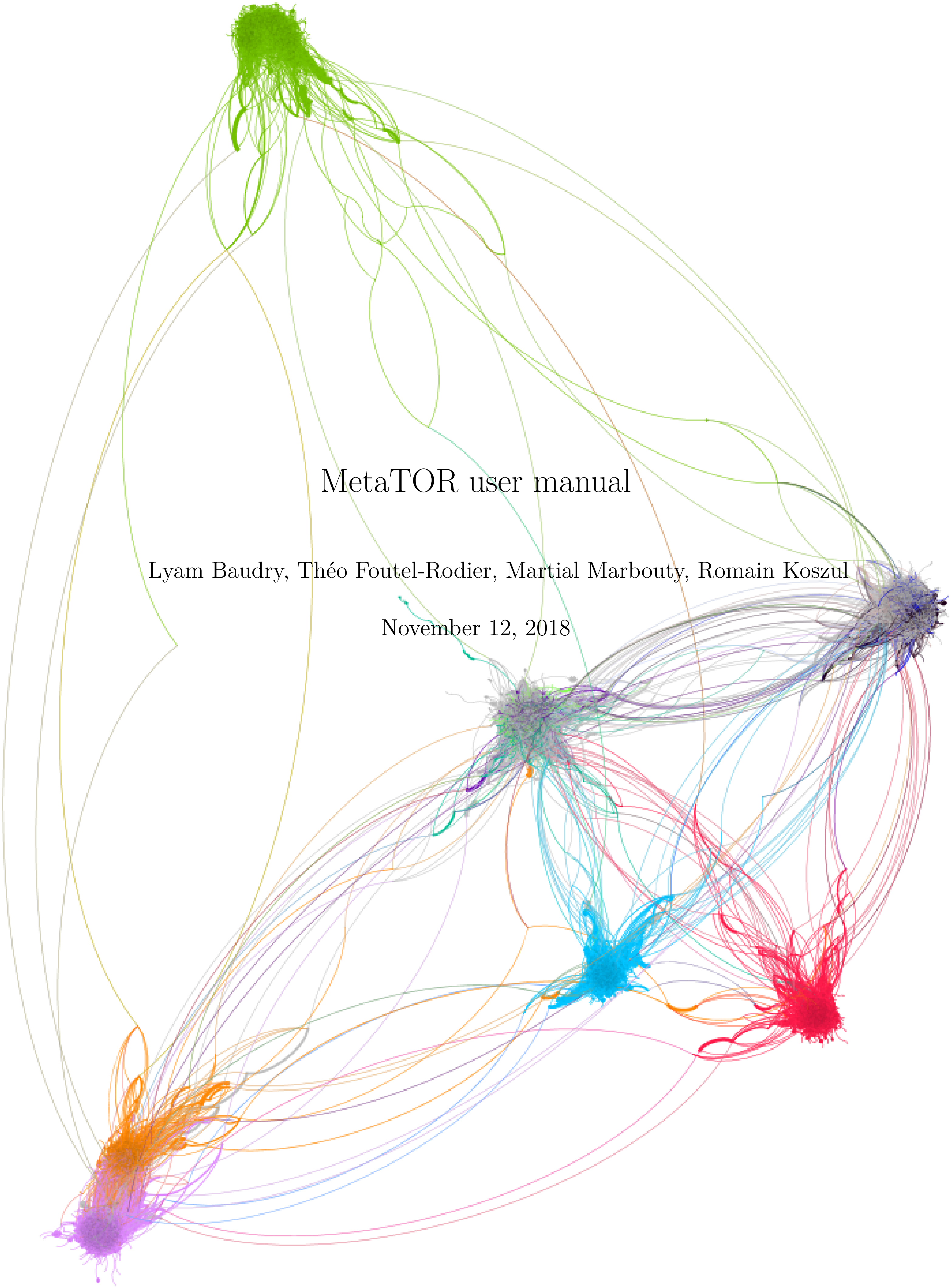

MetaTOR user manual

Lyam Baudry, Théo Foutel-Rodier, Martial Marbouty, Romain Koszul

November 12, 2018

Contents

1 For the impatient

If you know what you're doing and just want to quickly run the pipeline in a single step with no tweaking whatsoever:

```
pip3 install metator
metator dependencies
metator pipeline -1 reads_for.fastq -2 reads_rev.fastq -a assembly.fa
```

where `reads_for.fastq` and `reads_rev.fastq` are your forward and reverse paired-end read files, and `assembly.fa` is your preliminary metagenome assembly in FASTA format.

If you wish to know more about the way the pipeline works or you are encountering issues with the above, one-step approach, head down below to learn about the various parts and parameters of the pipeline.

2 Installation

MetaTOR is written in Python 3. You may install it from the PyPI package repository:

```
pip3 install metator
```

or, if you want to use the latest version:

```
pip3 install -e git+https://github.com/koszullab/metator.git@master#egg=metator
```

Older versions with no installer (just download, unzip and run) or with Python 2 versions are available on alternative branches of the github repository. Optionally, if you don't have hidden Markov model databases of your own, you may want to download the ones we provide and put them in the `HMM` folder.

2.1 Prerequisites

2.1.1 System requirements

Operating system MetaTOR was tested on GNU/Linux and OS X. It should run on any UNIX based platform provided it has the appropriate third-party tools. Windows platforms with environments like Cygwin or Windows Subsystem for Linux (WSL) *should* work, but there's no guarantee.

Memory consumption Some parts of the pipeline such as the alignment or the partitioning can be quite memory hungry, depending on the input size. Critical steps are confined in their own scripts so that they may be run on separate machines if needed.

2.1.2 Environment requirements

Libraries The pipeline was written in Python and Bash, which both should be available on practically any UNIX machine. Specifically, the Python parts make use of the `biopython`, `numpy`, `matplotlib` and `pysam` library, which can be easily installed via `pip` (using Python's package manager) or your system's own package manager (e.g. `apt` on Ubuntu):

- `sudo apt-get install python3-numpy` (or `sudo pip3 install numpy`)
- `sudo apt-get install python3-biopython` (or `sudo pip3 install biopython`)

and so on. Note that metaTOR will automatically try to resolve, download and install the dependencies as needed, so **this step should not be necessary** unless compatibility issues arise in the future, or `pip` fails for some reason.

Third-party tools The pipeline makes extensive use of third party tools along its progress. Because it can be cumbersome to maintain these on several machines, not all scripts require all such tools be present at the same time. Namely, the pipeline requires:

- `bowtie2`
- `hmmer` (and some HMM databases)
- `prodigal`
- The original implementation program of the Louvain algorithm, found at `https://sites.google.com/site/findcommunities/` (unless you skip the partitioning part).

Note that `bowtie2`, `hmmer` and `prodigal` are available on the standard Ubuntu package manager. Dependencies that can't be easily found in one's package manager can be downloaded with the following command, after installing metaTOR:

```
metator dependencies
```

This will fetch `prodigal`, `louvain` for your OS, as well as HMM databases, and place them the `tools` folder of the pipeline (resp. `hmm_databases` folder). You may manually put any missing tool in that folder, or even specify a new folder entirely. If the pipeline doesn't find a tool it needs on your system, (or you didn't add the tool's location to your `$PATH`), that folder will be the next place it will look for.

2.2 Set-up location

Prepare an empty directory wherever you like, as long as you have writing and executing permissions for where you set it up (this is important for generating outputs and running the aforementioned third-party tools). This directory will contain output, possible dependencies, and a configuration file. Note that the accumulation of outputs may take up a lot of storage space.

3 Preparing inputs

At the very least, the pipeline requires a FASTA file representing the preliminary assembly and two paired-end FASTQ files representing the 3C library, and will work its way from there. Other inputs are optional and only needed if one wishes to skip parts of the pipeline.

3.1 Preliminary assembly

This is the metagenome assembly metaTOR will be working on, mapping the reads against and generating the network nodes from. The metagenome assembly step isn't included here because it can be extremely resource consuming and the assembler isn't involved for the rest of the pipeline.

3.1.1 Assembler inputs

The preliminary assembly is normally generated from reads coming from the same metagenome. There is no reason not to generate the assembly from the same paired-end 3C library you'll be using for the pipeline. However, if you happen to have shotgun reads at your disposal anyway, performing a separate assembly or merging them with the actual 3C library will presumably improve the assembly's quality.

3.1.2 Choice of software

MetaTOR was tested on assemblies generated with IDBA-UD, MEGAHIT and metaSPAdes as they are specifically tailored for metagenomic reads. There should be no specific issue with any other assembler, however.

3.1.3 Assembly processing

Special characters Regardless of the assembler and reads used, it is probably safer to make sure the FASTA file doesn't contain special characters in the sequence headers (that is, any non-alphanumeric or non-ASCII character that isn't an hyphen or an underscore), as they have been known to cause issues. It will try to correct what it can but there is no guarantee that any of the many tools used won't break in subtle and unpredictable ways. Ambiguous IUPAC bases (N, B, D, H, V, etc.) should be fine.

Contig size The pipeline will discard any contig below a threshold size (500 bp by default). It is useless to include such contigs in the file. You should keep that in mind if your assembly happens to be highly fragmented, and perhaps lower the threshold accordingly.

3.2 3C reads

The pipeline currently supports a single, paired-end library. Reads should be processed, trimmed for adapters and filtered for quality beforehand. It works seamlessly with gzip'd or uncompressed reads.

4 Configuration

MetaTOR automatically generates a `config.sh` file that contains all relevant parameters to each step of the pipeline before running anything. It lets you control practically anything in any step, down to each specific output path, but it can also auto-configure everything if you want it to 'just work' out of the box.

You may either edit `config.sh` by hand or use arguments when running `metator` (which will then automatically generate `config.sh` for you). Arguments you don't specify will refer to the previous `config.sh` file by default. The arguments are described in detail in the subsections below.

You will notice there are different files beginning with `config`. The point is that you want your parameters from the previous run to be saved, but you also want to be able to start over with the defaults when needed. Each file has a different purpose:

- `config_current.sh` contains the configuration parameters for the *run you've just launched.* It's generated anew every time you run a pipeline step, so there's no use trying to modify it, but it can be consulted for troubleshooting purposes.
- `config.sh` contains the configuration parameters for the *next run you'll launch.* In other words, it is a template for what is going to be used to generate the next *config.sh* file for the next run, and it can be altered manually.
- `config_template.sh` contains the configuration file *with all the original default parameters.* You typically don't want to touch it unless you know what you are doing. When you launch the pipeline with the `--reset` option, it skips `config_current.sh` and directly uses that script as a template to generate your config file.

4.1 Overview

The first and foremost parameter you need to set is the name of your project, at the very beginning of the `config.sh` file. This is done by modifying the line containing the `project` variable (or using the `-p` or `--project` argument). Each project is by default given a distinct location in the output folder, and all outputs relevant to that project are contained within the project's folder.

4.1.1 Inputs

These are the most important variables and indeed the only mandatory ones that you have to specify yourself. The rest can be auto-configured.

- `assembly` (`-a` or `--assembly`): path to the initial metagenome the pipeline will be working on.
- `fastq_for` (`-1` or `--fastq-for`): forward paired-end 3C reads.
- `fastq_rev` (`-2` or `--fastq-rev`): reverse paired-end 3C reads.

4.1.2 Directories

- `working_dir` (`--working-dir`): directory where everything happens.
- `tools_dir` (`--tools-dir`): directory containing third-party tools, typically used for self-containing software that cannot be installed via your system's package manager. By default it contains `louvain`. When looking for any tool, the pipeline will first try to find it there, so you can put there specific versions of `bowtie2`, `hmmer` and so on to override the defaults.
- `model_dir` (`--model-dir`): directory for hidden Markov Chain models used for gene prediction and annotation purposes. Like bowtie2 index files, these can take up a fair bit of space so they get their own directory.
- `output_dir` (`-o` or `--model-dir`): directory where the pipeline writes all its outputs. Each project gets a different folder in the output directory. The following directories are normally located within this one by default.
- `assembly_dir` (`-A` or `--assembly-dir`): directory containing all outputs related to the metagenome assembly. This is distinct from the path to the initial one, which can be anywhere. This is where assemblies are stored after they are parsed for special characters, renamed, filtered, etc.
- `alignment_dir` (`--alignment-dir`): directory containing alignment files (in `sam` format) after mapping of the reads to the initial assembly. Be aware that these can reach enormous sizes.
- `tmp_dir` (`--tmp-dir`): directory containing all temporary and intermediary files written and read by the pipeline. A *lot* of files will be written there, so plan disk space accordingly.
- `network_dir` (`--network-dir`): directory containing 3C contact network files in sparse coordinate (aka edgelist) format.
- `partition_dir` (`--partition-dir`): directory containing files related to the network's partitioning. Each iteration gets its own separate file set.
- `annotation_dir` (`--annotation-dir`): directory containing all annotations of the initial metagenome and the core communities.

4.1.3 Quantitative parameters

- `size_contig_threshold` (`--size-contig-threshold`): any contig shorter than this will be discarded and ignored for the rest of the pipeline. This is done to alleviate size bias issues with minimal data loss (unless your assembly happens to be very fragmented, in which case you may wish to lower the threshold). Default is 500.

- `chunk_size` (`-C` or `--chunk-size`): when generating the network, each contig chunk of this size gets its own node and set of contacts as though they were separate from neighboring chunks. Chunks that are too big will result in high size heterogeneity, which may bias the contacts somewhat. Chunks that are too small will result in a very large, complex and sparse network on which annotations can be next to impossible to perform. Default is 1000.

- `size_chunk_threshold` (`-c` or `--size-chunk-threshold`): an edge case in contig chunk attribution, the tail-ends of chunks that do not reach this threshold are discarded as though they were separate contigs, and chunks that do are integrated into the network as if they were their own chunk. This means all chunks are between `size_chunk_threshold` and `chunk_size` in length. This should not be smaller than the normal contig size threshold. Default is 500.

- `iterations` (`--iterations`): Number of Louvain iterations to perform when partitioning. Default is 100.

- `iter` (`--iter`): Number of Louvain iterations to consider after partitioning, if some reason you wish it to be lower than the total number of iterations. Default is 100.

- `mapping_quality_threshold` (`-Q` or `--mapping-quality-threshold`): Mapping quality threshold to be incorporated into the final 3C contact network. Default is 10.

- `read_size` (`--read-size`): Total length of reads from *both* ends combined. Default is 130.

4.1.4 Miscellaneous

- `index_name` (`--index-name`): basename for bowtie2 index structure files.

5 Running the pipeline

The pipeline is decomposed into four steps. Each of these steps is run separately by calling `metator <action>` where `<action>` is one of the following:

- `align`
- `partition`
- `annotation`
- `binning`

There are additional actions available:

- `deploy` installs the requirements for an Ubuntu 14.04+ machine
- `pipeline` checks for requirements, then launches the four above steps sequentially

The four steps (and expected arguments) are described in detail below.

5.1 Alignment

This is the first step of the pipeline: first, reads are mapped onto the assembly. Then, each scaffold is split into 'chunks' of more or less equal size. A network is generated from contacts between chunks (aka read pair alignments): these are counted as edges and chunk act as nodes.

5.1.1 Relevant parameters

The most important parameters are `chunk_size` and `size_chunk_threshold`, as they determine the size range of your chunks. They can never be longer than `chunk_size`, but tail-ends and contigs that are shorter than that are retained as long as they are above `size_chunk_threshold`. If you want chunks to have the exact same size, set both values to be equal. If you want *no* chunking, set `chunk_size` to an arbitrarily high value and the pipeline will treat each contig as if it were a single chunk of its own.

There are additional relevant parameters such as `mapping_quality_threshold` and `size_contig_threshold` which should be self-explanatory.

5.1.2 Use cases

```
# Generate a network from alignments with mapping quality > 20
# and chunks all exactly equal to 10kb, discarding the rest
metator align -1 reads_forward.fastq -2 reads_reverse.fastq \
-a assembly.fa -Q 20 -c 10000 -C 10000 -p new_project

# Generate a network from alignments with mapping quality > 10
# and 1kb chunks, retaining tail-ends and short contigs above 500bp
# (what was used in Marbouty et al., 2017)
metator align -1 reads_forward.fastq -2 reads_reverse.fastq \
-a assembly.fa -Q 10 -c 500 -C 1000 -p another_new_project

# Generate a network from alignments with mapping quality > 10
# and no chunking, disregarding contigs below 500bp
# (what is used in the upcoming metaTOR paper)
metator align -1 reads_forward.fastq -2 reads_reverse.fastq \
-a assembly.fa -Q 10 -c 500 -C 100000000000000 -p yet_another_new_project

# Generate a network from all alignments and all contigs
# with no chunking (at your own risk)
metator align -1 reads_forward.fastq -2 reads_reverse.fastq \
-a assembly.fa -Q 1 -c 0 -C 100000000000000 -p and_yet_another_new_project
```

5.1.3 Inspecting outputs

The outputs are found in the `alignment_dir` and `network_dir`. In `alignment_dir` you will mainly find enormous BAM files (you may wish to change `alignment_dir` to a partition where you have enough disk space) sorted by name, which may be useful for other third party analyses. In `network_dir` you will find two files: `idx_contig_hit_size_cov.txt` and `network.txt`.

The `idx_contig_hit_size_cov.txt` file links chunk ids to the original sequences in the assembly in the form:

```
chunk_index contig_name total_contacts chunk_size chunk_coverage
```

such as *e.g.* the following:

```
1 contig1_0 0 1262 0.0
2 contig1_1 0 1279 0.0
3 contig2_0 1 568 0.114436619718
4 contig2_1 7 2193 0.207478340173
```

Chunk names take the form `contigX_N` where N represents the position on `contigX`, *i.e.* the first chunk of the contig will be named `contigX_0`, the second will be named `contigX_1` and so on. This makes it easy to identify the exact coordinate of the chunk sequences since `chunk_size` is constant (or otherwise runs all the way to the contig tail).

The `network.txt` file represents contacts between chunks in a plain edge list format. The first two columns represent the sorted indices of the source and target chunks and the third column contains the number of contacts between each pair, like in the following example:

```
source target contacts
1      2      5
1      3      1
1      4      1
2      3      2
3      4      5
```

Absent contacts (such as $(2, 4)$ in the example above) are not saved in the network. The format is deliberately barebones so that other third party tools may freely use it.

5.2 Partition

This is the second and crucial step of the pipeline, as it will determine the composition of the bins. Nodes in the network are clustered according to the Louvin community detection algorithm (using the original implementation found in https://sites.google.com/site/findcommunities/) a certain amount of times, and nodes that are always found together after all iterations are regrouped into so-called core communities. It also draws graphs to visualize the evolution of these cores as more iterations get piled up, notably their size distribution. Empirically, most of the time core size tends to stabilize after a few hundred iterations but this is obviously dataset-dependent.

5.2.1 Relevant parameters

The most important parameter is `iterations`, as it determines how stringent belonging to a core community is going to be. Few iterations are going to result in large and inclusive core communities (and therefore large bins). Many iterations are going to yield very tight core communities which will in turn lead to very conservative bins. There is a balance to be achieved to maximize bin completeness and minimize contamination, but a few hundreds of iterations were empirically found to be sufficient. Also, since the size distribution tends to plateau, most of the times the last iterations are not going to be as impactful.

There is an additional parameter called `iter` which lets you select only a subset of the iterations to determine your core communities. This may come in handy if you found out you ran too many iterations to your liking and don't want to start over.

5.2.2 Use cases

```
# Partition the network with 300 iterations and select 100
# of them for binning purposes
metator partition --iterations 300 \--iter 100 \-p new_project

# Partition the network with a single iteration
metator partition --iterations 1 --iter 1 -p another_new_project

# Partiton the network with 1000 partitions and keep them all
metator partition --iterations 1000 --iter 1000 -p yet_another_new_project
```

5.2.3 Inspecting outputs

There are a lot of outputs for this step. First, each iteration of the Louvain community partition is saved in an `iteration` folder found in `partition_dir/`. They are simple files (named `1.community`, `2.community` all the way to `<iterations>.community`) containing all community ids for all chunks (in chunk id order) in a single column each. The pipeline also takes a snapshot of different numbers of iterations (such as 1, 5, 10, 20, 30, etc. all the way to `iter`) which are are then *resolved* to construct *core communities* that are the basis of our metagenomic bins. Information about these are found in a folder called `partition` in `partition_dir`. For each snapshot, you will find three files named `chunkid_core_size_X.txt`, `chunkname_core_size_X.txt` and `core_size_indices_X.txt` (where `X` is the aforementioned number of iterations).

The first two files are almost identical and contain three columns representing, for each chunk in the network, its id (resp. name), the index of the core to which it was assigned, and the size of that core *in chunks*. The last one lists core communities in size order (so the core 1 will have the most chunks): the first column represents the core index, the second one represents its size *in chunks* and the last colmuns represent the ids of the chunks assigned to that core.

In that folder you will also find some figures plotting the size distribution of cores for each snapshot to help you visualize their relative stability.

5.3 Annotation

This is the third step of the pipeline, although it is independent from the other two. It runs standard gene prediction software on the initial assembly, then maps these putative genes onto public hidden Markov model (HMM) databases. By default, these are:

- Conjugative elements (taken from `https://research.pasteur.fr/en/software/conjscan-t4ssscan/`)
- Virus orthologous groups (VOGs)
- Single gene copies (SGCs)
- Essential genes

5.3.1 Relevant parameters

The most important parameter is the `evalue` as detected by `hmmer`. Hits with an E-value larger than that parameter will be discarded. You may also change `size_contig_threshold` if you do not wish to annotate contigs below a certain size.

5.3.2 Use cases

```
# Annotate contigs above 500bp in the assembly,
# keeping hits below 0.01
metator annotation --evalue 0.01 --size-contig-threshold 500 -p new_project

# Annotate the entire essembly, keeping hits below 0.0001
# It understands scientific notation as well
metator annotation --evalue 1e-4 --size-contig-threshold 0 \
-p another_new_project

# Keep all annotations (at your own risk)
metator annotation --evalue 1 --size-contig-threshold 0 \
-p yet_another_new_project
```

5.3.3 Inspecting outputs

Outputs will be found in the `annotation_dir` folder. First, there are the outputs of `prodigal`'s gene prediction, in all three formats (gene, FASTA/DNA and FASTA/protein).There's also a list of all headers with gene ids assigned to them.

Second, there are `hmmer`'s outputs, normally found in `annotation_dir/HMM`. For each database, there is an `*.out` file which is the raw output and `*_hit_weighted.txt` files listing the number of hits on each database for each contig, in a plain two column format.

5.4 Binning

This is the final and most important part of the pipeline, as it is the one that will yield your metagenomic bins. It also crosses annotations about the contigs with the contig make-up of the bins to annotate them as well, and draw figures about the distribution of these annotations.

5.4.1 Relevant parameters

The most important parameter is `n_bins`, which simply specifies how many bins you want to reconstruct this way (in size order). This is because there can be *many* core communities, most of them pretty small, so attempting to reconstruct them all may quickly saturate your hard drive with pretty inconsequential sequences. You may also change `iter` to one of the values you had previously picked during the partition step (or one of the default snapshots).

5.4.2 Use cases

```
# Bin the 100 biggest core communities after 300 iterations
metator binning --n-bins 100 --iter 300 -p new_project

# Bin all core communities after 100 iterations
metator annotation --n-bins 1000000000 --iter 100 -p another_new_project
```

5.4.3 Inspecting outputs

For a given number of iterations, you get a set of bins. For each bin, you will get:

- In `<partition_dir>/fasta/iteration<iter>`: the bin's chunks in FASTA format
- In `<partition_dir>/fasta_merged/iteration<iter>`: the bin's proper sequences with chunks being merged again when applicable
- In `<partition_dir>/subnetworks/iteration<iter>`: the bin's subnetwork in the same format as `network.txt`

- In the same folder, the subnetwork's corresponding contact map

There are also figures describing the distribution of annotations according to bin size (in `<partition_dir>/figures/iteration<iter>`). Additionally, there are files in `<partition_dir>/figures/iteration<iter>/data` reporting the number of hits (both in total counts and relative counts with respect to the other databases) for each bin.

6 Troubleshooting and contact

Please keep in mind this is an early version. The repository for the pipeline is available at `https://github.com/koszullab/metaTOR` and will regularly get updates. This manual will also get updated if more detailed documentation is required.

You may report issues, bugs and feature requests on the repo or contact the writers of the pipeline at one of the following addresses:

- lyam.baudry@pasteur.fr
- martial.marbouty@pasteur.fr
- thfoutel@pasteur.fr
- romain.koszul@pasteur.fr

We welcome feedback and suggestions.