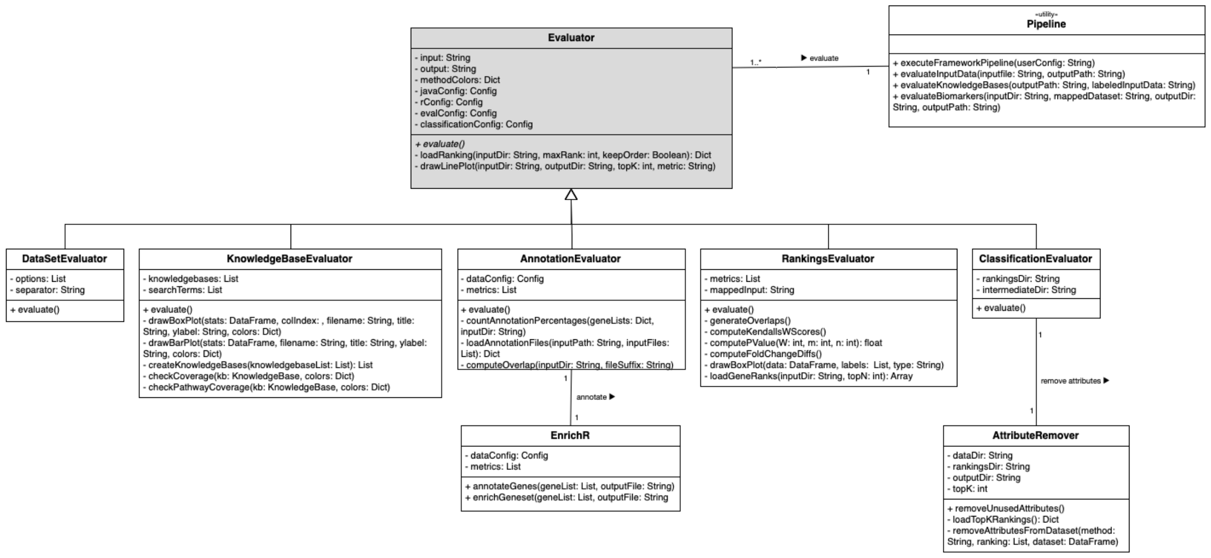
12.4 knowledgebases Class Diagram

This module contains all classes related to external knowledge retrieval. Every knowledge base is encapsulated in two classes, which must inherit from the abstract `knowledgebases.KnowledgeBase` class (the interface to the other components) and bioservices' REST class (the interface to the online web service). Abstract classes are marked in grey.


«Singleton»
KnowledgeBaseFactory
+ createKnowledgeBase(name: String): KnowledgeBase
1
create ▶
*
KnowledgeBase
- name: String
- config: Config
- webservice: REST
- haseGeneInformation: Boolean
- hasPathwayInformation: Boolean
+ getRelevantGenes(labels: List): List
+ getGeneScores(labels: List): Dict
+ getRelevantPathways(labels: List): List
+ getName(): String
+ hasPathways(): Boolean
+ hasGenes(): Boolean
PathwayParser
+ parsePathway(pathway: String, pathwayID: String): Network
KEGGPathwayParser
+ parsePathway(pathway: String, pathwayID: String): Network
- readInteractions(interactions: Dict, geneIDs: Dict): List
1
▶ parse pathway
1
OpenTargets
+ getRelevantGenes(labels: List): List
+ getGeneScores(labels: List): DataFrame
+ getRelevantPathways(labels: List): NotImplementedError
- getAssociations(labels: List): DataFrame
EnrichR
+ enrichGeneSet(geneList: List, filePrefix: String)
+ annotateGenes(geneList: List, filePrefix: String)
- annotateGene(gene: String): response
- downloadEnrichedTerms(userIdList: List, filePrefix: String)
+ getRelevantGenes(labels: List): NotImplementedError
+ getGeneScores(labels: List): NotImplementedError
+ getRelevantPathways(labels: List): NotImplementedError
PathwayCommons
+ getRelevantGenes(labels: List): NotImplementedError
+ getGeneScores(labels: List): NotImplementedError
+ getRelevantPathways(labels: List): Dict
- readPathway(pathway: String): Pathway
DisGeNET
+ getRelevantGenes(labels: List): List
+ getGeneScores(labels: List): DataFrame
+ getRelevantPathways(labels: List): NotImplementedError
Kegg
- pathwayparser: PathwayParser
+ getRelevantGenes(labels: List): List
+ getGeneScores(labels: List): DataFrame
+ getRelevantPathways(labels: List): Dict
- getPathwayNames(labels: List): List
- getPathwayGenes(pathways: List): List
1
1
ENRICHR
- config: Config
+ addList(geneList: List): response
+ export(params: **): response
+ geneMap(params: **): response
1
1
PATHWAYCOMMONSWS
- easyXMLconversion: Boolean
- _default_extension: String
+ getVersion(): String
+ get(uri: String, frmt: String): response
+ search(query: String, page: int, datasource: String, organism: String, type: String): response
1
1
DISGENET
- config: Config
- umls: UMLS
+ getVersion(): String
+ query(labels: List): DataFrame
1
▶ retrieve UMLS codes
1
UMLS
- config: Config
- auth: UMLS_AUTH
+ getCUIs(labels: List): Dict
1
1
«Singleton»
UMLS_AUTH
- config: Config
- tgt_timestamp:
- tgt
- service: Config
+ get_tgt(): String
+ get_st(): response
«bioservices»
REST
▶ authenticate

12.5 evaluation Class Diagram

This module contains all classes related to evaluation functionality. Every evaluation functionality is encapsulated in its own class, which must inherit from the abstract `evaluation.Evaluator` class (abstract classes are marked in grey) and implement its `evaluation.Evaluator.evaluate()` method.


Evaluator
- input: String
- output: String
- methodColors: Dict
- javaConfig: Config
- rConfig: Config
- evalConfig: Config
- classificationConfig: Config
+ evaluate()
- loadRanking(inputDir: String, maxRank: int, keepOrder: Boolean): Dict
- drawLinePlot(inputDir: String, outputDir: String, topK: int, metric: String)
1..*
▶ evaluate
1
«utility»
Pipeline
+ executeFrameworkPipeline(userConfig: String)
+ evaluateInputData(inputfile: String, outputPath: String)
+ evaluateKnowledgeBases(outputPath: String, labeledInputData: String)
+ evaluateBiomarkers(inputDir: String, mappedDataset: String, outputDir: String, outputPath: String)
DataSetEvaluator
- options: List
- separator: String
+ evaluate()
KnowledgeBaseEvaluator
- knowledgebases: List
- searchTerms: List
+ evaluate()
- drawBoxPlot(stats: DataFrame, colIndex: , filename: String, title: String, ylabel: String, colors: Dict)
- drawBarPlot(stats: DataFrame, filename: String, title: String, ylabel: String, colors: Dict)
- createKnowledgeBases(knowledgebaseList: List): List
- checkCoverage(kb: KnowledgeBase, colors: Dict)
- checkPathwayCoverage(kb: KnowledgeBase, colors: Dict)
AnnotationEvaluator
- dataConfig: Config
- metrics: List
+ evaluate()
- countAnnotationPercentages(geneLists: Dict, inputDir: String)
- loadAnnotationFiles(inputPath: String, inputFiles: List): Dict
- computeOverlap(inputDir: String, fileSuffix: String)
RankingsEvaluator
- metrics: List
- mappedInput: String
+ evaluate()
- generateOverlaps()
- computeKendallsWScores()
- computePValue(W: int, m: int, n: int): float
- computeFoldChangeDiffs()
- drawBoxPlot(data: DataFrame, labels: List, type: String)
- loadGeneRanks(inputDir: String, topN: int): Array
ClassificationEvaluator
- rankingsDir: String
- intermediateDir: String
+ evaluate()
1
annotate ▶
1
EnrichR
- dataConfig: Config
- metrics: List
+ annotateGenes(geneList: List, outputFile: String)
+ enrichGeneset(geneList: List, outputFile: String
1
remove attributes ▶
1
AttributeRemover
- dataDir: String
- rankingsDir: String
- outputDir: String
- topK: int
+ removeUnusedAttributes()
- loadTopKRankings(): Dict
- removeAttributesFromDataset(method: String, ranking: List, dataset: DataFrame)

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index