MINE-Database

Tyo Lab

Nov 04, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

MINE-Database, also referred to as Pickaxe, is a python library allows you to efficiently create reaction networks based on a set of reaction rules.

Some common use cases:

1. Predicting promiscuous enzymatic reactions in biological systems.
2. Searching for potential novel reaction pathways from starting compound(s) to target compound(s).
3. Annotating possible structures for unknown peaks in metabolomics datasets.
4. Predicting spontaneous chemical reactions which may be diverting flux from a pathway of interest.
5. Specifying custom reaction rules to extend reaction networks to include chemical reactions.

In all of these cases, you supply pickaxe with a set of starting compounds (as SMILES strings) and which set of reaction rules you would like to useand then Pickaxe does the rest. Pickaxe creates a network expansion by applying these reaction rules iteratively to your starting set of compounds, going for as many generations as you specify. There are many more advanced options and customizations you can add as well.

CHAPTER

TWO

GETTING STARTED

To get started, see *Installation*.

You can run pickaxe in two ways, in command-line mode (*Running Pickaxe via Command Line*) or using a template file (recommended) (*Running Pickaxe*). *Running Pickaxe* also provides information about different compound filters you can apply to your pickaxe expansions.

For a list of inputs required for pickaxe, see *Generating Pickaxe Inputs*.

To learn how to create your own custom filters, see *Custom Filters*.

An API reference is provided at *API Reference* if you need to see implementation details.

Finally, if you find yourself needing help or have feedback for us, please see *Support*!

CHAPTER

THREE

CONTENTS

3.1 Installation

MINE-Database requires the use of rdkit, which currently is unavailable to install on pip. Thus, we recommend you use conda to create a new environment and then install rdkit into that environment before proceeding:

conda create -n mine

conda activate mine

conda install -c rdkit rdkit

Then, use pip (in your conda environment) to install minedatabase:

pip install minedatabase

3.2 Running Pickaxe via Command Line

Pickaxe supports running through a command line interface, but does not offer the full functionality available through writing a python script pickaxe_run.rst.

3.2.1 Command Line Interface Features

```
$ python pickaxe.py -h
usage: pickaxe.py [-h] [-C COREACTANT_LIST] [-r RULE_LIST] [-c COMPOUND_FILE] [-v] [-H]␣
↪[-k] [-n] [-m PROCESSES] [-g GENERATIONS] [-q] [-s SMILES] [-p PRUNING_WHITELIST] [-o␣
↪OUTPUT_DIR] [-d DATABASE] [-u MONGO_URI] [-i IMAGE_DIR]

optional arguments:
-h, --help            show this help message and exit
-C COREACTANT_LIST, --coreactant_list COREACTANT_LIST
                        Specify a list of coreactants as a .tsv
-r RULE_LIST, --rule_list RULE_LIST
                        Specify a list of reaction rules as a .tsv
-c COMPOUND_FILE, --compound_file COMPOUND_FILE
                        Specify a list of starting compounds as .tsv or .csv
-v, --verbose         Display RDKit errors & warnings
-H, --explicit_h      Specify explicit hydrogen for use in reaction rules.
-k, --kekulize        Specify whether to kekulize compounds.
-n, --neutralise      Specify whether to neturalise compounds.
```

(continues on next page)

(continued from previous page)

```
-m PROCESSES, --processes PROCESSES
                        Set the max number of processes.
-g GENERATIONS, --generations GENERATIONS
                        Set the numbers of time to apply the reaction rules to the␣
↪compound set.
-q, --quiet           Silence warnings about imbalanced reactions
-s SMILES, --smiles SMILES
                        Specify a starting compound SMILES.
-p PRUNING_WHITELIST, --pruning_whitelist PRUNING_WHITELIST
                        Specify a list of target compounds to prune reaction network␣
↪down to.
-o OUTPUT_DIR, --output_dir OUTPUT_DIR
                        The directory in which to write files.
-d DATABASE, --database DATABASE
                        The name of the database to store results.
-u MONGO_URI, --mongo_uri MONGO_URI
                        The URI of the mongo database to connect to. Defaults to␣
↪mongodb://localhost:27017
-i IMAGE_DIR, --image_dir IMAGE_DIR
                        Specify a directory to store images of all created compounds
```

3.2.2 Examples

Generate and Save Data to Local directory

This is the simplest example of using the command line interface. It accepts coreactant, rule, and compound files and expands to generations before saving the results in .tsv files in a provided directory.

```
python pickaxe.py -r /path/to/rules.tsv -C path/to/coreactants.tsv -c /path/to/compounds.
↪tsv -g 2 -o /path/to/output/
```

Generate and Save Data to a Mongo Database

It is possible to save to a mongo database, either locally or remotely. This option works with writing a .tsv as well, and will write to both locations.

Local Mongo Server Running the following will use, by default, mongodb://localhost:27017 as the mongo URI.

```
python pickaxe.py -r /path/to/rules.tsv -C path/to/coreactants.tsv -c /path/to/compounds.
↪tsv -g 2 -d database_name
```

Specific Mongo Server Alternatively, a [specific Mongo URI can be specified](https://docs.mongodb.com/manual/reference/connection-string/), allowing for the use of password protected databases and remote databases.

```
python pickaxe.py -r /path/to/rules.tsv -C path/to/coreactants.tsv -c /path/to/compounds.
↪tsv -g 2 -d database_name -u mongodb://myDBReader:D1fficultP%40ssw0rd@mongodb0.example.
↪com:27017/?authSource=admin
```

Generate with Multiple Processes and Pruning Final Network

This example uses 4 processes to run and prunes the final network to contain only compounds that are specified and any compounds required to generate them from the starting compounds.

```
python pickaxe.py -r /path/to/rules.tsv -C path/to/coreactants.tsv -c /path/to/compounds.
↪tsv -g 2 -o /path/to/output/ -m 4 -p /path/to/pruning_targets.tsv
```

3.3 Running Pickaxe

Pickaxe is the program that is used to generate the data that is stored in the MINE-Database. The database is used for metabolomics applications, but pickaxe can be extended for use in general reaction network generation and analysis. An example run, pickaxe_run.py, is found in the github. This python script provides a template for producing pickaxe runs, exposing the key parameters for a user to modify and inputs these into the pickaxe class to run, greatly simplifying the process.

pickaxe_run.py highlights the key components for a pickaxe run block-by-block. This document also serves to highlight and explain the components of running pickaxe. Generally, pickaxe_run.py operates in the following steps:

1. Specify where the output of the run will be stored
2. Specifying the various run inputs
3. Core Pickaxe options
4. Specification of Filters

This document gives the relevant code snippets from a template and expands on existing comments. Additionally, brief examples of relevant inputs will be created. For more detailed descriptions please see *Generating Pickaxe Inputs* and *Filters*.

Tip: To create custom filters, see *Custom Filters*.

3.3.1 Example Template

This document details the specifics of a template file, pickaxe_run.py, that highlights common Pickaxe runs.

`pickaxe_run.py can be downloaded here.`

3.3.2 Run Output

There are two ways to output data:

1. Writing to a mongo database that is specified by a mongo uri, either local or in mongo_uri.csv
2. Local .tsv files

```
# Whether or not to write to a mongodb
write_db = False
database_overwrite = False
# database = "APAH_100Sam_50rule"
database = "example_pathway"
# Message to insert into metadata
```

(continues on next page)

(continued from previous page)

```
message = ("Example run to show how pickaxe is run.")

# mongo DB information
use_local = False
if write_db == False:
    mongo_uri = None
elif use_local:
    mongo_uri = 'mongodb://localhost:27017'
else:
    mongo_uri = open('mongo_uri1.csv').readline().strip('\n')

# Write output .csv files locally
write_to_csv = False
output_dir = '.'
```

3.3.3 Run Input

There are three key inputs for a Pickaxe run to be specified:

1. **input_cpds** specifying the compounds to be reacted
2. **coreactant_list** are coreactants that are required for the reaction rules
3. **rule_list** that specifies the reaction rules to be applied

Input Compounds Example

The file specified for `input_cpds` must be a .tsv or a .csv format. The file consists of an id and a SMILES string. An example of a .csv file is

```
id,SMILES
0,CC(=O)OC
1,CCO
```

Coreactant and Rule lists

Pickaxe is provided with a default rule list generated from approximately 70,000 MetaCyc reactions.

The following code allows you to select then number of rules by either a number or by coverage:

```
from minedatabase.rules import metacyc_generalized
# Select by number
rule_list, coreactant_list, rule_name = metacyc_generalized(n_rules=20)

# Select by fraction coverage
rule_list, coreactant_list, rule_name = metacyc_generalized(fraction_coverage=0.5)
```

When choosing how many reactions to use, you can refer to the following table:

Number of Rules	Percent Coverage of MetaCyc Reactions
20	50
84	75
100	78
272	90
500	95
956	99
1221	100

Note: Rules and coreactants can be generated manually as well, which is outlined in *Generating Pickaxe Inputs*.

Code snippet from Pickaxe_run.py

These input files are specified as follows:

```
input_cpds = './example_data/starting_cpds_single.csv'

# Generate rules automatically from metacyc generalized. n_rules takes precedence over
# fraction_coverage if both specified. Passing nothing returns all rules.
rule_list, coreactant_list, rule_name = metacyc_generalized(
    n_rules=20,
    fraction_coverage=None
)
```

If you generated a file manually then specify the file directly as follows:

```
rule_list = "path/to/rules"
coreactant_list = "path/to/coreactants"
rule_name = "rule name"
```

3.3.4 Core Pickaxe Options

Of these options the majority of uses will only require the changing of the following:

1. **generations** is the number of generations to expand, e.g. 2 generations will apply reaction rules twice
2. **num_works** specifies the number of processors to use

However, the remaining can be changed if needed:

3. **verbose** specifies if RDKit is suppressed or not
4. **kekulize** specifies whether or not to kekulize RDKit molecules
5. **neutralise** specifies whether or not to neutralise molecules
6. **image_dir** specifies the directory where to draw images of generated compounds
7. **quiet** specifies whether or not to suppress output
8. **indexing** specifies whether or not to index the databases

```
generations = 1
processes = 4     # Number of processes for parallelization
verbose = False     # Display RDKit warnings and errors
explicit_h = False
kekulize = True
neutralise = True
image_dir = None
quiet = True
indexing = False
```

3.3.5 Built-In Filters

Three general filters are supplied with Pickaxe:

1. A tanimoto threshold filters
2. A tanimoto sampling filters
3. A metabolomics filters

Specified filters are applied before each generation (and at the end of the run if specified) to reduce the number of compounds to be expanded. This allows for the removal of compounds that aren't of interest to reduce the number of non-useful compounds in the resultant network. Additionally, custom filters can be written. To write your own filter see:

General Filter Options

These options apply to every filter and are independent of the actual filter itself.

1. **target_cpds specifies where the target compound list is. This file is a csv**
   with the header id,SMILES
2. **react_targets** specifies whether a compound generated in the expansion should be further reacted
3. **prune_to_targets** specifies whether the network should be reduced to a minimal network containing only compounds directly connected to the targets from a source
4. **filter_after_final_gen** whether to apply the filter to the final application of reaction rules

```
# Path to target cpds file (not required for metabolomics filter)
target_cpds = './example_data/target_list_single.csv'

# Should targets be flagged for reaction
react_targets = True

# Prune results to remove compounds not required to produce targets
prune_to_targets = True

# Filter final generation?
filter_after_final_gen = True
```

Tanimoto Threshold Filter

The rational behind this filter is to generate a list of Tanimoto similarity scores (ranging from 0 to 1) for each generation in comparison to the targets and use this to trim compounds to only those above a certain similarity threshold. The maximum similarity of a given compound compared to all the targets is used. Similarity is calculated by using the default RDKFingerprints.

Before each generation the maximum similarity for each compound set to be reacted is compared to a threshold. Compounds greater than or equal to the threshold are reacted.

1. **tani_filter** whether or not to use this filter
2. **tani_threshold** is the threshold to cut off. Can be a single value or a list. If a list then the filter will use the next value in this list for each new generation
3. **increasing_tani** specifies whether the tanimoto value of compounds must increase each generation. I.e. a child compound must be more similar to a target than at least one of its parents

```
# Apply this filter?
tani_filter = False

# Tanimito filter threshold. Can be single number or a list with length at least
# equal to the number of generations (+1 if filtering after expansion)
tani_threshold = [0, 0.2, 0.7]

# Make sure tani increases each generation?
increasing_tani = False
```

Tanimoto Sampling Filter

For large expansions the tanimoto threshold filter does not work well. For example, expanding 10,000 compounds from KEGG with 272 rules from metacyc yields 5 million compounds. To expand this another generation the number of compounds has to be heavily reduced for the system resources to handle it and for analysis to be reasonable. The threshold filter will have to be at a large value, e.g. greater than 0.9, which leads to reduced chemical diversity in the final network.

To avoid this problem, the Tanimoto Sampling Filter was implemented. The same approach as the threshold filter is taken to get a list of maximum similarity score for compounds and the list of targets. This tanimoto score is scaled and then the distribution is sampled by inverse complementary distribution function sampling to select N compounds. This approach affords more diversity than the threshold and can be tuned by scaling the tanimoto similarity score scaling function. By default the function is T^4.

The filter is specified as follows:

1. **tani_sample** specifies whether to use the filter
2. **sample_size** specifies the number of compounds to expand each generation. If sample_size is greater than the total number of compounds all compounds are reacted
3. **weight** specifies the weighting function for the sampling. This function accepts a float and returns a float
4. **weight_representation** specifies how to display the weighting function in the database or stdout

```
# Apply this sampler?
tani_sample = False

# Number of compounds per generation to sample
```

(continues on next page)

(continued from previous page)

```
sample_size = 5

# weight is a function that specifies weighting of Tanimoto similarity
# weight accepts one input
# T : float in range 0-1
# and returns
# float in any range (will be rescaled later)
# weight = None will use a T^4 to weight.
def weight(T):
    return T**4

# How to represent the function in text
weight_representation = "T^4"
```

Metabolomics Filter

If you have a metabolomics dataset you would like to filter compounds against, you can use this filter. It will force pickaxe to only keep compounds with masses (and, optionally, retention time (RT)) within a set tolerance of a list of peaks. For example, if you had a dataset containing 3 peaks at 100, 200, and 300 m/z, you could do an expansion and only keep compounds with masses within 0.001 Da of those 3 values.

This is useful for trying to annotate unknown peaks starting from a set of known compounds in a specific organism from which metabolomics data was collected.

The filter is specified as follows. The following arguments are required:

1. **metabolomics_filter** specifies whether to use this filter
2. **met_data_path** specifies where to find your list of peaks in CSV format.

Format of CSV:

> Peak ID, Retention Time, Aggregate M/Z, Polarity, Compound Name, Predicted Structure (smile), ID
>
> Peak1, 6.33, 74.0373, negative, propionic acid, CCC(=O)O, yes
>
> Peak2, 26.31, 84.06869909, positive, , , no
>
> ...

Note that only unidentified peaks will be used by the filter.

3. **possible_adducts** specifies the possible adducts to consider when matching peaks, as different adducts cause different mass changes. For a list of options, see the first columns of "Negative Adducts full.txt" and "Positive Adducts full.txt" in minedatabase/data/adducts.
4. **mass_tolerance** specifies (in Da) the mass tolerance to use for matching peaks. For example, if 0.001, only compounds with masses between 99.999 and 100.001 would match a peak at 100 m/z.

The following optional arguments allow you to add retention time as an extra constraint in the filter. Note that this requires that you have built a RandomForestRegressor machine learning model to predict retention time for arbitrary compounds, using mordred fingerprints as input.

5. **rt_predictor_pickle_path** specifies the path to the built model (pickled). Make sure this is None, if you don't want to match based on retention time.
6. **rt_threshold** specifies the retention time tolerance (in whatever units RT is in the file at met_data_path)

7. **rt_important_features** specifies which mordred descriptors to use as input into the model (must be in same order as model expects them to be). If None, will use all (including 3D) mordred descriptors.

```
# Apply this filter?
metabolomics_filter = False

# Path to csv with list of detected masses (and optionally, retention times).
# For example: Peak ID, Retention Time, Aggregate M/Z, Polarity, Compound Name,
# Predicted Structure (smile), ID
#
# Peak1, 6.33, 74.0373, negative, propionic acid, CCC(=O)O, yes
# Peak2, 26.31, 84.06869909, positive, , , no
# ...
met_data_path = "./local_data/ADP1_Metabolomics_PeakList_final.csv"

# Name of dataset
met_data_name = "ADP1_metabolomics"

# Adducts to add to each mass in mass list to create final list of possible
# masses.
# See "./minedatabase/data/adducts/All adducts.txt" for options.
possible_adducts = ["[M+H]+", "[M-H]-"]

# Tolerance in Da
mass_tolerance = 0.001

# Retention Time Filter Options (optional but included in metabolomics filter)

# Path to pickled machine learning predictor (SMILES => RT)
rt_predictor_pickle_path = "../RT_Prediction/final_RT_model.pickle"

# Allowable deviation in predicted RT (units just have to be consistent with dataset)
rt_threshold = 4.5

# Mordred descriptors to use as input to model (must be in same order as in trained␣
↪model)
# If None, will try to use all (including 3D) mordred descriptors
rt_important_features = ["nAcid", "ETA_dEpsilon_D", "NsNH2", "MDEO-11"]
```

3.4 Generating Pickaxe Inputs

3.4.1 Compound Inputs

Pickaxe takes a few input files to specify compounds and rules for the expansion. One group of these files are simply compounds, some of which are required and others are option, depending on the desired functionality of a given Pickaxe run.

Required:

1. Compounds to react.

Optional:

1. Targets to filter for.

2. Metabolomic data to filter with (see **met_data_path** parameter in *Built-In Filters*).

Compound Input

Pickaxe accepts a .csv or a .tsv that consists of two columns, an *id* field and a *structure* field. The *id* field is used to label the final output and the structure field consists of SMILES representation of compounds.

Here is an example of a valid compound input file:

```
id,SMILES
glucose,C(C1C(C(C(C(C(O1)O)O)O)O)O)O
TAL,C/C1=CC(\O)=C/C(=O)O1
```

Target Input

The target compound input file takes the same form as the input compounds.:

```
id,SMILES
1,C=C(O)COCC(C)O
```

3.4.2 Reaction Operator Inputs

There are two files required for the application of reactions:

1. Reaction operators to use.
2. Coreactants required by the reaction operators.

Default rules are supplied with pickaxe, however custom rules can be written and used.

Default Rules

Overview

A set of biological reaction rules and cofactors are provided by default. These consist of approximately 70,000 MetaCyc reactions condensed into generic rules. Selecting all of these rules will result in a large expansion, but they can be trimmed down significantly while still retaining high coverage of MetaCyc reactions.

Number of Rules	Percent Coverage of MetaCyc Reactions
20	50
100	78
272	90
500	95
956	99
1221	100

Additionally, a set of intermediate reaction rule operators are provided as well. These operators are less generalized than the generalized ruleset and provide uniprot information for each operator.

Generating Default Rule Inputs

Default rules are imported from the rules module of minedatabase and have a few options to specify what is loaded:

1. Number of Rules
2. Fractional Coverage of MetaCyc
3. Anaerobic Rules only
4. Groups to Include
5. Groups to Ignore

Possible groups to ignore and include are: aromatic, aromatic_oxygen, carbonyl, nitrogen, oxygen, fluorine, phosphorus, sulfur, chlorine, bromine, iodine, halogen. Examples of Defining rules are given below.

The provided code returns the rule_list and coreactant_list that is passed to the pickaxe object.

Generalized Rules Mapping 90% Metacyc

```
from minedatabase.rules import metacyc_generalized
rule_list, coreactant_list, rule_name = metacyc_generalized(
    fraction_coverage=0.9
)
```

Generalized Rules with 200 Anaerobic and Halogens

```
from minedatabase.rules import metacyc_generalized
rule_list, coreactant_list, rule_name = metacyc_generalized(
    n_rules=200
    anaerobic=True,
    include_containing=["halogen"]
)
```

Intermediate Rules with all Halogens except Chlorine

```
from minedatabase.rules import metacyc_intermediate
rule_list, coreactant_list, rule_name = metacyc_intermediate(
    include_containing=["halogen"],
    exclude_containing=["chlorine"]
)
```

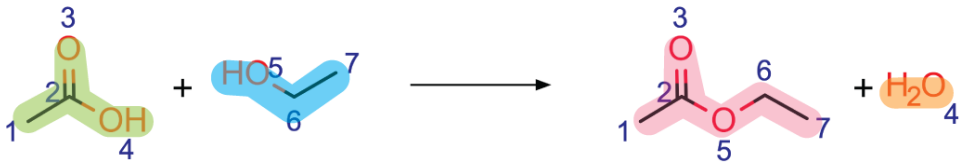
Generating Custom Rules

In the event that the default rules do not contain a reaction of interest, it is pososible to generate your own rules. Outlined below is the process to generate rules for esterification reactions, which consists of three parts

1. Writing the reaction SMARTS.
2. Writing the reaction rule.
3. Writing the coreactant list.

Writing Reaction SMARTS

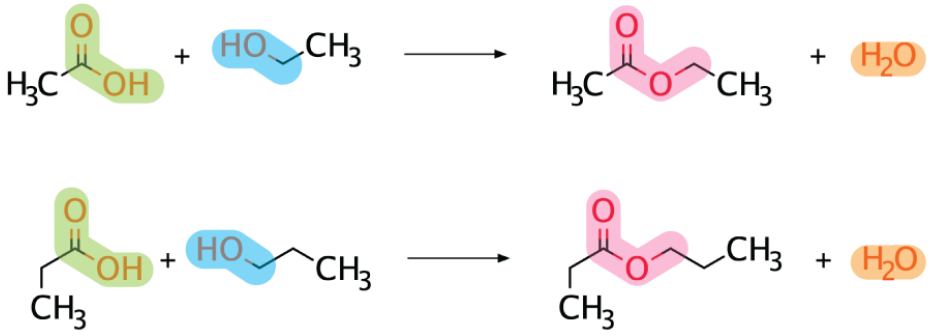
Rules are generated using SMARTS which represent reactions in a string. Importantly, these reaction rules specify atom mapping, which keeps track of the species throughout the reaction. To higlight a simple reaction rule generation, a deesterification reaction will be used.



[#6:1]-[#6:2](=[#8:3])-[#8:4].[#8:5]-[#6:6]-[#6:7]>>[#6:1]-[#6:2](=[#3])-[#8:5]-[#6:6]-[#6:7].[#8:4]

The reaction SMARTS is highighted the same color as the corresponding molecule in the reaction above. Ensuring correct atom mapping is important when writing these rules. This is an exact reaction rule and it matches the exact pattern of the reaction, which is not useful as it will not match many molecules.

Instead of using an exact reaction, a generic reaction rule can be used to match more molecules. In this case, the radius of the atom away from the reactive site is decreased.



[#6:2]-(=[#8:1])-[#8:3].[#8:4]-[#6:5]>>[#6:2]-(=[#8:1])-[#8:4]-[#6:5].[#8:3]

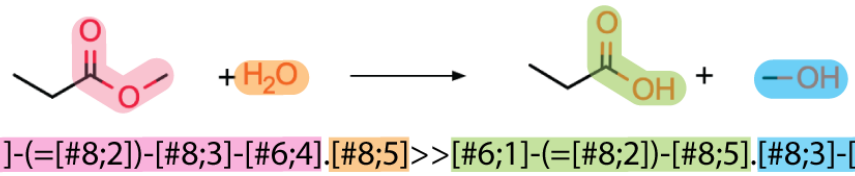
Writing Reaction Rules

With the reaction SMARTS written, now the whole rule for Pickaxe must be written. The rules are written as follows in a .tsv:

```
RULE_ID REACTANTS    RULE     PROODUCTS    NOTES
```

The rule_id is an arbitrary, unique value, the reactants and products specify how many compounds a rule should be expecting, and the rule is the reaction SMARTS. Notes can be provided, but have no effect on the running of Pickaxe. The reactants and products are specified as a generic compound, "Any", or as a predefined coreactant.

Below is an example of a reaction rule made for a deesterification reaction.



[#6;1]-(=[#8;2])-[#8;3]-[#6;4].[#8;5]>>[#6;1]-(=[#8;2])-[#8;5].[#8;3]-[#6;4]

```
RULE_ID REACTANTS    RULE     PROODUCTS    NOTES
rule1    Any;WATER      [#6:2]-(=[#8:1])-[#8:4]-[#6:5].[#8:3]>>[#6:2]-(=[#8:1])-[#8:3].[
↪#8:4]-[#6:5]     Any;Any
```

Note: Currently only one "Any" is allowed as a reactant and any other reactant must be defined as a coreactant.

Defining Coreactants

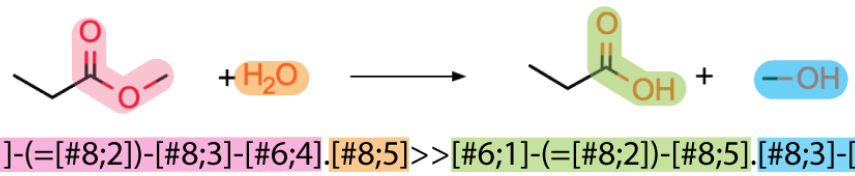
Coreactants are defined in their own file that the Pickaxe object will load and use alongside the reaction rules. The coreactant file for the example deesterification reaction is:

```
#ID Name     SMILES
WATER        WATER   O
```

Reaction Rule Example Summary

Summarized here is the input files for a deesterification reaction.

Reaction



[#6;1]-(=[#8;2])-[#8;3]-[#6;4].[#8;5]>>[#6;1]-(=[#8;2])-[#8;5].[#8;3]-[#6;4]

Reaction Rule Input

```
RULE_ID REACTANTS    RULE     PROODUCTS    NOTES
rule1    Any;WATER      [#6:2]-(=[#8:1])-[#8:4]-[#6:5].[#8:3]>>[#6:2]-(=[#8:1])-[#8:3].[
↪#8:4]-[#6:5]     Any;Any
```

Coreactant Input

```
#ID Name    SMILES
WATER       WATER   O
```

3.5 Custom Filters

3.5.1 Overview

Pickaxe expansions can grow extremely quickly in size, resulting in more compounds and reactions than a computer can efficiently handle, both during the expansion and during the subsequent analysis. An option to limit the chemical space explored during an expansion is to create a filter that selects only a subset of compounds to react in each generation. For example, you could create a filter that only allows compounds below a certain molecular weight or only compounds with specific structural features to react. A few filters have already been written by the Tyo lab, which you can find at *Built-In Filters*. We recommend looking at these as examples of how to write a filter as you write your own.

By creating and using a custom filter, you can control the scope of your expansion, allowing you to also expand for more generations. It also saves space in the database and should make downstream analysis faster.

3.5.2 Requirements

Creating a custom filter requires a working knowledge of python. Default filters are created using [RDKit](https://rdkit.org/docs/api-docs.html), a python library providing a collection of cheminformatic tools.

Ensure that that you have the [MINE-Database](https://github.com/tyo-nu/MINE-Database) github cloned on your machine.

The overall process for creating a filter is as follows:

1. Write custom Filter subclass in minedatabase/filters.py
2. Expose options for this filter subclass and add it to a pickaxe run in pickaxe_run.py
3. (optional) Write unit test(s) for this custom filter in tests/test_unit/test_filters.py

3.5.3 Writing Custom Filters

To write a custom filter, you need to subclass the Filter class in filters.py. The Filter class specifies the required functions your filter needs to implement as well as providing default methods that are inherited by your custom filter.

There are three methods you must implement (at a minimum). These are the __init__ method and the methods that are decorated with the @abc.abstractmethod decorator.

1. **__init__** - Initialize your filter's options and inputs here.
2. **filter_name** - This method just needs to return the filter name. This can be set to a constant string, set to a permanent self._filter_name (as in TanimotoSamplingFilter), or set to a custom self._filter_name (as in MetabolomicsFilter).
3. **_choose_cpds_to_filter** - This is the main method you need to implement, where you can loop through the compounds at each generation and decide which ones to keep and which ones to filter out. See the built-in filters' implementations of this method for examples. This method needs to return a set of compound IDs (e.g. "Ccffda1b2e82fcdb0e1e710cad4d5f70df7a5d74f") that you wish to **remove** from the expansion. Note that if a compound is a side-product of a reaction producing a *kept* compound, that compound will not be removed from the expansion, it just won't be reacted further.

There are two optional methods you may override as well. See the Filter ABC in filters.py (*Filters*) for more details.

1. **_pre_print** - This method prints to stdout just before the filter is applied. It should return None.
2. **_post_print** - This method prints to stdout just after the filter is applied. Useful for printing a summary of filtering results. It should return None.

3.5.4 Using Your Filter in a Pickaxe Run

Now that you have a filter defined, the next step is to import it and use it in a Pickaxe run. Refer to the example file, `pickaxe_run.py`, which is detailed more in *Running Pickaxe* to see an example of a Pickaxe run that uses filters. If you open pickaxe_run.py, you will notice different sections for the various built-in filters. Initialize your filter with any options that you have defined and then ensure you are appending your filter object to the pickaxe object.

You can find this in pickaxe_run.py by scrolling down to the comment that says "# Apply filters". The default filters all have an if statement associated with them when the filter is defined earlier in the file. Either replicate this format, or simply append your filter to the pickaxe object

```
pk.filters.append(my_filter)
```

That's it! Now, pickaxe will use that filter during any expansions.

If you have written tests for your filter and think it could be valuable to the community, feel free to submit a pull request at https://github.com/tyo-nu/MINE-Database to add your filter to the built-in set.

3.5.5 (Optional) Writing Tests for Your Filter

While it is not necessary, it is a good idea to write filters for your test to ensure the behavior of your tests don't change in the event of an update. There is an already existing file located at *tests/test_unit/test_filters.py* that you can add your tests to. We utilize [pytest](https://docs.pytest.org/en/stable/) and have defined useful fixtures for use in the tests. To run these tests run the following from the base MINE-Database directory

... codeblock:

```
pytest tests/test_unit/test_filters.py
```

3.6 Thermodynamic Calculations

3.6.1 Overview of Thermodynamics Module

eQuilibrator

Built into Pickaxe is the ability to estimate the Gibbs free energy of compounds and reactions. Pickaxe uses eQuilibrator to calculate thermodynamic values. More information about eQuilibrator can be found here.

Calculable Values

Pickaxe calculates the following values for compounds and reactions. More information about these conditions can be found here.

1. **Compounds**
    - **: fG'°: The Standard Gibbs Free Energy of Formation**
        a. Uses pH = 7 and ionic strength = 0.1M
2. **Reactions**
    - **rG'°: The Standard Gibbs Free Energy of Reaction**
        a. Uses pH = 7 and ionic strength = 0.1M
    - **rG'm: The Physiological Gibbs Free Energy of Reaction**
        a. Uses pH = 7 and ionic strength = 0.1M
        b. Assumes concentrations are 1mM.
    - **rG': Adjusted Gibbs Free Energy of Reaction**
        a. User-specified conditions

3.6.2 Calculating Thermodynamics of a Pickaxe Run

Set-up

Thermodynamics.py uses the compound ids (c_id) and reaction ids (r_id) of pickaxe runs to calculate values. This example assumes you have run a pickaxe run and have it accessible either from a MongoDB or in memory in a pickaxe object. Pickaxe runs can be stored later by using the pickleing functionality.

Additionally, an eQuilibrator database must be loaded.

Compound Value Calculations

If there is no eQuilibrator compounds.sqlite file present, generate one first.

```
>>> from equilibrator_assets.local_compound_cache import LocalCompoundCache
>>> lc = LocalCompoundCache()
>>> lc.generate_local_cache_from_default_zenodo("compounds.sqlite")
Copying default Zenodo compound cache to compounds.sqlite
```

Next, the thermodynamics class must be loaded and initialized, where *mongo_uri* is the uri to your mongo server. Providing None will use the default localhost.

```
>>> from minedatabase.thermodynamics import Thermodynamics
>>> thermo = Thermodynamics()
>>> thermo.load_thermo_from_sqlite("compounds.sqlite")
Loading compounds from compounds.sqlite
>>> thermo.load_mongo(mongo_uri=mongo_uri)
```

The following assumes you have a valid pickaxe object or a database to cross-reference the c_id and r_id from. No c_id or r_id is given here, but example outputs are.

Calculating Gf'°

```
>>> thermo.standard_dg_formation_from_cid(c_id=c_id, pickaxe=pickaxe, db_name=db_name)
-724.5684043965385
```

Calculating rG'm

```
>>> thermo.physiological_dg_prime_from_rid(r_id, pickaxe=pickaxe, db_name=db_name)
<Measurement(-5.432945798382008, 3.37496192184388, kilojoule / mole)>
```

Calculating rG' at pH = 4, ionic strength = 0.05M

```
>>> from equilibrator_api import Q_
>>> p_h = Q_("4")
>>> ionic_strength = Q_("0.05M)
>>> thermo.dg_prime_from_rid(r_id=r_id, db_name=db_name, p_h=p_h, ionic_strength=ionic_
↪strength)
<Measurement(11.68189173633911, 3.37496192184388, kilojoule / mole)>
```

3.7 API Reference

3.7.1 Compound I/O

Compound_io.py: Functions to load MINE databases from and dump compounds into common cheminformatics formats

minedatabase.compound_io.**export_inchi_rxns**(*mine_db: MINE*, *target: str*, *rxn_ids: Optional[List[str]] = None*) → None

Export reactions from a MINE db to a .tsv file.

Parameters

mine_db
[MINE] Name of MongoDB to export reactions from.

target
[str] Path to folder to save .tsv export file in.

rxn_ids
[Union[List[str], None], optional] Only export reactions with these ids, by default None.

minedatabase.compound_io.**export_kbase**(*mine_db: MINE*, *target: str*) → None

Exports MINE compound and reaction data as tab-separated values files amenable to use in ModelSEED.

Parameters

mine_db
[MINE] The database to export.

target
[str] Directory in which to place the files.

minedatabase.compound_io.**export_mol**(*mine_db: MINE*, *target: str*, *name_field: str = '_id'*) → None

Exports compounds from the database as a MDL molfiles

Parameters

mine_db
[MINE] MINE object that contains the database.

target
[str] Directory in which to place the files.

name_field
[str, optional] FIeld to provide names for the mol files. Must be unique and universal. By default, "_id".

minedatabase.compound_io.**export_sdf**(*mine_db: MINE*, *dir_path: str*, *max_compounds: Optional[int] = None*) → None

Exports compounds from the database as an MDL SDF file.

Parameters

mine_db
[MINE] MINE object that contains the database.

dir_path
[str] Directory for files.

max_compounds
[int, optional] Maximum number of compounds per file, by default None.

minedatabase.compound_io.**export_smiles**(*mine_db: MINE*, *dir_path: str*, *max_compounds: Optional[int] = None*) → None

Exports compounds from the database as a SMILES file.

Parameters

mine_db
[MINE] MINE object that contains the database.

dir_path
[str] Directory for files.

max_compounds
[int, optional] Maximum number of compounds per file, by default None.

minedatabase.compound_io.**export_tsv**(*mine_db: MINE*, *target: str*, *compound_fields: Tuple[str] = ('_id', 'Names', 'Model_SEED', 'Formula', 'Charge', 'Inchi')*, *reaction_fields: Tuple[str] = ('_id', 'SMILES_rxn', 'C_id_rxn')*) → None

Exports MINE compound and reaction data as tab-separated values files amenable to use in ModelSEED.

Parameters

mine_db
[MINE] The database to export.

target
[str] Directory, in which to place the files.

compound_fields
[Tuple[str], optional] Fields to export in the compound table, by default ('_id', 'Names', 'Model_SEED', 'Formula', 'Charge', 'Inchi').

reaction_fields
[Tuple[str], optional] Fields to export in the reaction table, by default ('_id', 'SMILES_rxn', 'C_id_rxn').

minedatabase.compound_io.**import_mol_dir**(*mine_db: MINE*, *target: str*, *name_field: str = 'Name'*, *overwrite: bool = False*) → None

Imports a directory of molfiles as a MINE database.

Parameters

mine_db
[MINE] The database to export.

target
[str] Directory in which to place the files.

name_field
[str, optional] Field for the compound name, by default "Name".

overwrite
[bool, optional] Replace old compounds with new ones if a collision happens, by default False.

`minedatabase.compound_io.`**`import_sdf`**(*mine_db:* MINE, *target: str*) → None

Imports a SDF file as a MINE database.

Parameters

mine_db
[MINE] The database to export.

target
[str] Directory in which to place the files.

`minedatabase.compound_io.`**`import_smiles`**(*mine_db:* MINE, *target: str*) → None

Imports a smiles file as a MINE database.

Parameters

mine_db
[MINE] The database to export.

target
[str] Directory in which to place the files.

3.7.2 Databases

Databases.py: This file contains MINE database classes including database loading and writing functions.

`class` `minedatabase.databases.`**`MINE`**(*name: str*, *uri: str = 'mongodb://localhost:27017/'*)

This class provides an interface to the MongoDB and some useful functions.

Parameters

name
[str] Name of the database to work with.

uri
[str, optional] uri of the mongo server, by default "mongodb://localhost:27017/".

Attributes

client
[pymongo.MongoClient] client connection to the MongoDB.

compounds
[Collection] Compounds collection.

core_compounds
[Collection] Core compounds collection.

meta_data
[Collection] Metadata collection.

models
[Collection] Models collection.

name
[str] Name of the database

operators
[Collection] Operators collection.

reactions
[Collection] Reactions collection.

target_compounds
[Collection] Target compounds collection.

uri
[str] MongoDB connection string.

`add_reaction_mass_change`(*reaction: Optional[str] = None*) → Optional[float]

Calculate the change in mass between reactant and product compounds.

This is useful for discovering compounds in molecular networking. If no reaction is specified then mass change of each reaction in the database will be calculated.

Parameters

reaction
[str, optional] Reaction ID to calculate the mass change for, by default None.

Returns

float, optional
Mass change of specified reaction. None if masses not all found.

`build_indexes`() → None

Build indexes for efficient querying of the database.

`generate_image_files`(*path: str*, *query: Optional[dict] = None*, *dir_depth: int = 0*, *img_type: str = 'svg:-a,nosource,w500,h500'*, *convert_r: bool = False*) → None

Generates image files for compounds in database using ChemAxon's MolConvert.

Parameters

path
[str] Target directory for image file.

query
[dict, optional] Query to limit number of files generated, by default None.

dir_depth
[int, optional] The number of directory levels to split the compounds into for files system efficiency. Ranges from 0 (all in top level directory) to the length of the file name (40 for MINE hashes), by default 0.

img_type
[str, optional] Type of image file to be generated. See molconvert documentation for valid options, by default 'svg:-a,nosource,w500,h500'.

convert_r
[bool, optional] Convert R in the smiles to *, by default False.

minedatabase.databases.**establish_db_client**(*uri: Optional[str] = None*) → MongoClient

Establish a connection to a mongo database given a URI.

Uses the provided URI to connect to a mongoDB. If none is given the default URI is used when using pymongo.

Parameters

uri
[str, optional] URI to connect to mongo DB, by default None.

Returns

pymongo.MongoClient
Connection to the specified mongo instance.

Raises

IOError
Attempt to connect to database timed out.

minedatabase.databases.**write_compounds_to_mine**(*compounds: List[dict]*, *db:* MINE, *chunk_size: int = 10000*, *processes: int = 1*) → None

Write compounds to reaction collection of MINE.

Parameters

compounds
[List[dict]] Dictionary of compounds to write.

db
[MINE] MINE object to write compounds with.

chunk_size
[int, optional] Size of chunks to break compounds into when writing, by default 10000.

processes
[int, optional] Number of processors to use, by default 1.

minedatabase.databases.**write_core_compounds**(*compounds: List[dict]*, *db:* MINE, *mine: str*, *chunk_size: int = 10000*, *processes=1*) → None

Write core compounds to the core compound database.

Calculates and formats compounds into appropriate form to insert into the core compound database in the mongo instance. Core compounds are attempted to be inserted and collisions are detected on the database. The list of MINEs a given compound is found in is updated as well.

Parameters

compounds
[dict] List of compound dictionaries to write.

db
[MINE] MINE object to write core compounds with.

mine
[str] Name of the MINE.

chunk_size
[int, optional] Size of chunks to break compounds into when writing, by default 10000.

processes
[int, optional] The number of processors to use, by default 1.

minedatabase.databases.**write_reactions_to_mine**(*reactions: List[dict]*, *db:* MINE, *chunk_size: int = 10000*) → None

Write reactions to reaction collection of MINE.

Parameters

reactions
[List[dict]] Dictionary of reactions to write.

db
[MINE] MINE object to write reactions with.

chunk_size
[int, optional] Size of chunks to break reactions into when writing, by default 10000.

minedatabase.databases.**write_targets_to_mine**(*targets: List[dict]*, *db:* MINE, *chunk_size: int = 10000*) → None

Write target compounds to target collection of MINE.

Parameters

targets
[List[dict]] Listt of target dictionaries to write.

db
[MINE] MINE object to write targets with.

chunk_size
[int, optional] Size of chunks to break compounds into when writing, by default 10000.

3.7.3 Filters

3.7.4 Metabolomics

Provides functionality to interact with metabolomics datasets and search MINE databases for metabolomics hits.

class minedatabase.metabolomics.**MetabolomicsDataset**(*name: str*, *adducts: Optional[List[str]] = None*, *known_peaks: Optional[List[*Peak*]] = None*, *unknown_peaks: Optional[List[*Peak*]] = None*, *native_set: Set[str] = {}*, *ppm: bool = False*, *tolerance: float = 0.001*, *halogens: bool = False*, *verbose: bool = False*)

A class containing all the information for a metabolomics data set.

annotate_peaks(*db:* MINE, *core_db:* MINE) → None

This function iterates through the unknown peaks in the dataset and searches the database for compounds that match a peak m/z given the adducts permitted. Statistics on the annotated data set are printed.

Parameters

db
[MINE] MINE database.

core_db
[MINE] Core database containing spectra info.

check_product_of_native(*cpd_ids: List[str]*, *db:* MINE) → List[str]

Filters list of compound IDs to just those associated with compounds produced from a native hit in the model (i.e. in native set).

enumerate_possible_masses(*tolerance: float*) → None

Generate all possible masses from unknown peaks and list of adducts. Saves these mass ranges to self.possible_ranges.

Parameters

tolerance
[float] Mass tolerance in Daltons.

find_db_hits(*peak: Peak*, *db: MINE*, *core_db: MINE*, *adducts: List[Tuple[str, float, float]]*) → None

This function searches the database for matches of a peak given adducts and updates the peak object with that information.

Parameters

peak
[Peak] Peak object to query against MINE compound database.

db
[MINE] MINE database to query.

adducts
[List[Tuple[str, float, float]]] List of adducts. Each adduct contains three values in a tuple: (adduct name, mass multiplier, ion mass).

get_rt(*peak_id: str*) → Optional[float]

Return retention time for peak with given ID. If not found, returns None.

Parameters

peak_id
[str] ID of peak as listed in dataset.

Returns

rt
[float, optional] Retention time of peak with given ID, None if not found.

class minedatabase.metabolomics.**Peak**(*name: str*, *r_time: float*, *mz: float*, *charge: str*, *inchi_key: str = None*, *ms2: List[float, float] = None*)

Peak object which contains peak metadata as well as mass, retention time, spectra, and any MINE database hits.

Parameters

name
[str] Name or ID of the peak.

r_time
[float] Retention time of the peak.

mz
[float] Mass-to-charge ratio (m/z) of the peak.

charge
[str] Charge of the peak, "+" or "-".

inchi_key
[str, optional] InChI key of the peak, if already identified, by default None.

ms2
[List[float], optional] MS2 spectra m/z values for this peak, by default None.

Attributes

isomers
[List[Dict]] List of compound documents in JSON (dict) format.

formulas
[Set[str]] All the unique compound formulas from compounds found for this peak.

total_hits
[int] Number of compound hits for this peak.

native_hit
[bool] Whether this peak matches a compound provided in the native set.

`score_isomers`(*metric: ~typing.Callable[[list, list], float] = <function dot_product>, energy_level: int = 20, tolerance: float = 0.005*) → None

Scores and sorts isomers based on mass spectra data.

Calculates the cosign similarity score between the provided ms2 peak list and pre-calculated CFM-spectra and sorts the isomer list according to this metric.

Parameters

metric
[function, optional] The scoring metric to use for the spectra. Function must accept 2 lists of (mz, intensity) tuples and return a score, by default dot_product.

energy_level
[int, optional] The Fragmentation energy level to use. May be 10, 20 or 40., by default 20.

tolerance
[float, optional] The precision to use for matching m/z in mDa, by default 0.005.

Raises

ValueError
Empty ms2 peak.

`class` minedatabase.metabolomics.`Struct`(***entries*)

convert key-value pairs into object-attribute pairs.

minedatabase.metabolomics.`dot_product`(*x: List[tuple], y: List[tuple], epsilon: float = 0.01*) → float

Calculate the dot product of two spectra, allowing for some variability in mass-to-charge ratios

Parameters

x
[List[tuple]] First spectra m/z values.

y
[List[tuple]] Second spectra m/z values.

epsilon
[float, optional] Mass tolerance in Daltons, by default 0.01.

Returns

dot_prod
[float] Dot product of x and y.

minedatabase.metabolomics.`get_KEGG_comps`(*db:* MINE, *core_db:* MINE, *kegg_db: Database*, *model_ids: List[str]*) → set

Get MINE IDs from KEGG MINE database for compounds in model(s).

Parameters

db
[MINE] MINE Mongo database.

kegg_db
[pymongo.database.Database] Mongo database with annotated organism metabolomes from KEGG.

model_ids
[List[str]] List of organism identifiers from KEGG.

Returns

set
MINE IDs of compounds that are linked to a KEGG ID in at least one of the organisms in model_ids.

`minedatabase.metabolomics.`**`jaccard`**(*x: List[tuple]*, *y: List[tuple]*, *epsilon: float = 0.01*) → float

Calculate the Jaccard Index of two spectra, allowing for some variability in mass-to-charge ratios

Parameters

x
[List[tuple]] First spectra m/z values.

y
[List[tuple]] Second spectra m/z values.

epsilon
[float, optional] Mass tolerance in Daltons, by default 0.01.

Returns

jaccard_index
[float] Jaccard Index of x and y.

`minedatabase.metabolomics.`**`ms2_search`**(*db:* MINE, *core_db:* MINE, *keggdb: Database*, *text: str*, *text_type: str*, *ms_params*) → List

Search for compounds matching MS2 spectra.

Parameters

db
[MINE] Contains compound documents to search.

core_db
[MINE] Contains extra info (including spectra) for compounds in db.

keggdb
[pymongo.database.Database] Contains models with associated compound documents.

text
[str] Text as in metabolomics datafile for specific peak.

text_type
[str] Type of metabolomics datafile (mgf, mzXML, and msp are supported). If text, assumes m/z values are separated by newlines (and set text_type to "form").

ms_params
[dict]

"tolerance": float specifying tolerance for m/z, in mDa by default.
Can specify in ppm if "ppm" key's value is set to True.

"charge": bool (1 for positive, 0 for negative). "energy_level": int specifying fragmentation energy level to use. May

be 10, 20, or 40.

"scoring_function": str describing which scoring function to use. Can
be either "jaccard" or "dot product".

"adducts": list of adducts to use. If not specified, uses all adducts. "models": List of model _ids. If supplied, score compounds higher if

present in model.

"ppm": bool specifying whether "tolerance" is in mDa or ppm. Default
value for ppm is False (so tolerance is in mDa by default).

"kovats": length 2 tuple specifying min and max kovats retention index
to filter compounds (e.g. (500, 1000)).

"logp": length 2 tuple specifying min and max logp to filter compounds
(e.g. (-1, 2)).

"halogens": bool specifying whether to filter out compounds containing
F, Cl, or Br. Filtered out if set to True. False by default.

Returns

ms_adduct_output
[list] Compound JSON documents matching ms2 search query.

minedatabase.metabolomics.**ms_adduct_search**(*db:* MINE, *core_db:* MINE, *keggdb: Database*, *text: str*, *text_type: str*, *ms_params*) → List

Search for compound-adducts matching precursor mass.

Parameters

db
[MINE] Contains compound documents to search.

core_db
[MINE] Contains extra info (including spectra) for compounds in db.

keggdb
[pymongo.database.Database] Contains models with associated compound documents.

text
[str] Text as in metabolomics datafile for specific peak.

text_type
[str] Type of metabolomics datafile (mgf, mzXML, and msp are supported). If text, assumes m/z values are separated by newlines (and set text_type to "form").

ms_params
[dict]

"tolerance": float specifying tolerance for m/z, in mDa by default.
Can specify in ppm if "ppm" key's value is set to True.

"adducts": list of adducts to use. If not specified, uses all adducts. "models": List of model _ids. If supplied, score compounds higher if

present in model. ["eco"] by default (E. coli).

"ppm": bool specifying whether "tolerance" is in mDa or ppm. Default value for ppm is False (so tolerance is in mDa by default).

"kovats": length 2 tuple specifying min and max kovats retention index to filter compounds (e.g. (500, 1000)).

"logp": length 2 tuple specifying min and max logp to filter compounds (e.g. (-1, 2)).

"halogens": bool specifying whether to filter out compounds containing F, Cl, or Br. Filtered out if set to True. False by default.

Returns

ms_adduct_output
[list] Compound JSON documents matching ms adduct query.

`minedatabase.metabolomics.`**`read_adduct_names`**(*filepath: str*) → List[str]

Read adduct names from text file at specified path into a list.

Parameters

filepath
[str] Path to adduct text file.

Returns

adducts
[list] Names of adducts in text file.

Notes

Not used in this codebase but used by MINE-Server to validate adduct input.

`minedatabase.metabolomics.`**`read_mgf`**(*input_string: str*, *charge: bool*, *ms2_delim='\t'*) → List[*Peak*]

Parse mgf metabolomics data file.

Parameters

input_string
[str] Metabolomics input data file.

charge
[bool] True if positive, False if negative.

ms2_delim
[str] Delimiter for whitespace between intensity and m/z value. Usually tab but can also be a space in some MGF files. Tab by default.

Returns

peaks
[List[Peak]] A list of Peak objects.

`minedatabase.metabolomics.`**`read_msp`**(*input_string: str*, *charge: bool*) → List[*Peak*]

Parse msp metabolomics data file.

Parameters

input_string
[str] Metabolomics input data file.

charge
[bool] True if positive, False if negative.

Returns

peaks
[List[Peak]] A list of Peak objects.

minedatabase.metabolomics.**read_mzxml**(*input_string: str*, *charge: bool*) → List[*Peak*]

Parse mzXML metabolomics data file.

Parameters

input_string
[str] Metabolomics input data file.

charge
[bool] True if positive, False if negative.

Returns

List[Peak]
A list of Peak objects.

minedatabase.metabolomics.**score_compounds**(*compounds: list*, *model_id: str = None*, *core_db: pymongo.database = None*, *mine_db: pymongo.database = None*, *kegg_db: pymongo.database = None*, *parent_frac: float = 0.75*, *reaction_frac: float = 0.25*, *get_native: bool = False*) → List[dict]

This function validates compounds against a metabolic model, returning only the compounds which pass.

Parameters

db
[Mongo DB] Should contain a "models" collection with compound and reaction IDs listed.

core_db
[Mongo DB] Core MINE database.

compounds
[list] Each element is a dict describing that compound. Should have an '_id' field.

model_id
[str] KEGG organism code (e.g. 'hsa').

parent_frac
[float, optional] Weighting for compounds derived from compounds in the provided model. 0.75 by default.

reaction_frac
[float, optional] Weighting for compounds derived from known compounds not in the model. 0.25 by default.

Returns

compounds
[List[dict]] Modified version of input compounds list, where each compound now has a 'Likelihood_score' key and value between 0 and 1.

minedatabase.metabolomics.**spectra_download**(*db: MINE*, *mongo_id: Optional[str] = None*) → str

Download one or more spectra for compounds matching a given query.

Parameters

db
[MINE] Contains compound documents to search.

mongo_query
[str, optional (default: None)] A valid Mongo query as a literal string. If None, all compound spectra are returned.

parent_filter
[str, optional (default: None)] If set to a metabolic model's Mongo _id, only get spectra for compounds in or derived from that metabolic model.

putative
[bool, optional (default: True)] If False, only find known compounds (i.e. in Generation 0). Otherwise, finds both known and predicted compounds.

Returns

spectral_library
[str] Text of all matching spectra, including headers and peak lists.

3.7.5 Pickaxe

Pickaxe.py: Create network expansions from reaction rules and compounds.

This module generates new compounds from user-specified starting compounds using a set of SMARTS-based reaction rules.

class minedatabase.pickaxe.**Pickaxe**(*rule_list: Optional[str] = None*, *coreactant_list: Optional[str] = None*, *explicit_h: bool = False*, *kekulize: bool = False*, *neutralise: bool = True*, *errors: bool = True*, *inchikey_blocks_for_cid: int = 1*, *database: Optional[str] = None*, *database_overwrite: bool = False*, *mongo_uri: bool = 'mongodb://localhost:27017'*, *image_dir: Optional[str] = None*, *quiet: bool = True*, *react_targets: bool = True*, *filter_after_final_gen: bool = True*, *prune_between_gens: bool = False*)

Class to generate expansions with compounds and reaction rules.

This class generates new compounds from user-specified starting compounds using a set of SMARTS-based reaction rules. It may be initialized with a text file containing the reaction rules and coreactants or this may be done on an ad hoc basis.

Parameters

rule_list
[str] Filepath of rules.

coreactant_list
[str] Filepath of coreactants.

explicit_h
[bool, optional] Whether rules utilize explicit hydrogens, by default True.

kekulize
[bool, optional] Whether or not to kekulize compounds before reaction, by default False.

neutralise
[bool, optional] Whether or not to neutralise compounds, by default True.

errors
[bool, optional] Whether or not to print errors to stdout, by default True.

inchikey_blocks_for_cid
[int, optional] How many blocks of the InChI key to use for the compound id, by default 1.

database
[str, optional] Name of the database where to save results, by default None.

database_overwrite
[bool, optional] Whether or not to erase existing database in event of a collision, by default False.

mongo_uri
[bool, optional] uri for the mongo client, by default 'mongodb://localhost:27017'.

image_dir
[str, optional] Filepath where images should be saved, by default None.

quiet
[bool, optional] Whether to silence warnings, by default False.

react_targets
[bool, optional] Whether or not to apply reactions to generated compounds that match targets, by default True.

filter_after_final_gen
[bool, optional] Whether to apply filters after final expansion, by default True.

prune_between_gens
[bool, optional] Whether to prune network between generations if using filters

Attributes

operators: dict
Reaction operators to transform compounds with.

coreactants: dict
Coreactants required by the operators.

compounds: dict
Compounds in the pickaxe network.

reactions: dict
Reactions in the pickaxe network.

generation: int
The current generation

explicit_h
[bool] Whether rules utilize explicit hydrogens.

kekulize
[bool] Whether or not to kekulize compounds before reaction.

neutralise
[bool] Whether or not to neutralise compounds.

fragmented_mols
[bool] Whether or not to allow fragmented molecules.

radical_check
[bool] Whether or not to check and remove radicals.

image_dir
[str, optional] Filepath where images should be saved.

errors
[bool] Whether or not to print errors to stdout.

quiet
[bool] Whether or not to silence warnings.

filters: List[object]
A list of filters to apply during the expansion.

targets
[dict] Molecules to be targeted during expansions.

target_smiles: List[str]
The SMILES of all the targets.

react_targets
[bool] Whether or not to react targets when generated.

filter_after_final_gen
[bool] Whether or not to filter after the last expansion.

prune_between_gens
[bool, optional] Whether to prune network between generations if using filters.

mongo_uri
[str] The connection string to the mongo database.

cid_num_inchi_blocks
[int] How many blocks of the inchi-blocks to use to generate the compound id.

`assign_ids()` → None

Assign a numerical ID to compounds (and reactions).

Assign IDs that are unique only to the CURRENT run.

`find_minimal_set(`*white_list: Set[str]*`)` → Tuple[set, set]

Find the minimal set of compounds and reactions given a white list.

Given a whitelist this function finds the minimal set of compound and reactions ids that comprise the set.

Parameters

white_list
[Set[str]] List of compound_ids to use to filter reaction network to.

Returns

Tuple[set, set]
The filtered compounds and reactions.

`load_compound_set(`*compound_file: Optional[str] = None*, *id_field: str = 'id'*`)` → str

Load compounds for expansion into pickaxe.

Parameters

compound_file
[str, optional] Filepath of compounds, by default None.

id_field
[str, optional] Header value of compound id in input file, by default 'id'.

Returns

str
List of SMILES that were succesfully loaded into pickaxe.

Raises

ValueError
No file specified for loading.

`load_pickled_pickaxe`(*fname: str*) → None

Load pickaxe from pickle.

Load pickled pickaxe object.

Parameters

fname
[str] filename to read (must be .pk).

`load_targets`(*target_compound_file: Optional[str]*, *id_field: str = 'id'*) → None

Load targets into pickaxe.

Parameters

target_compound_file
[str] Filepath of target compounds.

id_field
[str, optional] Header value of compound id in input file, by default 'id'.

`pickle_pickaxe`(*fname: str*) → None

Pickle key pickaxe items.

Pickle pickaxe object to be loaded in later.

Parameters

fname
[str] filename to save (must be .pk).

`prune_network`(*white_list: list*, *print_output: str = True*) → None

Prune the reaction network to a list of targets.

Prune the predicted reaction network to only compounds and reactions that terminate in a specified white list of compounds.

Parameters

white_list
[list] A list of compound ids to filter the network to.

print_output
[bool] Whether or not to print output

`prune_network_to_targets`() → None

Prune the reaction network to the target compounds.

Prune the predicted reaction network to only compounds and reactions that terminate in the target compounds.

`save_to_SBML`(*file_name: str*, *save_reactions_uniprot: bool = False*, *uniprot_save_style: str = 'grouped'*) → None

Save pickaxe run to an SBML file.

This function saves the species and reactions to an SBML file with annotations. Specifically, the species will be annotated with their SMILES and reactions annotated with their operator and uniprot ids (if available and desired).

Parameters

file_name
[str] The file name to save the SBML at.

save_reactions_uniprot
[bool] Whether or not to save the uniprot ids for a reaction operator (if available)

uniprot_save_style
[str]

The stle to save uniprot ids in. There are two options:
grouped : all uniprot information is stored in a semicolon delimited list individual : uniprot information is saved individually as a link to the uniprot website

`save_to_mine`(*processes: int = 1*, *indexing: bool = True*, *write_core: bool = False*) → None

Save pickaxe run to MINE database.

Parameters

processes
[int, optional] Number of processes to use, by default 1.

indexing
[bool, optional] Whether or not to add indexes, by default True.

write_core
[bool, optional] Whether or not to write to core database, by default False.

`transform_all`(*processes: int = 1*, *generations: int = 1*) → None

Transform compounds with reaction operators.

Apply reaction rules to compounds and generate a specified number of new generations.

Parameters

processes
[int, optional] Number of processes to run in parallel, by default 1.

generations
[int, optional] Number of generations to create, by default 1.

`write_compound_output_file`(*path: str*, *dialect: str = 'excel-tab'*) → None

Write compounds to an output file.

Parameters

path
[str] Path to write data.

dialect
[str, optional] Dialect of the output, by default 'excel-tab'.

`write_reaction_output_file`(*path: str*, *delimiter: str = '\t'*) → None

Write all reaction data to the specified path.

Parameters

path
[str] Path to write data.

delimiter
[str, optional] Delimiter for the output file, by default 't'.

3.7.6 Reactions

Reaction.py: Methods to execute reactions.

minedatabase.reactions.**transform_all_compounds_with_full**(*compound_smiles: list*, *coreactants: dict*, *coreactant_dict: dict*, *operators: dict*, *generation: int*, *explicit_h: bool*, *kekulize: bool*, *processes: int*) → Tuple[dict, dict]

Transform compounds given a list of rules.

Carry out the transformation of a list of compounds given operators. Generates new products and returns them to be processed by pickaxe.

Parameters

compound_smiles
[list] List of SMILES to react.

coreactants
[dict] Dictionary of correactants RDKit Mols defined in rules.

coreactant_dict
[dict] Dictionary of correactant compoudnds defined in rules.

operators
[dict] Dictionary of reaction rules.

generation
[int] Value of generation to expand.

explicit_h
[bool] Whether or not to have explicit Hs in reactions.

kekulize
[bool] Whether or not to kekulize compounds.

processes
[int] Number of processors being used.

Returns

Tuple[dict, dict]
Returns a tuple of New Compounds and New Reactants.

3.7.7 Rules

Generate rules to use in pickaxe runs.

minedatabase.rules.**BNICE**() → Tuple[Path, Path, str]

Generate BNICE rules.

Generate the original BNICE rules that were use before the improved MetaCyc rules were generated.

Returns

Tuple[Path, Path, str]
The path to the rules and coreactants and the rule name.

minedatabase.rules.**metacyc_generalized**(*n_rules: Optional[int] = None*, *fraction_coverage: Optional[float] = None*, *anaerobic: float = False*, *include_containing: Optional[List[str]] = None*, *exclude_containing: Optional[List[str]] = None*, ***kwargs*) → Tuple[StringIO, StringIO, str]

Generate generalize metacyc rule subsets.

Generate subsets of the metacyc generalized reaction opreators by specifying the number of rules of the fraction coverage of metacyc desired. Rules are chosen in the order of rules that map the most reactions to least. For fractional coverage the lowest number of rules that give a coverage less than or equal to the specified coverage is given.

Specific groups can be specified to be excluded or used as well to prune to specific rules. This is a two step process:

1) Select rules by include_containing. If none specified, use all rules.
2) Remove rules by excluded_containing.

Parameters

n_rules
[int, optional] Number of rules to use. If excluded rules result in less than specified number then all rules are taken, by default None.

fraction_coverage
[float, optional] The fraction of coverage desired. This may be impossible to reach depending on which rules are excluded, by default None.

anaerobic: float, optional
Whether to remove oxygen requiring reactions.

include_containing: List[str], optional

A list containing features to include. Valid features are:

- aromatic
- aromatic_oxygen
- carbonyl
- halogen
- nitrogen
- oxygen
- phosphorus
- sulfur
- fluorine
- chlorine
- bromine
- iodine

sending None gives all groups, by default None.

exclude_containing: List[str], optional

A list containing features to exclude.

- aromatic
- aromatic_oxygen
- carbonyl
- halogen
- nitrogen
- oxygen
- phosphorus
- sulfur
- fluorine
- chlorine
- bromine
- iodine

By default None.

Returns

Tuple[StringIO, StringIO, str]
A tuple containing two streams (reaction rules and cofactor) and the rule name.

minedatabase.rules.**metacyc_generalized_as_df**(*n_rules: Optional[int] = None*, *fraction_coverage: Optional[float] = None*, *anaerobic: float = False*, *include_containing: Optional[List[str]] = None*, *exclude_containing: Optional[List[str]] = None*) → DataFrame

Generate generalized metacyc rule subsets as Pandas Dataframe

Generate subsets of the metacyc intermediate reaction opreators by specifying the number of rules of the fraction coverage of metacyc desired. Coverage and number of rules are taken from the generalized operators and their intermediate operators are chosen.

Parameters

n_rules
[int, optional] Number of rules to use, by default None.

fraction_coverage
[float, optional] The fraction of coverage desired, by default None.

anaerobic: float, optional
Whether to remove oxygen requiring reactions.

include_containing: List[str], optional

A list containing features to include. Valid features are:

- aromatic
- aromatic_oxygen
- carbonyl
- halogen
- nitrogen

- oxygen
- phosphorus
- sulfur
- fluorine
- chlorine
- bromine
- iodine

sending None gives all groups, by default None.

exclude_containing: List[str], optional

A list containing features to exclude. Valid features are:

- aromatic
- aromatic_oxygen
- carbonyl
- halogen
- nitrogen
- oxygen
- phosphorus
- sulfur
- fluorine
- chlorine
- bromine
- iodine

by default None

Returns

pd.DataFrame
A Pandas DataFrame containig the ruleset.

minedatabase.rules.**metacyc_intermediate**(*n_rules: Optional[int] = None*, *fraction_coverage: Optional[float] = None*, *anaerobic: float = False*, *include_containing: Optional[List[str]] = None*, *exclude_containing: Optional[List[str]] = None*) → Tuple[StringIO, StringIO, str]

Generate intermediate metacyc rule subsets.

Generate subsets of the metacyc intermediate reaction opreators by specifying the number of rules of the fraction coverage of metacyc desired. Coverage and number of rules are taken from the generalized operators and their intermediate operators are chosen.

Parameters

n_rules
[int, optional] Number of rules to use, by default None.

fraction_coverage
[float, optional] The fraction of coverage desired, by default None.

anaerobic: float, optional
Whether to remove oxygen requiring reactions.

include_containing: List[str], optional

A list containing features to include. Valid features are:

- aromatic
- aromatic_oxygen
- carbonyl
- halogen
- nitrogen
- oxygen
- phosphorus
- sulfur
- fluorine
- chlorine
- bromine
- iodine

sending None gives all groups, by default None.

exclude_containing: List[str], optional

A list containing features to exclude. Valid features are:

- aromatic
- aromatic_oxygen
- carbonyl
- halogen
- nitrogen
- oxygen
- phosphorus
- sulfur
- fluorine
- chlorine
- bromine
- iodine

by default None

Returns

Tuple[StringIO, StringIO, str]
A tuple containing two streams that contain the reaction rule information and the rule name.

3.7.8 Thermodynamics

3.7.9 Utilities

Utils.py: contains basic functions reused in various contexts in other modules

class minedatabase.utils.**Chunks**(*it: Iterable*, *chunk_size: int = 1*, *return_list: bool = False*)

A class to chunk an iterator up into defined sizes.

next() → Union[List[chain], chain]

Returns the next chunk from the iterable. This method is not thread-safe.

Returns

next_slice
[Union[List[chain], chain]] Next chunk.

class minedatabase.utils.**StoichTuple**(*stoich*, *c_id*)

property c_id

Alias for field number 1

property stoich

Alias for field number 0

minedatabase.utils.**convert_sets_to_lists**(*obj: dict*) → dict

Recursively converts dictionaries that contain sets to lists.

Parameters

obj
[dict] Input object to convert sets from.

Returns

dict
dictionary with no sets.

minedatabase.utils.**file_to_dict_list**(*filepath: str*) → list

Accept a path to a CSV, TSV or JSON file and return a dictionary list.

Parameters

filepath
[str] File to load into a dictionary list.

Returns

list
Dictionary list.

minedatabase.utils.**get_atom_count**(*mol: rdkit.Chem.rdchem.Mol*, *radical_check: bool = False*) → Counter

Takes a mol object and returns a counter with each element type in the set.

Parameters

mol
[rdkit.Chem.rdchem.Mol] Mol object to count atoms for.

radical_check
[bool, optional] Check for radical electrons and count if present.

Returns

atoms
[collections.Counter] Count of each atom type in input molecule.

minedatabase.utils.**get_compound_hash**(*smi: str*, *cpd_type: str = 'Predicted'*, *inchi_blocks: int = 1*) → Tuple[str, Optional[str]]

Create a hash string for a given compound.

This function generates an unique identifier for a compound, ensuring a normalized SMILES. The compound hash is generated by sanitizing and neutralizing the SMILES and then generating a hash from the sha1 method in the haslib.

The hash is prepended with a character depending on the type. Default value is "C":

1. Coreactant: "X"
2. Target Compound: "T"
3. Predicted Compound: "C"

Parameters

smi
[str] The SMILES of the compound.

cpd_type
[str, optional] The Compound Type, by default 'Predicted'.

Returns

Tuple[str, Union[str, None]]
Compound hash, InChI-Key.

minedatabase.utils.**get_dotted_field**(*input_dict: dict*, *accessor_string: str*) → dict

Gets data from a dictionary using a dotted accessor-string.

Parameters

input_dict
[dict] A nested dictionary.

accessor_string
[str] The value in the nested dict.

Returns

dict
Data from the dictionary.

minedatabase.utils.**get_fp**(*smi: str*) → rdkit.Chem.AllChem.RDKFingerprint

Generate default RDKFingerprint.

Parameters

smi
[str] SMILES of the molecule.

Returns

AllChem.RDKFingerprint
Default fingerprint of the molecule.

minedatabase.utils.**get_reaction_hash**(*reactants: List[StoichTuple]*, *products: List[StoichTuple]*) → Tuple[str, str]

Hashes reactant and product lists.

Generates a unique ID for a given reaction for use in MongoDB.

Parameters

reactants
[List[StoichTuple]] List of reactants.

products
[List[StoichTuple]] List of products.

Returns

Tuple[str, str]
Reaction hash and SMILES.

minedatabase.utils.**get_size**(*obj_0*)

Recursively iterate to sum size of object & members.

minedatabase.utils.**mongo_ids_to_mine_ids**(*mongo_ids: List[str]*, *core_db*) → int

Convert mongo ID to a MINE ID for a given compound.

Parameters

mongo_id
[List[str]] List of IDs in Mongo (hashes).

core_db
[MINE] Core database connection. Type annotation not present to avoid circular imports.

Returns

mine_id
[int] MINE ID.

minedatabase.utils.**neutralise_charges**(*mol: rdkit.Chem.rdchem.Mol*, *reactions=None*) → rdkit.Chem.rdchem.Mol

Neutralize all charges in an rdkit mol.

Parameters

mol
[rdkit.Chem.rdchem.Mol] Molecule to neutralize.

reactions
[list, optional] patterns to neutralize, by default None.

Returns

mol
[rdkit.Chem.rdchem.Mol] Neutralized molecule.

minedatabase.utils.**postsanitize_smiles**(*smiles_list*)

Postsanitize smiles after running SMARTS. :returns tautomer list of list of smiles

minedatabase.utils.**prevent_overwrite**(*write_path: str*) → str

Prevents overwrite of existing output files by appending "_new" when needed.

Parameters

write_path
[str] Path to write.

Returns

str
Updated path to write.

`minedatabase.utils.`**`save_dotted_field`**(*accessor_string: str*, *data: dict*)

Saves data to a dictionary using a dotted accessor-string.

Parameters

accessor_string
[str] A dotted path description, e.g. "DBLinks.KEGG".

data
[dict] The value to be stored.

Returns

dict
The nested dictionary.

3.8 Support

Need help? Found a bug? Have an idea for a useful feature?

Feel free to open up an issue at https://github.com/tyo-nu/MINE-Database for any of these situations, and we will get back to you as soon as we can!

PYTHON MODULE INDEX

m

INDEX

L

M

N

P

R

S

T

W