

## *Supplementary Material for*

# **A modular protocol for metagenomic read classification: using disk storage dictionaries to transfer taxonomic and functional annotations**

Diego A. A. Morais, João V. F. Cavalcante, Matheus A. B. Pasquali, Rodrigo J. S. Dalmolin

## **1 Notation**

In what follows, all commands run at the command line and R scripts appear in **bold Courier** font. Unix commands are prefaced by a dollar sign (\$):

```
$ wc -l file.txt
```

R scripts are prefaced by a greater-than sign (>):

```
> mean(1:10)
```

Both left aligned.

## **2 Materials**

### **2.1 Operating system**

This protocol assumes users have a Linux operating system with a bash shell or similar. All commands given here were tested on Ubuntu 18.04, and are meant to be run in a terminal window.

## **2.2 Input file**

The initial input is meant to be a FASTQ file, a common format for reads from Illumina sequencing machines.

## **2.3 Example data**

We selected a public human gut metagenome shotgun data from the National Center for Biotechnology Information (NCBI). The run SRR579292 from the BioProject PRJNA175224 is a paired-end data obtained by an Illumina MiSeq instrument, a sequencing platform which produces a maximum output of a few Gb. The chosen run is one of the smallest datasets from a human host present in the NCBI Sequence Read Archive (SRA), being this data suited for a quick demonstration of our workflow. This data is from a patient with Crohn's disease and the following analysis aims to identify the microbial community and the functional activity present within it.

## **3 Setup**

An Anaconda environment was created to ease the installation of all required software and is available at the anaconda cloud (<https://anaconda.org/arthurvinx/protocolmeta>). A conda package manager, such as miniconda (<https://docs.conda.io/en/latest/miniconda.html>), must be installed to get this environment.

Download and install the latest miniconda release for Python 3.7 (adapt the commands if needed):

```
$ cd ~/Downloads  
$ chmod u+x Miniconda3-latest-Linux-x86_64.sh  
$ ./Miniconda3-latest-Linux-x86_64.sh
```

Check if the installation was successful (need to exit the terminal and then return to it):

```
$ conda -V
```

Get the environment containing the required tools to run the protocol:

```
$ conda install anaconda-client  
$ conda env create arthurvinx/protocolMeta
```

Activate the environment:

```
$ conda activate protocolMeta
```

This environment must be active to run the commands from the next sections.

## 4 Folder structure

This protocol assumes that the following folder structure will be used to organise the downloaded data, intermediate files, and outputs.

```
|-- Protocol
    |-- data
        |-- assembled
        |-- collapsed
        |-- removal
        |-- index
        |-- reference
        |-- trimmed
    |-- alignment
        |-- db
        |-- index
    |-- taxonomic
    |-- functional
```

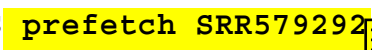

Create this folder structure:

```
$ mkdir -p
./Protocol/{data/{assembled,collapsed,removal/{index,reference},
trimmed},alignment/{db,index},taxonomic,functional}
```

## 5 Preprocessing

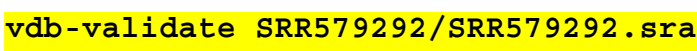

Change the current directory to “Protocol/data” and download the example data:

```
$ prefetch SRR579292
```

Check the downloaded data:

```
$ vdb-validate SRR579292/SRR579292.sra
```

Proceed if the important checks have an “ok” status as follows:

```
2019-12-11T14:23:06 vdb-validate.2.10.0 info: Database 'SRR579292.sra' metadata: md5 ok
2019-12-11T14:23:06 vdb-validate.2.10.0 info: Table 'SEQUENCE' metadata: md5 ok
2019-12-11T14:23:06 vdb-validate.2.10.0 info: Column 'ALTREAD': checksums ok
2019-12-11T14:23:06 vdb-validate.2.10.0 info: Column 'QUALITY': checksums ok
2019-12-11T14:23:06 vdb-validate.2.10.0 info: Column 'READ': checksums ok
2019-12-11T14:23:06 vdb-validate.2.10.0 info: Column 'READ_LEN': checksums ok
2019-12-11T14:23:06 vdb-validate.2.10.0 info: Column 'READ_START': checksums ok
2019-12-11T14:23:06 vdb-validate.2.10.0 info: Database 'SRR579292/SRR579292.sra' contains
only unaligned reads
```

2019-12-11T14:23:06 vdb-validate.2.10.0 info: Database 'SRR579292.sra' is consistent

Get the paired-end reads:

```
$ fasterq-dump SRR579292
```

Remove low-quality sequences and adapters:

```
$ fastp -i SRR579292_1.fastq -I SRR579292_2.fastq -o
trimmed/SRR579292_1_trim.fastq -O trimmed/SRR579292_2_trim.fastq
-q 20 -w 8 --detect_adapter_for_pe -h trimmed/report.html -j
trimmed/fastp.json
```

Note: In the fastp call, the -q argument sets the phred quality score threshold, and -w specifies the number of threads to use (adapt these arguments if needed).

A *Homo sapiens* reference genome is needed to remove the host sequences from this data. Change the current directory to “Protocol/data/removal/reference” and download a reference genome from Ensembl:

```
$ wget
ftp://ftp.ensembl.org/pub/release-98/fasta/homo_sapiens/dna/Homo
_sapiens.GRCh38.dna.primary_assembly.fa.gz
$ pigz -d Homo_sapiens.GRCh38.dna.primary_assembly.fa.gz -p 8
```

Note: The pigz software is a parallel implementation of gzip, the -p argument specifies the number of threads to use.

Build a bowtie2 index:

```
$ bowtie2-build Homo_sapiens.GRCh38.dna.primary_assembly.fa
../index/host --threads 8
```

Go back to “Protocol/data” and align the sequences against the reference:

```
$ bowtie2 -x removal/index/host -1
trimmed/SRR579292_1_trim.fastq -2 trimmed/SRR579292_2_trim.fastq
-S removal/all.sam -p 8
```

Extract the unaligned reads:

```
$ samtools view -bS removal/all.sam > removal/all.bam
$ samtools view -b -f 12 -F 256 removal/all.bam >
removal/unaligned.bam
$ samtools sort -n removal/unaligned.bam -o
removal/unaligned_sorted.bam
$ samtools bam2fq removal/unaligned_sorted.bam >
removal/unaligned.fastq
$ cat removal/unaligned.fastq | grep '^@.*1$' -A 3
--no-group-separator > removal/unaligned_1.fastq
```

```
$ cat removal/unaligned.fastq | grep '^@.*/2$' -A 3
--no-group-separator > removal/unaligned_2.fastq
```

Note: In the SAMtools call, the -f and -F arguments use flags to specify, respectively, which alignments must be extracted and which ones should not be extracted. See <http://broadinstitute.github.io/picard/explain-flags.html> to know more about the available SAMtools flags.

Merge the paired-end reads:

```
$ fastp -i removal/unaligned_1.fastq -I
removal/unaligned_2.fastq -o trimmed/unassembled_1.fastq -O
trimmed/unassembled_2.fastq -q 20 -w 8 --detect_adapter_for_pe
-h trimmed/report2.html -j trimmed/fastp2.json -m --merged_out
assembled/SRR579292_assembled.fastq
```

Remove duplicated sequences:

```
$ fastx_collapser -i assembled/SRR579292_assembled.fastq -o
collapsed/SRR579292.fasta
```

## 6 Alignment



Change the current directory to “Protocol/alignment/db” and download the NCBI non-redundant protein database:

```
$ wget ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/nr.gz
```

```
$ pigz -d nr.gz -p 8
```

Build a DIAMOND index:

```
$ diamond makedb --in nr -d ../index/nr
```

Change the current directory to “Protocol/alignment” and align the reads against the reference protein database:

```
$ touch unaligned.fasta
```

```
$ diamond blastx -d index/nr -q
```

```
../data/collapsed/SRR579292.fasta -o matches.m8 --top 3 --un  
unaligned.fasta
```

Note: In the DIAMOND call, the --un argument specifies the file used to write the unaligned sequences, and --top 3 report alignments within this percentage range of top alignment score. DIAMOND may use a lot of memory and temporary disk space, therefore, the program may fail due to running out of either one. The -b argument specifies the block size argument, and -c the number of chunks for index processing. The total memory usage can be roughly estimated by  $2(b+9b/c)$  GB. On a high memory server is recommended to set -c to 1.

Perform a more sensitive alignment using the unaligned sequences:

```
$ touch unaligned2.fasta
$ diamond blastx -d index/nr -q unaligned.fasta -o matches2.m8
--more-sensitive --top 3 --un unaligned2.fasta
```

Note: The default sensitive mode was mainly designed for short reads (around 100 bp). For longer sequences, the sensitive modes are recommended. DIAMOND is much more efficient for large query files (> 1 million reads).

Check the number of queries presenting hits with at least 80% of identity in matches2.m8:

```
$ awk '{ if ($3>=80) { print } }' matches2.m8 > check.m8
$ awk '{a[$1]++}END{for (i in a) sum+=1; print sum}' check.m8
```

Concatenate the alignment outputs:

```
$ cat matches.m8 matches2.m8 > all_matches.m8
```

## 7 Taxonomic classification

Download the NCBI taxonomy file:

```
$ basta taxonomy
```

Download the NCBI mapping file for the hit identifiers present in the alignment output:

```
$ basta download prot
```



Note: These files are downloaded at “~/basta/taxonomy” by default. Two mapping repositories will be created at this directory after the downloads.

Change the current directory to “Protocol/taxonomic” and perform the taxonomic classification:

```
$ basta sequence ../alignment/all_matches.m8 basta_out.txt prot
-l 1 -m 1 -b True
```

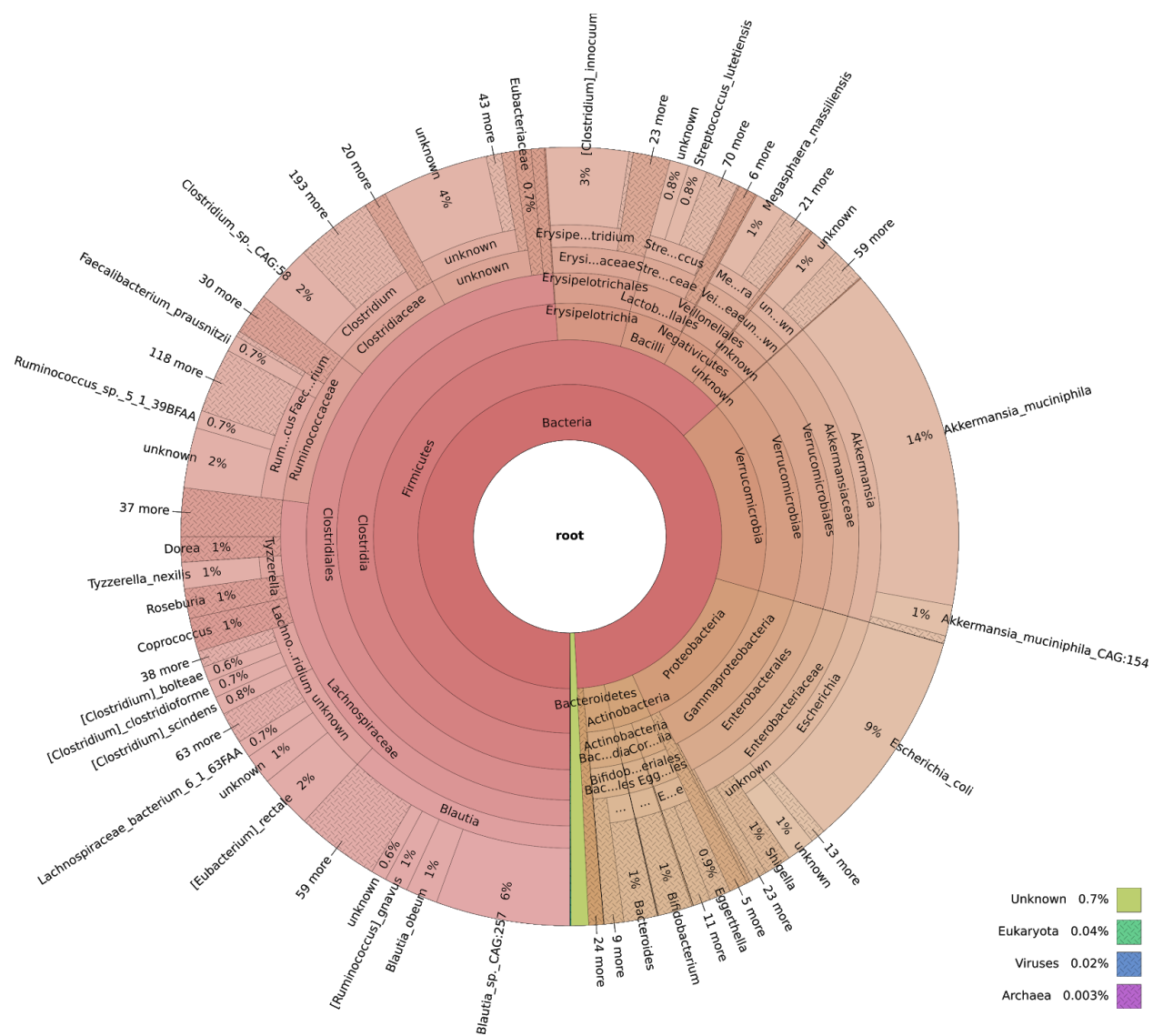
Note: In the basta call, the -l argument modifies the alignment length threshold, -m specifies the minimum number of hits used on the Last Common Ancestor estimation, and -b includes a column containing the taxonomy of the best hit.

Split the output into two files:

```
$ awk -F "\t" '{print $1"\t"$2}' basta_out.txt > lca.txt
$ awk -F "\t" '{print $1"\t"$3}' basta_out.txt > best_hit.txt
```

Generate the Krona plots:

```
$ basta2krona.py lca.txt lca.html
$ basta2krona.py best_hit.txt best_hit.html
```



Supplementary Figure S1 - Taxonomic classification of 216,828 queries, passing the identity threshold, using only the best hit.

## 8 Functional annotation

Change the current directory to “Protocol/functional” and download the UniProt ID mapping file:

```
$ wget
ftp://ftp.uniprot.org/pub/databases/uniprot/current_release/knowledgebase/idmapping/idmapping_selected.tab.gz
$ pigz -d idmapping_selected.tab.gz -p 8
```

Filter the columns containing Gene Ontology and GenBank identifiers:

```
$ awk -F "\t" '{if(($7!="") && ($18!="")){print $18"\t"$7}}'
idmapping_selected.tab > genbank2GO.txt
```

Filter the columns containing Gene Ontology and RefSeq identifiers:

```
$ awk -F "\t" '{if(($4!="") && ($7!="")){print $4"\t"$7}}'
idmapping_selected.tab > refseq2GO.txt
```

Use the R language to clean the dictionaries:

```
> library(dplyr)
> library(tidyr)
> library(data.table)
> fixCollapsed <- function(df){
  colnames(df) <- c("key", "value")
  df <- df %>%
```

```

      mutate(key = strsplit(key, "; ")) %>%
      unnest(key)
df <- df[, c(2, 1)]
return(df)
}
> fixDuplicated <- function(df){
  df <- df %>%
    group_by(key) %>%
    summarise(value = paste(value, collapse = "; "))
  values <- strsplit(df$value, "; ")
  values <- lapply(values, unique)
  values <- sapply(values, paste, collapse = "; ")
  df$value <- values
  return(df)
}
> removeUnknown <- function(df){
  idx <- grepl("^-", df$key)
  df <- df[!idx,]
  return(df)
}
> df <- fread("genbank2GO.txt", stringsAsFactors = FALSE,
head = FALSE, nThread = parallel::detectCores())

```

```

> df <- as.data.frame(df)

> df %>%

  fixCollapsed() %>%

  fixDuplicated() %>%

  removeUnknown() %>%

  fwrite(file = "dictionary.txt", sep = "\t",
nThread = parallel::detectCores())

> df <- fread("refseq2GO.txt", stringsAsFactors = FALSE,
head = FALSE, nThread = parallel::detectCores())

> df <- as.data.frame(df)

> df %>%

  fixCollapsed() %>%

  fixDuplicated() %>%

  removeUnknown() %>%

  fwrite(file = "dictionary.txt", sep = "\t",
append = TRUE, nThread = parallel::detectCores())

```

Note: This script requires a lot of memory to deal with the filtered UniProt ID mapping file contents. A 4.9 GB input requires roughly 78 GB of memory.

Generate a levelDB from the cleaned dictionary:

```
$ basta create_db dictionary.txt nr2GO_mapping.db 0 1
```

Annotate each query using the best alignment for which a mapping is known:

```
$ annotate ../alignment/all_matches.m8 functional.txt nr2GO -l 1
```

Note: In the annotate call, the -l argument modifies the alignment length threshold.

Use the R language to get the GO identifiers description:

```
> library(GO.db)

> library(data.table)

> library(dplyr)

> library(tidyr)

> fixCollapsed <- function(df){

  df <- df %>%

    mutate(Annotation = strsplit(Annotation, "; ")) %>%

    unnest(Annotation)

  df <- df[, c(2, 1)]

  return(df)

}

> getGODescription <- function(GOs){

  BP <- names(as.list(GO.db::GOBPCHILDREN))

  CC <- names(as.list(GO.db::GOCCCHILDREN))

  MF <- names(as.list(GO.db::GOMFCHILDREN))
```



```

result <- vapply(GOs, function(x){
  if(x %in% BP){
    return("BP")
  }else if(x %in% CC){
    return("CC")
  }else if(x %in% MF){
    return("MF")
  }else{
    return(NA_character_)
  }
}, character(1))

return(result)
}

> data <- as.data.frame(fread("functional.txt", header = T,
stringsAsFactors = F))

> data <- fixCollapsed(data)

> data <- data$Annotation

> data <- as.data.frame(table(data))

> data$data <- as.character(data$data)

> data$Ontology <- getGODescription(data$data)

> colnames(data)[1] <- "GOId"

> idx <- which(is.na(data$Ontology))

```

```

> data$Ontology[idx] <- "Unknown"

> xx <- as.list(GOTERM)

> data$desc <- sapply(data$GOId, function(i){

  flagError <- FALSE

  tryCatch({

    res <- xx[[i]]@Term

  }, error = function(e) {

    flagError <-<- TRUE

  })

  if(flagError){

    return("Unknown")

  }

  else{

    return(res)

  }

})

> data <- data[, c(2, 3, 1, 4)]

> write.table(data, file = "functional_description.txt",

col.names = TRUE, row.names = FALSE, quote = FALSE, sep = "\t")

```