UMI-tools Documentation

Tom Smith and Ian Sudbery

Jun 13, 2023

Contents:

Note: **Important update**: We now recommend the use of *alevin* for 10x Chromium and Drop-seq droplet-based scRNA-Seq. *alevin* is an accurate, fast and convenient end-to-end tool to go from fastq -> count matrix and extends the UMI error correction in *UMI-tools* within a framework that also enables quantification of droplet scRNA-Seq without discarding multi-mapped reads. See alevin documentation and alevin pre-print for more information

Welcome to the UMI-tools documentation. UMI-tools contains tools for dealing with Unique Molecular Identifiers (UMIs)/Random Molecular Tags (RMTs) and single cell RNA-Seq cell barcodes.

UMI-tools was published in Genome Research on 18 Jan '17 (open access)

CHAPTER 1

Installation Guide

Contents

We currently support installation on Linux (tested on Ubuntu and CentOS) and Mac OSX. We do not currently support Windows.

There are three possible ways to install UMI-tools: conda, pip or from source, in decending order of ease.

Note: As of version 1.0.0 UMI-tools requires python 3.5 or better. If you are still using python 2.7, we recommend you switch to python 3. If you use *conda* it is possible to have both python 2 and python 3 environments. If you can't do this and really need 2.7, we recommend you use UMI-tools 0.5.5, but note that this will not be updated.

1.1 Quick Start

Try one of the following:

```
$ conda install -c bioconda -c conda-forge umi_tools
```

or:

```
$ pip install umi_tools
```

or grab a zip of the latest release from github and unpack (replace *0.5.5* with version number; replace `wget` with `curl -O` for OS X):

```
$ unzip 1.0.0.zip
$ cd UMI-tools-1.0.0
$ python setup.py install --user
```

If these options don't work, see below.

1.2 Conda package manager

This is the easiest way to install `UMI-tools` if you are already using either anaconda python or miniconda: all depedencies, whether they be python libraries or system libraries are automatically installed. You can also do all your installations in seperate isolated "environments" where installing new software will not affect packages in other environments. The downside of `conda` installation is that if you do not already use anaconda or miniconda, you will be building a completely new python environment, would have to reinstall all of your libraries etc. You can read more about conda here.

1. Install miniconda (if not already installed), see conda installation instructions here.
2. Type:

   ```
   $ conda install -c bioconda -c conda-forge umi_tools
   ```

That's it, simple as that.

1.3 Installation from PyPI using the pip package manager

The pip python pacakge manager is the standard package manager. The advantage over conda is that it is probably already installed on your system, will use your existing python environments, and plays nicely with the virtualenv system. On the downside, the installation of dependencies is not handled as cleanly as in conda. You will need

- python version greater than 3.5
- gcc or compatible c compiler
- zlib with development headers
- the pip python package manager version at least 1.4

1.3.1 Linux

Most systems will already have `gcc`, `pip` and `zlib` installed, so its worth just trying:

```
$ pip install umi_tools
```

If you also have python 2 on your system, you may need to use `pip3` rather than `pip`. If you get a permissions error try adding `--user` to the `pip` command. Note that `umi_tools` will now only be installed for the current user.

If that doesn't work, then you need to find what is missing. You can check for gcc and pip by typing gcc or pip at a terminal prompt. Installing GCC and zlib is much easier if you have root access on your machine. In the unlikely event that you don't have these installed AND you don't have root access, please speak to someone who does have root access. Python and pip can be installed without root.

1. **Install gcc**: the easiest way is using your package manager. In ubuntu or debian:

   ```
   $ sudo apt-get install gcc
   ```

   or in CentOS/Redhat/Fedora:

   ```
   $ sudo yum install gcc
   ```

2. **Install zlib**: again, use the package manager. Ubuntu:

   ```
   $ sudo apt-get zlib1g-dev
   ```

   or in CentOS/Redhat/Fedora:

   ```
   $ sudo yum install zlib-devel
   ```

3. **Install pip**: pip is also probably available from your package manager. In ubuntu, Centos, RHEL and fedora the package is called *python-pip*. In CentOS/RHEL the package is located in the EPEL repository which needs to be installed first. You could also install pip from the web:

   ```
   $ wget https://bootstrap.pypa.io/get-pip.py
   $ python get-pip.py --user
   ```

   but in this case you'll need to make sure that the `python-dev` (Ubuntu) or `python-devel` (CentOS/RHEL/fedora) packages are installed.

The pip command at the top should now work.

1.3.2 Apple OS X

The good news is that *zlib* is installed by default of OS X. The bad news is that *gcc* and *pip* are generally not included (although many users may have installed them already). Furthermore, it's generally not advisable to use the default python since installation of third party python libraries leads to difficulties with permissions, especially since the introduction of System Integrity Protection (SIP) from OS X El Capitan onwards. For this reason, we recommend using a non-default python.

If you only have the default python (e.g /usr/local/bin/python) there are a number of ways to install another instance of python. Many OS X users recommend using the `homebrew` package manager to manage command line packages on OS X. You can find instructions here for installation python via `homebrew`. This will also install setuptools and pip. You can install gcc via homebrew by following these instructions:

```
$ brew install gcc48
```

You may also need to install `freetype`:

```
$ brew install freetype
```

Install UMI-tools: You should now have everything you need to install `UMI-tools`:

```
$ pip install umi_tools
```

We have had reports that the current version of one of the `UMI-tools` dependencies, `pysam`, is causing problems on the latest versions of OS X. If your installation is failing on the installation of pysam, try forcing an older version with:

```
$ pip install pysam==0.8.4
```

before installing `umi_tools`.

If you don't want to do use homebrew, here are non-homebrew instructions for installing gcc and pip as needed:

1. **Install gcc**: Apples XCode suite includes `gcc`. Installation depends on which version of OS X you are using

   - *Mac OS X 10.9* or higher: Open a terminal and run:

     ```
     $ xcode-select --install
     ```

   - *Mac OS X 10.8* or lower: go to Apple's developer download page and download Command Line Tools for XCode. You'll need a developer account.

2. **Install pip**: In a terminal type:

   ```
   $ curl -O https://bootstrap.pypa.io/get-pip.py
   $ python get-pip.py
   ```

1.4 Installing from source

There are several reaons you might want to install from source. If for example you need to install the most up-to-date version, or if you can't or don't want to use one of the package managers above. There are two levels of installing from source. The first is to install the dependencies using one of the pacakge managers above, and then just install `umi_tools` from source. The second is to install everything from source without the help of pip or conda.

1.4.1 Depedencies from conda/PyPI manager

1. Download the UMI-tools code, either the latest release or the master branch (which should contain the lastest development version) and unpack the zip or tar and enter the directory:

   ```
   $ unzip 1.0.0.zip
   $ cd UMI-tools-1.0.0
   ```

   or clone the repository:

   ```
   $ git clone https://github.com/CGATOxford/UMI-tools.git
   ```

3. Use your python package manager to install the dependencies. e.g. for `pip`

   $ pip install -r requirements.txt

   or with `conda`:

```
$ conda install setuptools
$ conda install pandas
$ conda install future
$ conda install scipy
$ conda install matplotlib
$ conda config --add channels bioconda
$ conda install regex
$ conda install pysam
```

4. Install UMI-tools using the `setup.py` script:

```
$ python setup.py install --user
```

1.4.2 Completely from source

Warning: **This section is deprecated and no longer updateed**. Once upon a time it was possible for us to provide complete instructions for installing completely from source without a package manager. Unfortunately, our dependencies have multiplied and the dependencies of our dependencies have also multiplied. You can try the below and it may work as the system libraries required are not particularly rare, especially if you are already doing bioinformatics. However, if one of the dependencies fails to install, I'm afraid you are on your own.

This method will allow you to install without installing pip or conda. It is in theory possible to install completely without root by installing gcc, zlib and python-dev in your home directory, but that is beyond the scope of this document. You are also going to need a `g++` compatiable compiler. On OS X `XCode` has one of these by default. On Linux install the `build-essential` or `g++` packages.

1. Download and install *Cython*. For OS X replace `wget` with `curl -O`:

```
$  wget https://pypi.python.org/packages/c6/fe/
↪97319581905de40f1be7015a0ea1bd336a756f6249914b148a17eefa75dc/Cython-0.24.1.tar.
↪gz
 $ tar -xzf Cython-0.24.1.tar.gz
 $ cd Cython-0.24.1.tar.gz
 $ python setup.py install --user
```

2. Download and install `UMI-tools`:

```
$ wget https://github.com/CGATOxford/UMI-tools/archive/master.zip
$ unzip master.zip
$ cd UMI-tools-master
$ python setup.py install --user
```

running this is probably going to take quite a long time. You will probably see quite a lot of warning messages that look like errors.

The most likely fail point is installing `pysam`. Due to a bug in pysam, when it is installed from source, the recorded install version is wrong. Thus, if you get the error:

```
$ pysam 0.2.3 is installed by 0.8.4 is required by umi_tools
```

try just running setup again.

In addition, as we pointed out above, we have had reports that installation of the lastest `pysam` fails on the latest OS X. If this is the case, try installing an older version of `pysam`:

```
$ curl -O https://pypi.python.org/packages/27/89/
↪bf8c44d0bfe9d0cadab062893806994c168c9f490f67370fc56d6e8ba224/pysam-0.8.4.tar.gz
$ tar -xzf pysam-0.8.4.tar.gz
$ cd pysam-0.8.4
$ python setup.py install --user
```

1.4.3 Running tests

After installing from source you can run the test suite to make sure everything is working. To do this you'll need to install *nose* and *pyyaml* using your favourite package manager and then run:

```
$ nosetests tests/test_umi_tools.py
```

1.5 Getting further help

If you are still having trouble with installation, contact us by by creating an issue on our github issues page.

CHAPTER 2

Quick start guide

This quick start guide uses an iCLIP dataset as an example. If you want to apply UMI-tools in a single cell RNA-Seq analysis, please see the *Single cell tutorial*

The following steps will guide you through a short example of how to use the `UMI-tools` package to process data with UMIs added to them. The data used comes from one of the control replicates from Mueller-Mcnicoll et al, *Genes Dev* (2016) 30: 553. We have adaptor trimmed and filtered the data to reduce its size.

The general pipeline is:

extract UMI from raw reads -> map reads -> deduplicate reads based on UMIs

The most computationally intensive part of this is the middle part - mapping the reads. It is also the least interesting for us here. To aid the ability of folks to follow along without having to worry if they have the correct indexs install etc, we have provided a BAM file of the mapped reads from this example. You can download it here. It will need indexing with `samtools index` before use. You can then skip straight to Step 5 below.

2.1 Step 1: Install `UMI-Tools`

The easiest way to install `UMI-Tools` is using your favorite python package manager, either `pip` or `conda`. If you don't know which you have installed, we recommend trying both, starting with `conda`:

```
$ conda install -c bioconda umi_tools
```

Alternatively, `pip`:

```
$ pip install umi_tools
```

If neither of these work, see our installation guide.

2.2 Step 2: Download the test data

The test data we are going to use is a control sample from a recent iCLIP experiment. We have processed this data to remove the various adaptors that you find in iCLIP data. You can download the trimmed file here:

```
$ wget https://github.com/CGATOxford/UMI-tools/releases/download/v0.2.3/example.fastq.
↪gz
```

If you're using macOS use:

```
$ curl -L "https://github.com/CGATOxford/UMI-tools/releases/download/v0.2.3/example.
↪fastq.gz" -o "example.fastq.gz"
```

The file is about 100Mb, and takes a couple of minutes to download on our system.

2.3 Step 3: Extract the UMIs

UMIs are strings of random nucleotides attached to the start of reads. Before the reads are mapped the random nucleotides must be removed from the reads, but the sequence must be kept. The `extract` command of `UMI-Tools` moves the UMI from the read to the read name.

Several techniques that use UMIs mix the UMI sequence in with a library barcode. In this case we want to remove the random part of the barcodes, but leave the library part so that the reads can be de-multiplexed. We specify this using the `--bc-pattern` parameter to `extract`. Ns represent the random part of the barcode and Xs the fixed part. For example, in a standard iCLIP experiment, the barcode is made of 3 random bases, followed by a 4 base library barcode, followed by 2 more random bases. Thus the `--bc-pattern` would be "NNNXXXXNN".

```
  Read:          TAGCCGGCTTTGCCCAATTGCCAAATTTTGGGGCCCCTATGAGCTAG
  Barcode:       NNNXXXXNN
                     |
                     v
                 TAGCCGGCT
                     |
                     V
       random-> TAG CCGG CT <- random
                     ^
                     |
                  library

```

(continues on next page)

(continued from previous page)

```
Processed read: CCGGTTGCCCAATTGCCAAATTTTGGGGCCCCTATGAGCTAG
                ^^^^
```

The processed reads could then be passed to a demultiplexing tool to deal with the library part of the barcode.

Since the file we have downloaded contains only one library, here we will treat the whole barcode as a UMI, and so the pattern will contain only Ns.

```
$ umi_tools extract --stdin=example.fastq.gz --bc-pattern=NNNNNNNNN --log=processed.
↪log --stdout processed.fastq.gz
```

Note that `extract` can output to a gzipped or uncompressed file depending on the file extension. It can also output to `stdout` if no output is specified. A log file is saved containing the parameters with which `extract` was run and the frequency of each UMI encountered. This can be redirected with `--log` or suppressed with `--supress-stats` (run parameters are still output).

2.4 Step 4: Mapping

The next step is to map the reads (in real life, you might also want to demultiplex, trim and quality filter the reads). Below we will use `bowtie` to map the reads to the mouse genome and `samtools` to create a BAM file from the results. If you don't wish to spend the time doing this, or don't have access to `bowtie` or `samtools` (or suitable alternatives), we provide a premapped BAM file (see command at the end of this step).

First map the reads with your favorite read mapper, here `bowtie`, using parameters from the paper which we stole the sample from. This assumes the mm9 `bowtie` index and fasta are in your current directory.

```
$ bowtie --threads 4 -v 2 -m 10 -a mm9 <( gunzip < processed.fastq.gz ) --sam >␣
↪mapped.sam
```

Next we need to convert the SAM file to BAM (actually `dedup` will use SAM files for single ended analysis, but it's much slower).

```
$ samtools import mm9.fa mapped.sam mapped.bam
```

The BAM now needs to be sorted and indexed:

```
$ samtools sort mapped.bam -o example.bam
$ samtools index example.bam
```

If you want to skip the mapping, you can get the file here. It will still need indexing (see "samtools index" command above):

```
$ wget https://github.com/CGATOxford/UMI-tools/releases/download/v0.2.3/example.bam
```

Again for macOS use the following to download:

```
$ curl -L "https://github.com/CGATOxford/UMI-tools/releases/download/v0.2.3/example.
↪bam" -o "example.bam"
```

2.5 Step 5: Deduplication

Now that we have a mapped, sorted, indexed file, we can proceed to run the deduplication procedure on it:

```
$ umi_tools dedup -I example.bam --output-stats=deduplicated -S deduplicated.bam
```

The `--output-stats` option is optional, but selecting it will provide a range of statistics about the run. One of the most interesting is the distribution of edit distances (here named deduplicated_edit_distance.tsv). The content of this file after running the above will look something like:

The first two columns show the distribution of average edit distances between UMIs found at a single base in the genome after deduplication with the directional-adjacency method (the default). Thus in the third line we see that there are 2 bases in the genome where the average edit distance between the UMIs found at that base is 1. The second column is what we would expect to see if UMIs were randomly distributed between mapping locations (taking into account any biases in the overall usage of particular UMI sequences). The last two columns the same, but for the naive `unique` deduplication method where every UMI is assumed to represent an independent molecule in the biological sample. Looking at the third row, we see that there are 1167 positions where the average edit distance between UMIs is 1, whereas in the random null (in the final column) we would only expect to see 33 such bases.

The statistics options signficantly reduce the speed at which deduplication is performed and increase the memory usage. If time or memory usage is an issue, try running without the `--output-stats` option.

2.6 Common variations

2.6.1 Paired-end sequencing

If paired-end sequencing has been performed, it is necessary to make sure that the UMI sequence is added to both reads. When processing, provide the second read like so:

```
$ umi_tools extract -I pair.1.fastq.gz --bc-pattern=NNNXXXXNN \
  --read2-in=pair.2.fastq.gz --stdout=processed.1.fastq.gz \
  --read2-out=processed.2.fastq.gz
```

This assumes the UMI is on the 5' end of read1. For other possibilities (such as a UMI on both reads) see `umi_tools extract --help`. After paired-end mapping, paired end deduplication can be achieved by adding the `--paired` option to the call to `dedup`:

```
$ umi_tools dedup -I mapped.bam --paired -S deduplicated.bam
```

Paired deduplicating is signficantly slower and more memory intensive than single-ended.

2.6.2 Read grouping

For some applications it may be neccessary to mark the duplicates but retain all reads, for example, where the PCR duplicates are used to correct sequence errors by generating a consensus sequence. In these cases, the *group* command can be used to mark each read with its read group. Optionally a flatfile detailing the read groups and read identifiers can also be output using the `--group-out` option:

```
$ umi_tools group -I mapped.bam --paired --group-out=groups.tsv --output-bam -S␣
↪mapped_grouped.bam
```

The output bam will contain two tags: UG = read group id, BX = read group UMI. The tag containing the read group UMI can be modified with the `--umi-group-tag` option.

The groups flatfile contains the following columns:

- read_id

- contig
- position
- umi = raw umi
- umi_count = how many times was this umi observed at the same alignment coordinates
- final_umi = the error corrected umi
- final_umi_count = how many times was the umi observed at the same alignment coordinates, inc. error correction
- unique_id = the unique identifier for this group

Example UMI extraction:

In the case above the UMIs are extracted with the pattern –bc-pattern=NNXXXXNN. Below is an example of how the fastq should be formatted following extraction:

UMI is bases 3-7, bases 1-2 and 7-8 are the sample barcode and need to be removed

```
@HISEQ:87:00000000 read1
AAGGTTGCTGATTGGATGGGCTAG
+
DA1AEBFGGCG01DFH00B1FF0B
```

will become:

```
 @HISEQ:87:00000000_GGTT read1
 TGATTGGATGGGCTAG
 +
 1AFGGCG01DFH00B1
```

2.6.3 Other options

See `umi_tools extract --help` and `umi_tools dedup --help` for details of futher possibilities.

CHAPTER 3

The network-based deduplication methods

`dedup` enables deduplication of UMIs by a variety of schemes. These are explained in the UMI-tools publication (open access). A modified Figure 1e from the publication is reproduced below.

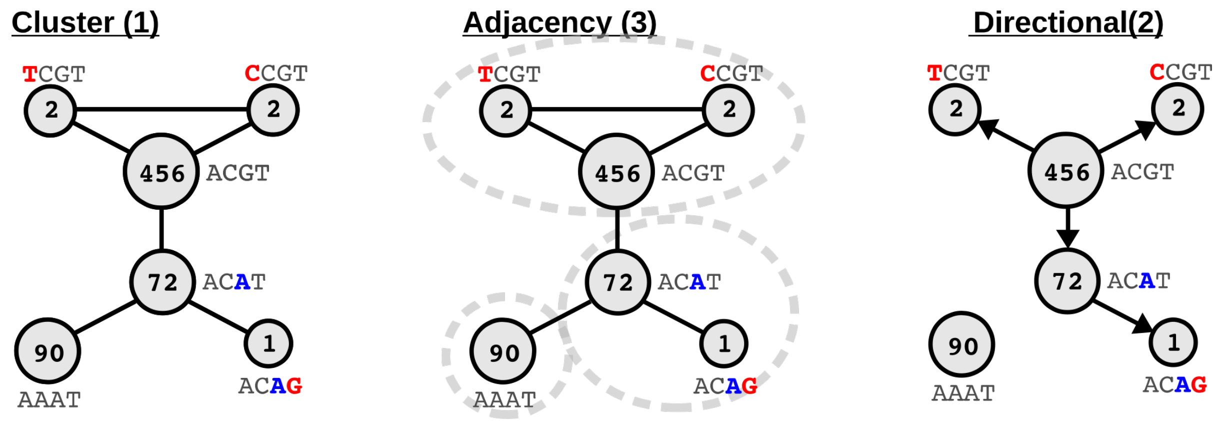


Fig. 1: Schematic representation of UMI deduplication by the 6 methods.

All methods generate read groups which are inferred to represent duplicate reads from a single unique molecule prior to PCR. Where the read group contains more than one UMI, the UMI with the highest frequency is selected as the representative UMI for the group. Ties are broken randomly.

The simplest methods, *unique* and *percentile*, group reads with the exact same UMI.

The network-based methods, *cluster*, *adjacency* and *directional*, build networks where nodes are UMIs and edges connect UMIs with an edit distance <= threshold (usually 1). The groups of reads are then defined from the network in a method-specific manner.

cluster: Form networks of connected UMIs (based on hamming distance threshold). Each connected component is a read group. In the above example, all the UMIs are contained in a single connected component and thus there is one read group containing all reads, with ACGT as the 'selected' UMI.

adjacency: Form networks as above. For each connected component, select the node (UMI) with the highest counts. Visit all nodes one edge away. If all nodes have been visited, stop. Otherwise, select the top 2 nodes with highest counts and visit all nodes one edge away. Repeat process until all nodes have been visited. Nodes which are not one of the n selected nodes are placed in a read group with the selected node with the highest counts for which there is an edge.

In the example above, ACGT (456) is selected first. Visiting all nodes one edge away leaves AAAT and ACAG unvisted. AAAT (90) is then additionally selected. Visiting all nodes one edge away from the selected nodes now leaves ACAG unvisited. Finally, ACAT (72) is additionally selected. Visiting all nodes one edge away now leaves no nodes unvisited. TCGT and CCGT are assigned to the ACGT read group, AAAT is in read group by itself and ACAG is assigned to the ACAT read group.

directional (default): Form networks with edges defined based on hamming distance threshold and node A counts >= (2 * node B counts) - 1. Each connected component is a read group, with the node with the highest counts selected as the top node for the component. In the example above, the directional edges yield two connected components. One with AAAT by itself and the other with the remaining UMIs with ACGT as the selected node.

CHAPTER 4

Single cell tutorial

Important update: We now recommend the use of `alevin` for droplet-based scRNA-Seq (e.g 10X, inDrop etc). `alevin` extends the directional method used in `UMI-tools` to correct UMI errors with droplet scRNA-Seq within a framework that also enables quantification using multi-mapped reads. `alevin` is an accurate, fast and convenient end-to-end tool to go from fastq -> count matrix. See alevin documentation and alevin pre-print for more information

This tutorial will go through an end to end analysis for single cell analysis using UMI-tools. Before you start, you will need:

- An installed copy of UMI-tools (see the installation guide)
- The STAR aligner
- A STAR index for the human genome
- A transcriptome annotation.
- The Subread package, version 1.5.3 or greater.

This tutorial will use an example dataset generated by 10X Genomics on their Chromium platform. At the end of the tutorial we cover variations on this workflow. We will continue to expand this section and cover other other data types here as well.

With 4 cores at your disposal, the whole thing should be do-able in less than an hour for this dataset.

The below assumes you are working in a linux environment. Some changes might be necessary if you are working in OSX, notably `wget` should be replace with `curl -O`

Outline of the process:

4.1 TL;DR

```
#! /bin/env bash
# Step 1: get data
wget http://cf.10xgenomics.com/samples/cell-exp/1.3.0/hgmm_100/hgmm_100_fastqs.tar;
tar -xf hgmm_100_fastqs.tar;
```

(continues on next page)

(continued from previous page)

```
cat fastqs/hgmm_100_S1_L00?_R1_001.fastq.gz > hgmm_100_R1.fastq.gz;
cat fastqs/hgmm_100_S1_L00?_R2_001.fastq.gz > hgmm_100_R2.fastq.gz;

# Step 2: Identify correct cell barcodes
umi_tools whitelist --stdin hgmm_100_R1.fastq.gz \
                    --bc-pattern=CCCCCCCCCCCCCCCCNNNNNNNNNN \
                    --set-cell-number=100 \
                    --log2stderr > whitelist.txt;

# Step 3: Extract barcdoes and UMIs and add to read names
umi_tools extract --bc-pattern=CCCCCCCCCCCCCCCCNNNNNNNNNN \
                  --stdin hgmm_100_R1.fastq.gz \
                  --stdout hgmm_100_R1_extracted.fastq.gz \
                  --read2-in hgmm_100_R2.fastq.gz \
                  --read2-out=hgmm_100_R2_extracted.fastq.gz \
                  --whitelist=whitelist.txt;
# Step 4: Map reads
STAR --runThreadN 4 \
     --genomeDir hg38_noalt_junc85_99.dir \
     --readFilesIn hgmm_100_R2_extracted.fastq.gz \
     --readFilesCommand zcat \
     --outFilterMultimapNmax 1 \
     --outSAMtype BAM SortedByCoordinate;

# Step 5: Assign reads to genes
featureCounts -a geneset.gtf \
              -o gene_assigned \
              -R BAM Aligned.sortedByCoord.out.bam \
              -T 4;
samtools sort Aligned.sortedByCoord.out.bam.featureCounts.bam -o assigned_sorted.bam;
samtools index assigned_sorted.bam;

# Step 6: Count UMIs per gene per cell
umi_tools count --per-gene --gene-tag=XT --assigned-status-tag=XS --per-cell -I␣
↪assigned_sorted.bam -S counts.tsv.gz
```

4.2 Step 1: Obtaining the data

The dataset was produced from around 100 cells from a mixture of human and mouse cells. The download is 760MB, so make sure you've got space. It can be downloaded thus:

```
$ wget http://cf.10xgenomics.com/samples/cell-exp/1.3.0/hgmm_100/hgmm_100_fastqs.tar
$ tar -x hgmm_100_fastqs.tar
```

out of the archive you will get a directory called `fastqs` that will contain data from 8 sequencing lanes, with three files for each lane: a read 1 file, a read 2 file and file containing the Sample Barcodes. We are not interested in the sample barcodes files, but for the read 1 and read2 files, we need to combine the eight lanes into one. By the magic of gzip, we can do with `cat`:

```
$ cat fastqs/hgmm_100_S1_L00?_R1_001.fastq.gz > hgmm_100_R1.fastq.gz
$ cat fastqs/hgmm_100_S1_L00?_R2_001.fastq.gz > hgmm_100_R2.fastq.gz
```

If we look at the content of of the files we will see that the first read in the pair contains 26nt, which correspond to a

16nt Cell barcode (CB) and 10nt the Unique Molecular Identifier (UMI):

```
$ zcat hgmm_100_R1.fastq.gz | head -n2
@ST-K00126:308:HFLYFBBXX:1:1101:25834:1173 1:N:0:NACCACCA
NGGGTCAGTCTAGTGTGGCGATTCAC
+
#AAFFJJJJJJJJJJJJJJJJJJJJJ

-------CB-------|---UMI---
```

We will need to remember this for the next two steps

4.3 Step 2: Identifying the real cells

Cell barcodes are short nucleotide sequences, very much like UMIs, except instead of identifying independent molecules, they identify independent cells. We generally observe more of them in an experiment than there were cells. This could be for several reasons, including sequencing or PCR errors and the sequencing of empty droplets or those containing contaminants. Thus we must identify which cell barcodes we wish to use downstream. UMI-Tools `whitelist` command is used to produce a list of CB to use downstream.

`whitelist` currently allows the common method of taking the top X most abundant barcodes. X can be estimated automatically from the data using the `knee` method (for more detail see this blog post). However, it is just an estimate and for this data we've been told that there were 100 cells, so we can just supply that number (see variations section for performing the estimation for data sets where cell number is unknown).

```
umi_tools whitelist --stdin hgmm_100_R1.fastq.gz \
                    --bc-pattern=CCCCCCCCCCCCCCCCNNNNNNNNNN \
                    --set-cell-number=100 \
                    --log2stderr > whitelist.txt
```

A couple of things about this command:

4.3.1 Input files

Firstly, note that FASTQ file that contains the barcodes is passed to `--stdin`, this tells `whitelist` to read the data from this file rather than from stdin. This of course means you could pass data using standard pipes.

4.3.2 Plots

The `--plot-prefix` option tells `whitelist` to output summary plots for the frequency of each CB. We always recommend running with this option in order to visualise whether the number of cells accepted seems reasonable. See [Variations](Automatic estimation of cell number) for a more complete explanation of what these plots show.

4.3.3 Specifying barcode locations

Second, the `--bc-pattern`. This tells `whitelist` where to find the CB and UMI in the read sequence. By default we assume the barcodes are at the 5' end of the read (this can be changed with `--3prime`). We then use `C` characters to show where CB bases are and `N` characters to show were UMI bases are. Thus, in the above we have 16 `C`s followed by 10 `N`s to denote that the first 16 bases of the read are CB bases and the second 16 are UMI bases. Alternatively, you can also define the barcode pattern using a regex instead (–extract-method=regex) in which there are named groups to define the positions for the cell barcode and UMI. For example we would change the above command to:

```
umi_tools whitelist --stdin hgmm_100_R1.fastq.gz \
                    --bc-pattern='(?P<cell_1>.{16})(?P<umi_1>.{10})' \
                    --extract-method=regex \
                    --set-cell-number=100 \
                    --log2stderr > whitelist.txt
```

This interface is very powerful and flexible and allows for the specification of all sorts of interesting things, like variable length CBs (UMIs do have to be a fixed length) and tolerant linker sequences (see the inDrop example in *Variations* and `umi_tools whitelist --help`).

4.3.4 Specifying outputs

By default UMI-tools outputs everything, final output, logging info and progress report, to the standard out pipe. Downstream sections will take this output just fine, but you might want to put the log somewhere else, like a separate file, or on the terminal. This can be achieved in one of several ways:

- You could redirect the final output using the `--stdout` or `-S` options. The final output will be directed to a file and the log will continue to come on stdout.
- You could redirect the log to a file with `--stdlog` or `-L`. The log and progress will be saved to a file and the final output sent to the standard out.
- You can redirect the log and progress to the standard err using `--log2stderr` as above.
- You can switch off the log/progress with `-v 0`

This applies to all the UMI-Tools commands.

4.3.5 Using UMI counts rather than read counts in umi_tools whitelist

Many published protocols rank CBs by the number of reads the CBs appear in. However you could also use the number of unique UMIs a CB is associated with. Note that this is still an approximation to the number of transcripts captured because the same UMI could be associated with two different transcripts and be counted as independent. Activate this with `--method=umis`.

4.3.6 Contents of `whitelist.txt`

The output of the whitelist command is a table containing the accepted CBs. It has four columns:

1. The accepted CB
2. Comma separated list of other CBs within an edit distance of the CB in columns 1 and >1 edit away from any other accepted CB.
3. The abundance (read or UMI count) of the accepted.
4. Comma separated list of abundances for the CBs in column 2

e.g:

```
$ head whitelist.txt
AAAGATGAGAAACGAG        AAAAATGAGAAACGAG,AAACATGAGAAACGAG,...        53122        4,6,
↪...
AAAGCAAGTACCTACA        AAAACAAGTACCTACA,AAACCAAGTACCTACA,...        36255        2,3,
↪...
AACACGTCAGCGTAAG        AAAACGTCAGCGTAAG,AACAAGTCAGCGTAAG,...        53133        4,
↪11,...
```

(I have truncated columns 2 and 4 for easy viewing)

As we asked for 100 cells, this file should contain 100 lines:

```
$ wc -l whitelist.txt
100 whitelist.txt
```

4.4 Step 3: Extract the barcodes and filter the reads

The next step is to extract the CB and UMI from Read 1 and add it to the Read 2 read name. We will also filter out reads that do not match one of the accepted cell barcode.

The most basic form of this is executed with:

```
umi_tools extract --bc-pattern=CCCCCCCCCCCCCCCCNNNNNNNNNN \
                  --stdin hgmm_100_R1.fastq.gz \
                  --stdout hgmm_100_R1_extracted.fastq.gz \
                  --read2-in hgmm_100_R2.fastq.gz \
                  --read2-out=hgmm_100_R2_extracted.fastq.gz \
                  --whitelist=whitelist.txt
```

The `--bc-pattern` and `--stdin` options are as before. Note that we send the standard out (which contains the extract Read 1s) to the a file (add `.gz` to the end of the name will trigger automatic compression). `--read2-in` and `--read2-out` specify the output files for read 2. Finally `--whitelist` passes the list of accepted CBs generated in the previous step and tells `extract` to only output those reads that contain accepted CBs.

As the end of the log makes clear, the command above took about 20 minutes

```
2017-08-10 16:29:45,449 INFO Parsed 7100000 reads
2017-08-10 16:30:01,896 INFO Input Reads: 7197662
2017-08-10 16:30:01,896 INFO Reads output: 5963891
2017-08-10 16:30:01,896 INFO Filtered cell barcode: 1233771
# job finished in 1175 seconds at Thu Aug 10 16:30:01 2017 -- 1153.42 24.13  0.00  0.
↪00 -- f5227f66-4f36-4862-9c7a-87caf3f81870
```

There should now be two output files in your directory: `hgmm_100_R1_extracted.fastq.gz` and `hgmm_100_R2_extracted.fastq.gz`. Looking at the first read we see:

```
$ zcat hgmm_100_R1_extracted.fastq.gz | head -n4
@ST-K00126:308:HFLYFBBXX:1:1101:31345:1261_AACTCTTGTTCTGAAC_CGGTTGGGAT 1:N:0:NACCACCA

+
```

The reads are now empty because the CBs and UMIs have now been moved from the read sequence to the read name, and there is nothing left. We will not need this file again.

If we look at the Read 2:

```
$ zcat hgmm_100_R2_extracted.fastq.gz | head -n4
@ST-K00126:308:HFLYFBBXX:1:1101:31345:1261_AACTCTTGTTCTGAAC_CGGTTGGGAT 2:N:0:NACCACCA
CCTTTTTGGAACCAACAATAGCAGCTCCATTTCTGGAGTCTGGGTCTTCCGAGGCCAGGAGCTCGCCTTTCCGCCGAGCCCAGATTGGCAGGTGGACT
+
A<<AFAFFFJJJJFJJJFA<7<FJF-AAJF7-FFF<FA7AFFFJ-77<JJFFFJJJJFAJFJ7-7AJ-7-FJJJ--)7-77F-F--
↪AAAJAA-7-7F7
```

We can see that the CB and UMI from read 1 has been added to the name of read 2 as well.

These reads are now ready to map!

4.4.1 Discarding read 1

There are many single cell RNA-Seq techniques where read 1 only contains barcodes, as above. If we only want to write out the extracted read 2 file we can supply the `--read2-stdout` option to output read 2 on the stdout and discard read 1.

4.5 Step 4: Mapping reads

At this point there are two different ways we might proceed. We could map to the transcriptome, where each each contig represents a different transcript. Alternatively we could map to the genome and then assign reads to genes. While transcriptome mapping allows for easier downstream analysis, we recommend mapping to the genome because it doesn't force reads to map to transcripts when a better match outside an annotated gene exists. However, there are cases when transcriptome mapping should be favoured, and we discuss this in the *Variations* section.

We are going to use STAR here as our mapper, but you can substitute whichever splice-aware aligner you prefer. We are mapping to an index built on the human NCBI38 reference, with the alternate contigs removed (referred to by NCBI as the "analysis read set") using junctions from ENSEMBL85 and a 99nt overhang. We do not allow multimapping reads.

```
$ STAR --runThreadN 4 \
       --genomeDir hg38_noalt_junc85_99.dir \
       --readFilesIn hgmm_100_R2_extracted.fastq.gz \
       --readFilesCommand zcat \
       --outFilterMultimapNmax 1 \
       --outSAMtype BAM SortedByCoordinate
```

If we have a look at the stats for this, we will see that 63.02% of reads mapped (depending on exactly how you've built your STAR index you may get a slightly different number). For 8.1% of these this is because of mapping to more than one location. The rest is probably because the sample contains both mouse and human cells, but we are only aligning to the human genome. In fact, given that, the mapping rate looks a bit on the high side: we are probably mapping some reads to the human genome that actually came from the mouse genome. Ideally we would perform this mapping using references that combined both the mouse and human genomes/transcriptomes.

On our systems, using 4 threads, this mapping takes about 7 minutes.

4.6 Step 5: Assigning reads to genes

Many of the single cell techniques that use UMIs are 3' tag count methods. This is important because it means that the UMIs are generally added before PCR and then fragmentation. This means that two reads from the same gene might have different mapping locations and still represent duplicates, as long as they come from the same gene. Therefore, UMI-tools need to know which gene each read came from. When reads are aligned to the transcriptome, this is easy as it is given by the contig (and possibly a gene to transcript look up table). However, with a genome alignment we need to assign reads to genes.

We can do this with the `featureCounts` tool from the `subread` package. As well as outputting a table of (undeduplicated) counts, we can also instruct `featureCounts` to output a BAM with a new tag containing the identity of any gene the read maps to.

warning: This only works on featureCounts from subread 1.5.3 and above, which was released July 2017. If you already have subread it could well need updating

You will need a geneset to assign genes to. Here ours is called `geneset.gtf`. It is the same geneset used to generate the mapping index above, and is based on the GENCODE annotations.

```
$ featureCounts -a geneset.gtf -o gene_assigned -R BAM Aligned.sortedByCoord.out.bam -
↪T 4
```

Unfortunately the BAM output by featureCounts is no longer sorted, so we now need to sort the BAM.

```
$ samtools sort Aligned.sortedByCoord.out.bam.featureCounts.bam -o assigned_sorted.bam
$ samtools index assigned_sorted.bam
```

We now have a sorted, indexed set of alignments assigned to the appropriate genes and we can now count the number of distinct UMIs mapping to each gene in each cell.

4.7 Step 6: Counting molecules

We are finally ready to process the UMIs aligned to each gene in each cell to find the number of distinct, error corrected UMIs mapping to each gene.

```
$ umi_tools count --per-gene --gene-tag=XT --assigned-status-tag=XS --per-cell -I␣
↪assigned_sorted.bam -S counts.tsv.gz
```

Since we want to count the number of UMIs per gene, and the gene assignment is encoded in the XT tag, we use the `--per-gene --gene-tag=XT` options. FeatureCounts also uses the XS tag to describe the gene assignment so we use the `--assigned-status-tag=XS` option. Any reads with an assignment matching `^[__|Unassigned]` (default value for `--skip-tags-regex`) will be skipped. `--per-cell` tells UMI-tools to also consider the CB and produce a separate count for each cell. The counts are output in a table with three columns: the gene_id, the cell barcode and the count of deduplicated UMIs.

Running this command on this input takes 72 seconds on our system.

```
$ zcat counts.tsv.gz | head
gene        cell        count
ENSG00000000003       AAAGATGAGAAACGAG        3
ENSG00000000003       AACTCTTGTTCTGAAC        4
ENSG00000000003       ACACCGGGTACGACCC        2
ENSG00000000003       ACACTGAGTCGGGTCT        5
ENSG00000000003       ACTATCTCAAGGTGTG        2
ENSG00000000003       ACTGTCCCATATGGTC        11
ENSG00000000003       ACTGTCCTCATGCTCC        5
ENSG00000000003       AGAGCTTCACGACGAA        2
ENSG00000000003       AGCGGTCTCGATAGAA        2
```

If you'd rather have this as a matrix, which each cell as a sepearte column, you can specify `--wide-format-cell-counts`. You can also output deduplicated reads using the `dedup` command in place of `count`. You can even output all reads, with each assigned to an UMI group, using the `group` command, although see below for some important points.

4.8 Time and memory for larger data sets

While the above was pretty quick, 7 million reads over 100 cells is a pretty tiny dataset. The most memory intensive part was the `STAR` alignment, and this is unlikely to change with larger datasets. Below we provide some figures for larger datasets. These times will obviously vary depending on your setup, in particular many of these tasks are I/O bound, and the nature of your storage will make a big difference. Running on local storage, particularly if its SSD, will probably give substantially quicker performance. Conversely if storage is accessed over standard network connections, it will probably be much slower.

* Much of this time is spent compressing the output. Outputting to uncompressed files is significantly faster.

** memory used by `samtools sort` can be specified with the `-m` option.

*** memory used counting is significaitonly reduced by mapping to the transcriptome rather than the genome.

4.9 Variations

4.9.1 Barcode extraction for Drop-seq

For Drop-seq, the current protocol includes a 12bp CB, followed by an 8bp UMI. We can therefore extract the barcodes using the following options:

```
--extract-method=string
--bc-pattern=CCCCCCCCCCCCNNNNNNNN
```

4.9.2 Barcode extraction for inDrop

The CBs for inDrop are different lengths. Thus, we must use `--extract-method=regex` since we can no longer encode the barcode pattern in a simple string. If we look at the sequence of an inDrop read 1, we can demark the barcodes as follows:

```
TGAACATCTATGAGTGATTGCTTGTGACGCCTTGAGCCCATCCTGCTTTTTTTTT
---CB-1---|------ADAPTER--------|--CB-2-|-UMI-|poly(T)
```

The CB is split into two halves. CB-1 can be 8-12bp, the adapter sequence is always "GAGTGATTGCTTGTGACGCCTT", CB-2 is 8bp, UMI is 6bp and the read ends in a poly(T) seqeunce. In the case above, the barcode sequences are:

CB-1: TGAACATCTAT CB-2: GAGCCCAT CB (CB-1 + CB-2): TGAACATCTATGAGCCCAT UMI: CCTGCT

To encode this in the barcode pattern, we must provide the option `--extract-method=regex` and provide `--bc-pattern` with a regex with groups named accordingly. Each group must have a integer suffix: `--bc-pattern="(?P<cell_1>.{8,12})(?P<discard_1>GAGTGATTGCTTGTGACGCCTT)(?P<cell_2>.{8})(?P<umi_1>.{6})T{3}.*"`

This translates to: 8-12 characters (group="cell_1"), followed by "GAGTGATTGCTTGTGACGCCTT" (group="discard_1"), followed by 8 characters (group="cell_2"), followed by 6 characters (group="umi_1"), followed by 3 or more "T"s.

The other benefit of using a regex is that we can perform 'fuzzy'. For example, the inDrop adapter sequence is 22bp long so it may sometimes contain base calling errors. We can allow usp to two substituions like so:

`--bc-pattern="(?P<cell_1>.{8,12})(?P<discard_1>GAGTGATTGCTTGTGACGCCTT){s<=2}(?P<cell_2>.{8})(?P<umi_1>.{6})T{3}.*"`

It should be possible to encode any conceivable pattern of barcodes using a regex.

4.9.3 Automatic estimation of cell number

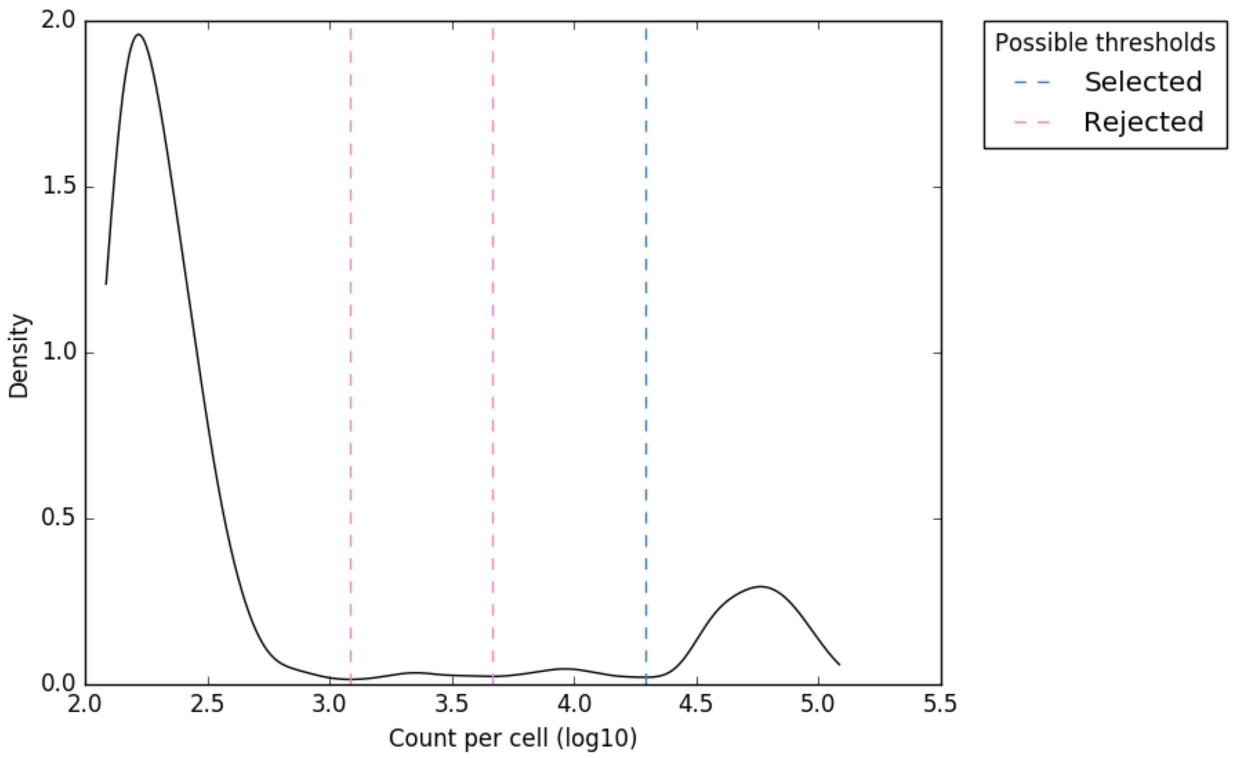
In *Step 2* above, we generated a whitelist of accepted ("true") CBs and provided the option `--set-cell-number=100` since we a prior belief about the number of cells we had sequenced. However, what if we didn't know how many cells we had actually sequenced? Well, in that case, we can leave out the `--set-cell-number` option and let `whitelist` estimate the number of true CBs automatically. Even better, if we have an estimate on the number of cells we have input into the system and we know the capture rate is at least 10% (this assumption holds true for inDrop and 10X Chromium platforms), we can provide the estimated number of input cells as an upper limit on the number of true CBs using the `--expect-cells` option. So, imagine with our experiment above, we estimated that around 200 cells were input into the system, we can generate the whitelist with the following command:

```
umi_tools whitelist --stdin hgmm_100_R1.fastq.gz \
                    --bc-pattern=CCCCCCCCCCCCCCCCNNNNNNNNNN \
                    --expect-cells=200 \
                    --plot-prefix=expect_whitelist \
                    --log2stderr > whitelist.txt
```

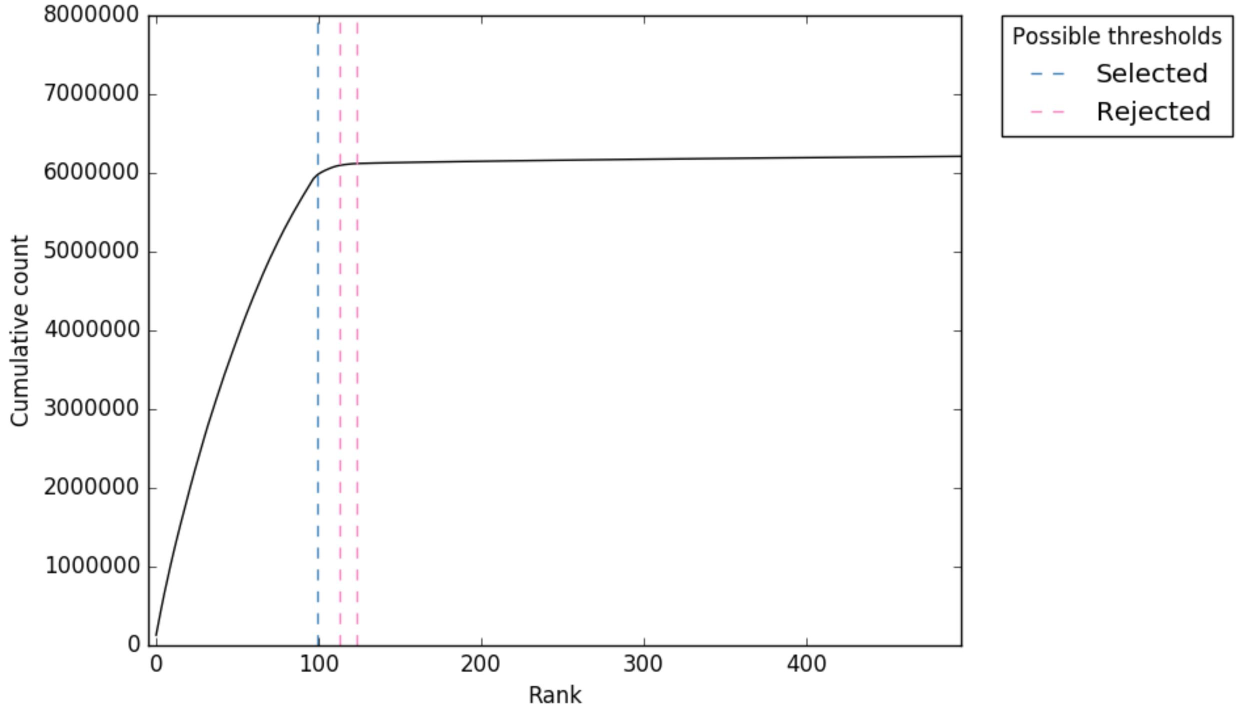
We can check how many CBs we accepted:

```
$ wc -l whitelist.txt
119 whitelist.txt
```

Based on our prior knowledge that there were ~100 true CBs, we've probably slightly over-estimated the number of CBs here. We can visualise this by looking at the following plots which present the same data in different ways to help assess the threshold:

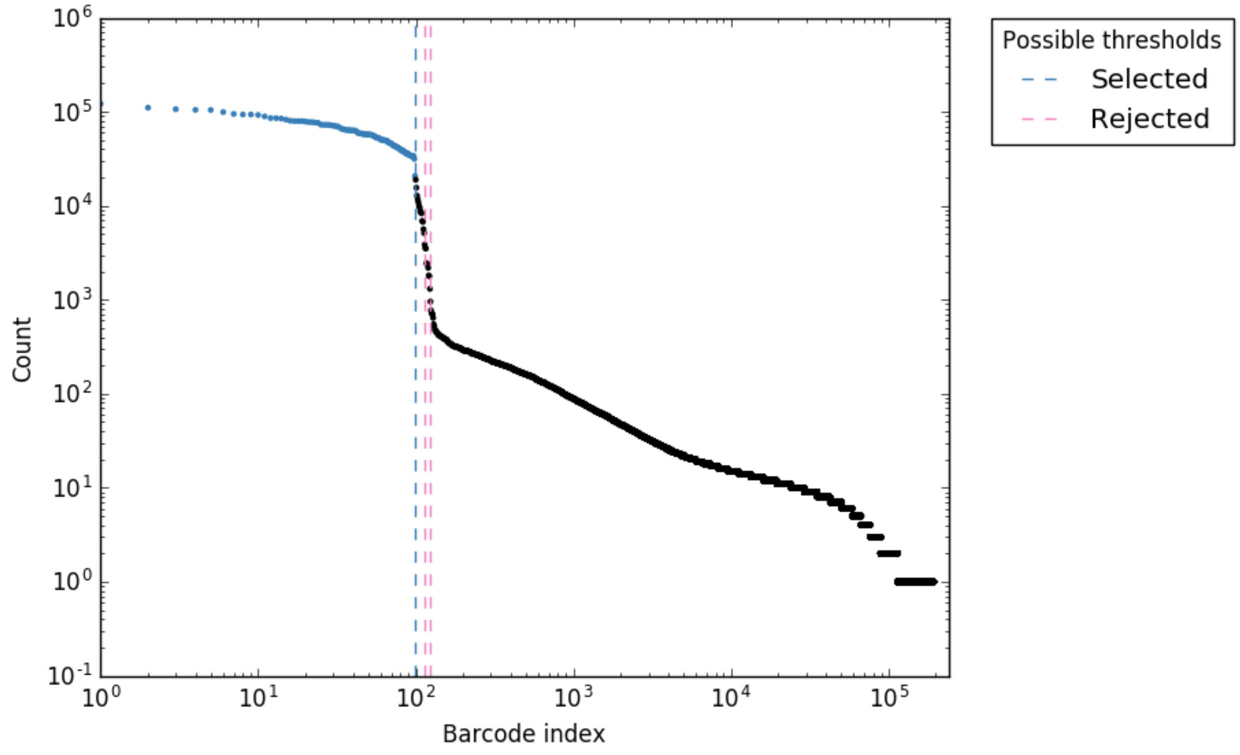


Counts distribution Figure 1. "expect_whitelist_cell_barcode_counts_density.png" Distribution of read counts per CB. The least abundant CBs are removed from this plot



Cumulative counts Figure 2. “expect_whitelist_cell_barcode_knee.png” Cumulative counts as number of accepted barcodes is increased

---



Counts per barcode Figure 3. “expect_whitelist_cell_barcode_counts.png” Logs counts per barcode (ranked)

For all plots, the 'Selected' local minima (or threshold) is shown by the dashed line. If we had other local minima which were 'Rejected' these would also be shown. The thresholds are also tabulated in the file `expect_whitelist_cell_thresholds.tsv`.

In Figure 1, we can see two clear peaks, with the right peak representing the true cell barcodes. In this case, the selected local minima appears to be in the correct place. This is confirmed in Figures 2 & 3, where the threshold appears to be at the 'knee' in the cumulative counts, and counts per barcode. So it looks like in this case, the automatic detection of the 'knee' has worked well. However, in some cases, the automatic selection may not work so well. In these case we can re-run `whitelist` and set the number of cells with the `--set-cell-number` option as described above.

In most cases, there will be many local minima and sometimes the wrong one may be selected. If so, you can insepect the `[PLOT_PREFIX]_cell_thresholds.tsv` file and re-run the `whitelist` command using the `--set-cell-number` option with the threshold which you have deemed to be most appropriate. We've not yet come across a case where no local minima identified near the 'knee', but if so, you can inspect the counts per barcode and cumulative counts figures and estimate a number to use with the `--set-cell-number` option.

4.9.4 Correction of cell barcodes

In *Step 2* above, we generated a whitelist of accepted ("true") CBs. By default, we also identified the CBs that were not accepted but were a single edit from a just one accepted barcode (column 2) in `whitelist.txt`. These are likely to be errors and can be corrected. By default, `extract` will only retain reads where the CB matches the whitelist. However, we can also accept these "error" CBs and correct the sequence to the accepted CB using the `--error-correct-cell` option.

4.9.5 Quality filtering and masking barcodes

In some cases, the UMI bases may have very low base call quality scores. There are three options to deal with these reads:

1. Retain
2. Discard
3. Mask

By defualt, `extract` takes option 1 and does not considered quality scores. If you want to take option 2 and discard these reads you can do so with `--quality-filter-threshold=[FILTER]`, whereby all reads where the UMI contains a base with a Phred scrore below `FILTER` are discarded. However, if you are concerned about low quality bases, our recommendation is that you take option 3 and use `--quality-filter-mask=[FILTER]` to replace all UMI bases with a Phred score below `FILTER` with an "N". UMIs with an "N" will be 1-edit distance away from their cognate error-free UMI sequence and will therefore be handled sensibly by the downstream UMI-tools commands.

4.9.6 Going back a step in the group>dedup>count proceedure

In this guide we have discussed the the UMI-tools `whitelist`, `extract` and `count` commands. There are two further UMI-tools commands which deal with BAM files, `group` and `dedup`. `group`, `dedup` and `count` can be thought of as performing sequential levels of analysis in the process of obtaining read counts per gene: First the UMIs are "grouped" according to the gene to which they are assigned, then the UMIs are "deduplicated" to leave a single UMI per group, then the UMIs per gene are "counted". In some cases, you may wish to stop the proceedure prior to counting to inspect the grouping or deduplication process. This can be easily achieved by simply replacing the `count` command with `dedup` or `group`, with a very few considerations to command specific options:

- `--wide-format-cell-counts` is a `count`-specific option

- When running `dedup`, you may want to include the `--output-stats` option
- To output a BAM with `group`, you need to supply the `--output-bam` option

warning: Running the `group` command with the `--per-gene` option in order to assess the groups used in the process of counting is very memory intensive, especially when the reads have been aligned to a genome, since all reads for a given contig in the BAM must be kept in memory as python objects until the whole contig has been parsed. This is not the case for `count` since we only retain the tally of reads. We recommend if you want to `group` together with `--per-gene` you used a transcriptome mapping (see below)

4.9.7 Mapping to the transcriptome rather than genome

Many single cell protocols recommend mapping to the transcriptome rather than then genome. This makes downstream processing easier. In paritcular it makes it easier to know when you've seen all the reads from a gene, so deduplication can happen. This leads to a reduced memory footprint, which is particularly obvious in `dedup` and `group`. However, there are problems.

1. When you map to the transcriptome almost all reads are multimapping as most sequence in a gene is part of more than one transcript. This makes it difficult to distinguish between genuninely multimapping reads and reads that just come from multiple transcripts of the same gene.
2. If we include these reads we also face the problem that reads will be double counted, and each time the deduplication is done, it will be done using a different set of other reads, thus a read might filtered as a duplicate for one transcript but not another.
3. Reads that should map elsewhere might be forced to map to the transcriptome.

It turns out we can solve either problems 1 + 2 or problem 3. This is why we recommend mapping to the genome and annotating with `featureCounts` above.

Method 1 We solve problems 1 and 2 by using a collapsed transcriptome where all transcripts for a gene are merged into a single model. This is similar to the superTranscript concept proposed by Davidison *et al*. You can download our merged transcriptome annotations for human GENCODE 26 and mouse GENCODE 15 from ==here and here==, or see below for instructions on generating your own. You should generate FASTA from these, and then genome indexes for your favorite spliced aligner. After alignment the contigs of the aligned BAM file will contain the gene_id. You can then run `count`, `dedup` or `group` with:

```
$ umi_tools [COMMAND] --per-contig -I aligned.bam
```

You'll find that after the effort of generating the genome and its indexes, the `umi_tools` step is quicker and uses less memory, and the `featureCounts` step is unecessary.

Method 2 The above doesn't solve the third problem above however. We can get transcriptome aligned reads without the problem three using `STAR`. STAR has an output mode `--quantMode TranscriptomeSAM` where reads are mapped to the genome, but then their mapping coordinates are translated to the transcriptome and output in that form. For this you would pass `STAR` a normal transcriptome (i.e. not one of the collasped ones from above) using `--sjdbGTFfile` option. You could then sort the `BAM` file and run `umi_tools` with:

```
$ umi_tools [COMMAND] --per-contig --per-gene --gene-transcript-map=[MAPFILE]
```

where `MAPFILE` is a tab-sepearted text file mapping transcript_ids (in column 1) to gene_ids (in column 2).

However, since we might encounter the same read twice (mapped to different transcripts of the same gene), UMI counts might be inaccurate.

Ideally we'd like to combine these two methods: map to the genome and then have STAR translate to merged transcriptome coordinates. Unfortunately `STAR` will only transfer reads that fit the intron/exon structure of the collapsed gene, which looses many splice junctions, and therefore about 10% of reads are lost.

4.9.8 Generating merge transcriptome annotations

We generated the above linked collapsed transcriptomes using the `cgat` pacakge and files from gencode. This involves repeated use of the `gtf2gtf` tool, which first merges the transcripts, and then sets the `transcript_id` of every exon to be equal to the `gene_id`.

For example, for the human transcriptome:

```
$ wget ftp://ftp.sanger.ac.uk/pub/gencode/Gencode_human/release_26/gencode.v26.
↪annotation.gtf.gz
$   cgat gtf2gtf --method=merge-exons -I gencode.v26.annotation.gtf.gz \
  | cgat gtf2gtf --method=set-transcript-to-gene -S hg38_genocode26_merged.gtf.gz
```

CHAPTER 5

Specifiying cell barcode and UMI

There are two ways to specify the location of cell barcodes and UMI bases when using `whitelist` and `extract`: string/basic (default) or regex. Sting or basic mode is simple and easy to understand and can be used for most techniques, like iCLIP, 10x chromium or Drop-seq. The regex mode is a little harder to explain, but allows a great deal of flexibility in specifying the layout of your read.

5.1 Basic string mode

In the default basic mode we specify the location of barcodes and UMIs in a read using a simple string. `N`s specify the location of bases to be treated as UMIs, `C`s as bases to treated as cell barcode and `X`s as bases that are neither and that should be retained on the read. By default this pattern is applied to the 5' end of the read, but we can tell `extract` to look on the 3' end of the read using `--3-prime`.

So for example in 10x Chromium, read one consists of 16 bases of cell barcode followed by 10 bases of UMI, so the correct pattern to use is `CCCCCCCCCCCCCCCCNNNNNNNNNN`:

```
@ST-K00126:308:HFLYFBBXX:1:1101:25834:1173 1:N:0:NACCACCA
NGGGTCAGTCTAGTGTGGCGATTCAC
+
#AAFFJJJJJJJJJJJJJJJJJJJJJ

-------CB-------|---UMI---
```

To understand how `X`s are treated, consider the read layout for an iCLIP experiment following the protocol of Mueller-Mcnicoll et al, *Genes Dev* (2016) 30: 553. Here at the start of each read 1 there are three UMI bases. This is then followed by 4 bases representing a library barcode, and then there are two more UMI bases. The correct pattern here is `NNNXXXXNN`.

```
  Read:          TAGCCGGCTTTGCCCAATTGCCAAATTTTGGGGCCCCTATGAGCTAG
  Barcode:       NNNXXXXNN
                     |
                     v
```

(continues on next page)

(continued from previous page)

```
                  TAGCCGGCT
                      |
                      V
        random-> TAG CCGG CT <- random
                      ^
                      |
                   library


Processed read: CCGGTTGCCCAATTGCCAAATTTTGGGGCCCCTATGAGCTAG
                ^^^^
```

5.2 Regex (regular expression) mode

Regexes provide a more flexible way to describe the pattern of UMI +/- cell barcode in the reads and can be used with (`--extract-method=regex`). Its needed for techniques such as inDrop, SLiT-seq and ddSeq among others. If you know nothing about regular expressions, see *regular what-nows?* below.

Regexes provide a number of advantages over the simpler "string" extraction method:

1. Reads not matching the regex will be discarded. E.g. this can be used to filter reads which do not contain adapter sequences.
2. Variable cell barcode lengths can be encoded.
3. Finally, regexes allow fuzzy matching (error-aware) in, for example, adaptors. Note that to enable fuzzy matching, `umi_tools` uses the `regex` library rather than the more standard `re` library.

The regex must contain named capture groups to define how the barcodes are encoded in the read. Named capture groups are a non-standard regex feature available in the python regex dialect. A capture group is a sub part of a pattern enclosed in brackets. Matches to sub-pattern are "captured" and can be extracted for reuse. So the pattern `(.{4})TTTTT` will match any four characters followed by 5 Ts, and return what those 4 characters were. In most cases we refer to a capture group by its positions (1st group, 2nd group etc). *Named* capture groups allow use to give names to each group using a `(?P<name>` syntax. Thus, `(?P<prefix>.{4})TTTTT` matches the same as the pattern above, but the captured group is given the name "prefix".

When passing a regex to `whitelist`/`extract`, the allowable groups in the regex are:

- `umi_n` = UMI positions, where n can be any value (required)
- `cell_n` = cell barcode positions, where n can be any value (optional)
- `discard_n` = positions to discard, where n can be any value (optional)

We specify fuzzy matching by adding something like `{s<=X}` after a group. This specifies that the group should be matched with up to X **s**ubstitutions. The allowed error types are `s`: substitutions, `i`: insertions, `d`:deletions, `e`: any error (or the Levenshtein distance). See the `regex` package documentation for more details.

5.2.1 Example: extracting UMIs

Suppose we have a FASTQ file with reads with a 4 base UMI at their 5' end and a 4 base UMI at their 3' end. For example:

```
@EWSim-1.1-umi5-reada-umix
AAAAATGGCATCCACCGATTTCTCCAAGATTGAACGTA
+
IHHHIIIHHHHIHHIIHIIHIIHHIHHHIIIIIIHIII
@EWSim-2.1-umi5-reada-umiy
AAAAATGGCATCCACCGATTTCTCCAAGATTGAAATAT
+
GABIFCD@ABEAA?EAHH?AFA@IEDGGA@CCDGFI@C
@EWSim-3.1-umi5-readb-umix
AAAATCTAGATTAGAAAGATTGACCTCATTAACGTA
+
E?IHIIH@DAC?CIAGDCIIEF@BIEDG?EDH@I?A
```

To extract these UMIs, we can use the regular expression:

```
^(?P<umi_1>.{4}).+(?P<umi_2>.{4})$
```

To break this down:

- `^(?P<umi_1>.{4})` matches the 4 bases at the start of the read and extracts this as a UMI, named `<umi_1>`.
- `.+` matches all the bases between the first 4 bases and last 4 bases (`+` indicates that there must be at least one).
- `(?P<umi_2>.{4})$` matches the 4 bases at the end of the read and extracts this as a UMI, named `<umi_2>`.

Applying this to each example read in our FASTQ file would give the following values:

```
@EWSim-1.1-umi5-reada-umix
AAAAATGGCATCCACCGATTTCTCCAAGATTGAACGTA # Read
AAAA                                   # <umi_1>
    ATGGCATCCACCGATTTCTCCAAGATTGAA     # Read-post UMI extraction
                                  CGTA # <umi_2>
@EWSim-2.1-umi5-reada-umiy
AAAAATGGCATCCACCGATTTCTCCAAGATTGAAATAT # Read
AAAA                                   # <umi_1>
    ATGGCATCCACCGATTTCTCCAAGATTGAA     # Read-post UMI extraction
                                  ATAT # <umi_2>
@EWSim-3.1-umi5-readb-umix
AAAATCTAGATTAGAAAGATTGACCTCATTAACGTA   # Read
AAAA                                   # <umi_1>
    TCTAGATTAGAAAGATTGACCTCATTAA       # Read-post UMI extraction
                                CGTA   # <umi_2>
```

If we were to use `umi_tools extract` to extract these UMIs, using this regex, our resulting FASTQ file would look like the following:

```
@EWSim-1.1-umi5-reada-umix_AAAACGTA
ATGGCATCCACCGATTTCTCCAAGATTGAA
+
IIIHHHHIHHIIHIIHIIHHIHHHIIIIII
@EWSim-2.1-umi5-reada-umiy_AAAAATAT
ATGGCATCCACCGATTTCTCCAAGATTGAA
+
FCD@ABEAA?EAHH?AFA@IEDGGA@CCDG
@EWSim-3.1-umi5-readb-umix_AAAACGTA
TCTAGATTAGAAAGATTGACCTCATTAA
+
IIH@DAC?CIAGDCIIEF@BIEDG?EDH
```

The extracted UMIs are appended together (`<umi_1><umi2>`) and are appended to the end of the head header with a delimiter, _.

5.2.2 Example: extracting barcodes and UMIs

Suppose we have a multiplexed FASTQ file with reads with a 4 base UMI at their 5' end and a 4 base UMI at their 3' end, and with a 3 base barcode at the 3' end. For example:

```
@EWSim-1.1-umi5-reada-umix-bar0.0
AAAAATGGCATCCACCGATTTCTCCAAGATTGAACGTAACG
+
IHIHIIIHHIIHHIHIIHHHIIHIHIIIHIIIHHIIIIHIH
@EWSim-2.1-umi5-reada-umiy-bar1.0
AAAAATGGCATCCACCGATTTCTCCAAGATTGAAATATGAC
+
?I?DDIECDEIA?GBGEGGB?EG@GDFEG?GDB?A?IDIDA
@EWSim-3.1-umi5-readb-umix-bar2.0
AAAATCTAGATTAGAAAGATTGACCTCATTAACGTACGA
+
?I?CGGIFCICHHCFFDFB@DDGA??BFDDHABBIF@GC
```

To extract these UMIs, and also the barcode, we can use the regular expression:

```
^(?P<umi_1>.{4}).+(?P<umi_2>.{4})(?P<cell_1>.{3})$
```

To break this down:

- `^(?P<umi_1>.{4})` matches the 4 bases at the start of the read and extracts this as a UMI, named `<umi_1>`.
- `.+` matches all the bases between the first 4 bases and last 7 bases (+ indicates that there must be at least one).
- `(?P<umi_2>.{4})$` matches the first 4 of the 7 bases at the end of the read and extracts this as a UMI, named `<umi_2>`.
- `(?P<cell_1>.{4})$` matches the 3 bases at the end of the read and extracts this as a barcode, named `<cell_1>`.

Applying this to each example read in our multiplexed FASTQ file would give the following values:

```
@EWSim-1.1-umi5-reada-umix-bar0.0
AAAAATGGCATCCACCGATTTCTCCAAGATTGAACGTAACG # Read
AAAA                                      # <umi_1>
    ATGGCATCCACCGATTTCTCCAAGATTGAA        # Read-post UMI extraction
                                  CGTA    # <umi_2>
                                      ACG # <cell_1>
@EWSim-2.1-umi5-reada-umiy-bar1.0
AAAAATGGCATCCACCGATTTCTCCAAGATTGAAATATGAC # Read
AAAA                                      # <umi_1>
    ATGGCATCCACCGATTTCTCCAAGATTGAA        # Read-post UMI extraction
                                  ATAT    # <umi_2>
                                      GAC # <cell_1>
@EWSim-3.1-umi5-readb-umix-bar2.0
AAAATCTAGATTAGAAAGATTGACCTCATTAACGTACGA   # Read
AAAA                                      # <umi_1>
    TCTAGATTAGAAAGATTGACCTCATTAA        # Read-post UMI extraction
                                CGTA      # <umi_2>
                                    CGA   # <cell_1>
```

If we were to use `umi_tools extract` to extract these UMIs and barcode, using this regex, our resulting FASTQ file would look like the following:

```
@EWSim-1.1-umi5-reada-umix-bar0.0_ACG_AAAACGTA
ATGGCATCCACCGATTTCTCCAAGATTGAA
+
IIIHHIIHHIHIIHHHIIHIHIIIHIIIHH
@EWSim-2.1-umi5-reada-umiy-bar1.0_GAC_AAAAATAT
ATGGCATCCACCGATTTCTCCAAGATTGAA
+
DIECDEIA?GBGEGGB?EG@GDFEG?GDB?
@EWSim-3.1-umi5-readb-umix-bar2.0_CGA_AAAACGTA
TCTAGATTAGAAAGATTGACCTCATTAA
+
GGIFCICHHCFFDFB@DDGA??BFDDHA
```

The extracted UMIs are appended together (`<umi_1><umi2>`) and are then appended to the barcode, with a delimiter, _. Together the barcode and UMIs are then appended to the end of the read header, again with a delimiter _.

5.2.3 Example: the inDrop barcode read

Read 1 from the inDrop technique of Klein *et al* consists of a two part cell barcode separated by an a 22 base adapter sequence. The first part of the cell barcode can be between 8 and 12 bases in length, and the second part is always 8 bases long. A 6 base UMI follows the second part of the cell barcode, and this is then followed by at least three T bases.

This gives us the regex: `(?P<cell_1>.{8,12})(?P<discard_1>GAGTGATTGCTTGTGACGCCTT){s<=2}(?P<cell_2>.{8})(?P<umi_1>.{6})T{3}.*`

The first part of the cell barcode is matched by `.{8,12}`. We wish to capture this and use it as the first part of the cell barcode, so we name this group `cell_1`. This is followed by the adapter sequence `GAGTGATTGCTTGTGACGCCTT` which is captured in a group `discard_1` so that it is discarded. We wish to allow up to two mismatches when matching the adapter sequence, so we follow the group with `{s<=2}` (this is a syntax specific to the `regex` library). Next up is the second part of the cell barcode, 8 anythings in a group called `cell_2`: `(?<cell_2>.{8})`, the final cell barcode attached to the read header will consist of the concatenation of these two parts. `(?<umi_1?.{6})` captures the six base UMI. Finally three Ts and then any number of any base is matched with `T{3}.*`. Because these are not in a capture group they are left on the read and not discarded or moved to the read header.

5.2.4 Regular what-nows?

Regular expressions (abbreviated regex) are a language used for flexible string matching. We will not go into the full details of this here, there are many tutorials available on the web. A good reference manual is the python regular expression page.

The basics you will need are that patterns consist of characters that must be matched. For for example `A` matches the character A. If we put sets of characters in square brackets, we match any of the characters, so the pattern `[AT]` matches an A or a T. We also have wildcards. There are many wildcards, but the most useful is `.` which matches anything.

We can also repeat things. `*` and `+` mean zero-or-more and one-or-more respectively, so `A*` matches zero-or-more A characters and `.+` matches or or more of any character. We can be more restrictive using curly braces: `A{3}` matches exactly 3 A characters and `A{1,3}` matches 1 to 3 A characters.

Sometimes we want to "capture" part of the match to the pattern so that we can tell what it was. For example the pattern `.{4}TTTT` matches any four characters and then four Ts. After testing if a string matches a pattern we might want to do what the four characters were. We do this by enclosing the bit we are interested in round brackets - this is a so called "capture group". So if we test the pattern `(.{4})TTTT` against the string `ATCGTTTT` it will match and the content of the group will be `ATCG`. Sometimes we want to capture more than one group, so if we search `(.{4})TTTT(.{4})` we have two groups (numbered from the left). If we search against `ATCGTTTTAAAA`, when we fetch group 1, if will contain `ATCG` and when we fetch group 2 it will contain `AAAA`.

CHAPTER 6

FAQ

- **Why is my `umi_tool group/dedup` command taking so long?**

  The time taken to resolve each position is dependent on how many unique UMIs there are and how closely related they are since these factors will affect how large the network of connected UMIs is. Some of the factors which can affect run time are:

  1. UMI length (shorter => fewer possible UMIs => increased connectivity between UMIs => larger networks => longer run time)
  2. Sequencing error rate (higher => more error UMIs => larger networks => longer run time)
  3. Sequencing depth (higher = greater proportion of possible UMIs observed => larger network => longer run time)
  4. Running `umi_tools dedup --output-stats` requires a considerable amount of time and memory to generate the null distributions. If you want these stats, consider obtaining stats for just a single contig, eg. `--chrom=chr22`.
  5. If you are doing single-cell RNA-seq and you have reads from more than one cell in your BAM file, make sure you are running with the `--per-cell` swtich.

- **Why is my `umi_tool group/dedup` command taking so much memory?**

  There are a few reasons why your command could require an excessive amount of memory:

  1. Most of the above factors increasing run time can also increase memory
  2. Running `umi_tools dedup` with `chimeric-reads=use` (default) can take a large amount of memory. This is because `dedup` will wait until a whole contig has been deduplicated before re-parsing the reads from the contig and writing out the read2s which match the retained read1s. For chimeric read pairs, the read2s will not be found on the same contig and will be kept in a buffer of "orphan" read2s which may take up a lot of memory. Consider using `chimeric-reads=discard` instead to discard chimeric read pairs. Similarly, use of `--unmapped-reads=use` with `--paired` can also increase memory requirements.

- **Can I run `umi_tools` with parallel threads?**

Not yet! This is something we have been discussing for a while (see #203 & #257) but haven't found the time to actually implement. If you'd like to help us out, get in touch!

- **What's the correct regex to use for technique X?**

  Here is a table of the techniques we have come across that are not easily processed with the basic barcode pattern syntax, and the correct regex's to use with them:

If you know of other, please drop us a PR with them!

- **Can I use `umi_tools` to determine consensus sequences?**

  Right now, you can use `umi_tools group` to identify the duplicated read groups. From this, you can then derive consensus sequences as you wish. We have discussed adding consense sequence calling as a separate `umi_tools` command (see #203). If you'd like to help us out, get in touch!

- **What do the `--per-gene`, `--gene-tag` and `--per-contig` options do in `umi_tools group/dedup/count`?**

  These options are designed to handle samples from sequence library protocols where fragmentation occurs post amplification, e.g CEL-Seq for single cell RNA-Seq. For such sample, mapping coordinates of duplicate reads will no longer be identical and `umi_tools` can instead use the gene to which the read is mapped. This behaviour is switched on with `--per-gene`. `umi_tools` then needs to be told if the reads are directly mapped to a transcriptome (`--per-contig`), or mapped to a genome and the transcript/gene assignment is contained in a read tag (`--gene-tag=[TAG]`). If you have assigned reads to transcripts but want to use the gene IDs to determine UMI groups, there is a further option to provide a file mapping transcript IDs to gene IDs (`--gene-transcript-map`). Finally, for single cell RNA-Seq, you must specify `--per-cell`. See `umi_tools dedup/group/count --help` for details of further related options and the UMI-tools single cell RNA-Seq guide.

- **Should I use `--per-gene` with my sequencing from method X?**

  It can be difficult to work this out sometimes! So far we have come across the following technqies that require the use of `--per-gene`: CEL-seq2, SCRB-seq, 10x Chromium, inDrop, Drop-seq and SPLiT-seq. Let us know if you know of more

- **Has the whitelist command been peer-reviewed and compared to alternatives?**

  No. At the time of the UMI-tools publication on 18 Jan '17, the only tools available were `extract`, `dedup` and `group`. The `count` and `whitelist` commands were added later. With `count`, the deduplication part is identical to `dedup`, so it's reasonable to say the underlying agorithm has been peer-reviewed. However, `whitelist` is using an entirely different approach (see here which has not been rigourously tested, compared to alternative algorithms or peer-reviewed. We recommend users to explore other options for whitelisting also.

- **Can I use whitelist without UMIs?**

  Strickly speaking, yes, but only with `--extract-method string`. If you use `--extract-method regex` and don't provide both a UMI and Cell barcode position in the regex, you'll get an error.

CHAPTER 7

Making use of our Alogrithmns: The API

7.1 The UMIClusterer class

There are occasions when you would like to make use of the deduplication/clustering algorithmns that we use here, but can't quite get the commandline tools to do exactly what you want. To help you use our methods in your tool we export the key class from the UMI-Tools package: *umi_tools.UMIClusterer*. It allows access to all the algorithmns detailed in the UMI-Tools paper.

To use the class, first create an instance, specifying the deduplication method. The default is the *directional* method, that is also the default for the command likne tools:

```
from umi_tools import UMIClusterer
clusterer = UMIClusterer(cluster_method="directional")
```

The returned instance is a *functor*. That is a class instance that can be called like a function. The function takes as its main argument a dictionary where the keys are UMI sequences and the value is the count of that UMI sequence. For example you might have reads with three different UMIs at one position: "ATAT", "GTAT", "CCAT". You may seen the first one 10 times, the second 5 times and the third 3 times:

```
umis = {b"ATAT": 10,
        b"GTAT": 5,
        b"CCAT": 3}
```

We pass this to our cluster, and it will return to us a list of lists, with each sub-list being a cluster of UMIs which we predict arose from the same original molecule:

```
clustered_umis = clusterer(umis, threshold=1)
print clustered_umis


[["ATAT", "GTAT"], ["CCAT"]]
```

ATAT and *GTAT* are only a single base apart, and the count of *ATAT* is >= 2n-1, where n is the count of *GTAT*. *CCAT* is two base edits away from the others and so is in a seperate cluster. The order of the UMIs in the groups is meaningful:

we predict that *ATAT* was the "true" UMI and so it is listed first.

Thus if you were in the bussiness of deduplicating reads, you'd keep one read associated with the *ATAT* and *CCAT* UMIs, and discard the reads associated with *GTAT*. Or if you were, for example, building a tools to geneate concensus read sequences, then you'd build one consensus from the reads with *ATAT* and *GTAT* and a different consensus read from the reads with the *CCAT* UMI.

Note that the *threshold* argument to the *UMIClusterer* allows the use of different edit distance threshold (although it is optional and the default is 1).

The *UMIClusterer* class is the only part of the API we publically export, and therefore guarentee won't change between major versions. You are welcome to peruse the rest of the API, especially if you are interested in contributing, but we can't guarentee that it will stay stable.

CHAPTER 8

Common options

Each of the `umi_tools` commands has a set of common options to deal with input and output files, logging, profiling and debugging

8.1 Input/Output options

By default, each tool reads from `stdin` and outputs to `stdout`, with the exception of `dedup`, `group` and `count`, which cannot work from `stdin` since in some cases they need to parse the input multiple times.

By default, logging is sent to `stdout`. In most cases, you will want to direct this to a dedicated logfile (`-L`/ `--log`) or send the logging to `stderr` (`--log2stderr`).

8.1.1 `-I, --stdin`

File to read stdin from [default = stdin].

8.1.2 `-S, --stdout`

File where output is to go [default = stdout].

8.1.3 `-L, --log`

File with logging information [default = stdout].

8.1.4 `--log2stderr`

Send logging information to stderr [default = False].

8.1.5 `-v, --verbose`

Log level. The higher, the more output [default = 1].

8.1.6 `-E, --error`

File with error information [default = stderr].

8.1.7 `--temp-dir`

Directory for temporary files. If not set, the bash environmental variable TMPDIR is used[default = None].

8.1.8 `--compresslevel`

Level of Gzip compression to use. Default=6 matches GNU gzip rather than python gzip default (which is 9)

8.2 profiling and debugging options

8.2.1 `--timeit`

Store timeing information in file [default=none].

8.2.2 `--timeit-name`

Name in timing file for this class of jobs [default=all].

8.2.3 `--timeit-header`

Add header for timing information [default=none].

8.2.4 `--random-seed`

Random seed to initialize number generator with [default=none].

CHAPTER 9

Release notes

9.1 1.1.2

07 May 2021

Bugfix

- `whitelist --filtered-out` with SE reads threw an unassigned error. Thanks @yech1990 for rectifying this (#453)

Also includes a very minor update of syntax (#455)

9.2 1.1.1

18 Nov 2020

Updates requirements for pysam version to >0.16.0.1. Thanks @sunnymouse25 (#444)

9.3 1.1.0

4 Nov 2020

Additional functionality

- Write out reads failing regex matching with extract/whitelist (see options –filtered-out, –filtered-out2). See #328 for motivation
- Ignore template length with paired-end dedup/group (see option –ignore-tlen). See #357 for motivation. Thanks @skitcattCRUKMI
- Ignore read pair suffixes with extract/whitelist e.g /1 or /2. (see option –ignore-read-pair-suffixes). See (#325, #391, #418, PierreBSC/Viral-Track issue 9) for motivation

Performance

- Sped up error correction mapping for cell barcodes in whitelist by using BKTree. Thanks @redst4r. Note that this adds a new python dependency (pybktree) which is available via pip and conda-forge.
- Very slight reduction in memory usage for dedup/group via bugfix to reduce the amount of reads being retained in the buffer. Thanks to @mitrinh1 for spotting this (#428). The bug was equivalent to hardcoding the option -buffer-whole-contig on, which ensures all reads with the same start position are grouped together for deduplication, but at the cost of not yielding reads until the end of each contig, thus increasing memory usage. As such, the bug was not detrimental to results output.

Bugfixes

- Unmapped mates were not properly discarded with dedup and group. Thanks @Daniel-Liu-c0deb0t for rectifying this.

9.4 1.0.1

6 Dec 2019

Debug for KeyError when some reads are missing a cell barode tag and stats output required from umi_tools dedup. See comments from @ZHUwj0 in #281

9.5 1.0.0

14 Feb 2019

This release is intended to be a stable release with no plans for significant updates to UMI-tools functionality in the near future. As part of this release, much of the code base has been refactored. It is possible this may have introduced bugs which have not been picked up by the regression testing. If so, please raise an issue and we'll try and rectify with a minor release update ASAP.

Documentation

UMI-tools documentation is now available online: https://umi-tools.readthedocs.io/en/latest/index.html

Along with the previous documentation, the readthedocs pages also include new pages:

- FAQ
- Making use of our Alogrithmns: The API

New knee method for whitelist

- The method to detect the "knee" in whitelist has been updated (#317). This method should always identify a threshold and is now set as the default method. Note that this knee method appears to be slightly more conservative (fewer cells above threshold) but having identified the knee, one can always re-run whitelist and use `--set-cell-number` to expand the whitelist if desired
- The old method is still available via `--knee-method=density`
- In addition, to run the old knee method but allow whitelist to exit without error even if a suitable knee point isn't identified, use the new `--allow-threshold-error` option (#249)
- Putative errors in CBs above the knee can be detected using `--ed-above-threshold` (#309)

Explicit options for handling chimeric & inproper read pairs (#312)

The behaviour for chimeric read pairs, inproper read pairs and unmapped reads can now be explictly set with the `--chimeric-pairs`, `--unpaired-reads` and `--unmapped-reads` options.

New options

- `--temp-dir`: Set the directory for temporary files (#254)
- `--either-read` & `--either-read-resolve`: Extract the UMI from either read (#175)

Misc

- Updates python testing version to 3.6.7 and drops python 2 testing
- Replace deprecated imp import (#318)
- Debug error with pysam <0.14 (#319)
- Refactor module files
- Moves documentation into dedicated module

9.6 0.5.5

16 Nov 2018

Mainly minor debugs and improved detection of incorrect command line options. Minor updates to documentation.

- Resolves issues correctly skipping reads which have not been assigned (#191 & #273).

This involves the addition of the `--assigned-status-tag` option

- Testing for OSX has been dropped due to unresolved issues with travis. We hope to resurrect this in the future!
- In line with major python packages (e.g https://www.numpy.org/neps/nep-0014-dropping-python2.7-proposal.html), support for python 2 will be dropped from January 1st 2019.

9.7 0.5.4

16 Jul 2018

- The defualt value for `--skip_regex` was incorrectly formatted. Thanks to @ekernf01 for spotting (#231 / #256)

9.8 0.5.3

2 Jan 2018

- Debugs wide-format output for count (#227). Thanks @kevin199011

9.9 0.5.2

21 Dec 2017

- Adds options to specify a delimiter for a cell barcode or UMI which should be concatenated + options to specify a string splitting the cell barcode or UMI into multiple parts, of which only the first will be used. Note, this options will only work if the barcodes are contained in the BAM tag - if they were appended to the read name using umi_tools extract there is no need for these options. See #217 for motivation:

    - **`--umi-tag-delimiter=[STRING]`** remove the delimeter STRING from the UMI. Defaults to None
    - **`--umi-tag-split=[STRING]`** split UMI by STRING and take only the first portion. Defaults to None
    - **`--cell-tag-delimiter=[STRING]`** remove the delimeter STRING from the cell barcode. Defaults to None
    - **`--cell-tag-split=[STRING]`** split cell barcode by STRING and take only the first portion. Defaults to - to deal with 10X GEMs
- Reduced memory requirements for `count --wide-format-cell-counts` (#222)
- Debugs issues with –bc-pattern2 (#201, #221)
- Updates documentation (#204, #210, #211). Thanks @kohlkopf, @hy09 & @cbrueffer.

9.10 0.5.1

16 Oct 2017

- Minor update. Improves detection of duplicate reads with paired end reads, reduces run time with dedup `--output-stats` and a few simple debugs.
- Improved identification of duplicate reads from paired end reads - will now use the position of the FIRST splice junction in the read (in reference coords) (#187)
- Speeds up dedup when running with `--output-stats` - (#184)
- **Fixes bugs:**
    - `whitelist --set-cell-number --plot-prefix` -> unwanted error
    - dedup gave non-informative error when input contains zero valid reads/read pairs. Now raises a warning but exits with status 0 (#190, #195)
    - count errored if gene identifier contained a ":" (#198)
    - Renames `--whole-contig option` to `--buffer-whole-contig` to avoid confusion with *–per-contig*' option. `--whole-contig` option will still work but will not be visible in documentation (#196)

9.11 0.5.0

18 Aug 2017

Version 0.5.0 introduces new commands to support single-cell RNA-Seq and reduces run-time. The underlying methods have not changed hence the minor release number uptick.

UMI-tools goes single cell

New commands for single cell RNA-Seq (scRNA-Seq):

- **`whitelist`** Extract cell barcodes (CB) from droplet-based scRNA-Seq fastqs and

  estimate the number of "true" CBs. Outputs a flatfile listing the true cell barcodes and 'error' barcodes within a set distance. See #97 for a motivating example. Thanks to @Hoohm for input and patience in testing. Thanks to @k3yavi for input in discussions about implementing a 'knee' method.

- **count** Count the number of reads per cell per gene after de-duplication. This tool uses the same underlying methods as group and dedup and acts to simplify scRNA-Seq read-counting with umi_tools. See #114, #131.
- **count_tab** As per count but works from a flatfile input from e.g featureCounts - See #44, #121, #125

In the process of creating these commands, the options for dealing with UMIs on a "per-gene" basis have been re-jigged to make their purpose clearer. See e.g #127 for a motvating example.

To perform group, dedup or count on a per-gene, basis, the `--per-gene` option should be provided. This must be combined with either `--gene-tag` if the BAM contains gene assignments in a tag, or `--per-contig` if the reads have been aligned to a transcriptome. In the later case, if the reads have been aligned to a transcriptome where each contig is a transcript, the option `--gene-transcript-map` can be used to operate at the gene level. These options are standardised across all tools such that one can easily change e.g a `count` command into a `dedup` command.

Additional updates

- `extract` can now accept regex patterns to describe UMI +/- CB encoding in read(s). See `--extract-method=regex` option.
- We have written a guide for how to use UMI-tools for scRNA-Seq analysis including estimation of the number of true CBs, flexible extraction of cell barcodes and UMIs and `--per-cell` read-counting as well as common workflow variations.
- Reduced run-time (#156)
- Introduced a hashing step to limit the scope of the edit-distance comparisons required to build the networks. Big thanks to @mparker2 for this!
- Simplified installation (#145)
- Previously extensions were cythonized and compiled on the fly using `pyximport`, requiring users to have access to the install directory the first time the extension was required. Now the cythonized extension is provided, and is compiled at install-time.

9.12 0.4.4

8 May 2017

- Tweaks the way group handles paired end BAMs. To simplify the process and ensure all reads are written out, the paired end read (read 2) is now outputted without a group or UMI tag. (#115).
- Introduces the `--skip-tags-regex` option to enable users to skip descriptive gene tags, such as "Unassigned" when using the –gene-tag option. See #108.

Bugfixes:

- If the `--transcript-gene-map` included transcripts not observed in the BAM, this caused an error when trying to retrieve reads aligned to the transcript. This has been resolved. See #109
- Allow output to zipped file with extract using python 3 #104
- Improved test coverage (`--chrom` and `--gene-tag` options). Thanks @MarinusVL for kindly sharing a BAM with gene tags.

9.13 0.4.3

28 Mar 2017

- Improves run time for large networks (see #94, #31). Thanks to @gpratt for identifying the issue and implementing the solution

9.14 0.4.2

22 Mar 2017

- When using the directional method with the group command, the 'top' UMI within each group was not always the most abundant (see comments in #96). This has now been resolved

9.15 0.4.1

16 Mar 2017

- Due to a bug in `pysam.fetch()` paired end files with a large number of contigs could take a long time to process (see #93). This has now been resolved. Thanks to @gpratt for spotting and resolving this.

9.16 0.4.0

9 Mar 2017

Added functionality:

- Deduplicating on gene ids (*#44 <https://github.com/CGATOxford/UMI-tools/issues/44>*'_ for motivation) - The user can now group/dedup according to the gene which the read

    aligns to. This is useful for single cell RNA-Seq methods such as e.g CEL-Seq where the position of the read on a transcript may be different for reads generated from the same initial molecule. The following options may be used define the gene_id for each read:

    - `--per-gene`
    - `--gene-transcript-map`
    - `--gene-tag`

- Working with BAM tags (#73, #76, #89):
- UMIs can now be extracted from the BAM tags and *group* will add a tag to each read describing the read group and UMI. See following options for controlling this behaviour:
    - `--extract-umi-method`
    - `--umi-tag`
    - `--umi-group-tag`
- Ouput unmapped reads (#78)

    The group command will now output unmapped reads if the `--output-unmapped` is supplied. These reads will not be assigned to any group.

- bug fixes for `group` command (#67, #81)
- updated documentation (#77, #79)

9.17 0.3.6

1 Feb 2017

Improves the group command:

- Adds the `--subset option` as per the dedup command (#74)
- Corrects the flatfile output from the dedup command (#72)

9.18 0.3.5

27 Jan 2017

- The code has been tweaked to improve run-time. See #69 for a discussion about the changes implemented.

9.19 0.3.4

23 Jan 2017

- Corrects the edit distance comparison used to generate the network for the `directional` method.
- This will only affect results generated using the directional method and `--edit-distance-threshold` >1.
- Previously, using the `directional` method with the option `--edit-distance-threshold` set to > 1 did not return the expected set of de-duplicated reads. If you have used the `directional` method with a threshold >1, we recommend updating UMI-tools and re-running dedup.

9.20 0.3.3

19 Jan 2017

- Debugs `python 3` compatibility issues
- Adds `python 3` tests

9.21 0.3.2

17 Jan 2017)

Minor bump:

- Resolves setuptools-based installation issue

9.22 0.3.1

1 Dec 2016

Version bump to allow pypi update. No code changes

9.23 0.3.0

1 Dec 2016

- Adds the new `group` command to group PCR duplicates and return the groups in a tagged BAM file and/or flat file format. This was motivated by multiple requests to group PCR duplicated reads for downstream processes, e,g #45, #54. Special thanks to Nils Koelling (@koelling) for testing the group command.
- Adds the –umi-separator option for dedup and group for workflow where umi_tools extract is not used to extract the UMI. This was motivated by #58

9.24 0.2.6

8 Nov 2016

- directional-adjacency method is renamed directional

9.25 0.2.5

2 Nov 2016

- Debugs writing out paired end
- Debugs installation

9.26 0.2.3

7 Jun 2016

- Debugs pip installation

9.27 0.2.0

31 May 2016

extract

- New feature: Filter out read by UMI base-call quality score `--quality-threshold` and `--quality-encoding` options (#29, #33)

dedup

- Improved performance for paired end files (#31, #35)

9.28 0.0.11

23 May 2016

- Debugs read extraction from 3' end

9.29 0.0.10

- Improved memory performace for UMI extraction from paired end reads

9.30 0.0.9

29 Apr 2016

UMI-Tools Manuscript Release

- Merge pull request #18 from CGATOxford/TS-RefactorTools

CHAPTER 10

Tools

Currently there are 6 commands. The `extract` and `whitelist` commands are used to prepare a fastq containg UMIs +/- cell barcodes for alignment.

- **whitelist:**

  Builds a whitelist of the 'real' cell barcodes This is useful for droplet-based single cell RNA-Seq where the identity of the true cell barcodes is unknown. The whitelist can then be used to filter cell barcodes with extract (see below)

- **extract:**

  Flexible removal of UMI sequences from fastq reads. UMIs are removed and appended to the read name. Any other barcode, for example a library barcode, is left on the read. Can also filter reads by quality or against a whitelist (see above)

The remaining commands, `group`, `dedup` and `count`/`count_tab`, are used to identify PCR duplicates using the UMIs and perform different levels of analysis depending on the needs of the user. A number of different UMI deduplication schemes are enabled - The recommended method is *directional*. For more deails about the deduplication schemes see *The network-based deduplication methods*

- **dedup:**

  Groups PCR duplicates and deduplicates reads to yield one read per group Use this when you want to remove the PCR duplicates prior to any downstream analysis

- **group:**

  Groups PCR duplicates using the same methods available through 'dedup'. This is useful when you want to manually interrogate the PCR duplicates or perform bespoke downstream processing such as generating consensus sequences

- **count:**

  Groups and deduplicates PCR duplicates and counts the unique molecules per gene Use this when you want to obtain a matrix with unique molecules per gene, per cell, for scRNA-Seq

- **count_tab: As per count except input is a flatfile**

Each tool has a set of *Common options* for input/output, profiling and debugging.

10.1 whitelist - Identify the likely true cell barcodes

Extract cell barcodes and identify the most likely true cell barcodes

10.1.1 Usage:

For single ended reads, the following reads from stdin and outputs to stdout:

```
umi_tools whitelist --bc-pattern=[PATTERN] -L extract.log
[OPTIONS]
```

For paired end reads where the cell barcodes is split across the read pairs, the following reads end one from stdin and end two from FASTQIN and outputs to stdin:

```
umi_tools whitelist --bc-pattern=[PATTERN]
--bc-pattern2=[PATTERN] --read2-in=[FASTQIN] -L extract.log
[OPTIONS]
```

10.1.2 Output:

The whitelist is outputted as 4 tab-separated columns:

1. whitelisted cell barcode
2. Other cell barcode(s) (comma-separated) to correct to the whitelisted barcode
3. Count for whitelisted cell barcodes
4. Count(s) for the other cell barcode(s) (comma-separated)

example output:

```
AAAAAA      AGAAAA          146     1
AAAATC                      22
AAACAT                      21
AAACTA      AAACTN,GAACTA   27      1,1
AAATAC                      72
AAATCA      GAATCA          37      3
AAATGT      AAAGGT,CAATGT   41      1,1
AAATTG      CAATTG          36      1
AACAAT                      18
AACATA                      24
```

If `--error-correct-threshold` is set to 0, columns 2 and 4 will be empty.

10.1.3 Identifying the true cell barcodes

In the absence of the `--set-cell-number` option, `whitelist` finds the knee in the curve for the cumulative read counts per CB or unique UMIs per CB (`--method=[reads|umis]`). This point is referred to as the 'knee'. Previously this point was identified using the distribution of read counts per CB or unique UMIs per CB. The old behaviour can be activated using `--knee-method=density`

See this blog post for a more detailed exploration of the previous method:

https://cgatoxford.wordpress.com/2017/05/18/estimating-the-number-of-true-cell-barcodes-in-single-cell-rna-seq/

Counts per cell barcode can be performed using either read or unique UMI counts. Use `--method=[read|umis]` to set the counting method.

The process of selecting the "best" local minima with `--knee-method=density` is not completely foolproof. We recommend users always run whitelist with the `--plot-prefix` option to visualise the set of thresholds considered for defining cell barcodes. This option will also generate a table containing the thresholds which were rejected if you want to manually adjust the threshold. In addition, if you expect that a local minima will not be found, you can use the `--allow-threshold-error` option to allow `whitelist` to proceed proceed past this stage. In addition, if you have some prior expectation on the maximum number of cells which may have been sequenced, you can provide this using the option `--expect-cells` (see below).

If you don't mind if `whitelist --knee-method=density` cannot identify a suitable threshold as you intend to inspect the plots and identify the threshold manually, provide the following options: `--allow-threshold-error`, `--plot-prefix=[PLOT_PREFIX]`

We expect that the default distance-based knee method should be more robust than the density-based method. However, we haven't extensively tested this method. If you have a dataset where you believe the density-based method is better, please share this information with us: https://github.com/CGATOxford/UMI-tools/issues

Finally, in some datasets there may be a risk that CBs above the selected threshold are actually errors from another CB. We can detect potential instances of this by looking for CBs within one error (substition, insertion or deletion) of another CB with higher counts. One can then either take a conservate approach (remove CB with lower counts), or a more relaxed approach (correct CB with lower counts to CB with higher counts). Note that correction is only possible for substitutions since insertions & deletions may also affect the UMI so these are always discarded. See `--ed-above-threshold=[discard/correct]` below. Of course, the risk with the relaxed approach is that this may erroneously merge two truly different CBs together and create an in-silico "doublet". The end of the log file (–log) will detail the number of reads from CBs above the threshold which may be errors. In most cases, we expect the number of reads to be a very small fraction of the total reads and therefore recommend taking the conservative approach. See https://cgatoxford.wordpress.com/2017/05/23/estimating-the-number-of-true-cell-barcodes-in-single-cell-rna-seq-part-2/ for an analysis of errors in barcodes above the knee threshold.

10.1.4 whitelist-specific options

`--method`

"reads" or "umis". Use either reads or unique UMI counts per cell

`--knee-method`

"distance" or "density". Two methods are available to detect the 'knee' in the cell barcode count distributions. "distance" identifies the maximum distance between the cumulative distribution curve and a straight line between the first and last points on the cumulative distribution curve. "density" transforms the counts per UMI into a gaussian density and then finds the local minima which separates "real" from "error" cell barcodes. The gaussian method was the only method available prior to UMI-tools v1.0.0. "distance" is now the default method.

`--set-cell-number`

Use this option to explicity set the number of cell barcodes which should be accepted. Note that the exact number of cell barcodes in the outputted whitelist may be slightly less than this if there are multiple cells observed with the same frequency at the threshold between accepted and rejected cell barcodes.

`--expect-cells`

An upper limit estimate for the number of inputted cells. The knee method will now select the first threshold (order ascendingly) which results in the number of cell barcodes accepted being <= EXPECTED_CELLS and > EXPECTED_CELLS * 0.1. Note: This is not compatible with the default `--knee-method=distance` since there is always as single solution using this method.

`--allow-threshold-error`

This is useful if you what the command to exit with just a warning if a suitable threshold cannot be selected

`--error-correct-threshold`

Hamming distance for correction of barcodes to whitelist barcodes. This value will also be used for error detection above the knee if required (`--ed-above-threshold`)

`--plot-prefix`

Use this option to indicate the prefix for the plots and table describing the set of thresholds considered for defining cell barcodes

`--ed-above-threshold=[discard|correct]`

Detect CBs above the threshold which may be sequence errors:

- **"discard"** Discard all putative error CBs.
- **"correct"** Correct putative substituion errors in CBs above the threshold. Discard putative insertions/deletions. Note that correction is only possible when the CB contains only substituions since insertions and deletions may cause errors in the UMI sequence too

Where a CB could be corrected to two other CBs, correction is not possible. In these cases, the CB will be discarded regardless of which option is used.

`--subset-reads`

Use the first N reads to automatically identify the true cell barcodes. If N is greater than the number of reads, all reads will be used. Default is 100000000 (100 Million).

10.1.5 Barcode extraction

`--bc-pattern`

Pattern for barcode(s) on read 1. See `--extract-method`

`--bc-pattern2`

Pattern for barcode(s) on read 2. See `--extract-method`

`--extract-method`

There are two methods enabled to extract the umi barcode (+/- cell barcode). For both methods, the patterns should be provided using the `--bc-pattern` and `--bc-pattern2` options.x

- **`string`** This should be used where the barcodes are always in the same place in the read.
  - N = UMI position (required)
  - C = cell barcode position (optional)
  - X = sample position (optional)

  Bases with Ns and Cs will be extracted and added to the read name. The corresponding sequence qualities will be removed from the read. Bases with an X will be reattached to the read.

  E.g. If the pattern is *NNNNCC*, Then the read:

```
@HISEQ:87:00000000 read1
AAGGTTGCTGATTGGATGGGCTAG
+
DA1AEBFGGCG01DFH00B1FF0B
```

  will become:

```
@HISEQ:87:00000000_TT_AAGG read1
GCTGATTGGATGGGCTAG
+
1AFGGCG01DFH00B1FF0B
```

  where 'TT' is the cell barcode and 'AAGG' is the UMI.

- **`regex`** This method allows for more flexible barcode extraction and should be used where the cell barcodes are variable in length. Alternatively, the regex option can also be used to filter out reads which do not contain an expected adapter sequence. UMI-tools uses the regex module rather than the more standard re module since the former also enables fuzzy matching

  The regex must contain groups to define how the barcodes are encoded in the read. The expected groups in the regex are:

  umi_n = UMI positions, where n can be any value (required) cell_n = cell barcode positions, where n can be any value (optional) discard_n = positions to discard, where n can be any value (optional)

  UMI positions and cell barcode positions will be extracted and added to the read name. The corresponding sequence qualities will be removed from the read.

  Discard bases and the corresponding quality scores will be removed from the read. All bases matched by other groups or components of the regex will be reattached to the read sequence

  For example, the following regex can be used to extract reads from the Klein et al inDrop data:

```
(?P<cell_1>.{8,12})(?P<discard_1>GAGTGATTGCTTGTGACGCCTT)(?P<cell_2>.
↪{8})(?P<umi_1>.{6})T{3}.*
```

  Where only reads with a 3' T-tail and *GAGTGATTGCTTGTGACGCCTT* in the correct position to yield two cell barcodes of 8-12 and 8bp respectively, and a 6bp UMI will be retained.

  You can also specify fuzzy matching to allow errors. For example if the discard group above was specified as below this would enable matches with up to 2 errors in the discard_1 group.

```
(?P<discard_1>GAGTGATTGCTTGTGACGCCTT){s<=2}
```

Note that all UMIs must be the same length for downstream processing with dedup, group or count commands

`--3prime`

By default the barcode is assumed to be on the 5' end of the read, but use this option to sepecify that it is on the 3' end instead. This option only works with `--extract-method=string` since 3' encoding can be specified explicitly with a regex, e.g `.*(?P<umi_1>.{5})$`

`--read2-in`

Filename for read pairs

`--filtered-out`

Write out reads not matching regex pattern or cell barcode whitelist to this file

`--filtered-out2`

Write out read pairs not matching regex pattern or cell barcode whitelist to this file

`--ignore-read-pair-suffixes`

Ignore and read name suffixes. Note that this options is required if the suffixes are not whitespace separated from the rest of the read name

Each tool has a set of *Common options* for input/output, profiling and debugging.

10.2 extract - Extract UMI from fastq

Extract UMI barcode from a read and add it to the read name, leaving any sample barcode in place

Can deal with paired end reads and UMIs split across the paired ends. Can also optionally extract cell barcodes and append these to the read name also. See the section below for an explanation for how to encode the barcode pattern(s) to specficy the position of the UMI +/- cell barcode.

10.2.1 Usage:

For single ended reads, the following reads from stdin and outputs to stdout:

```
umi_tools extract --extract-method=string
--bc-pattern=[PATTERN] -L extract.log [OPTIONS]
```

For paired end reads, the following reads end one from stdin and end two from FASTQIN and outputs end one to stdout and end two to FASTQOUT:

```
umi_tools extract --extract-method=string
--bc-pattern=[PATTERN] --bc-pattern2=[PATTERN]
--read2-in=[FASTQIN] --read2-out=[FASTQOUT] -L extract.log [OPTIONS]
```

Using regex and filtering against a whitelist of cell barcodes:

```
umi_tools extract --extract-method=regex
--bc-pattern=[REGEX] --whitlist=[WHITELIST_TSV]
-L extract.log [OPTIONS]
```

10.2.2 Filtering and correcting cell barcodes

umi_tools extract can optionally filter cell barcodes against a user-supplied whitelist (`--whitelist`). If a whitelist is not available for your data, e.g if you have performed droplet-based scRNA-Seq, you can use the whitelist tool.

Cell barcodes which do not match the whitelist (user-generated or automatically generated) can also be optionally corrected using the `--error-correct-cell` option.

`--error-correct-cell`

Error correct cell barcodes to the whitelist (see `--whitelist`)

`--whitelist`

Whitelist of accepted cell barcodes. The whitelist should be in the following format (tab-separated):

```
AAAAAA    AGAAAA
AAAATC
AAACAT
AAACTA    AAACTN,GAACTA
AAATAC
AAATCA    GAATCA
AAATGT    AAAGGT,CAATGT
```

Where column 1 is the whitelisted cell barcodes and column 2 is the list (comma-separated) of other cell barcodes which should be corrected to the barcode in column 1. If the `--error-correct-cell` option is not used, this column will be ignored. Any additional columns in the whitelist input, such as the counts columns from the output of umi_tools whitelist, will be ignored.

`--blacklist`

BlackWhitelist of cell barcodes to discard

`--subset-reads=[N]`

Only parse the first N reads

`--quality-filter-threshold`

Remove reads where any UMI base quality score falls below this threshold

`--quality-filter-mask`

If a UMI base has a quality below this threshold, replace the base with 'N'

`--quality-encoding`

Quality score encoding. Choose from:

- 'phred33' [33-77]
- 'phred64' [64-106]
- 'solexa' [59-106]

`--reconcile-pairs`

Allow read 2 infile to contain reads not in read 1 infile. This enables support for upstream protocols where read one contains cell barcodes, and the read pairs have been filtered and corrected without regard to the read2s

10.2.3 Experimental options

Note: These options have not been extensively testing to ensure behaviour is as expected. If you have some suitable input files which we can use for testing, please contact us.

If you have a library preparation method where the UMI may be in either read, you can use the following options to search for the UMI in either read:

```
--either-read --extract-method --bc-pattern=[PATTERN1] --bc-pattern2=[PATTERN2]
```

Where both patterns match, the default behaviour is to discard both reads. If you want to select the read with the UMI with highest sequence quality, provide `--either-read-resolve=quality`.

10.2.4 Barcode extraction

`--bc-pattern`

Pattern for barcode(s) on read 1. See `--extract-method`

`--bc-pattern2`

Pattern for barcode(s) on read 2. See `--extract-method`

`--extract-method`

There are two methods enabled to extract the umi barcode (+/- cell barcode). For both methods, the patterns should be provided using the `--bc-pattern` and `--bc-pattern2` options.x

- **`string`** This should be used where the barcodes are always in the same place in the read.

  - N = UMI position (required)
  - C = cell barcode position (optional)
  - X = sample position (optional)

  Bases with Ns and Cs will be extracted and added to the read name. The corresponding sequence qualities will be removed from the read. Bases with an X will be reattached to the read.

  E.g. If the pattern is *NNNNCC*, Then the read:

```
@HISEQ:87:00000000 read1
AAGGTTGCTGATTGGATGGGCTAG
+
DA1AEBFGGCG01DFH00B1FF0B
```

  will become:

```
@HISEQ:87:00000000_TT_AAGG read1
GCTGATTGGATGGGCTAG
+
1AFGGCG01DFH00B1FF0B
```

  where 'TT' is the cell barcode and 'AAGG' is the UMI.

- **`regex`** This method allows for more flexible barcode extraction and should be used where the cell barcodes are variable in length. Alternatively, the regex option can also be used to filter out reads which do not contain an expected adapter sequence. UMI-tools uses the regex module rather than the more standard re module since the former also enables fuzzy matching

  The regex must contain groups to define how the barcodes are encoded in the read. The expected groups in the regex are:

  umi_n = UMI positions, where n can be any value (required) cell_n = cell barcode positions, where n can be any value (optional) discard_n = positions to discard, where n can be any value (optional)

  UMI positions and cell barcode positions will be extracted and added to the read name. The corresponding sequence qualities will be removed from the read.

  Discard bases and the corresponding quality scores will be removed from the read. All bases matched by other groups or components of the regex will be reattached to the read sequence

  For example, the following regex can be used to extract reads from the Klein et al inDrop data:

```
(?P<cell_1>.{8,12})(?P<discard_1>GAGTGATTGCTTGTGACGCCTT)(?P<cell_2>.
↪{8})(?P<umi_1>.{6})T{3}.*
```

  Where only reads with a 3' T-tail and *GAGTGATTGCTTGTGACGCCTT* in the correct position to yield two cell barcodes of 8-12 and 8bp respectively, and a 6bp UMI will be retained.

  You can also specify fuzzy matching to allow errors. For example if the discard group above was specified as below this would enable matches with up to 2 errors in the discard_1 group.

```
(?P<discard_1>GAGTGATTGCTTGTGACGCCTT){s<=2}
```

  Note that all UMIs must be the same length for downstream processing with dedup, group or count commands

`--3prime`

By default the barcode is assumed to be on the 5' end of the read, but use this option to sepecify that it is on the 3' end instead. This option only works with `--extract-method=string` since 3' encoding can be specified explicitly with a regex, e.g `.*(?P<umi_1>.{5})$`

`--read2-in`

Filename for read pairs

`--filtered-out`

Write out reads not matching regex pattern or cell barcode whitelist to this file

`--filtered-out2`

Write out read pairs not matching regex pattern or cell barcode whitelist to this file

`--ignore-read-pair-suffixes`

Ignore and read name suffixes. Note that this options is required if the suffixes are not whitespace separated from the rest of the read name

Each tool has a set of *Common options* for input/output, profiling and debugging.

10.3 Group - Group reads based on their UMI and mapping coordinates

Identify groups of reads based on their genomic coordinate and UMI

The group command can be used to create two types of outfile: a tagged BAM or a flatfile describing the read groups

To generate the tagged-BAM file, use the option `--output-bam` and provide a filename with the `--stdout/-S` option. Alternatively, if you do not provide a filename, the bam file will be outputted to the stdout. If you have provided the `--log/-L` option to send the logging output elsewhere, you can pipe the output from the group command directly to e.g samtools view like so:

```
umi_tools group -I inf.bam --group-out=grouped.tsv --output-bam
--log=group.log --paired | samtools view - |less
```

The tagged-BAM file will have two tagged per read:

- UG Unique_id. 0-indexed unique id number for each group of reads with the same genomic position and UMI or UMIs inferred to be from the same true UMI + errors
- BX Final UMI. The inferred true UMI for the group

To generate the flatfile describing the read groups, include the `--group-out=<filename>` option. The columns of the read groups file are below. The first five columns relate to the read. The final 3 columns relate to the group.

- **read_id** read identifier
- **contig** alignment contig

- **position** Alignment position. Note that this position is not the start position of the read in the BAM file but the start of the read taking into account the read strand and cigar
- **gene** The gene assignment for the read. Note, this will be NA unless the –per-gene option is specified
- **umi** The read UMI
- **umi_count** The number of times this UMI is observed for reads at the same position
- **final_umi** The inferred true UMI for the group
- **final_umi_count** The total number of reads within the group
- **unique_id** The unique id for the group

10.3.1 group-specific options

–group-out

Outfile name for file mapping read id to read group

–out-bam

Output a bam file with read groups tagged using the UG tag

–umi-group-tag

BAM tag for the error corrected UMI selected for the group. Default=BX

10.3.2 Extracting barcodes

It is assumed that the FASTQ files were processed with *umi_tools extract* before mapping and thus the UMI is the last word of the read name. e.g:

```
@HISEQ:87:00000000_AATT
```

where *AATT* is the UMI sequeuence.

If you have used an alternative method which does not separate the read id and UMI with a "_", such as bcl2fastq which uses ":", you can specify the separator with the option `--umi-separator=<sep>`, replacing <sep> with e.g ":".

Alternatively, if your UMIs are encoded in a tag, you can specify this by setting the option –extract-umi-method=tag and set the tag name with the –umi-tag option. For example, if your UMIs are encoded in the 'UM' tag, provide the following options: `--extract-umi-method=tag --umi-tag=UM`

Finally, if you have used umis to extract the UMI +/- cell barcode, you can specify `--extract-umi-method=umis`

The start position of a read is considered to be the start of its alignment minus any soft clipped bases. A read aligned at position 500 with cigar 2S98M will be assumed to start at position 498.

`--extract-umi-method`

How are the barcodes encoded in the read?

Options are:

- **read_id (default)** Barcodes are contained at the end of the read separated as specified with `--umi-separator` option
- **tag** Barcodes contained in a tag(s), see `--umi-tag`/`--cell-tag` options
- **umis** Barcodes were extracted using umis (https://github.com/vals/umis)

`--umi-separator=[SEPARATOR]`

Separator between read id and UMI. See `--extract-umi-method` above. Default=‘‘_‘‘

`--umi-tag=[TAG]`

Tag which contains UMI. See `--extract-umi-method` above

`--umi-tag-split=[SPLIT]`

Separate the UMI in tag by SPLIT and take the first element

`--umi-tag-delimiter=[DELIMITER]`

Separate the UMI in by DELIMITER and concatenate the elements

`--cell-tag=[TAG]`

Tag which contains cell barcode. See *–extract-umi-method* above

`--cell-tag-split=[SPLIT]`

Separate the cell barcode in tag by SPLIT and take the first element

`--cell-tag-delimiter=[DELIMITER]`

Separate the cell barcode in by DELIMITER and concatenate the elements

10.3.3 UMI grouping options

`--method`

What method to use to identify group of reads with the same (or similar) UMI(s)?

All methods start by identifying the reads with the same mapping position.

The simplest methods, unique and percentile, group reads with the exact same UMI. The network-based methods, cluster, adjacency and directional, build networks where nodes are UMIs and edges connect

UMIs with an edit distance <= threshold (usually 1). The groups of reads are then defined from the network in a method-specific manner. For all the network-based methods, each read group is equivalent to one read count for the gene.

- **unique** Reads group share the exact same UMI
- **percentile** Reads group share the exact same UMI. UMIs with counts < 1% of the median counts for UMIs at the same position are ignored.
- **cluster** Identify clusters of connected UMIs (based on hamming distance threshold). Each network is a read group
- **adjacency** Cluster UMIs as above. For each cluster, select the node (UMI) with the highest counts. Visit all nodes one edge away. If all nodes have been visited, stop. Otherwise, repeat with remaining nodes until all nodes have been visted. Each step defines a read group.
- **directional (default)** Identify clusters of connected UMIs (based on hamming distance threshold) and umi A counts >= (2* umi B counts) - 1. Each network is a read group.

`--edit-distance-threshold`

For the adjacency and cluster methods the threshold for the edit distance to connect two UMIs in the network can be increased. The default value of 1 works best unless the UMI is very long (>14bp).

`--spliced-is-unique`

Causes two reads that start in the same position on the same strand and having the same UMI to be considered unique if one is spliced and the other is not. (Uses the 'N' cigar operation to test for splicing).

`--soft-clip-threshold`

Mappers that soft clip will sometimes do so rather than mapping a spliced read if there is only a small overhang over the exon junction. By setting this option, you can treat reads with at least this many bases soft-clipped at the 3' end as spliced. Default=4.

`--multimapping-detection-method=[NH/X0/XT]`

If the sam/bam contains tags to identify multimapping reads, you can specify for use when selecting the best read at a given loci. Supported tags are "NH", "X0" and "XT". If not specified, the read with the highest mapping quality will be selected.

`--read-length`

Use the read length as a criteria when deduping, for e.g sRNA-Seq.

10.3.4 Single-cell RNA-Seq options

`--per-gene`

Reads will be grouped together if they have the same gene. This is useful if your library prep generates PCR duplicates with non identical alignment positions such as CEL-Seq. Note this option is hardcoded to be on with the count command. I.e counting is always performed per-gene. Must be combined with either `--gene-tag` or `--per-contig` option.

`--gene-tag`

Deduplicate per gene. The gene information is encoded in the bam read tag specified

`--assigned-status-tag`

BAM tag which describes whether a read is assigned to a gene. Defaults to the same value as given for `--gene-tag`

`--skip-tags-regex`

Use in conjunction with the `--assigned-status-tag` option to skip any reads where the tag matches this regex. Default (`"^[__|Unassigned]"`) matches anything which starts with “__” or “Unassigned”:

`--per-contig`

Deduplicate per contig (field 3 in BAM; RNAME). All reads with the same contig will be considered to have the same alignment position. This is useful if you have aligned to a reference transcriptome with one transcript per gene. If you have aligned to a transcriptome with more than one transcript per gene, you can supply a map between transcripts and gene using the `--gene-transcript-map` option

`--gene-transcript-map`

File mapping genes to transcripts (tab separated), e.g:

```
gene1   transcript1
gene1   transcript2
gene2   transcript3
```

`--per-cell`

Reads will only be grouped together if they have the same cell barcode. Can be combined with `--per-gene`.

10.3.5 SAM/BAM Options

`--mapping-quality`

Minimium mapping quality (MAPQ) for a read to be retained. Default is 0.

`--unmapped-reads`

How should unmapped reads be handled. Options are:

- **discard (default)** Discard all unmapped reads
- **use** If read2 is unmapped, deduplicate using read1 only. Requires `--paired`
- **output** Output unmapped reads/read pairs without UMI grouping/deduplication. Only available in umi_tools group

`--chimeric-pairs`

How should chimeric read pairs be handled. Options are:

- **discard** Discard all chimeric read pairs
- **use (default)** Deduplicate using read1 only
- **output** Output chimeric read pairs without UMI grouping/deduplication. Only available in umi_tools group

`--unpaired-reads`

How should unpaired reads be handled. Options are:

- **discard** Discard all unpaired reads
- **use (default)** Deduplicate using read1 only
- **output** Output unpaired reads without UMI grouping/deduplication. Only available in umi_tools group

`--ignore-umi`

Ignore the UMI and group reads using mapping coordinates only

`--subset`

Only consider a fraction of the reads, chosen at random. This is useful for doing saturation analyses.

`--chrom`

Only consider a single chromosome. This is useful for debugging/testing purposes

10.3.6 Input/Output Options

`--in-sam, --out-sam`

By default, inputs are assumed to be in BAM format and outputs are written in BAM format. Use these options to specify the use of SAM format for input or output.

`--paired`

BAM is paired end - output both read pairs. This will also force the use of the template length to determine reads with the same mapping coordinates.

10.3.7 Group/Dedup options

`--no-sort-output`

By default, output is sorted. This involves the use of a temporary unsorted file since reads are considered in the order of their start position which may not be the same as their alignment coordinate due to soft-clipping and reverse alignments. The temp file will be saved (in `--temp-dir`) and deleted when it has been sorted to the outfile. Use this option to turn off sorting.

`--buffer-whole-contig`

forces dedup to parse an entire contig before yielding any reads for deduplication. This is the only way to absolutely guarantee that all reads with the same start position are grouped together for deduplication since dedup uses the start position of the read, not the alignment coordinate on which the reads are sorted. However, by default, dedup reads for another 1000bp before outputting read groups which will avoid any reads being missed with short read sequencing (<1000bp).

Each tool has a set of *Common options* for input/output, profiling and debugging.

10.4 dedup - Deduplicate reads using UMI and mapping coordinates

Deduplicate reads based on the mapping co-ordinate and the UMI attached to the read

The identification of duplicate reads is performed in an error-aware manner by building networks of related UMIs (see `--method`). `dedup` can also handle cell barcoded input (see `--per-cell`).

Usage:

```
umi_tools dedup --stdin=INFILE --log=LOGFILE [OPTIONS] > OUTFILE
```

10.4.1 Selecting the representative read

For every group of duplicate reads, a single representative read is retained.The following criteria are applied to select the read that will be retained from a group of duplicated reads:

1. The read with the lowest number of mapping coordinates (see `--multimapping-detection-method` option)

2. The read with the highest mapping quality. Note that this is not the read sequencing quality and that if two reads have the same mapping quality then one will be picked at random regardless of the read quality.

Otherwise a read is chosen at random.

10.4.2 Dedup-specific options

`--output-stats=[PREFIX]`

One can use the edit distance between UMIs at the same position as an quality control for the deduplication process by comparing with a null expectation of random sampling. For the random sampling, the observed frequency of UMIs is used to more reasonably model the null expectation.

Use this option to generate a stats outfile called:

[PREFIX]_stats_edit_distance.tsv Reports the (binned) average edit distance between the UMIs at each position. Positions with a single UMI are reported seperately. The edit distances are reported pre- and post-deduplication alongside the null expectation from random sampling of UMIs from the UMIs observed across all positions. Note that separate null distributions are reported since the null depends on the observed frequency of each UMI which is different pre- and post-deduplication. The post-duplication values should be closer to their respective null than the pre-deduplication vs null comparison

In addition, this option will trigger reporting of further summary statistics for the UMIs which may be informative for selecting the optimal deduplication method or debugging.

Each unique UMI sequence may be observed [0-many] times at multiple positions in the BAM. The following files report the distribution for the frequencies of each UMI.

[PREFIX]_stats_per_umi_per_position.tsv The *_stats_per_umi_per_position.tsv* file simply tabulates the counts for unique combinations of UMI and position. E.g if prior to deduplication, we have two positions in the BAM (POSa, POSb), at POSa we have observed 2*UMIa, 1*UMIb and at POSb: 1*UMIc, 3*UMId, then the stats file is populated thus:

counts	instances_pre
1	2
2	1
3	1

If post deduplication, UMIb is grouped with UMIa such that POSa: 3*UMIa, then the *instances_post* column is populated thus:

counts	instances_pre	instances_post
1	2	1
2	1	0
3	1	2

[PREFIX]_stats_per_umi_per.tsv The *_stats_per_umi_per.tsv* table provides UMI-level summary statistics. Keeping in mind that each unique UMI sequence can be observed at [0-many] times across multiple positions in the BAM,

times_observed How many positions the UMI was observed at

total_counts The total number of times the UMI was observed across all positions

median_counts The median for the distribution of how often the UMI was observed at each position (excluding zeros)

Hence, whenever times_observed=1, total_counts==median_counts.

10.4.3 Extracting barcodes

It is assumed that the FASTQ files were processed with *umi_tools extract* before mapping and thus the UMI is the last word of the read name. e.g:

```
@HISEQ:87:00000000_AATT
```

where *AATT* is the UMI sequeuence.

If you have used an alternative method which does not separate the read id and UMI with a "_", such as bcl2fastq which uses ":", you can specify the separator with the option `--umi-separator=<sep>`, replacing <sep> with e.g ":".

Alternatively, if your UMIs are encoded in a tag, you can specify this by setting the option –extract-umi-method=tag and set the tag name with the –umi-tag option. For example, if your UMIs are encoded in the 'UM' tag, provide the following options: `--extract-umi-method=tag` `--umi-tag=UM`

Finally, if you have used umis to extract the UMI +/- cell barcode, you can specify `--extract-umi-method=umis`

The start position of a read is considered to be the start of its alignment minus any soft clipped bases. A read aligned at position 500 with cigar 2S98M will be assumed to start at position 498.

`--extract-umi-method`

How are the barcodes encoded in the read?

Options are:

- **read_id (default)** Barcodes are contained at the end of the read separated as specified with `--umi-separator` option
- **tag** Barcodes contained in a tag(s), see `--umi-tag`/`--cell-tag` options
- **umis** Barcodes were extracted using umis (https://github.com/vals/umis)

`--umi-separator=[SEPARATOR]`

Separator between read id and UMI. See `--extract-umi-method` above. Default=``_``

`--umi-tag=[TAG]`

Tag which contains UMI. See `--extract-umi-method` above

`--umi-tag-split=[SPLIT]`

Separate the UMI in tag by SPLIT and take the first element

`--umi-tag-delimiter=[DELIMITER]`

Separate the UMI in by DELIMITER and concatenate the elements

`--cell-tag=[TAG]`

Tag which contains cell barcode. See *–extract-umi-method* above

`--cell-tag-split=[SPLIT]`

Separate the cell barcode in tag by SPLIT and take the first element

`--cell-tag-delimiter=[DELIMITER]`

Separate the cell barcode in by DELIMITER and concatenate the elements

10.4.4 UMI grouping options

`--method`

What method to use to identify group of reads with the same (or similar) UMI(s)?

All methods start by identifying the reads with the same mapping position.

The simplest methods, unique and percentile, group reads with the exact same UMI. The network-based methods, cluster, adjacency and directional, build networks where nodes are UMIs and edges connect UMIs with an edit distance <= threshold (usually 1). The groups of reads are then defined from the network in a method-specific manner. For all the network-based methods, each read group is equivalent to one read count for the gene.

- **unique** Reads group share the exact same UMI
- **percentile** Reads group share the exact same UMI. UMIs with counts < 1% of the median counts for UMIs at the same position are ignored.
- **cluster** Identify clusters of connected UMIs (based on hamming distance threshold). Each network is a read group
- **adjacency** Cluster UMIs as above. For each cluster, select the node (UMI) with the highest counts. Visit all nodes one edge away. If all nodes have been visited, stop. Otherwise, repeat with remaining nodes until all nodes have been visted. Each step defines a read group.
- **directional (default)** Identify clusters of connected UMIs (based on hamming distance threshold) and umi A counts >= (2* umi B counts) - 1. Each network is a read group.

`--edit-distance-threshold`

For the adjacency and cluster methods the threshold for the edit distance to connect two UMIs in the network can be increased. The default value of 1 works best unless the UMI is very long (>14bp).

`--spliced-is-unique`

Causes two reads that start in the same position on the same strand and having the same UMI to be considered unique if one is spliced and the other is not. (Uses the ‘N’ cigar operation to test for splicing).

`--soft-clip-threshold`

Mappers that soft clip will sometimes do so rather than mapping a spliced read if there is only a small overhang over the exon junction. By setting this option, you can treat reads with at least this many bases soft-clipped at the 3’ end as spliced. Default=4.

`--multimapping-detection-method=[NH/X0/XT]`

If the sam/bam contains tags to identify multimapping reads, you can specify for use when selecting the best read at a given loci. Supported tags are “NH”, “X0” and “XT”. If not specified, the read with the highest mapping quality will be selected.

`--read-length`

Use the read length as a criteria when deduping, for e.g sRNA-Seq.

10.4.5 Single-cell RNA-Seq options

`--per-gene`

Reads will be grouped together if they have the same gene. This is useful if your library prep generates PCR duplicates with non identical alignment positions such as CEL-Seq. Note this option is hardcoded to be on with the count command. I.e counting is always performed per-gene. Must be combined with either `--gene-tag` or `--per-contig` option.

`--gene-tag`

Deduplicate per gene. The gene information is encoded in the bam read tag specified

`--assigned-status-tag`

BAM tag which describes whether a read is assigned to a gene. Defaults to the same value as given for `--gene-tag`

`--skip-tags-regex`

Use in conjunction with the `--assigned-status-tag` option to skip any reads where the tag matches this regex. Default (`"^[__|Unassigned]"`) matches anything which starts with “__” or “Unassigned”:

`--per-contig`

Deduplicate per contig (field 3 in BAM; RNAME). All reads with the same contig will be considered to have the same alignment position. This is useful if you have aligned to a reference transcriptome with one transcript per gene. If you have aligned to a transcriptome with more than one transcript per gene, you can supply a map between transcripts and gene using the `--gene-transcript-map` option

`--gene-transcript-map`

File mapping genes to transcripts (tab separated), e.g:

```
gene1   transcript1
gene1   transcript2
gene2   transcript3
```

`--per-cell`

Reads will only be grouped together if they have the same cell barcode. Can be combined with `--per-gene`.

10.4.6 SAM/BAM Options

`--mapping-quality`

Minimium mapping quality (MAPQ) for a read to be retained. Default is 0.

`--unmapped-reads`

How should unmapped reads be handled. Options are:

- **discard (default)** Discard all unmapped reads
- **use** If read2 is unmapped, deduplicate using read1 only. Requires `--paired`
- **output** Output unmapped reads/read pairs without UMI grouping/deduplication. Only available in umi_tools group

`--chimeric-pairs`

How should chimeric read pairs be handled. Options are:

- **discard** Discard all chimeric read pairs
- **use (default)** Deduplicate using read1 only
- **output** Output chimeric read pairs without UMI grouping/deduplication. Only available in umi_tools group

`--unpaired-reads`

How should unpaired reads be handled. Options are:

- **discard** Discard all unpaired reads
- **use (default)** Deduplicate using read1 only
- **output** Output unpaired reads without UMI grouping/deduplication. Only available in umi_tools group

`--ignore-umi`

Ignore the UMI and group reads using mapping coordinates only

`--subset`

Only consider a fraction of the reads, chosen at random. This is useful for doing saturation analyses.

`--chrom`

Only consider a single chromosome. This is useful for debugging/testing purposes

10.4.7 Input/Output Options

`--in-sam, --out-sam`

By default, inputs are assumed to be in BAM format and outputs are written in BAM format. Use these options to specify the use of SAM format for input or output.

`--paired`

BAM is paired end - output both read pairs. This will also force the use of the template length to determine reads with the same mapping coordinates.

10.4.8 Group/Dedup options

`--no-sort-output`

By default, output is sorted. This involves the use of a temporary unsorted file since reads are considered in the order of their start position which may not be the same as their alignment coordinate due to soft-clipping and reverse alignments. The temp file will be saved (in `--temp-dir`) and deleted when it has been sorted to the outfile. Use this option to turn off sorting.

`--buffer-whole-contig`

forces dedup to parse an entire contig before yielding any reads for deduplication. This is the only way to absolutely guarantee that all reads with the same start position are grouped together for deduplication since dedup uses the start position of the read, not the alignment coordinate on which the reads are sorted. However, by default, dedup reads for another 1000bp before outputting read groups which will avoid any reads being missed with short read sequencing (<1000bp).

Each tool has a set of *Common options* for input/output, profiling and debugging.

10.5 count - Count reads per gene from BAM using UMIs and mapping coordinates

Count the number of reads per gene based on the mapping co-ordinate and the UMI attached to the read

This tool is only designed to work with library preparation methods where the fragmentation occurs after amplification, as per most single cell RNA-Seq methods (e.g 10x, inDrop, Drop-seq, SCRB-seq and CEL-seq2). Since the precise mapping co-ordinate is not longer informative for such library preparations, it is simplified to the gene. This is a reasonable approach providing the number of available UMIs is sufficiently high and the sequencing depth is sufficiently low that the probability of two reads from the same gene having the same UMIs is acceptably low.

If you want to count reads per gene for library preparations which fragment prior to amplification (e.g bulk RNA-Seq), please use `umi_tools dedup` to remove the duplicate reads as this will use the full information from the mapping co-ordinate. Then use a read counting tool such as FeatureCounts or HTSeq to count the reads per gene.

In the rare case of bulk RNA-Seq using a library preparation method with fragmentation after amplification, one can still use `count` but note that it has not been tested on bulk RNA-Seq.

This tool deviates from group and dedup in that the `--per-gene` option is hardcoded on.

10.5.1 Extracting barcodes

It is assumed that the FASTQ files were processed with *umi_tools extract* before mapping and thus the UMI is the last word of the read name. e.g:

```
@HISEQ:87:00000000_AATT
```

where *AATT* is the UMI sequeuence.

If you have used an alternative method which does not separate the read id and UMI with a "_", such as bcl2fastq which uses ":", you can specify the separator with the option `--umi-separator=<sep>`, replacing <sep> with e.g ":".

Alternatively, if your UMIs are encoded in a tag, you can specify this by setting the option –extract-umi-method=tag and set the tag name with the –umi-tag option. For example, if your UMIs are encoded in the 'UM' tag, provide the following options: `--extract-umi-method=tag --umi-tag=UM`

Finally, if you have used umis to extract the UMI +/- cell barcode, you can specify `--extract-umi-method=umis`

The start position of a read is considered to be the start of its alignment minus any soft clipped bases. A read aligned at position 500 with cigar 2S98M will be assumed to start at position 498.

`--extract-umi-method`

How are the barcodes encoded in the read?

Options are:

- **read_id (default)** Barcodes are contained at the end of the read separated as specified with `--umi-separator` option
- **tag** Barcodes contained in a tag(s), see `--umi-tag`/`--cell-tag` options
- **umis** Barcodes were extracted using umis (https://github.com/vals/umis)

`--umi-separator=[SEPARATOR]`

Separator between read id and UMI. See `--extract-umi-method` above. Default=‘‘_‘‘

`--umi-tag=[TAG]`

Tag which contains UMI. See `--extract-umi-method` above

`--umi-tag-split=[SPLIT]`

Separate the UMI in tag by SPLIT and take the first element

`--umi-tag-delimiter=[DELIMITER]`

Separate the UMI in by DELIMITER and concatenate the elements

`--cell-tag=[TAG]`

Tag which contains cell barcode. See *–extract-umi-method* above

`--cell-tag-split=[SPLIT]`

Separate the cell barcode in tag by SPLIT and take the first element

`--cell-tag-delimiter=[DELIMITER]`

Separate the cell barcode in by DELIMITER and concatenate the elements

10.5.2 UMI grouping options

`--method`

What method to use to identify group of reads with the same (or similar) UMI(s)?

All methods start by identifying the reads with the same mapping position.

The simplest methods, unique and percentile, group reads with the exact same UMI. The network-based methods, cluster, adjacency and directional, build networks where nodes are UMIs and edges connect UMIs with an edit distance <= threshold (usually 1). The groups of reads are then defined from the network in a method-specific manner. For all the network-based methods, each read group is equivalent to one read count for the gene.

- **unique** Reads group share the exact same UMI
- **percentile** Reads group share the exact same UMI. UMIs with counts < 1% of the median counts for UMIs at the same position are ignored.
- **cluster** Identify clusters of connected UMIs (based on hamming distance threshold). Each network is a read group
- **adjacency** Cluster UMIs as above. For each cluster, select the node (UMI) with the highest counts. Visit all nodes one edge away. If all nodes have been visited, stop. Otherwise, repeat with remaining nodes until all nodes have been visted. Each step defines a read group.
- **directional (default)** Identify clusters of connected UMIs (based on hamming distance threshold) and umi A counts >= (2* umi B counts) - 1. Each network is a read group.

`--edit-distance-threshold`

For the adjacency and cluster methods the threshold for the edit distance to connect two UMIs in the network can be increased. The default value of 1 works best unless the UMI is very long (>14bp).

`--spliced-is-unique`

Causes two reads that start in the same position on the same strand and having the same UMI to be considered unique if one is spliced and the other is not. (Uses the 'N' cigar operation to test for splicing).

`--soft-clip-threshold`

Mappers that soft clip will sometimes do so rather than mapping a spliced read if there is only a small overhang over the exon junction. By setting this option, you can treat reads with at least this many bases soft-clipped at the 3' end as spliced. Default=4.

`--multimapping-detection-method=[NH/X0/XT]`

If the sam/bam contains tags to identify multimapping reads, you can specify for use when selecting the best read at a given loci. Supported tags are "NH", "X0" and "XT". If not specified, the read with the highest mapping quality will be selected.

`--read-length`

Use the read length as a criteria when deduping, for e.g sRNA-Seq.

10.5.3 Single-cell RNA-Seq options

`--per-gene`

Reads will be grouped together if they have the same gene. This is useful if your library prep generates PCR duplicates with non identical alignment positions such as CEL-Seq. Note this option is hardcoded to be on with the count command. I.e counting is always performed per-gene. Must be combined with either `--gene-tag` or `--per-contig` option.

`--gene-tag`

Deduplicate per gene. The gene information is encoded in the bam read tag specified

`--assigned-status-tag`

BAM tag which describes whether a read is assigned to a gene. Defaults to the same value as given for `--gene-tag`

`--skip-tags-regex`

Use in conjunction with the `--assigned-status-tag` option to skip any reads where the tag matches this regex. Default (`"^[__|Unassigned]"`) matches anything which starts with "__" or "Unassigned":

`--per-contig`

Deduplicate per contig (field 3 in BAM; RNAME). All reads with the same contig will be considered to have the same alignment position. This is useful if you have aligned to a reference transcriptome with one transcript per gene. If you have aligned to a transcriptome with more than one transcript per gene, you can supply a map between transcripts and gene using the `--gene-transcript-map` option

`--gene-transcript-map`

File mapping genes to transcripts (tab separated), e.g:

```
gene1   transcript1
gene1   transcript2
gene2   transcript3
```

`--per-cell`

Reads will only be grouped together if they have the same cell barcode. Can be combined with `--per-gene`.

10.5.4 SAM/BAM Options

`--mapping-quality`

Minimium mapping quality (MAPQ) for a read to be retained. Default is 0.

`--unmapped-reads`

How should unmapped reads be handled. Options are:

- **discard (default)** Discard all unmapped reads
- **use** If read2 is unmapped, deduplicate using read1 only. Requires `--paired`
- **output** Output unmapped reads/read pairs without UMI grouping/deduplication. Only available in umi_tools group

`--chimeric-pairs`

How should chimeric read pairs be handled. Options are:

- **discard** Discard all chimeric read pairs
- **use (default)** Deduplicate using read1 only
- **output** Output chimeric read pairs without UMI grouping/deduplication. Only available in umi_tools group

`--unpaired-reads`

How should unpaired reads be handled. Options are:

- **discard** Discard all unpaired reads
- **use (default)** Deduplicate using read1 only
- **output** Output unpaired reads without UMI grouping/deduplication. Only available in umi_tools group

`--ignore-umi`

Ignore the UMI and group reads using mapping coordinates only

`--subset`

Only consider a fraction of the reads, chosen at random. This is useful for doing saturation analyses.

`--chrom`

Only consider a single chromosome. This is useful for debugging/testing purposes

10.5.5 Input/Output Options

`--in-sam, --out-sam`

By default, inputs are assumed to be in BAM format and outputs are written in BAM format. Use these options to specify the use of SAM format for input or output.

`--paired`

BAM is paired end - output both read pairs. This will also force the use of the template length to determine reads with the same mapping coordinates.

Each tool has a set of *Common options* for input/output, profiling and debugging.

10.6 count_tab - Count reads per gene from flatfile using UMIs

10.6.1 Purpose

The purpose of this command is to count the number of reads per gene based on the read's gene assignment and UMI. See the count command if you want to perform per-cell counting using a BAM file input.

The input must be in the following format (tab separated), where the first column is the read identifier (including UMI) and the second column is the assigned gene. The input must be sorted by the gene identifier.

Input template:

```
read_id[SEP]_UMI    gene
```

Example:

```
NS500668:144:H5FCJBGXY:2:22309:18356:15843_TCTAA     ENSG00000279457.3
NS500668:144:H5FCJBGXY:3:23405:39715:19716_CGATG     ENSG00000225972.1
```

You can perform any required file transformation and pipe the output directly to count_tab. For example to pipe output from featureCounts with the '-R CORE' option you can do the following:

```
awk '$2=="Assigned" {print $1"      "$4}' my.bam.featureCounts | sort -k2 |
umi_tools count_tab -S gene_counts.tsv -L count.log
```

The tab file is assumed to contain each read id once only. For paired end reads with featureCounts you must include the "-p" option so each read id is included once only.

Per-cell counting can be enable with `--per-cell`. For per-cell counting, the input must be in the following format (tab separated), where the first column is the read identifier (including UMI and Cell Barcode) and the second column is the assigned gene. The input must be sorted by the gene identifier:

Input template:

```
read_id[SEP]_UMI_CB    gene
```

Example:

```
NS500668:144:H5FCJBGXY:2:22309:18356:15843_TCTAA_AGTCGA     ENSG00000279457.3
NS500668:144:H5FCJBGXY:3:23405:39715:19716_CGATG_GGAGAA     ENSG00000225972.1
```

10.6.2 Extracting barcodes

It is assumed that the FASTQ files were processed with *umi_tools extract* before mapping and thus the UMI is the last word of the read name. e.g:

```
@HISEQ:87:00000000_AATT
```

where *AATT* is the UMI sequeuence.

If you have used an alternative method which does not separate the read id and UMI with a "_", such as bcl2fastq which uses ":", you can specify the separator with the option `--umi-separator=<sep>`, replacing <sep> with e.g ":".

Alternatively, if your UMIs are encoded in a tag, you can specify this by setting the option –extract-umi-method=tag and set the tag name with the –umi-tag option. For example, if your UMIs are encoded in the 'UM' tag, provide the following options: `--extract-umi-method=tag --umi-tag=UM`

Finally, if you have used umis to extract the UMI +/- cell barcode, you can specify `--extract-umi-method=umis`

The start position of a read is considered to be the start of its alignment minus any soft clipped bases. A read aligned at position 500 with cigar 2S98M will be assumed to start at position 498.

`--extract-umi-method`

How are the barcodes encoded in the read?

Options are:

- **read_id (default)** Barcodes are contained at the end of the read separated as specified with `--umi-separator` option

- **tag** Barcodes contained in a tag(s), see `--umi-tag`/`--cell-tag` options
- **umis** Barcodes were extracted using umis (https://github.com/vals/umis)

`--umi-separator=[SEPARATOR]`

Separator between read id and UMI. See `--extract-umi-method` above. Default=‘‘_‘‘

`--umi-tag=[TAG]`

Tag which contains UMI. See `--extract-umi-method` above

`--umi-tag-split=[SPLIT]`

Separate the UMI in tag by SPLIT and take the first element

`--umi-tag-delimiter=[DELIMITER]`

Separate the UMI in by DELIMITER and concatenate the elements

`--cell-tag=[TAG]`

Tag which contains cell barcode. See *–extract-umi-method* above

`--cell-tag-split=[SPLIT]`

Separate the cell barcode in tag by SPLIT and take the first element

`--cell-tag-delimiter=[DELIMITER]`

Separate the cell barcode in by DELIMITER and concatenate the elements

10.6.3 UMI grouping options

`--method`

What method to use to identify group of reads with the same (or similar) UMI(s)?

All methods start by identifying the reads with the same mapping position.

The simplest methods, unique and percentile, group reads with the exact same UMI. The network-based methods, cluster, adjacency and directional, build networks where nodes are UMIs and edges connect UMIs with an edit distance <= threshold (usually 1). The groups of reads are then defined from the network in a method-specific manner. For all the network-based methods, each read group is equivalent to one read count for the gene.

- **unique** Reads group share the exact same UMI
- **percentile** Reads group share the exact same UMI. UMIs with counts < 1% of the median counts for UMIs at the same position are ignored.

- **cluster** Identify clusters of connected UMIs (based on hamming distance threshold). Each network is a read group
- **adjacency** Cluster UMIs as above. For each cluster, select the node (UMI) with the highest counts. Visit all nodes one edge away. If all nodes have been visited, stop. Otherwise, repeat with remaining nodes until all nodes have been visted. Each step defines a read group.
- **directional (default)** Identify clusters of connected UMIs (based on hamming distance threshold) and umi A counts >= (2* umi B counts) - 1. Each network is a read group.

`--edit-distance-threshold`

For the adjacency and cluster methods the threshold for the edit distance to connect two UMIs in the network can be increased. The default value of 1 works best unless the UMI is very long (>14bp).

`--spliced-is-unique`

Causes two reads that start in the same position on the same strand and having the same UMI to be considered unique if one is spliced and the other is not. (Uses the 'N' cigar operation to test for splicing).

`--soft-clip-threshold`

Mappers that soft clip will sometimes do so rather than mapping a spliced read if there is only a small overhang over the exon junction. By setting this option, you can treat reads with at least this many bases soft-clipped at the 3' end as spliced. Default=4.

`--multimapping-detection-method=[NH/X0/XT]`

If the sam/bam contains tags to identify multimapping reads, you can specify for use when selecting the best read at a given loci. Supported tags are "NH", "X0" and "XT". If not specified, the read with the highest mapping quality will be selected.

`--read-length`

Use the read length as a criteria when deduping, for e.g sRNA-Seq.

10.6.4 Single-cell RNA-Seq options

`--per-gene`

Reads will be grouped together if they have the same gene. This is useful if your library prep generates PCR duplicates with non identical alignment positions such as CEL-Seq. Note this option is hardcoded to be on with the count command. I.e counting is always performed per-gene. Must be combined with either `--gene-tag` or `--per-contig` option.

`--gene-tag`

Deduplicate per gene. The gene information is encoded in the bam read tag specified

`--assigned-status-tag`

BAM tag which describes whether a read is assigned to a gene. Defaults to the same value as given for `--gene-tag`

`--skip-tags-regex`

Use in conjunction with the `--assigned-status-tag` option to skip any reads where the tag matches this regex. Default (`"^[__|Unassigned]"`) matches anything which starts with "__" or "Unassigned":

`--per-contig`

Deduplicate per contig (field 3 in BAM; RNAME). All reads with the same contig will be considered to have the same alignment position. This is useful if you have aligned to a reference transcriptome with one transcript per gene. If you have aligned to a transcriptome with more than one transcript per gene, you can supply a map between transcripts and gene using the `--gene-transcript-map` option

`--gene-transcript-map`

File mapping genes to transcripts (tab separated), e.g:

```
gene1   transcript1
gene1   transcript2
gene2   transcript3
```

`--per-cell`

Reads will only be grouped together if they have the same cell barcode. Can be combined with `--per-gene`.

10.6.5 SAM/BAM Options

`--mapping-quality`

Minimium mapping quality (MAPQ) for a read to be retained. Default is 0.

`--unmapped-reads`

How should unmapped reads be handled. Options are:

- **discard (default)** Discard all unmapped reads
- **use** If read2 is unmapped, deduplicate using read1 only. Requires `--paired`
- **output** Output unmapped reads/read pairs without UMI grouping/deduplication. Only available in umi_tools group

`--chimeric-pairs`

How should chimeric read pairs be handled. Options are:

- **discard** Discard all chimeric read pairs
- **use (default)** Deduplicate using read1 only
- **output** Output chimeric read pairs without UMI grouping/deduplication. Only available in umi_tools group

`--unpaired-reads`

How should unpaired reads be handled. Options are:

- **discard** Discard all unpaired reads
- **use (default)** Deduplicate using read1 only
- **output** Output unpaired reads without UMI grouping/deduplication. Only available in umi_tools group

`--ignore-umi`

Ignore the UMI and group reads using mapping coordinates only

`--subset`

Only consider a fraction of the reads, chosen at random. This is useful for doing saturation analyses.

`--chrom`

Only consider a single chromosome. This is useful for debugging/testing purposes

10.6.6 Input/Output Options

`--in-sam, --out-sam`

By default, inputs are assumed to be in BAM format and outputs are written in BAM format. Use these options to specify the use of SAM format for input or output.

`--paired`

BAM is paired end - output both read pairs. This will also force the use of the template length to determine reads with the same mapping coordinates.

Each tool has a set of *Common options* for input/output, profiling and debugging.

See *Quick start guide* for a quick tutorial on the most common usage pattern.

If you want to use UMI-tools in single-cell RNA-Seq data processing, see *Single cell tutorial*

The `dedup`, `group`, and `count` / `count_tab` commands make use of network-based methods to resolve similar UMIs with the same alignment coordinates. For a background regarding these methods see:

Genome Research Publication

Blog post discussing network-based methods.

CHAPTER 11

Installation

If you're using Conda, you can use:

```
$ conda install -c bioconda -c conda-forge umi_tools
```

Or pip:

```
$ pip install umi_tools
```

Or if you'd like to work directly from the git repository:

```
$ git clone https://github.com/CGATOxford/UMI-tools.git
```

Enter repository and run:

```
$ python setup.py install
```

For more detail see *Installation Guide*

To get detailed help on umi_tools run

```
$ umi_tools --help
```

To get help on a specific [COMMAND] run

```
$ umi_tools [COMMAND] --help
```

CHAPTER 12

Dependencies

umi_tools is dependent on *numpy*, *pandas*, *scipy*, *cython*, *pysam*, *future*, *regex* and *matplotlib*

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index