chemfp Documentation

Release 1.6.1

Andrew Dalke

Aug 27, 2020

Table of Contents

chemfp is a set of tools for working with cheminformatics fingerprints in the FPS format.

This is the documentation for the no-cost/free software/open source version of chemfp. To see the documentation for the commercial version of chemfp, go to http://chemfp.readthedocs.io/.

The commercial version is 20-50% faster than chemfp 1.6.1 on modern hardware, has an improved API for web applications development, supports the FPB binary format for fast loading, implements Tversky similarity search, runs on Python 3, and much more.

Chemfp 1.6.1 only supports Python 2.7. It is still being maintained, primarily to provide a baseline for similarity search benchmarking. If a new search implementation is not faster than chemfp 1.6.1 then it cannot be considered "high-performance", nor can its search algorithm be considered an improvement over Swamidass and Baldi's Bit-Bound algorithm.

Note: If you are benchmarking chemfp, ask me for an evaluation license so you can include chemfp 3 timings. If you are benchmarking free software/open source methods, let me know as I may be able to optimize chemfp 1.6.1 for your test case.

Chemfp 1.6.1 also exists for those decreasing number of people who are still using Python 2.7. As they likely know, Python 2.7 has reached its end-of-life and is no longer supported by the core Python developers. Several vendors offer extended support for Python 2.7. Furthermore, I can provide some help to compile Python 2.7 for those people who want to benchmark chemfp 1.6.1.

Most people will use the command-line programs to generate and search fingerprint files. *ob2fps*, *oe2fps*, and *rdkit2fps* use respectively the Open Babel, OpenEye, and RDKit chemistry toolkits to convert structure files into fingerprint files. *sdf2fps* extracts fingerprints encoded in SD tags to make the fingerprint file. *simsearch* finds targets in a fingerprint file which are sufficiently similar to the queries. *fpcat* can be used to merge fingerprint files.

Be aware that all of those those vendors dropped Python 2.7 support by 2019. Use the commercial version of chemfp if you need to generate fingerprints using those toolkits.

The programs are built using the *chemfp Python library API*, which in turn uses a C extension for the performance critical sections. The parts of the library API documented here are meant for public use, and include examples.

Remember: chemfp cannot generate fingerprints from a structure file without a third-party chemistry toolkit.

Chemfp 1.6.1 was released on 21 August 2020. It supports Python 2.7 and can be used with any historic version of OEChem/OEGraphSim, Open Babel, or RDKit which supports Python 2.7. Python 3 support is available in the commerical version of chemfp. If you are interested in paying for a copy, send an email to sales@dalkescientific.com .

To cite chemfp use: Dalke, A. The chemfp project. J Cheminform 11, 76 (2019). https://doi.org/10.1186/s13321-019-0398-8 .

CHAPTER 1

Installing

Chemfp requires that Python and a C compiler be installed in your machines. Since chemfp doesn't run on Microsoft Windows (for tedious technical reasons), then your machine likely already has both Python and a C compiler installed. In case you don't have Python, or you want to install a newer version, you can download a copy of Python from http://www.python.org/download/ . If you don't have a C compiler, .. well, do I really need to give you a pointer for that?

Chemfp 1.6.1 only supports Python 2.7. It might work under Python 2.6 but that configuration hasn't been tested. It will not work under Python 2.5.

The core chemfp functionality (e.g. similarity search) does not depend on a third-party library but you will need a chemistry toolkit in order to generate new fingerprints from structure files. chemfp supports the free Open Babel and RDKit toolkits and the proprietary OEChem toolkit. Make sure you install the Python libraries for the toolkit(s) you select.

Chemfp 1.5 was tested with Open Babel 2.4.1, RDKit 2013.03, RDKit 2016.09, RDKit 2017.03, RDKit 2017.09 (dev), OEChem/OEGraphSim 2014.07, OEChem/OEGraphSim 2016.10, and OEChem/OEGraphSim 2017.10 (beta).

Chemfp 1.6.1 was not tested against any of those toolkits because I don't have working copies of them any more. As far as I know, they should still work.

The easiest way to install chemfp is with the pip installer. This comes with Python 2.7.9 or later so it may already be installed. Chemfp 1.6.1 is available through PyPI (the Python Package Index) so you can install it over the web as:

```
python2.7 -m pip install chemfp
```

To install the `tar.gz` file with pip:

```
python2.7 -m pip install chemfp-1.6.1.tar.gz
```

Otherwise you can use Python's standard "setup.py". Read http://docs.python.org/install/index.html for details of how to use it. The short version is to do the following:

```
tar xf chemfp-1.6.1.tar.gz
cd chemfp-1.6.1
```

(continues on next page)

(continued from previous page)

```
python setup.py build
python setup.py install
```

The last step may need a `sudo` if you otherwise cannot write to your Python site-package. Another option, almost certainly better, is to use a virtual environment.

You can use Python 3's virtualenv to create a Python 2 environment.

1.1 Configuration options

The setup.py file has several compile-time options which can be set either from the `python setup.py build` command-line or through environment variables. The environment variable solution is the easiest way to change the settings under pip.

`--with-openmp, --without-openmp`

Chemfp uses OpenMP to parallelize multi-query searches. The default is *`--with-openmp`*. If you have a very old version of gcc, or an older version of clang, or are on a Mac where the clang version doesn't support OpenMP, then you will need to use *`--without-openmp`* to tell setup.py to compile without OpenMP:

```
python setup.py build --without-openmp
```

You can also set the environment variable CHEMFP_OPENMP to "1" to compile with OpenMP support, or to "0" to compile without OpenMP support:

```
CHEMFP_OPENMP=0 pip install chemfp-1.6.1.tar.gz
```

Note: you can use the environment variable `CC` to change the C compiler. For example, the clang compiler on Mac doesn't support OpenMP so I installed gcc-10 and compile using:

```
CC=gcc-10 pip install chemfp-1.6.1.tar.gz
```

`--with-ssse3, --without-ssse3`

Chemfp by default compiles with SSSE3 support, which was first available in 2006 so almost certainly available on your Intel-like processor. In case I'm wrong (are you compiling for ARM? If so, send my any compiler patches), you can disable SSSE3 support using the *`--without-ssse3`*, or set the environment variable `CHEMFP_SSSE3` to "0".

Compiling with SSSE3 support has a very odd failure case. If you compile with the SSSE3 flag enabled, then take the binary to a machine without SSSE3 support, then it will crash because all of the code will be compiled to expect the SSSE3 instruction set even though only one file, popcount_SSSE3.c, should be compiledthat way.

The solution is to compile popcount_SSSE3.c with the SSSE3 flag enabled and all of the other files without that flag. Unfortunately, Python's setup.py doesn't make that easy to do. If this is a problem for you, take a look at `filter_gcc` in the chemfp distribution. It's used like this:

```
CC=$PWD/filter_gcc python setup.py build
```

It's a bit of a hack so contact me if you have problems.

CHAPTER 2

Working with the command-line tools

The sections in this chapter describe examples of using the command-line tools to generate fingerprint files and to do similarity searches of those files.

2.1 Generating fingerprint files from PubChem SD files

In this section you'll learn how to create a fingerprint file from an SD file which contains pre-computed CACTVS fingerprints. You do not need a chemistry toolkit for this section.

PubChem is a great resource of publically available chemistry information. The data is available for ftp download. We'll use some of their SD formatted files. Each record has a PubChem/CACTVS fingerprint field, which we'll extract to generate an FPS file.

Start by downloading the files Compound_099000001_099500000.sdf.gz (from ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/SDF/Compound_099000001_099500000.sdf.gz) and Compound_048500001_049000000.sdf.gz (from ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/SDF/Compound_048500001_049000000.sdf.gz). At the time of writing they contain 10,826 and 14,967 records, respectively. (I chose some of the smallest files so they would be easier to open and review.)

Start by downloading the files Compound_099000001_099500000.sdf.gz (from ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/SDF/Compound_099000001_099500000.sdf.gz) and Compound_048500001_049000000.sdf.gz (from ftp://ftp.ncbi.nlm.nih.gov/pubchem/Compound/CURRENT-Full/SDF/Compound_048500001_049000000.sdf.gz). At the time of writing they contain 10,826 and 14,967 records, respectively. (I chose some of the smallest files so they would be easier to open and review.)

Next, convert the files into fingerprint files. On the command line do the following two commands:

```
sdf2fps --pubchem Compound_099000001_099500000.sdf.gz -o pubchem_queries.fps
sdf2fps --pubchem Compound_048500001_049000000.sdf.gz -o pubchem_targets.fps
```

Congratulations, that was it!

How does this work? Each PubChem record contains the precomputed CACTVS substructure keys in the PUBCHEM_CACTVS_SUBSKEYS tag. The `--pubchem` flag tells sdf2fps to get the value of that tag and decode it to

get the fingerprint. It also adds a few metadata fields to the fingerprint file header. Here's are the first few lines of pubchem_queries.fps. The header lines start with "#", followed by the fingerprint lines. The fingerprints are hex-encoded, followed by a tab, followed by an identifier.:

```
#FPS1
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_099000001_099500000.sdf.gz
#date=2020-05-06T12:40:21
07de0d0000000000000000000000000000000000003c060100a0010000008d2f00007800080000
0030148379203c034f13080015c0acee2a00410104ac4004101b851d261b10065f03ab8f29a41106
69001393e338d10171000000002040000000000001020000000000000000      99000039
07de1c0002000000000000000000000000800400000003c020000000000000800300007820080200
00b034870b604ce0410320421100954a090e4310082404001011997130137066 4c21addce99c1427
6b881995e1398a40500001000000000008000000000000000000000000000      99000230
```

The order of the fingerprints are the same as the order of the corresponding record in the SDF, although unconvertable records might be skipped, depending on the `--errors` flag.

If you store records in an SD file then you almost certainly don't use the same fingerprint encoding as PubChem. sdf2ps can decode from a number of encodings. Use `--help` to see the list of available decoders.

2.2 k-nearest neighbor search

In this section you'll learn how to search a fingerprint file to find the k-nearest neighbors. You will need the fingerprint files generated in *Generating fingerprint files from PubChem SD files* but you do not need a chemistry toolkit.

We'll use the pubchem_queries.fps as the queries for a k=2 nearest neighor similarity search of the target file puchem_targets.gps:

```
simsearch -k 2 -q pubchem_queries.fps pubchem_targets.fps
```

That's all! You should get output which starts:

```
#Simsearch/1
#num_bits=881
#type=Tanimoto k=2 threshold=0.0
#software=chemfp/1.6
#queries=pubchem_queries.fps
#targets=pubchem_targets.fps
#query_source=Compound_099000001_099500000.sdf.gz
#target_source=Compound_048500001_049000000.sdf.gz
2   99000039        48503376        0.8785  48503380        0.8729
2   99000230        48563034        0.8588  48731730        0.8523
2   99002251        48798046        0.8110  48625236        0.8107
2   99003537        48997075        0.9036  48997697        0.8985
```

How do you interpret the output? The lines starting with '#' are header lines. It contains metadata information describing that this is a similarity search report. You can see the search parameters, the name of the tool which did the search, and the filenames which went into the search.

After the '#' header lines come the search results, with one result per line. There are in the same order as the query fingerprints. Each result line contains tab-delimited columns. The first column is the number of hits. The second column is the query identifier used. The remaining columns contain the hit data, with alternating target id and its score.

For example, the first result line contains the 2 hits for the query 99000039. The first hit is the target id 48503376 with score 0.8785 and the second hit is 48503380 with score 0.8729. Since this is a k-nearest neighor search, the hits are sorted by score, starting with the highest score. Do be aware that ties are broken arbitrarily. There may be additional hits with the score 0.8729 which are not reported.

2.3 Threshold search

In this section you'll learn how to search a fingerprint file to find all of the neighbors at or above a given threshold. You will need the fingerprint files generated in *Generating fingerprint files from PubChem SD files* but you do not need a chemistry toolkit.

Let's do a threshold search and find all hits which are at least 0.85 similar to the queries:

```
simsearch --threshold 0.85 -q pubchem_queries.fps pubchem_targets.fps
```

The first 15 lines of output from this are:

```
#Simsearch/1
#num_bits=881
#type=Tanimoto k=all threshold=0.85
#software=chemfp/1.6
#queries=pubchem_queries.fps
#targets=pubchem_targets.fps
#query_source=Compound_099000001_099500000.sdf.gz
#target_source=Compound_048500001_049000000.sdf.gz
4   99000039        48732162        0.8596  48503380        0.8729  48503376
    0.8785  48520532        0.8541
2   99000230        48563034        0.8588  48731730        0.8523
0   99002251
4   99003537        48566113        0.8724  48998000        0.8535  48997697
    0.8985  48997075        0.9036
4   99003538        48566113        0.8724  48998000        0.8535  48997697
    0.8985  48997075        0.9036
0   99005028
0   99005031
```

Take a look at the first result line, which contains the 4 hits for the query id 99000039. As before, the hit information alternates between the target ids and the target scores, but unlike the k-nearest search, the hits are not in a particular order. You can see that here where the scores are 0.8596, 0.8729, 0.8785, and 0.8541.

You might be wondering why I chose the 0.85 threshold, or decided to show only the first 15 lines of output. Quite simply, it was for presentation. With a threshold of 0.8, the first record has 41 hits, which requires 84 columns to show, which is a bit overwhelming.

2.4 Combined k-nearest and threshold search

In this section you'll learn how to search a fingerprint file to find the k-nearest neighbors, where all of the hits must be at or above given threshold. You will need the fingerprint files generated in *Generating fingerprint files from PubChem SD files* but you do not need a chemistry toolkit.

You can combine the `-k` and `--threshold` queries to find the k-nearest neighbors which are all above a given threshold:

```
simsearch -k 3 --threshold 0.7 -q pubchem_queries.fps pubchem_targets.fps
```

This find the nearest 3 structures, which all must be at least 0.7 similar to the query fingerprint. The output from the above starts:

```
#Simsearch/1
#num_bits=881
#type=Tanimoto k=3 threshold=0.7
#software=chemfp/1.6
#queries=pubchem_queries.fps
#targets=pubchem_targets.fps
#query_source=Compound_099000001_099500000.sdf.gz
#target_source=Compound_048500001_049000000.sdf.gz
3    99000039        48503376        0.8785  48503380        0.8729  48732162        0.
↪8596
3    99000230        48563034        0.8588  48731730        0.8523  48583483        0.
↪8412
3    99002251        48798046        0.8110  48625236        0.8107  48500395        0.
↪7927
3    99003537        48997075        0.9036  48997697        0.8985  48566113        0.
↪8724
3    99003538        48997075        0.9036  48997697        0.8985  48566113        0.
↪8724
3    99005028        48651160        0.8288  48848576        0.8167  48660867        0.
↪8000
3    99005031        48651160        0.8288  48848576        0.8167  48660867        0.
↪8000
3    99006292        48945841        0.9652  48737522        0.8793  48575758        0.
↪8537
3    99006293        48945841        0.9652  48737522        0.8793  48575758        0.
↪8537
0    99006597
3    99006753        48655580        0.9310  48662591        0.9249  48654553        0.
↪9096
3    99009085        48561250        0.8503  48588162        0.8027  48675288        0.
↪7973
```

The output format is identical to the previous two search examples, and because this is a k-nearest search, the hits are sorted from higest score to lowest.

2.5 NxN (self-similar) searches

Use the –NxN option if you want to use the same fingerprints as both the queries and targets:

```
simsearch -k 3 --threshold 0.7 --NxN pubchem_queries.fps
```

This is about twice as fast and uses half as much memory compared to:

```
simsearch -k 3 --threshold 0.7 -q pubchem_queries.fps pubchem_queries.fps
```

Plus, the –NxN option excludes matching a fingerprint to itself (the diagonal term).

2.6 Using a toolkit to process the ChEBI dataset

In this section you'll learn how to create a fingerprint file from a structure file. The structure processing and fingerprint generation are done with a third-party chemisty toolkit. chemfp supports Open Babel, OpenEye, and RDKit. (OpenEye users please note that you will need an OEGraphSim license to use the OpenEye-specific fingerprinters.)

NOTE: All of these toolkit vendors dropped support for Python 2.7 by 2019, so this is mostly of historical note.

We'll work with data from ChEBI http://www.ebi.ac.uk/chebi/ which contains "Chemical Entities of Biological Interest". They distribute their structures in several formats, including as an SD file. For this section, download the "lite" version from ftp://ftp.ebi.ac.uk/pub/databases/chebi/SDF/ChEBI_lite.sdf.gz . It contains the same structure data as the complete version but many fewer tag data fields. For ChEBI 155 this file contains 95,955 records and the compressed file is 28MB.

Unlike the PubChem data set, the ChEBI data set does not contain fingerprints so we'll need to generate them using a toolkit.

2.6.1 ChEBI record titles don't contain the id

Strangely, the ChEBI dataset does not use the title line of the SD file to store the record id. A simple examination shows that 47,376 of the title lines are empty, 39,615 have the title "null", 4,499 have the title " ", 2,033 have the title "ChEBI", 45 of them are labeled "Structure #1", and the others are usually compound names.

(I've asked ChEBI to fix this, to no success. Perhaps you have more influence?)

Instead, the id is stored as the value of the "ChEBI ID" tag, which in the SD file looks like:

```
> <ChEBI ID>
CHEBI:776
```

By default the toolkit-based fingerprint generation tools use the title as the identifier, and print a warning and skip the record if the identifier is missing. Here's an example with *rdkit2fps*:

```
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 1, record #1.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 62, record #2.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 100, record #3.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 135, record #4.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 201, record #5.␣
↪Skipping.
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 236, record #6.␣
↪Skipping.
[22:53:43]  S group MUL ignored on line 103
     ... skipping many lines ...
ERROR: Missing title in SD record, file 'ChEBI_lite.sdf.gz', line 22392, record #343.␣
↪Skipping.
#FPS1
#num_bits=2048
#type=RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1
#software=RDKit/2018.03.1.dev1 chemfp/1.4
#source=ChEBI_lite.sdf.gz
#date=2018-03-16T21:53:43
031087be231150242e714400920000a193c1080c02858a1116a68100a588063428404052530040080c8cc3c4811
4101b25081a10c025e634c08a1c00088102c0400121040a2080505188a9c0a150000028211219c1001000981c4
```

(continues on next page)

(continued from previous page)

```
804417180aca0401408500180182210716db1580708a0b8a080282053285441120 0c1101040404001118600d0a
518402385dc00011290602205a070480c148f240421000c321801922c7808740cd0b10ea4c40000403dc180121
94d8d120020150b3d00043a24370000201042881d15018c0e0901442881d68604c4a83808110c772a824051948
003c801360600221040010e20418381668404b0424ec130f05a090c94960e0      ChEBI
000080000000000000000028800000000000000020000004008000000000000002000400000020000c000000
000000000800800000002004001000000000000000010000040000100000000000000800000000000000100
000008010020000000010000004000040000000000000800004000000001102000000200004000000100300
080000000000000000000000000000000000820000404000000800000400000200c000008040000000000000000
200101008000000000000000000202000002008000000000000002000000000008000400000000000000000100
400001000200800000010003002800000020020000000000000000000000    ChEBI
210809600d11180010010200820108302804406016040100a4019100001204a12800000c400202200286000491
800080c00019050000630a8222b4a10c10450170048100a00206002000930205220 88a900504002810000089 00
48004af130e28000044500052649604 4c228041380403000006206080 4c52000220003006411 4f2001803401af
120100043248000c2002008092020c6a042925c0800008c140848448541a42205c030558481078844 1610a0400
000c8100088c4064000105128a824284300648008900000100c00201c41027400c8a20908700440a0012012180
410291002200024002a1100b5038410206a000090040440000115000 0a020a null
    ... and more ...
```

That output I showed contains only three fingerprint records, the first two with the id "ChEBI" and the last with the id of 'null'. The earlier records had no title or the title was a space character, so they were skipped, with a message sent to stderr describing the problem and the location of the record containing the problem.

(If the first 100 records have no identifiers then the command-line tools will exit even if `--errors` is ignore. This is a safety mechanism. Let me know if it's a problem.)

Instead, use the `--id-tag` option to specify of the name of the data tag containing the id. For this data set you'll need to write it as:

```
--id-tag "ChEBI ID"
```

The quotes are important because of the space in the tag name. For example:

```
rdkit2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz
```

Here's what the first few lines of that output looks like:

```
[22:58:35]  S group MUL ignored on line 103
[22:58:35]  Unhandled CTAB feature: S group SRU on line: 31. Molecule skipped.
#FPS1
#num_bits=2048
#type=RDKit-Fingerprint/2 minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1
#software=RDKit/2018.03.1.dev1 chemfp/1.4
#source=ChEBI_lite.sdf.gz
#date=2018-03-16T21:58:35
10208220141258c184490038b4124609db0030024a0765883c62c9e1288a1dc224de62f445743b8b
30ad542718468104d521a214227b29ba3822fbf20e15491802a051532cd10d902c39b02b51648981
9c87eb41142811026d510a890a711cb02f2090ddacd990c5240cc282090640103d0a0a8b460184f5
11114e2a806020080452980453231 3bb03912d5e2857a6028960189e370100052c63474748a1c000
8079f49c484ca04c0d0bcb2c64b72401042a1f82002b097e852830e5898302021a1203e412064814
a598741c014e9210bc30ab180f0162029d4c446aa01c34850071e4ff037a60e732fd85014344f82a
344aa98398654481b003a84f201f518f    CHEBI:90
000000000802004120080000080000040000101000220080004000020000201000200060008000 01
010001000800010000100000020022000002000000080000004000021000000000800000044010 00
802000208002000020000014000220640000042448100000000000800 00a80012002020004198002
000802000200201200402030010008020101000242110000044000000001002000030000010001 00
0100021000a200601080002a000020200480040300008840840000080000020402000108000000 00
```

(continues on next page)

(continued from previous page)

```
2000010022000800002000020001400020800100025040000000200a080244000060008000000802
8100c801108000000041c00200800002    CHEBI:165
```

In addition to "ChEBI ID" there's also a "ChEBI Name" tag which includes data values like "tropic acid" and "(+)-guaia-6,9-diene". Every ChEBI record has a unique name so the names could also be used as the primary identifier.

The FPS fingerprint file format allows identifiers with a space, or comma, or anything other tab, newline, and a couple of other special bytes, so it's no problem using those names directly.

To use the ChEBI Name as the primary chemfp identifier, specify:

```
--id-tag "ChEBI Name"
```

2.6.2 Generating fingerprints with Open Babel

If you have the Open Babel Python library installed then you can use *ob2fps* to generate fingerprints:

```
ob2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o ob_chebi.fps
```

This takes just under 3 minutes on my ca. 2009 Mac desktop to process all of the records.

The default uses the FP2 fingerprints, so the above is the same as:

```
ob2fps --FP2 --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o ob_chebi.fps
```

ob2fps can generate several other types of fingerprints. (Use `--help` for a list.) For example, to generate the Open Babel implementation of the MACCS definition use:

```
ob2fps --MACCS --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o chebi_maccs.fps
```

2.6.3 Generating fingerprints with OpenEye

If you have the OEChem Python library installed, with licenses for OEChem and OEGraphSim, then you can use *oe2fps* to generate fingerprints:

```
oe2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o oe_chebi.fps
```

This takes about 40 seconds on my desktop and generates a number of warnings like "Stereochemistry corrected on atom number 17 of", "Unsupported Sgroup information ignored", and "Invalid stereochemistry specified for atom number 9 of". Normally the record title comes after the "... of", but the title is blank for most of the records.

OEChem could not parse 7 of the 95,955 records. I looked at the failing records and noticed that all of them had 0 atoms and 0 bonds.

The default settings produce OEGraphSim path fingerprint with the values:

```
numbits=4096 minbonds=0 maxbonds=5
   atype=Arom|AtmNum|Chiral|EqHalo|FCharge|HvyDeg|Hyb btype=Order|Chiral
```

Each of these can be changed through command-line options.

oe2fps can generate several other types of fingerprints. For example, to generate the OpenEye implementation of the MACCS definition specify:

```
oe2fps --maccs166 --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o chebi_maccs.fps
```

Use `--help` for a list of available oe2fps fingerprints or to see more configuration details.

2.6.4 Generating fingerprints with RDKit

If you have the RDKit Python library installed then you can use *rdkit2fps* to generate fingerprints. Based on the previous examples you probably guessed that the command-line is:

```
rdkit2fps --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o rdkit_chebi.fps
```

This takes just under 6 minutes on my desktop, and RDKit did not generate fingerprints for 1,101 of the 95,955 records.

You can see some of the RDKit error messages in the output, like:

```
[00:47:02] Explicit valence for atom # 12 N, 4, is greater than permitted
[00:47:02]  S group DAT ignored on line 102
```

These come from RDKit's error log. RDKit is careful to check that structures make chemical sense, and in this case it didn't like the 4-valent nitrogen. It refuses to process this molecule.

The default generates RDKit's path fingerprints with parameters:

```
minPath=1 maxPath=7 fpSize=2048 nBitsPerHash=2 useHs=1
```

(NOTE! In chemfp 1.1 the default nBitsPerHash was 4. The RDKit default nBitsPerHash is 2.)

Each of those can be changed through command-line options. See rdkit2fps `--help` for details, where you'll also see a list of the other available fingerprint types.

For example, to generate the RDKit implementation of the MACCS definition use:

```
rdkit2fps --maccs166 --id-tag "ChEBI ID" ChEBI_lite.sdf.gz -o chebi_maccs.fps
```

while the following generates the Morgan/circular fingerprint with radius 3:

```
rdkit2fps --morgan --radius 3 --id-tag "ChEBI ID" ChEBI_lite.sdf.gz
```

2.7 Alternate error handlers

In this section you'll learn how to change the error handler for rdkit2fps using the `--errors` option.

By default the "<toolkit>2fps" programs "ignore" structures which could not be parsed into a molecule option. There are two other options. They can "report" more information about the failure case and keep on processing, or they can be "strict" and exit after reporting the error.

This is configured with the `--errors` option.

Here's the rdkit2fps output using `--errors report`:

```
[00:52:39]  S group MUL ignored on line 103
[00:52:39]  Unhandled CTAB feature: S group SRU on line: 36. Molecule skipped.
ERROR: Could not parse molecule block, file 'ChEBI_lite.sdf.gz', line 12036, record
↪#179. Skipping.
[00:52:39] Explicit valence for atom # 12 N, 4, is greater than permitted
ERROR: Could not parse molecule block, file 'ChEBI_lite.sdf.gz', line 16213, record
↪#265. Skipping.
```

(continues on next page)

(continued from previous page)

The first two lines come from RDKit. The third line is from chemfp, reporting which record could not be parsed. (The record starts at line 12036 of the file and the SRU is on line 36 of the record, so the SRU is at line 12072.) The fourth line is another RDKit error message, and the last line is another chemfp error message.

Here's the rdkit2fps output using `--errors strict`:

```
[00:54:30]  S group MUL ignored on line 103
[00:54:30]  Unhandled CTAB feature: S group SRU on line: 36. Molecule skipped.
ERROR: Could not parse molecule block, file 'ChEBI_lite.sdf.gz', line 12036, record
↪#179. Exiting.
```

Because this is strict mode, processing exits at the first failure.

The ob2fps and oe2fps tools implement the `--errors` option, but they aren't as useful as rdkit2fps because the underlying APIs don't give useful feedback to chemfp about which records failed. For example, the standard OEChem file reader automatically skips records that it cannot parse. Chemfp can't report anything when it doesn't know there was a failure.

The default error handler in chemfp 1.1 was "strict". In practice this proved more annoying than useful because most people want to skip the records which could not be processed. They would then contact me asking what was wrong, or doing some pre-processing to remove the failure cases.

One of the few times when it is useful is for records which contain no identifier. When I changed the default from "strict" to "ignore" and tried to process ChEBI, I was confused at first about why the output file was so small. Then I realized that it's because the many records without a title were skipped, and there was no feedback about skipping those records.

I changed the code so missing identifiers are always reported, even if the error setting is "ignore". Missing identifiers will still stop processing if the error setting is "strict".

2.8 Alternate fingerprint file formats

In this section you'll learn about chemfp's support for other fingerprint file formats.

Chemfp started as a way to promote the FPS file format for fingerprint exchange. Chemfp 2.0 added the FPB format, which is a binary format designed around chemfp's internal search data structure so it can be loaded quickly. (For FPB support you will need to get a copy of the commercial version of chemfp.)

There are many other fingerprint formats. Perhaps the best known is the Open Babel FastSearch format. Two others are Dave Cosgrove's flush format, and OpenEye's "fpbin" format.

The chemfp_converters package contains utilities to convert between the chemfp formats and these other formats.:

```
# Convert from/to Dave Cosgrove's Flush format
flush2fps drugs.flush
fps2flush drugs.fps -o drugs.flush

# Convert from/to OpenEye's fpbin format
fpbin2fps drugs.fpbin --moldb drugs.sdf
fps2fpbin drugs_openeye_path.fps --moldb drugs.sdf -o drugs.fpbin

# Convert from/to Open Babel's FastSearch format
fs2fps drugs.fs --datafile drugs.sdf
fps2fs drugs_openbabel_FP2.fps  --datafile drugs.sdf  -o drugs.fs
```

Of the three formats, the flush format is closest to the FPS data model. That is, it stores fingerprint records as an identifier and the fingerprint bytes. By comparison, the FastSearch and fpbin formats store the fingerprint bytes and an index into another file containing the structure and identifier. It's impossible for chemfp to get the data it needs without reading both files.

Chemfp has special support for the flush format. If chemfp_converters is installed, chemfp will use it to read and write flush files nearly everywhere that it accepts FPS files. You can use it at the output to oe2fps, rdkit2fps, and ob2fps, and as the input queries to simsearch. (You cannot use it as the simsearch targets because that code has been optimized for FPS and FPB search, and I haven't spent the time to optimize flush file support.)

This means that if chemfp_converters is installed then you can use *fpcat* (see also the next section) to convert between FPS and flush file formats.

In addition, you can use it at the API level in `chemfp.open()`, `chemfp.load_fingerprints()`, `chemfp.open_fingerprint_writer()`, and `FingerprintArena.save()`.

Note that the flush format does not support the FPS metadata fields, like the fingerprint type, and it only support fingerprints which are a multiple of 32 bits long.

2.9 Convert formats with fpcat

In this section you'll learn how to use the command-line tool *fpcat* to convert between fingerprint file formats.

Chemfp 1.4 included a backport of fpcat from the commercial version of chemfp. In the commerical version, the fpcat program is often used to convert from the text-based FPS files into the binary FPB format, and vice versa.

The no-cost version of chemfp does not include the FPB format, but it does include support for Dave Cosgrove's flush file format (see also the previous section). The fpcat program can be used to convert flush files to FPS format and vice-versa:

```
fpcat drugs.flush -o drugs.fps
fpcat drugs.fps -o drugs.flush
```

For more control over the conversion, use flush2fps and fps2flush respectively, from the chemfp_converters package.

2.10 Merge multiple fingerprint files with fpcat

In this section you'll learn how to merge multiple fingerprint files into one using the command-line tool *fpcat*, and how to get slightly faster FPS arena load times by reordering the fingerprints.

The previous section showed how use fpcat to convert from one fingerprint format to another.

You can also use the fpcat program to merge multiple fingerprint files. It's based on the general idea of the Unix 'cat' program. In the following example, I'll give it three filenames, and have it save the concatenated fingerprints to an fps.gz file:

```
fpcat filename1.fps filename2.fps filename3.fps -o output.fps.gz
```

Note: fpcat uses the metadata from the first file to generate the metadata for the output. The output metadata does not currently include the 'sources' metadata lines because that would require opening all of the files first to get that information, then closing the files, and reopening them to get the fingerprint data. A future version of chemfp may support this option, and/or some way to specify the source line(s) directly.

For example, if you generate fingerprints for a lot of structures, you might split them up into multiple files, process them in parallel, and use fpcat to merge the results into a single file.

More concretely, I used RDKit to convert the ChEMBL 23 SD file into a SMILES file, which I want to process to get the MACCS fingerprints. I'll break it up into three parts, so lines 1, 4, 7, etc. go into one file, lines 2, 5, 8, etc. go into another, and lines 3, 6, 9, etc. go into a third:

```
% awk 'NR % 3 == 0' chembl_23.rdkit.smi > subset0.smi
% awk 'NR % 3 == 1' chembl_23.rdkit.smi > subset1.smi
% awk 'NR % 3 == 2' chembl_23.rdkit.smi > subset2.smi
```

I'll have rdkit2fps process each subset independently in the background (my laptop has more than 3 cores, so each job will get its own core):

```
% rdkit2fps --maccs166 subset0.smi -o subset0.fps &
[1] 13935
% rdkit2fps --maccs166 subset1.smi -o subset1.fps &
[2] 13943
% rdkit2fps --maccs166 subset2.smi -o subset2.fps &
[3] 13952
```

You may want to use something like GNU parallel for a more automated solution.

Once those are done, I'll merge them using fpcat:

```
% fpcat subset0.fps subset1.fps subset2.fps -o chembl_23.maccs.fps
```

By default the output fingerprints contain the fingerprints from the first file, in the order they appear in the file, followed by the fingerprints from the second file, and so on.

Chemfp goes through several steps to load an FPS file into an arena. It loads the fingerprints into memory, it sorts them by population count, so that fingerprints with 0 bits set come first, then those with 1 bit set, etc., and finally it creates an index describing the offset to each of those popcount boundaries.

As an optimization, if the fingerprints are already ordered, then there's no need to sort them, so it skips that step. Here's an example of the time needed to load the 1.7M ChEMBL 23 MACCS fingerprints:

```
% time python -c 'import chemfp; chemfp.load_fingerprints("chembl_23.maccs.fps")'
7.762u 0.251s 0:08.01 100.0%  0+0k 0+0io 0pf+0w
```

(This was the best of 3 times.)

I can ask fpcat to reorder the fingerprints by population count. This loads all of the fingerprints into memory, sorts them, and then saves the fingerprints in sorted order.:

```
% fpcat subset0.fps subset1.fps subset2.fps -o chembl_23.maccs.fps --reorder
```

As a result, the load time decreases by about 10-15%:

```
% time python -c 'import chemfp; chemfp.load_fingerprints("chembl_23.maccs.fps")'
6.681u 0.246s 0:06.94 99.7%   0+0k 0+0io 0pf+0w
```

Of course, if you really want fast load performance, you should use the FPB format in the commercial version:

```
% time python -c 'import chemfp; print(len(chemfp.load_fingerprints("chembl_23.maccs.
↪fpb")))'
1727081
0.078u 0.013s 0:00.09 88.8%   0+0k 0+0io 0pf+0w
```

About half of the 0.09 seconds is the startup overhead for Python itself.

2.11 chemfp's two cross-toolkit substructure fingerprints

In this section you'll learn how to generate the two substructure-based fingerprints which come as part of chemfp. These are based on cross-toolkit SMARTS pattern definitions and can be used with Open Babel, OpenEye, and RDKit. (For OpenEye users, these fingerprints use the base OEChem library and not the separately licensed OEGraphSim add-on.)

NOTE: All of these toolkit vendors dropped support for Python 2.7 by 2019, so this is mostly of historical note. I have not updated the examples to use chemfp 1.6 as I no longer have versions of those toolkits to test against.

Chemfp implements two platform-independent fingerprints where were originally designed for substructure filters but which are also used for similarity searches. One is based on the 166-bit MACCS implementation in RDKit and the other is derived from the 881-bit PubChem/CACTVS substructure fingerprints.

The chemfp MACCS definition is called "rdmaccs" because it closely derives from the MACCS SMARTS patterns used in RDKit. (These pattern definitions are also used in Open Babel and the CDK, but are completely independent from the OpenEye implementation.)

Here are example of the respective rdmaccs fingerprint for phenol using each of the toolkits.

Open Babel:

```
% echo "c1ccccc1O phenol" | ob2fps --in smi --rdmaccs
#FPS1
#num_bits=166
#type=RDMACCS-OpenBabel/2
#software=OpenBabel/2.4.1 chemfp/1.4
#date=2018-03-16T21:47:36
00000000000000000000000000000140004480101e  phenol
```

OpenEye:

```
% echo "c1ccccc1O phenol" | oe2fps --in smi --rdmaccs
#FPS1
#num_bits=166
#type=RDMACCS-OpenEye/2
#software=OEChem/2.1.3.b.1_debug (20170816) chemfp/1.4
#date=2018-03-16T21:47:54
00000000000000000000000000000140004480101e  phenol
```

RDKit:

```
% echo "c1ccccc1O phenol" | rdkit2fps --in smi --rdmaccs
#FPS1
#num_bits=166
#type=RDMACCS-RDKit/2
#software=RDKit/2018.03.1.dev1 chemfp/1.4
#date=2018-03-16T21:48:12
00000000000000000000000000000140004480101e  phenol
```

For more complex molecules it's possible that different toolkits produce different fingerprint rdmaccs, even though the toolkits use the same SMARTS definitions. Each toolkit has a different understanding of chemistry. The most notable is the different definition of aromaticity, so the bit for "two or more aromatic rings" will be toolkit dependent.

2.11.1 substruct fingerprints

chemp also includes a "substruct" substructure fingerprint. This is an 881 bit fingerprint derived from the PubChem/CACTVS substructure keys. They do not match the CACTVS fingerprints exactly, in part due to differences in ring perception. Some of the substruct bits will always be 0. With that caution in mind, if you want to try them out, use the `--substruct` option.

The term "substruct" is a horribly generic name, but I couldn't think of a better one. Until chemfp 3.0 I said these fingerprints were "experimental", in that I hadn't fully validated them against PubChem/CACTVS and could not tell you the error rate. I still haven't done that.

What's changed is that I've found out over the years that people are using the substruct fingerprints, even without full validatation. That surprised me, but use is its own form of validation. I still would like to validate the fingerprints, but it's slow, tedious work which I am not really interested in doing. Nor does it earn me any money. Plus, if the validation does lead to any changes, it's easy to simply change the version number.

CHAPTER 3

Help for the command-line tools

3.1 ob2fps command-line options

The following comes from `ob2fps --help`:

```
usage: ob2fps [-h]
              [--FP2 | --FP3 | --FP4 | --MACCS | --substruct | --rdmaccs | --rdmaccs/
↪1]
              [--id-tag NAME] [--in FORMAT] [-o FILENAME] [--out FORMAT]
              [--errors {strict,report,ignore}] [--version]
              [filenames [filenames ...]]

Generate FPS fingerprints from a structure file using Open Babel

positional arguments:
  filenames             input structure files (default is stdin)

optional arguments:
  -h, --help            show this help message and exit
  --FP2                 linear fragments up to 7 atoms
  --FP3                 SMARTS patterns specified in the file patterns.txt
  --FP4                 SMARTS patterns specified in the file
                        SMARTS_InteLigand.txt
  --MACCS               Open Babel's implementation of the MACCS 166 keys
  --substruct           generate ChemFP substructure fingerprints
  --rdmaccs, --rdmaccs/2
                        166 bit RDKit/MACCS fingerprints (version 2)
  --rdmaccs/1           use the version 1 definition for --rdmaccs
  --id-tag NAME         tag name containing the record id (SD files only)
  --in FORMAT           input structure format (default autodetects from the
                        filename extension)
  -o FILENAME, --output FILENAME
                        save the fingerprints to FILENAME (default=stdout)
  --out FORMAT          output structure format (default guesses from output
```

(continues on next page)

(continued from previous page)

```
                        filename, or is 'fps')
  --errors {strict,report,ignore}
                        how should structure parse errors be handled?
                        (default=ignore)
  --version             show program's version number and exit
```

3.2 oe2fps command-line options

The following comes from `oe2fps --help`:

```
usage: oe2fps [-h] [--path] [--circular] [--tree] [--numbits INT]
              [--minbonds INT] [--maxbonds INT] [--minradius INT]
              [--maxradius INT] [--atype ATYPE] [--btype BTYPE] [--maccs166]
              [--substruct] [--rdmaccs] [--rdmaccs/1] [--aromaticity NAME]
              [--id-tag NAME] [--in FORMAT] [-o FILENAME] [--out FORMAT]
              [--errors {strict,report,ignore}] [--version]
              [filenames [filenames ...]]

Generate FPS fingerprints from a structure file using OEChem

positional arguments:
  filenames             input structure files (default is stdin)

optional arguments:
  -h, --help            show this help message and exit
  --aromaticity NAME    use the named aromaticity model
  --id-tag NAME         tag name containing the record id (SD files only)
  --in FORMAT           input structure format (default guesses from filename)
  -o FILENAME, --output FILENAME
                        save the fingerprints to FILENAME (default=stdout)
  --out FORMAT          output structure format (default guesses from output
                        filename, or is 'fps')
  --errors {strict,report,ignore}
                        how should structure parse errors be handled?
                        (default=ignore)
  --version             show program's version number and exit

path, circular, and tree fingerprints:
  --path                generate path fingerprints (default)
  --circular            generate circular fingerprints
  --tree                generate tree fingerprints
  --numbits INT         number of bits in the fingerprint (default=4096)
  --minbonds INT        minimum number of bonds in the path or tree
                        fingerprint (default=0)
  --maxbonds INT        maximum number of bonds in the path or tree
                        fingerprint (path default=5, tree default=4)
  --minradius INT       minimum radius for the circular fingerprint
                        (default=0)
  --maxradius INT       maximum radius for the circular fingerprint
                        (default=5)
  --atype ATYPE         atom type flags, described below (default=Default)
  --btype BTYPE         bond type flags, described below (default=Default)

166 bit MACCS substructure keys:
```

(continues on next page)

(continued from previous page)

```
  --maccs166            generate MACCS fingerprints

881 bit ChemFP substructure keys:
  --substruct           generate ChemFP substructure fingerprints

ChemFP version of the 166 bit RDKit/MACCS keys:
  --rdmaccs, --rdmaccs/2
                        generate 166 bit RDKit/MACCS fingerprints (version 2)
  --rdmaccs/1           use the version 1 definition for --rdmaccs

ATYPE is one or more of the following, separated by the '|' character
  Arom AtmNum Chiral EqArom EqHBAcc EqHBDon EqHalo FCharge HCount HvyDeg
  Hyb InRing
The following shorthand terms and expansions are also available:
 DefaultPathAtom = AtmNum|Arom|Chiral|FCharge|HvyDeg|Hyb|EqHalo
 DefaultCircularAtom = AtmNum|Arom|Chiral|FCharge|HCount|EqHalo
 DefaultTreeAtom = AtmNum|Arom|Chiral|FCharge|HvyDeg|Hyb
and 'Default' selects the correct value for the specified fingerprint.
Examples:
  --atype Default
  --atype Arom|AtmNum|FCharge|HCount

BTYPE is one or more of the following, separated by the '|' character
  Chiral InRing Order
The following shorthand terms and expansions are also available:
 DefaultPathBond = Order|Chiral
 DefaultCircularBond = Order
 DefaultTreeBond = Order
and 'Default' selects the correct value for the specified fingerprint.
Examples:
   --btype Default
   --btype Order|InRing

To simplify command-line use, a comma may be used instead of a '|' to
separate different fields. Example:
  --atype AtmNum,HvyDegree

OEChem guesses the input structure format based on the filename
extension and assumes SMILES for structures read from stdin.
Use "--in FORMAT" to select an alternative, where FORMAT is one of:

  File Type      Valid FORMATs (use gz if compressed)
  ---------      ------------------------------------
   SMILES        smi, ism, usm, can, smi.gz, ism.gz, can.gz
   SDF           sdf, mol, sdf.gz, mol.gz
   SKC           skc, skc.gz
   CDK           cdk, cdk.gz
   MOL2          mol2, mol2.gz
   PDB           pdb, ent, pdb.gz, ent.gz
   MacroModel    mmod, mmod.gz
   OEBinary v2   oeb, oeb.gz
```

3.3 rdkit2fps command-line options

The following comes from `rdkit2fps --help`:

```
usage: rdkit2fps [-h] [--fpSize INT] [--RDK] [--minPath INT] [--maxPath INT]
                 [--nBitsPerHash INT] [--useHs 0|1] [--morgan] [--radius INT]
                 [--useFeatures 0|1] [--useChirality 0|1] [--useBondTypes 0|1]
                 [--torsions] [--targetSize INT] [--pairs] [--minLength INT]
                 [--maxLength INT] [--maccs166] [--avalon] [--isQuery 0|1]
                 [--bitFlags INT] [--pattern] [--substruct] [--rdmaccs]
                 [--rdmaccs/1] [--from-atoms INT,INT,...] [--id-tag NAME]
                 [--in FORMAT] [-o FILENAME] [--out FORMAT]
                 [--errors {strict,report,ignore}] [--version]
                 [filenames [filenames ...]]

Generate FPS fingerprints from a structure file using RDKit

positional arguments:
  filenames              input structure files (default is stdin)

optional arguments:
  -h, --help             show this help message and exit
  --fpSize INT           number of bits in the fingerprint. Default of 2048 for
                         RDK, Morgan, topological torsion, atom pair, and
                         pattern fingerprints, and 512 for Avalon fingerprints
  --from-atoms INT,INT,...
                         fingerprint generation must use these atom indices
                         (out of range indices are ignored)
  --id-tag NAME          tag name containing the record id (SD files only)
  --in FORMAT            input structure format (default guesses from filename)
  -o FILENAME, --output FILENAME
                         save the fingerprints to FILENAME (default=stdout)
  --out FORMAT           output structure format (default guesses from output
                         filename, or is 'fps')
  --errors {strict,report,ignore}
                         how should structure parse errors be handled?
                         (default=ignore)
  --version              show program's version number and exit

RDKit topological fingerprints:
  --RDK                  generate RDK fingerprints (default)
  --minPath INT          minimum number of bonds to include in the subgraph
                         (default=1)
  --maxPath INT          maximum number of bonds to include in the subgraph
                         (default=7)
  --nBitsPerHash INT     number of bits to set per path (default=2)
  --useHs 0|1            include information about the number of hydrogens on
                         each atom (default=1)

RDKit Morgan fingerprints:
  --morgan               generate Morgan fingerprints
  --radius INT           radius for the Morgan algorithm (default=2)
  --useFeatures 0|1      use chemical-feature invariants (default=0)
  --useChirality 0|1     include chirality information (default=0)
  --useBondTypes 0|1     include bond type information (default=1)

RDKit Topological Torsion fingerprints:
  --torsions             generate Topological Torsion fingerprints
  --targetSize INT       number of bits in the fingerprint (default=4)

RDKit Atom Pair fingerprints:
```

(continues on next page)

(continued from previous page)

```
  --pairs               generate Atom Pair fingerprints
  --minLength INT       minimum bond count for a pair (default=1)
  --maxLength INT       maximum bond count for a pair (default=30)

166 bit MACCS substructure keys:
  --maccs166            generate MACCS fingerprints

Avalon fingerprints:
  --avalon              generate Avalon fingerprints
  --isQuery 0|1         is the fingerprint for a query structure? (1 if yes, 0
                        if no) (default=0)
  --bitFlags INT        bit flags, SSSBits are 32767 and similarity bits are
                        15761407 (default=15761407)

RDKit Pattern fingerprints:
  --pattern             generate (substructure) pattern fingerprints

881 bit substructure keys:
  --substruct           generate ChemFP substructure fingerprints

ChemFP version of the 166 bit RDKit/MACCS keys:
  --rdmaccs, --rdmaccs/2
                        generate 166 bit RDKit/MACCS fingerprints (version 2)
  --rdmaccs/1           use the version 1 definition for --rdmaccs

This program guesses the input structure format based on the filename
extension. If the data comes from stdin, or the extension name us
unknown, then use "--in" to change the default input format. The
supported format extensions are:

  File Type      Valid FORMATs (use gz if compressed)
  ---------      ------------------------------------
   SMILES        smi, ism, usm, can, smi.gz, ism.gz, usm.gz, can.gz
   SDF           sdf, mol, sd, mdl, sdf.gz, mol.gz, sd.gz, mdl.gz
```

3.4 sdf2fps command-line options

The following comes from `sdf2fps --help`:

```
usage: sdf2fps [-h] [--id-tag TAG] [--fp-tag TAG] [--in FORMAT]
               [--num-bits INT] [--errors {strict,report,ignore}]
               [-o FILENAME] [--out FORMAT] [--software TEXT] [--type TEXT]
               [--version] [--binary] [--binary-msb] [--hex] [--hex-lsb]
               [--hex-msb] [--base64] [--cactvs] [--daylight]
               [--decoder DECODER] [--pubchem]
               [filenames [filenames ...]]

Extract a fingerprint tag from an SD file and generate FPS fingerprints

positional arguments:
  filenames             input SD files (default is stdin)

optional arguments:
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  --id-tag TAG          get the record id from TAG instead of the first line
                        of the record
  --fp-tag TAG          get the fingerprint from tag TAG (required)
  --in FORMAT           Specify if the input SD file is uncompressed or gzip
                        compressed
  --num-bits INT        use the first INT bits of the input. Use only when the
                        last 1-7 bits of the last byte are not part of the
                        fingerprint. Unexpected errors will occur if these
                        bits are not all zero.
  --errors {strict,report,ignore}
                        how should structure parse errors be handled?
                        (default=strict)
  -o FILENAME, --output FILENAME
                        save the fingerprints to FILENAME (default=stdout)
  --out FORMAT          output structure format (default guesses from output
                        filename, or is 'fps')
  --software TEXT       use TEXT as the software description
  --type TEXT           use TEXT as the fingerprint type description
  --version             show program's version number and exit

Fingerprint decoding options:
  --binary              Encoded with the characters '0' and '1'. Bit #0 comes
                        first. Example: 00100000 encodes the value 4
  --binary-msb          Encoded with the characters '0' and '1'. Bit #0 comes
                        last. Example: 00000100 encodes the value 4
  --hex                 Hex encoded. Bit #0 is the first bit (1<<0) of the
                        first byte. Example: 01f2 encodes the value \x01\xf2 =
                        498
  --hex-lsb             Hex encoded. Bit #0 is the eigth bit (1<<7) of the
                        first byte. Example: 804f encodes the value \x01\xf2 =
                        498
  --hex-msb             Hex encoded. Bit #0 is the first bit (1<<0) of the
                        last byte. Example: f201 encodes the value \x01\xf2 =
                        498
  --base64              Base-64 encoded. Bit #0 is first bit (1<<0) of first
                        byte. Example: AfI= encodes value \x01\xf2 = 498
  --cactvs              CACTVS encoding, based on base64 and includes a
                        version and bit length
  --daylight            Daylight encoding, which is is base64 variant
  --decoder DECODER     import and use the DECODER function to decode the
                        fingerprint

shortcuts:
  --pubchem             decode CACTVS substructure keys used in PubChem. Same
                        as --software=CACTVS/unknown --type 'CACTVS-
                        E_SCREEN/1.0 extended=2' --fp-
                        tag=PUBCHEM_CACTVS_SUBSKEYS --cactvs
```

3.5 simsearch command-line options

The following comes from `simsearch --help`:

```
usage: simsearch [-h] [-k INT] [-t FLOAT] [--queries FILENAME] [--NxN]
                 [--query STRING] [--hex-query HEX] [--query-id ID]
```

(continues on next page)

(continued from previous page)

```
                [--query-structures FILENAME] [--query-format FORMAT]
                [--target-format FORMAT] [--id-tag NAME]
                [--errors {strict,report,ignore}] [-o FILENAME] [-c] [-b INT]
                [--scan] [--memory] [--times] [--version]
                target_filename

Search an FPS or FPB file for similar fingerprints

positional arguments:
  target_filename       target filename

optional arguments:
  -h, --help            show this help message and exit
  -k INT, --k-nearest INT
                        select the k nearest neighbors (use 'all' for all
                        neighbors)
  -t FLOAT, --threshold FLOAT
                        minimum similarity score threshold
  --queries FILENAME, -q FILENAME
                        filename containing the query fingerprints
  --NxN                 use the targets as the queries, and exclude the self-
                        similarity term
  --query STRING        query as a structure record (default format: 'smi')
  --hex-query HEX       query in hex
  --query-id ID         id for the query or hex-query (default: 'Query1'
  --query-structures FILENAME, -S FILENAME
                        read strutures
  --query-format FORMAT, --in FORMAT
                        input query format (default uses the file extension,
                        else 'fps' for --queries and 'smi' for query
                        structures)
  --target-format FORMAT
                        input target format (default uses the file extension,
                        else 'fps')
  --id-tag NAME         tag containing the record id if --query-structures is
                        an SD file)
  --errors {strict,report,ignore}
                        how should structure parse errors be handled?
                        (default=ignore)
  -o FILENAME, --output FILENAME
                        output filename (default is stdout)
  -c, --count           report counts
  -b INT, --batch-size INT
                        batch size
  --scan                scan the file to find matches (low memory overhead)
  --memory              build and search an in-memory data structure (faster
                        for multiple queries)
  --times               report load and execution times to stderr
  --version             show program's version number and exit
```

3.6 fpcat command-line options

The following comes from `fpcat --help`:

```
usage: fpcat [-h] [--in FORMAT] [--merge] [-o FILENAME] [--out FORMAT]
             [--reorder] [--preserve-order] [--show-progress] [--version]
             [filename [filename ...]]

Combine multiple fingerprint files into a single file.

positional arguments:
  filename              input fingerprint filenames (default: use stdin)

optional arguments:
  -h, --help            show this help message and exit
  --in FORMAT           input fingerprint format. One of fps or fps.gz.
                        (default guesses from filename or is fps)
  --merge               assume the input fingerprint files are in popcount
                        order and do a merge sort
  -o FILENAME, --output FILENAME
                        save the fingerprints to FILENAME (default=stdout)
  --out FORMAT          output fingerprint format. One of fps or fps.gz.
                        (default guesses from output filename, or is 'fps')
  --reorder             reorder the output fingerprints by popcount
  --preserve-order      save the output fingerprints in the same order as the
                        input (default for FPS output)
  --show-progress       show progress
  --version             show program's version number and exit

Examples:

fpcat can be used to merge multiple FPS files. For example, you might
have used GNU parallel to generate FPS files for each of the PubChem
files, which you want to merge into a single file.:

    fpcat Compound_*.fps -o pubchem.fps

The --merge option is experimental. Use it if the input fingerprints
are in popcount order, because sorted output is a simple merge sort of
the individual sorted inputs. However, this option opens all input
files at the same time, which may exceed your resource limit on file
descriptors. The current implementation also requires a lot of disk
seeks so is slow for many files.
```

CHAPTER 4

The chemfp Python library

The chemfp command-line programs use a Python library called chemfp. Portions of the API are in flux and subject to change. The stable portions of the API which are open for general use are documented in *chemfp API*.

The API includes:

- low-level Tanimoto and popcount operations
- Tanimoto search algorithms based on threshold and/or k-nearest neighbors
- a cross-toolkit interface for reading fingerprints from a structure file

The following chapters give examples of how to use the API.

4.1 Byte and hex fingerprints

In this section you'll learn how chemfp stores fingerprints and some of the low-level bit operations on those fingerprints.

chemfp stores fingerprints as byte strings. Here are two 8 bit fingerprints:

```
>>> fp1 = "A"
>>> fp2 = "B"
```

The *chemfp.bitops* module contains functions which work on byte fingerprints. Here's the Tanimoto of those two fingerprints:

```
>>> from chemfp import bitops
>>> bitops.byte_tanimoto(fp1, fp2)
0.3333333333333331
```

To understand why, you have to know that ASCII character "A" has the value 65, and "B" has the value 66. The bit representation is:

```
"A" = 01000001   and   "B" = 01000010
```

so their intersection has 1 bit and the union has 3, giving a Tanimoto of 1/3 or 0.33333333333333331 when represented as a 64 bit floating point number on the computer.

You can compute the Tanimoto between any two byte strings with the same length, as in:

```
>>> bitops.byte_tanimoto("apples&", "oranges")
0.58333333333333337
```

You'll get a chemfp exception if they have different lengths.

Most fingerprints are not as easy to read as the English ones I showed above. They tend to look more like:

```
P1@\x84K\x1aN\x00\n\x01\xa6\x10\x98\\\x10\x11
```

which is hard to read. I usually show hex-encoded fingerprints. The above fingerprint in hex is:

```
503140844b1a4e000a01a610985c1011
```

which is simpler to read, though you still need to know your hex digits. There are two ways to hex-encode a byte string. I suggest using chemfp's *hex_encode()* function:

```
>>> bitops.hex_encode("P1@\x84K\x1aN\x00\n\x01\xa6\x10\x98\\\x10\x11")
'503140844b1a4e000a01a610985c1011'
```

Older versions of chemfp recommended using the s.encode() method of strings:

```
>>> "P1@\x84K\x1aN\x00\n\x01\xa6\x10\x98\\\x10\x11".encode("hex")
'503140844b1a4e000a01a610985c1011'
```

However, this will not work on Python 3. That version of Python distinguishes between text/Unicode strings and byte strings. There is no "hex" encoding for text strings, and byte strings do not implement the "encode()" method.

Use chemfp's *hex_decode()* function to decode a hex string to a fingerprint byte string.

The bitops module includes other low-level functions which work on byte fingerprints, as well as corresponding functions which work on hex fingerprints. (Hex-encoded fingerprints are decidedly second-class citizens in chemfp, but they are citizens.)

4.2 Fingerprint collections and metadata

In this section you'll learn the basic operations on a fingerprint collection and the fingerprint metadata.

A fingerprint record is the fingerprint plus an identifier. In chemfp, a fingerprint collection is a object which contains fingerprint records and which follows the common API providing access to those records.

That's rather abstract, so let's work with a few real examples. You'll need to create a copy of the "pubchem_targets.fps" file generated in *Generating fingerprint files from PubChem SD files* in order to follow along.

Here's how to open an FPS file:

```
>>> import chemfp
>>> reader = chemfp.open("pubchem_targets.fps")
```

Every fingerprint collection has a metadata attribute with details about the fingerprints. It comes from the header of the FPS file. You can view the metadata in Python repr format:

```
>>> reader.metadata
Metadata(num_bits=881, num_bytes=111,
type=u'CACTVS-E_SCREEN/1.0 extended=2', aromaticity=None,
sources=[u'Compound_048500001_049000000.sdf.gz'],
software=u'CACTVS/unknown', date='2020-05-06T12:40:32')
```

but I think it's easier to view it in string format, which matches the format of the FPS header:

```
>>> print reader.metadata
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_048500001_049000000.sdf.gz
#date=2020-05-06T12:40:32
```

All fingerprint collections support iteration. Each step of the iteration returns the fingerprint identifier and its score. Since I know the 6th record has the id 48500164, I can write a simple loop which stops with that record:

```
>>> from chemfp.bitops import hex_encode
>>> for (id, fp) in reader:
...   print id, "starts with", hex_encode(fp)[:20]
...   if id == "48500164":
...     break
...
48500020 starts with 07de0500000000000000
48500053 starts with 07de0c00000000000000
48500091 starts with 07de8c00000000000000
48500092 starts with 07de0d00020000000000
48500110 starts with 075e0c00000000000000
48500164 starts with 07de0c00000000000000
```

Fingerprint collections also support iterating via arenas, and several support Tanimoto search functions.

4.3 FingerprintArena

In this section you'll learn about the FingerprintArena fingerprint collection and how to iterate through arenas in a collection.

The FPSReader reads through or searches a fingerprint file once. If you want to read the file again you have to reopen it.

Reading from disk is slow, and the FPS format is designed for ease-of-use and not performance. If you want to do many queries then it's best to store everything in memory. The *FingerprintArena* is a fingerprint collection which does that.

Here's how to load fingerprints into an arena:

```
>>> import chemfp
>>> arena = chemfp.load_fingerprints("pubchem_targets.fps")
>>> print arena.metadata
#num_bits=881
#type=CACTVS-E_SCREEN/1.0 extended=2
#software=CACTVS/unknown
#source=Compound_048500001_049000000.sdf.gz
#date=2020-05-06T12:40:32
```

This implements the fingerprint collection API, so you can do things like iterate over an arena and get the id/fingerprint pairs.:

```
>>> from chemfp import bitops
>>> for id, fp in arena:
...     print id, "with popcount", bitops.byte_popcount(fp)
...     if id == "48656867":
...         break
...
48942244 with popcount 33
48941399 with popcount 39
48940284 with popcount 40
48943050 with popcount 40
48656359 with popcount 41
48656867 with popcount 42
```

If you look closely you'll notice that the fingerprint record order has changed from the previous section, and that the population counts are suspiciously non-decreasing. By default *load_fingerprints()* reorders the fingerprints into a data structure which is faster to search, although you can disable that if you want the fingerprints to be the same as the input order.

The *FingerprintArena* has new capabilities. You can ask it how many fingerprints it contains, get the list of identifiers, and look up a fingerprint record given an index, as in:

```
>>> len(arena)
14967
>>> arena.ids[:5]
['48942244', '48941399', '48940284', '48943050', '48656359']
>>> id, fp = arena[6]
>>> id
'48839855'
>>> arena[-1][0]
'48985180'
>>> bitops.byte_popcount(arena[-1][1])
253
```

An arena supports iterating through subarenas. This is like having a long list and being able to iterate over sublists. Here's an example of iterating over the arena to get subarenas of size 1000 (the last subarea may have fewer elements), and print information about each subarena.:

```
>>> for subarena in arena.iter_arenas(1000):
...   print subarena.ids[0], len(subarena)
...
48942244 1000
48867092 1000
48629741 1000
48795302 1000
48848217 1000
48689418 1000
48873983 1000
48503654 1000
48575094 1000
48575460 1000
48531270 1000
48960181 1000
48806978 1000
48837835 1000
48584671 967
```

(continues on next page)

(continued from previous page)

```
>>> arena[0][0]
'48942244'
>>> arena[1000][0]
'48867092'
```

To help demonstrate what's going on, I showed the first id of each record along with the main arena ids for records 0 and 1000, so you can verify that they are the same.

Arenas are a core part of chemfp. Processing one fingerprint at a time is slow, so the main search routines expect to iterate over query arenas, rather than query fingerprints.

Thus, the FPSReaders – and all chemfp fingerprint collections – also support the `iter_arenas()` interface. Here's an example of reading the targets file 2000 records at a time:

```
>>> queries = chemfp.open("pubchem_queries.fps")
>>> for arena in queries.iter_arenas(2):
...   print len(arena)
...
2000
2000
2000
2000
2000
826
```

Those add up to 10,826, which you can verify is the number of structures in the original source file.

If you have a *FingerprintArena* instance then you can also use Python's slice notation to make a subarena:

```
>>> queries = chemfp.load_fingerprints("pubchem_queries.fps")
>>> queries[10:15]
<chemfp.arena.FingerprintArena object at 0x552c10>
>>> queries[10:15].ids
['99110546', '99110547', '99123452', '99123453', '99133437']
>>> queries.ids[10:15]
['99110546', '99110547', '99123452', '99123453', '99133437']
```

The big restriction is that slices can only have a step size of 1. Slices like *[10:20:2]* and *[::-1]* aren't supported. If you want something like that then you'll need to make a new arena instead of using a subarena slice.

In case you were wondering, yes, you can use *iter_arenas* or the other FingerprintArena methods on a subarena:

```
>>> queries[10:15][1:3].ids
['99110547', '99123452']
>>> queries.ids[11:13]
['99110547', '99123452']
```

4.4 How to use query fingerprints to search for similar target fingerprints

In this section you'll learn how to do a Tanimoto search using the previously created PubChem fingerprint files for the queries and the targets.

It's faster to search an arena, so I'll load the target fingerprints:

```
>>> import chemfp
>>> targets = chemfp.load_fingerprints("pubchem_targets.fps")
>>> len(targets)
14967
```

and open the queries as an FPSReader.

```
>>> queries = chemfp.open("pubchem_queries.fps")
```

I'll use *threshold_tanimoto_search()* to find, for each query, all hits which are at least 0.7 similar to the query.

```
>>> for (query_id, hits) in chemfp.threshold_tanimoto_search(queries, targets,␣
↪threshold=0.7):
...   print query_id, len(hits), list(hits)[:2]
...
99000039 641 [(3619, 0.7085714285714285), (4302, 0.7371428571428571)]
99000230 373 [(2747, 0.703030303030303), (3608, 0.7041420118343196)]
99002251 270 [(2512, 0.7006369426751592), (2873, 0.7088607594936709)]
99003537 523 [(6697, 0.7230769230769231), (7478, 0.7085427135678392)]
99003538 523 [(6697, 0.7230769230769231), (7478, 0.7085427135678392)]
99005028 131 [(772, 0.7589285714285714), (796, 0.7522123893805309)]
99005031 131 [(772, 0.7589285714285714), (796, 0.7522123893805309)]
99006292 308 [(805, 0.7058823529411765), (808, 0.7)]
99006293 308 [(805, 0.7058823529411765), (808, 0.7)]
99006597 0 []
      # ... many lines omitted ...
```

I'm only showing the first two hits for the sake of space. It seems rather pointless, after all, to show all 641 hits of query id 99000039.

What you don't see is that the implementation uses the iter_arenas() interface on the queries so that it processes only a subarena at a time. There's a tradeoff between a large arena, which is faster because it doesn't often go back to Python code, or a small arena, which uses less memory and is more responsive. You can change the tradeoff using the *arena_size* parameter.

If all you care about is the count of the hits within a given threshold then use *chemfp.count_tanimoto_hits()*:

```
>>> queries = chemfp.open("pubchem_queries.fps")
>>> for (query_id, count) in chemfp.count_tanimoto_hits(queries, targets, threshold=0.
↪7):
...     print query_id, count
...
99000039 641
99000230 373
99002251 270
99003537 523
99003538 523
99005028 131
99005031 131
99006292 308
99006293 308
99006597 0
     # ... many lines omitted ...
```

Or, if you only want the k=2 nearest neighbors to each target within that same threshold of 0.7 then use *chemfp.knearest_tanimoto_search()*:

```
>>> queries = chemfp.open("pubchem_queries.fps")
>>> for (query_id, hits) in chemfp.knearest_tanimoto_search(queries, targets, k=2,␣
↪threshold=0.7):
...     print query_id, list(hits)
...
99000039 [(10706, 0.8784530386740331), (10551, 0.8729281767955801)]
99000230 [(8201, 0.8588235294117647), (10267, 0.8522727272727273)]
99002251 [(6939, 0.8109756097560976), (8628, 0.8106508875739645)]
99003537 [(13023, 0.9035532994923858), (12924, 0.8984771573604061)]
99003538 [(13023, 0.9035532994923858), (12924, 0.8984771573604061)]
99005028 [(906, 0.8288288288288288), (1746, 0.8166666666666667)]
     # ... many lines omitted ...
```

4.5 How to search an FPS file

In this section you'll learn how to search an FPS file directly, without loading it into a FingerprintArena.

The previous example loaded the fingerprints into a FingerprintArena. That's the fastest way to do multiple searches. Sometimes though you only want to do one or a couple of queries. It seems rather excessive to read the entire targets file into an in-memory data structure before doing the search when you could search will processing the file.

For that case, use an FPSReader as the target file. Here I'll get the first two records from the queries file and use them to search the targets file:

```
>>> query_arena = next(chemfp.open("pubchem_queries.fps").iter_arenas(2))
```

This line opens the file, iterates over its fingerprint records, and return the two as an arena. Perhaps a slightly less confusing way to write the above is:

```
>>> for query_arena in chemfp.open("pubchem_queries.fps").iter_arenas(1):
...   break
```

Here are the k=5 closest hits against the targets file:

```
>>> targets = chemfp.open("pubchem_targets.fps")
>>> for query_id, hits in chemfp.knearest_tanimoto_search(query_arena, targets, k=5,␣
↪threshold=0.0):
...   print "Hits for", query_id
...   for hit in hits:
...     print "", hit
...
Hits for 99000039
 ('48503376', 0.8784530386740331)
 ('48503380', 0.8729281767955801)
 ('48732162', 0.8595505617977528)
 ('48520532', 0.8540540540540541)
 ('48985130', 0.8449197860962567)
Hits for 99000230
 ('48563034', 0.8588235294117647)
 ('48731730', 0.8522727272727273)
 ('48583483', 0.8411764705882353)
 ('48563042', 0.8352941176470589)
 ('48935653', 0.8333333333333334)
```

Remember that the FPSReader is based on reading an FPS file. Once you've done a search, the file is read, and you can't do another search. You'll need to reopen the file.

Each search processes *arena_size* query fingerprints at a time. You will need to increase that value if you want to search more than that number of fingerprints with this method. The search performance tradeoff between a FPSReader search and loading the fingerprints into a FingerprintArena occurs with under 10 queries, so there should be little reason to worry about this.

4.6 FingerprintArena searches returning indices instead of ids

In this section you'll learn how to search a FingerprintArena and use hits based on integer indices rather than string ids.

The previous sections used a high-level interface to the Tanimoto search code. Those are designed for the common case where you just want the query id and the hits, where each hit includes the target id.

Working with strings is actually rather inefficient in both speed and memory. It's usually better to work with indices if you can, and in the next section I'll show how to make a distance matrix using this interface.

The index-based search functions are in the *chemfp.search* module. They can be categorized into three groups:

1. Count the number of hits:

- *chemfp.search.count_tanimoto_hits_fp()* - search an arena using a single fingerprint
- *chemfp.search.count_tanimoto_hits_arena()* - search an arena using an arena
- *chemfp.search.count_tanimoto_hits_symmetric()* - search an arena using itself

2. Find all hits at or above a given threshold, sorted arbitrarily:

- *chemfp.search.threshold_tanimoto_search_fp()* - search an arena using a single fingerprint
- *chemfp.search.threshold_tanimoto_search_arena()* - search an arena using an arena
- *chemfp.search.threshold_tanimoto_search_symmetric()* - search an arena using itself

3. Find the k-nearest hits at or above a given threshold, sorted by decreasing similarity:

- *chemfp.search.knearest_tanimoto_search_fp()* - search an arena using a single fingerprint
- *chemfp.search.knearest_tanimoto_search_arena()* - search an arena using an arena
- *chemfp.search.knearest_tanimoto_search_symmetric()* - search an arena using itself

The functions ending '_fp' take a query fingerprint and a target arena. The functions ending '_arena' take a query arena and a target arena. The functions ending '_symmetric' use the same arena as both the query and target.

In the following example, I'll use the first 5 fingerprints of a data set to search the entire data set. To do this, I load the data set as an arena, read the 5 records of the same file as a query arena, and do the search.

```
>>> import chemfp
>>> from chemfp import search
>>> targets = chemfp.load_fingerprints("pubchem_queries.fps")
>>> queries = next(chemfp.open("pubchem_queries.fps").iter_arenas(5))
>>> results = search.threshold_tanimoto_search_arena (queries, targets, threshold=0.7)
```

The threshold_tanimoto_search_arena search finds the target fingerprints which have a similarity score of at least 0.7 compared to the query.

You can iterate over the results to get the list of hits for each of the queries. The order of the results is the same as the order of the records in the query.:

```
>>> for hits in results:
...   print len(hits), hits.get_ids_and_scores()[:3]
...
261 [('99115962', 0.7005649717514124), ('99115963', 0.7005649717514124), ('99103967',␣
↪0.7303370786516854)]
281 [('99141183', 0.7202380952380952), ('99174339', 0.7017543859649122), ('99275524',␣
↪0.7093023255813954)]
118 [('99123562', 0.7564102564102564), ('99104138', 0.7080745341614907), ('99104141',␣
↪0.7080745341614907)]
223 [('99121080', 0.7591623036649214), ('99210542', 0.7106598984771574), ('99210544',␣
↪0.7106598984771574)]
223 [('99121080', 0.7591623036649214), ('99210542', 0.7106598984771574), ('99210544',␣
↪0.7106598984771574)]
```

This result is like what you saw earlier, except that it doesn't have the query id. You can get that from the arena's *id* attribute, which contains the list of fingerprint identifiers.

```
>>> for query_id, hits in zip(queries.ids, results):
...   print "Hits for", query_id
...   for hit in hits.get_ids_and_scores()[:3]:
...     print "", hit
Hits for 99000039
('99115962', 0.7005649717514124)
('99115963', 0.7005649717514124)
('99103967', 0.7303370786516854)
Hits for 99000230
('99141183', 0.7202380952380952)
('99174339', 0.7017543859649122)
('99275524', 0.7093023255813954)
Hits for 99002251
   ...
```

What I really want to show is that you can get the same data only using the offset index for the target record instead of its id. The result from a Tanimoto search is a *SearchResults* instance, with methods that include `SearchResults.get_indices_and_scores()`, `SearchResults.get_ids()`, and `SearchResults.get_scores()`:

```
>>> for hits in results:
...   print len(hits), hits.get_indices_and_scores()[:3]
...
261 [(2998, 0.7005649717514124), (2999, 0.7005649717514124), (3816, 0.
↪7303370786516854)]
281 [(2953, 0.7202380952380952), (3162, 0.7017543859649122), (3543, 0.
↪7093023255813954)]
118 [(2491, 0.7564102564102564), (2584, 0.7080745341614907), (2585, 0.
↪7080745341614907)]
223 [(5509, 0.7591623036649214), (5793, 0.7106598984771574), (5794, 0.
↪7106598984771574)]
223 [(5509, 0.7591623036649214), (5793, 0.7106598984771574), (5794, 0.
↪7106598984771574)]
>>>
>>> targets.ids[0]
```

(continues on next page)

(continued from previous page)

```
'99116624'
>>> targets.ids[3]
'99116668'
>>> targets.ids[15]
'99134597'
```

I did a few id lookups given the target dataset to show you that the index corresponds to the identifiers from the previous code.

These examples iterated over each individual `SearchResult` to fetch the ids and scores, or indices and scores. Another possibility is to ask the *SearchResults* collection to iterate directly over the list of fields you want.

```
>>> for row in results.iter_indices_and_scores():
...   print len(row), row[:3]
...
261 [(2998, 0.7005649717514124), (2999, 0.7005649717514124), (3816, 0.
↪7303370786516854)]
281 [(2953, 0.7202380952380952), (3162, 0.7017543859649122), (3543, 0.
↪7093023255813954)]
118 [(2491, 0.7564102564102564), (2584, 0.7080745341614907), (2585, 0.
↪7080745341614907)]
223 [(5509, 0.7591623036649214), (5793, 0.7106598984771574), (5794, 0.
↪7106598984771574)]
223 [(5509, 0.7591623036649214), (5793, 0.7106598984771574), (5794, 0.
↪7106598984771574)]
```

This was added to get a bit more performance out of chemfp and because the API is sometimes cleaner one way and sometimes cleaner than the other. Yes, I know that the Zen of Python recommends that "there should be one– and preferably only one –obvious way to do it." Oh well.

4.7 Computing a distance matrix for clustering

In this section you'll learn how to compute a distance matrix using the chemfp API.

chemfp does not do clustering. There's a huge number of tools which already do that. A goal of chemfp in the future is to provide some core components which clustering algorithms can use.

That's in the future. Right now you can use the following to build a distance matrix and pass that to one of those tools.

Since we're using the same fingerprint arena for both queries and targets, we know the distance matrix will be symmetric along the diagonal, and the diagonal terms will be 1.0. The *chemfp.search.threshold_tanimoto_search_symmetric()* functions can take advantage of the symmetry for a factor of two performance gain. There's also a way to limit it to just the upper triangle, which gives a factor of two memory gain as well.

Most of those tools use NumPy, which is a popular third-party package for numerical computing. You will need to have it installed for the following to work.

```
import numpy  # NumPy must be installed
from chemfp import search

# Compute distance[i][j] = 1-Tanimoto(fp[i], fp[j])

def distance_matrix(arena):
    n = len(arena)
```

(continues on next page)

(continued from previous page)

```
    # Start off a similarity matrix with 1.0s along the diagonal
    similarities = numpy.identity(n, "d")

    ## Compute the full similarity matrix.
    # The implementation computes the upper-triangle then copies
    # the upper-triangle into lower-triangle. It does not include
    # terms for the diagonal.
    results = search.threshold_tanimoto_search_symmetric(arena, threshold=0.0)

    # Copy the results into the NumPy array.
    for row_index, row in enumerate(results.iter_indices_and_scores()):
        for target_index, target_score in row:
            similarities[row_index, target_index] = target_score

    # Return the distance matrix using the similarity matrix
    return 1.0 - similarities
```

Once you've computed the distance matrix, clustering is easy. I installed the hcluster package, as well as matplotlib, then ran the following to see the hierarchical clustering:

```
import chemfp
import hcluster # Clustering package from http://code.google.com/p/scipy-cluster/

# ... insert the 'distance_matrix' function definition here ...

dataset = chemfp.load_fingerprints("pubchem_queries.fps")
distances  = distance_matrix(dataset)

linkage = hcluster.linkage(distances, method="single", metric="euclidean")

# Plot using matplotlib, which you must have installed
hcluster.dendrogram(linkage, labels=dataset.ids)

import pylab
pylab.show()
```

In practice you'll almost certainly want to use one of the scikit-learn clustering algorithms.

4.8 Convert SearchResults to a SciPy csr matrix

In this section you'll learn how to convert a SearchResults object into a SciPy compressed sparse row matrix.

In the previous section you learned how to use the chemfp API to create a NumPy similarity matrix, and convert that into a distance matrix. The result is a dense matrix, and the amount of memory goes as the square of the number of structures.

If you have a reasonably high similarity threshold, like 0.7, then most of the similarity scores will be zero. Internally the *SearchResults* object only stores the non-zero values for each row, along with an index to specify the column. This is a common way to compress sparse data.

SciPy has its own compressed sparse row ("csr") matrix data type, which can be used as input to many of the scikit-learn clustering algorithms.

If you want to use those algorithms, call the *SearchResults.to_csr()* method to convert the SearchResults scores (and only the scores) into a csr matrix. The rows will be in the same order as the SearchResult (and the original

queries), and the columns will be in the same order as the target arena, including its ids.

I don't know enough about scikit-learn to give a useful example. (If you do, let me know!) Instead, I'll start by doing an NxM search of two sets of fingerprints:

```
from __future__ import print_function
import chemfp
from chemfp import search

queries = chemfp.load_fingerprints("pubchem_queries.fps")
targets = chemfp.load_fingerprints("pubchem_targets.fps")
results = search.threshold_tanimoto_search_arena(queries, targets, threshold = 0.8)
```

The SearchResults attribute *shape* describes the number of rows and columns:

```
>>> results.shape
(10826, 14967)
>>> len(queries)
10826
>>> len(targets)
14967
>>> results[2001].get_indices_and_scores()
[(2031, 0.8770491803278688), (2032, 0.8770491803278688)]
```

I'll turn it into a SciPy csr:

```
>>> csr = results.to_csr()
>>> csr
<10826x14967 sparse matrix of type '<type 'numpy.float64'>'
    with 369471 stored elements in Compressed Sparse Row format>
>>> csr.shape
(10826, 14967)
```

and look at the same row to show it has the same indices and scores:

```
>>> csr[2001]
<1x14967 sparse matrix of type '<type 'numpy.float64'>'
    with 2 stored elements in Compressed Sparse Row format>
>>> csr[2001].indices
array([2031, 2032], dtype=int32)
>>> csr[2001].data
array([0.87704918, 0.87704918])
```

4.9 Taylor-Butina clustering

For the last clustering example, here's my (non-validated) variation of the Butina algorithm from JCICS 1999, 39, 747-750. See also http://www.redbrick.dcu.ie/~noel/R_clustering.html . You might know it as Leader clustering.

First, for each fingerprint find all other fingerprints with a threshold of 0.8:

```
import chemfp
from chemfp import search

arena = chemfp.load_fingerprints("pubchem_targets.fps")
results = search. threshold_tanimoto_search_symmetric (arena, threshold = 0.8)
```

Sort the results so that fingerprints with more hits come first. This is more likely to be a cluster centroid. Break ties arbitrarily by the fingerprint id; since fingerprints are ordered by the number of bits this likely makes larger structures appear first.:

```
# Reorder so the centroid with the most hits comes first.
# (That's why I do a reverse search.)
# Ignore the arbitrariness of breaking ties by fingerprint index
results = sorted( (  (len(indices), i, indices)
                          for (i,indices) in enumerate(results.iter_indices())  ),
                  reverse=True)
```

Apply the leader algorithm to determine the cluster centroids and the singletons:

```
# Determine the true/false singletons and the clusters
true_singletons = []
false_singletons = []
clusters = []

seen = set()
for (size, fp_idx, members) in results:
    if fp_idx in seen:
        # Can't use a centroid which is already assigned
        continue
    seen.add(fp_idx)

    # Figure out which ones haven't yet been assigned
    unassigned = set(members) - seen

    if not unassigned:
        false_singletons.append(fp_idx)
        continue

    # this is a new cluster
    clusters.append( (fp_idx, unassigned) )
    seen.update(unassigned)
```

Once done, report the results:

```
print len(true_singletons), "true singletons"
print "=>", " ".join(sorted(arena.ids[idx] for idx in true_singletons))
print

print len(false_singletons), "false singletons"
print "=>", " ".join(sorted(arena.ids[idx] for idx in false_singletons))
print

# Sort so the cluster with the most compounds comes first,
# then by alphabetically smallest id
def cluster_sort_key(cluster):
    centroid_idx, members = cluster
    return -len(members), arena.ids[centroid_idx]

clusters.sort(key=cluster_sort_key)

print len(clusters), "clusters"
for centroid_idx, members in clusters:
    print arena.ids[centroid_idx], "has", len(members), "other members"
    print "=>", " ".join(arena.ids[idx] for idx in members)
```

The algorithm is quick for this small data set. (Less than a second.)

Out of curiosity, I tried this on 100,000 compounds selected arbitrarily from PubChem. It took 35 seconds on my desktop (a 3.2 GHZ Intel Core i3) with a threshold of 0.8. In the Butina paper, it took 24 hours to do the same, although that was with a 1024 bit fingerprint instead of 881. It's hard to judge the absolute speed differences of a MIPS R4000 from 1998 to a desktop from 2011, but it's less than the factor of about 2000 you see here.

More relevent is the comparison between these numbers for the 1.1 release compared to the original numbers for the 1.0 release. On my old laptop, may it rest it peace, it took 7 minutes to compute the same benchmark. Where did the roughly 16-fold peformance boost come from? Money. After 1.0 was released, Roche funded me to add various optimizations, including taking advantage of the symmetery (2x) and using hardware POPCNT if available (4x). Roche and another company helped fund the OpenMP support, and when my desktop reran this benchmark it used 4 cores instead of 1.

The wary among you might notice that 2*4*4 = 32x faster, while I said the overall code was only 16x faster. Where's the factor of 2x slowdown? It's in the Python code! The *chemfp.search.threshold_tanimoto_search_symmetric()* step took only 13 seconds. The remaining 22 seconds was in the leader code written in Python. To make the analysis more complicated, improvements to the chemfp API sped up the clustering step by about 40%.

With chemfp 1.0 version, the clustering performance overhead was minor compared to the full similarity search, so I didn't keep track of it. With chemfp 1.1, those roles have reversed!

Update for chemfp 1.6 in 2020: I re-ran the same algorithm on an even newer Mac laptop, with a single thread on an 2.3 GHz Intel Core i5. It took 19 seconds. The laptop is more powerful, and chemfp 1.6 added an even faster search implementation. (The commercial version, chemfp 3, is faster still.)

4.10 Reading structure fingerprints using a toolkit

In this section you'll learn how to use a chemistry toolkit in order to compute fingerprints from a given structure file.

NOTE: this is here mostly for historical interest. chemfp 1.6 was released in 2020. None of the underlying chemistry toolkits support Python 2.7 and I no longer have a working setup where I can test the older toolkits. This section shows the output from chemfp 1.4.

What happens if you're given a structure file and you want to find the two nearest matches in an FPS file? You'll have to generate the fingerprints for the structures in the structure file, then do the comparison.

For this section you'll need to have a chemistry toolkit. I'll use the "chebi_maccs.fps" file generated in *Using a toolkit to process the ChEBI dataset* as the targets, and the PubChem file Compound_027575001_027600000.sdf.gz as the source of query structures:

```
>>> import chemfp
>>> from chemfp import search
>>> targets = chemfp.load_fingerprints("chebi_maccs.fps")
>>> queries = chemfp.read_molecule_fingerprints(targets.metadata, "Compound_027575001_
↪027600000.sdf.gz")
>>> for (query_id, hits) in chemfp.knearest_tanimoto_search(queries, targets, k=2,␣
↪threshold=0.4):
...   print query_id, "=>",
...   for (target_id, score) in hits.get_ids_and_scores():
...     print "%s %.3f" % (target_id, score),
...   print
...
27575190 => CHEBI:116551 0.779 CHEBI:105622 0.771
27575192 => CHEBI:105622 0.809 CHEBI:108425 0.809
27575198 => CHEBI:109833 0.736 CHEBI:105937 0.730
```

(continues on next page)

(continued from previous page)

```
27575208 => CHEBI:105622 0.783 CHEBI:108425 0.783
27575240 => CHEBI:91516 0.747 CHEBI:111326 0.737
27575250 => CHEBI:105622 0.809 CHEBI:108425 0.809
27575257 => CHEBI:105622 0.732 CHEBI:108425 0.732
27575282 => CHEBI:126087 0.764 CHEBI:127676 0.764
27575284 => CHEBI:105622 0.900 CHEBI:108425 0.900
     # ... many lines omitted ...
```

That's it! Pretty simple, wasn't it? You didn't even need to explictly specify which toolkit you wanted to use.

The only new thing here is *chemfp.read_molecule_fingerprints()*. The first parameter of this is the metadata used to configure the reader. In my case it's:

```
>>> print targets.metadata
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2017.09.1.dev1 chemfp/1.4
#source=ChEBI_lite.sdf.gz
#date=2017-09-14T11:19:31
```

The "type" told chemfp which toolkit to use to read molecules, and how to generate fingerprints from those molecules, while "aromaticity" told it which aromaticity model to use when reading the molecule file.

You can instead course pass in your own metadata as the first parameter to read_molecule_fingerprints, and as a shortcut, if you pass in a string then it will be used as the fingerprint type.

For examples, if you have OpenBabel installed then you can do:

```
>>> from chemfp.bitops import hex_encode
>>> reader = chemfp.read_molecule_fingerprints("OpenBabel-MACCS", "Compound_027575001_
↪027600000.sdf.gz")
>>> for i, (id, fp) in enumerate(reader):
...   print id, hex_encode(fp)
...   if i == 3:
...     break
...
27575433 800404000840549e848189cca1f132aedfab6eff1b
27575577 800400000000449e850581c22190022f8a8baadf1b
27575602 000000000000449e840191d820a0122eda9abaff1b
27575603 000000000000449e840191d820a0122eda9abaff1b
```

If you have OEChem and OEGraphSim installed then you can do:

```
>>> from chemfp.bitops import hex_encode
>>> reader = chemfp.read_molecule_fingerprints("OpenEye-MACCS166", "Compound_
↪027575001_027600000.sdf.gz")
>>> for i, (id, fp) in enumerate(reader):
...   print id, hex_encode(fp)
...   if i == 3:
...     break
...
27575433 000000080840448e8481cdccb1f1b216daaa6a7e3b
27575577 000000080000448e850185c2219082178a8a6a5e3b
27575602 000000080000448e8401d14820a01216da983b7e3b
27575603 000000080000448e8401d14820a01216da983b7e3b
```

And if you have RDKit installed then you can do:

```
>>> from chemfp.bitops import hex_encode
>>> reader = chemfp.read_molecule_fingerprints("RDKit-MACCS166", "Compound_027575001_
↪027600000.sdf.gz")
>>> for i, (id, fp) in enumerate(reader):
...   print id, hex_encode(fp)
...   if i == 3:
...     break
...
27575433 000000000840549e84818dccb1f1323cdfab6eff1f
27575577 000000000000449e850185c22190023d8a8beadf1f
27575602 000000000000449e8401915820a0123eda98bbff1f
27575603 000000000000449e8401915820a0123eda98bbff1f
```

4.11 Select a fingerprint subset using a list of indicies

In this section you'll learn how to make a new arena given a list of indices for the fingerprints to select from an old arena.

For this section, one example will use indices will be a randomly selected subset of the indices in the fingerprint. If you want to sample a next section for an easier way to do this using *FingerprintArena.sample()*. If you want to split the arena into a training set and a test set, see the section after that which shows how to use *FingerprintArena.train_test_split()*.

A FingerprintArena slice creates a subarena. Technically speaking, this is a "view" of the original data. The subarena doesn't actually copy its fingerprint data from the original arena. Instead, it uses the same fingerprint data, but keeps track of the start and end position of the range it needs. This is why it's not possible to slice with a step size other than +1.

This also means that memory for a large arena won't be freed until all of its subarenas are also removed.

You can see some evidence for this because a *FingerprintArena* stores the entire fingerprint data as a set of bytes named *arena*:

```
>>> import chemfp
>>> targets = chemfp.load_fingerprints("pubchem_targets.fps")
>>> subset = targets[10:20]
>>> targets.arena is subset.arena
True
```

This shows that the *targets* and *subset* share the same raw data set. At least it shows that to me, the person who wrote the code.

You can ask an arena or subarena to make a *FingerprintArena.copy()*. This allocates new memory for the new arena and copies all of its fingerprints there.

```
>>> new_subset = subset.copy()
>>> len(new_subset) == len(subset)
>>> new_subset.arena is subset.arena
False
>>> subset[7][0]
'48637548'
>>> new_subset[7][0]
'48637548'
```

The *FingerprintArena.copy()* method can do more than just copy the arena. You can give it a list of indices and it will only copy those fingerprints:

```
>>> three_targets = targets.copy([3112, 0, 1234])
>>> three_targets.ids
['48942244', '48568841', '48628197']
>>> [targets.ids[3112], targets.ids[0], targets.ids[1234]]
['48628197', '48942244', '48568841']
```

Are you confused about why the identifiers aren't in the same order? That's because when you specify indicies, the copy automatically reorders them by popcount and stores the popcount information. This extra work help makes future searches faster. Use `reorder=False` to leave the order unchanged

```
>>> my_ordering = targets.copy([3112, 0, 1234], reorder=False)
>>> my_ordering.ids
['48628197', '48942244', '48568841']
```

This is interesting, I guess, in a boring sort of way.

Suppose you want to partition the data set into two parts; one containing the fingerprints at positions 0, 2, 4, ... and the other containing the fingerprints at positions 1, 3, 5, ... The *range()* function returns a list of the right length, and you can have it start from either 0 or 1 and count by twos, like this:

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

so the following will create the correct indices and from that the correct arena subsets:

```
>>> evens = targets.copy(range(0, len(targets), 2))
>>> odds = targets.copy(range(1, len(targets), 2))
>>> len(evens)
7484
>>> len(odds)
7483
```

(Use *FingerprintArena.train_test_split()* if you want to select two disjoint subsets selected at random without replacement.)

What about getting a random subset of the data? I want to select *m* records at random, without replacement, to make a new data set. (See the next section for a better way to do this using *FingerprintArena.sample()*.)

You can see this just means making a list of indices with *m* different index values. Python's built-in random.sample function makes this easy:

```
>>> import random
>>> random.sample("abcdefgh", 3)
['b', 'h', 'f']
>>> random.sample("abcdefgh", 2)
['d', 'a']
>>> random.sample([5, 6, 7, 8, 9], 2)
[7, 9]
>>> help(random.sample)
sample(self, population, k) method of random.Random instance
   Chooses k unique random elements from a population sequence.
   ...
   To choose a sample in a range of integers, use xrange as an argument.
   This is especially fast and space efficient for sampling from a
   large population:   sample(xrange(10000000), 60)
```

The last line of the help points out what do next!:

```
>>> random.sample(xrange(len(targets)), 5)
[610, 2850, 705, 1402, 2635]
>>> random.sample(xrange(len(targets)), 5)
[1683, 2320, 1385, 2705, 1850]
```

Putting it all together, and here's how to get a new arena containing 100 randomly selected fingerprints, without replacement, from the *targets* arena:

```
>>> sample_indices = random.sample(xrange(len(targets)), 100)
>>> sample = targets.copy(indices=sample_indices)
>>> len(sample)
100
```

4.12 Sample N fingerprints at random

In this section you'll learn how to select a random subset of the fingerprints in an arena.

The previous section showed how to use the *FingerprintArena.copy()* method to create a new arena containing a randomly selected subset of the fingerprints in an arena. This required writing some code to specify the randomly samples indices.

Chemfp 1.6.1 added the method *FingerprintArena.sample()* which lets you make a random sample using a single call:

```
>>> import chemfp
>>> targets = chemfp.load_fingerprints("pubchem_targets.fps")
>>> sample_arena = targets.sample(10000)
>>> len(sample_arena)
10000
>>> sample_arena.ids[:5]
['48942244', '48941399', '48940284', '48943050', '48656867']
```

If you do the sample a few times you'll see that many of the elements occur often:

```
>>> targets.sample(10000).ids[:5]
['48940284', '48943050', '48656359', '48966209', '48946425']
>>> targets.sample(10000).ids[:5]
['48940284', '48943050', '48656867', '48839855', '48946668']
>>> targets.sample(10000).ids[:5]
['48942244', '48941399', '48940284', '48943050', '48656359']
>>> targets.sample(10000).ids[:5]
['48942244', '48656359', '48656867', '48839855', '48966209']
```

This is for two reasons. First, the sample size is about 2/3rds of the size of the the data set:

```
>>> len(targets)
14967
```

which means there's a roughly 2/3rds chance that a given record will be in the sample. Second, by default the sampled fingerprints are reordered by popcount when making the arena, which means many of the first few identifiers are the same.

Set *reorder* to *False* to keep the fingerprints in random sample order:

```
>>> targets.sample(10000, reorder=False).ids[:5]
['48650979', '48932835', '48741156', '48946513', '48518719']
>>> targets.sample(10000, reorder=False).ids[:5]
['48526403', '48599308', '48645719', '48575346', '48736396']
>>> targets.sample(10000, reorder=False).ids[:5]
['48583570', '48666587', '48862252', '48942877', '48574505']
>>> targets.sample(10000, reorder=False).ids[:5]
['48666554', '48676514', '48586264', '48688145', '48634017']
```

Remember that similarity search performance is better if the the fingerprints are sorted by popcount.

The above examples used *num_samples=10000*. If *num_samples* is an integer, then it's used as the number of samples to make. (Chemfp raises a ValueError if the size is negative or too large.) If *num_samples* is a float between 0.0 and 1.0 inclusive then it's used as the fraction of the dataset to sample. For example, the following samples 10% of the arena, rounded down:

```
>>> len(targets.sample(0.1))
1496
```

If no *rng* is given then the underlying implementation uses Python's random.sample function. That in turn uses a random number generator (RNG) which was initialized with a hard-to-guess seed.

If you need a reproducible sample, you can pass in an integer *rng* value. This is used to seed a new RNG for the sampling. In the following example, using the same seed always returns the same fingerprints:

```
>>> targets.sample(2, rng=123).ids
['48914963', '48920139']
>>> targets.sample(2, rng=123).ids
['48914963', '48920139']
>>> targets.sample(2, rng=789).ids
['48966001', '48982750']
>>> targets.sample(2, rng=789).ids
['48966001', '48982750']
```

Another option is pass in a random.Random() instance, which will be used directly as the RNG:

```
>>> import random
>>> my_rng = random.Random(123)
>>> targets.sample(2, rng=my_rng).ids
['48914963', '48920139']
>>> targets.sample(2, rng=my_rng).ids
['48574157', '48626955']
>>> targets.sample(2, rng=my_rng).ids
['48940920', '48983572']
```

This may be useful if you need to make several random samples, want reproducibility, and only want to specify one RNG seed.

4.13 Split into training and test sets

In this section you'll learn how to split an arena into two disjoint arenas, which can be then be used as a training set and a test set.

The previous section showed how to use chemfp to select N fingerprints at random from an arena. Sometimes you need two randomly selected subsets, with no overlap between the two. For example, one might be used as a training set and the other as a test set.

Chemfp 1.6.1 added the method *FingerprintArena.train_test_split()* which does that. You give it the number of fingerprints you want in the training set and/or the test set, and it returns two arenas; the first is the training set and the second is the test set:

```
>>> import chemfp
>>> targets = chemfp.load_fingerprints("pubchem_targets.fps")
>>> len(targets)
14967
>>> train_arena, test_arena = targets.train_test_split(train_size=10, test_size=5)
>>> len(train_arena)
10
>>> len(test_arena)
5
```

This function is modeled on the scikit learn function train_test_split() , which allows for the sizes to be specified as an integer number or a floating point fraction.

If a specified size is an integer, it is interpreted at the number of fingerprints to have in the corresponding set. If a specified size is a float between 0.0 and 1.0 inclusive then it's interpreted as the fraction of fingerprints to select. For example, the following puts 10% of the fingerprints into the training arena and 20 fingerprints

```
>>> train_arena, test_arena = targets.train_test_split(train_size=0.1, test_size=20)
>>> len(train_arena), len(test_arena)
(1496, 20)
```

If you don't specify the test or arena size then the training set gets 75% of the fingerprints and the test set gets the rest:

```
>>> train_arena, test_arena = targets.train_test_split()
>>> len(train_arena), len(test_arena)
(11226, 3741)
```

If only one of *train_size* or *test_size* is specified then the other value is interpreted as the complement size, so the entire arena is split into the two sets. In the following, 75% of the fingerprints are put into the training arena and 25% into the test arena:

```
>>> train_arena, test_arena = targets.train_test_split(train_size=0.75)
>>> len(train_arena), len(test_arena)
(11225, 3742)
```

It is better to let chemp figure out the complement size than to specify both sizes as a float because integer rounding may cause a fingerprint to be left out (the test arena size is 3741 in the following when it should be 3742):

```
>>> train_arena, test_arena = targets.train_test_split(train_size=0.75, test_size=0.
↪25)
>>> len(train_arena), len(test_arena)
(11225, 3741)
```

By default, after the random sampling the fingerprints in each set are reordered by population count and indexed for fast similarity search.

```
>>> from chemfp import bitops
>>> train_arena, test_arena = targets.train_test_split(10, 10)
>>> [bitops.byte_popcount(train_arena.get_fingerprint(i)) for i in range(10)]
[71, 118, 119, 145, 146, 159, 162, 167, 176, 196]
>>> [bitops.byte_popcount(test_arena.get_fingerprint(i)) for i in range(10)]
[87, 116, 117, 121, 129, 131, 139, 183, 184, 193]
```

To keep the fingerprints in random sample order, specify *reorder=False*:

```
>>> train_arena, test_arena = targets.train_test_split(10, 10, reorder=False)
>>> [bitops.byte_popcount(train_arena.get_fingerprint(i)) for i in range(10)]
[118, 53, 170, 110, 138, 169, 129, 125, 129, 151]
>>> [bitops.byte_popcount(test_arena.get_fingerprint(i)) for i in range(10)]
[172, 167, 123, 152, 147, 162, 156, 197, 45, 151]
```

The *rng* parameter affects how the fingerprints are samples. By default (if *rng=None*), Python's default RNG is used. If *rng* is an integer then it's used as the seed for a new random.Random() instance. Otherwise it's assumed to be an RNG object and its sample() method is used to make the sample.

The parameter works the same as *FingerprintArena.sample()* so for examples see the previous section.

4.14 Look up a fingerprint with a given id

In this section you'll learn how to get a fingerprint record with a given id.

All fingerprint records have an identifier and a fingerprint. Identifiers should be unique. (Duplicates are allowed, and if they exist then the lookup code described in this section will arbitrarily decide which record to return. Once made, the choice will not change.)

Let's find the fingerprint for the record in "pubchem_targets.fps" which has the identifier *48626981*. One solution is to iterate over all of the records in a file, using the FPS reader:

```
>>> import chemfp
>>> for id, fp in chemfp.open("pubchem_targets.fps"):
...   if id == "48626981":
...     break
... else:
...   raise KeyError("%r not found" % (id,))
...
>>> fp[:5]
'\x07\xde\x1c\x00\x00'
```

I used the somewhat obscure *else* clause to the *for* loop. If the *for* finishes without breaking, which would happen if the identifier weren't present, then it will raise an exception saying that it couldn't find the given identifier.

If the fingerprint records are already in a *FingerprintArena* then there's a better solution. Use the *FingerprintArena.get_fingerprint_by_id()* method to get the fingerprint byte string, or *None* if the identifier doesn't exist:

```
>>> arena = chemfp.load_fingerprints("pubchem_targets.fps")
>>> fp = arena.get_fingerprint_by_id("48626981")
>>> fp[:5]
'\x07\xde\x1c\x00\x00'
>>> missing_fp = arena.get_fingerprint_by_id("does-not-exist")
>>> missing_fp
>>> missing_fp is None
True
```

Internally this does about what you think it would. It uses the arena's *id* list to make a lookup table mapping identifier to index, and caches the table for later use. Given the index, it's very easy to get the fingerprint.

In fact, you can get the index and do the record lookup yourself:

```
>>> arena.get_index_by_id("48626981")
11223
```

(continues on next page)

(continued from previous page)

```
>>> fp_index = arena.get_index_by_id("48626981")
>>> arena[fp_index]
('48626981', '\x07\xde\x1c\x00  ... many bytes deleted ...')
```

4.15 Sorting search results

In this section you'll learn how to sort the search results.

The k-nearest searches return the hits sorted from highest score to lowest, and break ties arbitrarily. This is usually what you want, and the extra cost to sort is small (k*log(k)) compared to the time needed to maintain the internal heap (N*log(k)).

By comparison, the threshold searches return the hits in arbitrary order. Sorting takes up to N*log(N) time, which is extra work for those cases where you don't want sorted data. Use the `SearchResult.reorder()` method if you want the hits sorted in-place:

```
>>> import chemfp
>>> arena = chemfp.load_fingerprints("pubchem_queries.fps")
>>> query_fp = arena.get_fingerprint_by_id("99129158")
>>> from chemfp import search
>>> result = search.threshold_tanimoto_search_fp(query_fp, arena, threshold=0.90)
>>> len(result)
5
>>> for pair in result.get_ids_and_scores():
...   print pair
...
('99129178', 0.9733333333333334)
('99129047', 0.9166666666666666)
('99129278', 0.9166666666666666)
('99129158', 1.0)
('99129260', 0.9548387096774194)
>>> result.reorder("decreasing-score")
>>> result.reorder("decreasing-score")
>>> for pair in result.get_ids_and_scores():
...   print pair
...
('99129158', 1.0)
('99129178', 0.9733333333333334)
('99129260', 0.9548387096774194)
('99129047', 0.9166666666666666)
('99129278', 0.9166666666666666)
>>> result.reorder("increasing-score")
>>> for pair in result.get_ids_and_scores():
...   print pair
...
('99129047', 0.9166666666666666)
('99129278', 0.9166666666666666)
('99129260', 0.9548387096774194)
('99129178', 0.9733333333333334)
('99129158', 1.0)
```

There are currently six different sort methods, all specified by name. These are

- increasing-score: sort by increasing score
- decreasing-score: sort by decreasing score

- increasing-index: sort by increasing target index
- decreasing-index: sort by decreasing target index
- reverse: reverse the current ordering
- move-closest-first: move the hit with the highest score to the first position

The first two should be obvious from the examples. If you find something useful for the next two then let me know. The "reverse" option reverses the current ordering, and is most useful if you want to reverse the sorted results from a k-nearest search.

The "move-closest-first" option exists to improve the leader algorithm stage used by the Taylor-Butina algorithm. The newly seen compound is either in the same cluster as its nearest neighbor or it is the new centroid. I felt it best to implement this as a special reorder term, rather than one of the other possible options.

If you are interested in other ways to help improve your clustering performance, let me know.

Each *SearchResult* has a `SearchResult.reorder()` method. If you want to reorder all of the hits of a *SearchResults* then use its *SearchResults.reorder_all()* method:

```
>>> similarity_matrix = search.threshold_tanimoto_search_symmetric(
...                         arena, threshold=0.8)
>>> for query_id, row in zip(arena.ids, similarity_matrix):
...   if len(row) == 3:
...     print query_id, "->", row.get_ids_and_scores()
...
99110554 -> [('99110555', 1.0), ('99110552', 0.8214285714285714), ('99110553', 0.
↪8214285714285714)]
99110555 -> [('99110552', 0.8214285714285714), ('99110553', 0.8214285714285714), (
↪'99110554', 1.0)]
99110556 -> [('99110557', 1.0), ('99110552', 0.8214285714285714), ('99110553', 0.
↪8214285714285714)]
       ... many lines omitted ...
>>> similarity_matrix.reorder_all("decreasing-score")
>>> for query_id, row in zip(arena.ids, similarity_matrix):
...   if len(row) == 3:
...     print query_id, "->", row.get_ids_and_scores()
...
99110554 -> [('99110555', 1.0), ('99110552', 0.8214285714285714), ('99110553', 0.
↪8214285714285714)]
99110555 -> [('99110554', 1.0), ('99110552', 0.8214285714285714), ('99110553', 0.
↪8214285714285714)]
99110556 -> [('99110557', 1.0), ('99110552', 0.8214285714285714), ('99110553', 0.
↪8214285714285714)]
```

It takes the same set of ordering names as *SearchResult.reorder()*.

4.16 Working with raw scores and counts in a range

In this section you'll learn how to get the hit counts and raw scores for a interval.

The length of the *SearchResult* is the number of hits it contains:

```
>>> import chemfp
>>> from chemfp import search
>>> arena = chemfp.load_fingerprints("pubchem_targets.fps")
>>> fp = arena.get_fingerprint_by_id("48692333")
```

(continues on next page)

(continued from previous page)

```
>>> result = search.threshold_tanimoto_search_fp(fp, arena, threshold=0.2)
>>> len(result)
14888
```

This gives you the number of hits at or above a threshold of 0.2, which you can also get by doing *chemfp.search.count_tanimoto_hits_fp()*. The result also stores the hits, and you can get the number of hits which are within a specified interval. Here are the hits counts at or above 0.5, 0.80, and 0.95:

```
>>> result.count(0.5)
8976
>>> result.count(0.8)
150
>>> result.count(0.85)
24
>>> result.count(0.9)
0
```

The first parameter, *min_score*, specifies the minimum threshold. The second, *max_score*, specifies the maximum. Here's how to get the number of hits with a score of at most 0.95 and 0.5:

```
>>> result.count(max_score=0.95)
14865
>>> result.count(max_score=0.5)
6035
```

If you work do the addition for the min/max score of 0.5 you'll realize that 8976 + 6035 equals 15011 which is 123 elements larger than the result size of 14888. This is because the default interval uses a closed range, and there are 123 hits with a score of exactly 0.5:

```
>>> result.count(0.5, 0.5)
26
```

The third parameter, *interval*, specifies the end conditions. The default is "[]" which means that both ends are closed. The interval "()" means that both ends are open, and "[)" and "(]" are the two half-open/half-closed ranges. To get the number of hits below 0.5 and the number of hits at or above 0.5 then you might use:

```
>>> result.count(None, 0.5, "[)")
5912
>>> result.count(0.5, None, "[]")
8976
```

to get the expected results. (A min or max of *None* means that there is respectively no lower or no upper bound.)

Now for something a bit fancier. Suppose you have two sets of structures. How well do they compare to each other? I can think of various ways to do it. One is to look at a comparison profile. Find all NxM comparisons between the two sets. How many of the hits have a threshold of 0.2? How many at 0.5? 0.95?

If there are "many", then the two sets are likely more similar than not. If the answer is "few", then they are likely rather distinct.

I'll be more specific. Are the coenzyme A-like structures in ChEBI more similar to the penicillin-like structures than you would expect by comparing two randomly chosen subsets? By similar, I'll use Tanimoto similarity of the "chebi_maccs.fps" file created in the *Using a toolkit to process the ChEBI dataset* command-line tool example.

The CHEBI id for coenzyme A is CHEBI:15346 and for penicillin is CHEBI:17334. I'll define the "coenzyme A-like" structures as the 117 structures where the fingerprint is at least 0.95 similar to coenzyme A, and "penicillin-like" as the 15 structures at least 0.90 similar to penicillin. This gives 1755 total comparisons.

You know enough to do this, but there's a nice optimization I haven't told you about. You can get the total count of all of the threshold hits using the *SearchResults.count_all()* method, instead of looping over each *SearchResult* and calling its *SearchResult.count()*:

```
import chemfp
from chemfp import search

def get_neighbors_as_arena(arena, id, threshold):
    fp = arena.get_fingerprint_by_id(id)
    neighbor_results =  search.threshold_tanimoto_search_fp(fp, chebi,␣
↪threshold=threshold)
    neighbor_arena = arena.copy(neighbor_results.get_indices())
    return neighbor_arena

chebi = chemfp.load_fingerprints("chebi_maccs.fps")

# coenzyme A
coA_arena = get_neighbors_as_arena(chebi, "CHEBI:15346", threshold=0.95)
print len(coA_arena), "coenzyme A-like structures"

# penicillin
penicillin_arena = get_neighbors_as_arena(chebi, "CHEBI:17334", threshold=0.9)
print len(penicillin_arena), "penicillin-like structures"

# I'll compute a profile at different thresholds
thresholds = [0.3, 0.35, 0.4, 0.45, 0.5, 0.6, 0.7, 0.8, 0.9]

# Compare the two sets. (For this case the speed difference between a threshold
# of 0.25 and 0.0 is not noticible, but having it makes me feel better.)
coA_against_penicillin_result= search.threshold_tanimoto_search_arena(
    coA_arena, penicillin_arena, threshold=min(thresholds))

# Show a similarity profile
print "Counts  coA/penicillin"
for threshold in thresholds:
    print " %.2f      %5d" % (threshold,
                              coA_against_penicillin_result.count_all(min_
↪score=threshold))
```

This gives a not very useful output:

```
261 coenzyme A-like structures
8 penicillin-like structures
Counts  coA/penicillin
 0.30       2088
 0.35       2088
 0.40       2087
 0.45       1113
 0.50          0
 0.60          0
 0.70          0
 0.80          0
 0.90          0
```

It's not useful because it's not possible to make any decisions from this. Are the numbers high or low? It should be low, because these are two quite different structure classes, but there's nothing to compare it against.

I need some sort of background reference. What I'll two is construct two randomly chosen sets, one with 117 fingerprints and the other with 15, and generate the same similarity profile with them. That isn't quite fair, since randomly

chosen sets will most likely be diverse. Instead, I'll pick one fingerprint at random, then get its 117 or 15, respectively, nearest neighbors as the set members:

```
# Get background statistics for random similarity groups of the same size
import random

# Find a fingerprint at random, get its k neighbors, return them as a new arena
def get_random_fp_and_its_k_neighbors(arena, k):
    fp = arena[random.randrange(len(arena))][1]
    similar_search = search.knearest_tanimoto_search_fp(fp, arena, k)
    return arena.copy(similar_search.get_indices())
```

I'll construct 1000 pairs of sets this way, accumulate the threshold profile, and compare the CoA/penicillin profile to it:

```
# Initialize the threshold counts to 0
total_background_counts = dict.fromkeys(thresholds, 0)

REPEAT = 1000
for i in range(REPEAT):
    # Select background sets of the same size and accumulate the threshold count␣
↪totals
    set1 = get_random_fp_and_its_k_neighbors(chebi, len(coA_arena))
    set2 = get_random_fp_and_its_k_neighbors(chebi, len(penicillin_arena))
    background_search = search.threshold_tanimoto_search_arena(set1, set2,␣
↪threshold=min(thresholds))
    for threshold in thresholds:
        total_background_counts[threshold] += background_search.count_all(min_
↪score=threshold)

print "Counts  coA/penicillin  background"
for threshold in thresholds:
    print " %.2f       %5d           %5d" % (threshold,
                                            coA_against_penicillin_result.count_
↪all(min_score=threshold),
                                            total_background_counts[threshold] /␣
↪(REPEAT+0.0))
```

Your output should look something like:

```
Counts  coA/penicillin  background
 0.30       2088            882
 0.35       2088            698
 0.40       2087            550
 0.45       1113            413
 0.50          0            322
 0.60          0            156
 0.70          0             58
 0.80          0             20
 0.90          0              5
```

This is a bit hard to interpret. Clearly the coenzyme A and penicillin sets are not closely similar, but for low Tanimoto scores the similarity is higher than expected.

That difficulty is okay for now because I mostly wanted to show an example of how to use the chemfp API. If you want to dive deeper into this sort of analysis then read a three-part series I wrote at http://www.dalkescientific.com/writings/diary/archive/2017/03/20/fingerprint_set_similarity.html on using chemfp to build a target set association network using ChEMBL.

I first learned about this approach from the *Similarity Ensemble Approach* (SEA) work of Keiser, Roth, Armbruster, Ernsberger, and Irwin. The paper is available online from http://sea.bkslab.org/ .

That paper actually wants you to use the "raw score". This is the sum of the hit scores in a given range, and not just the number of hits. No problem! Use *SearchResult.cumulative_score()* for an individual result or *SearchResults.cumulative_score_all()* for the entire set of results:

```
>>> sum(row.cumulative_score(min_score=0.5, max_score=0.9)
...             for row in coA_against_penicillin_result)
224.83239025119906
>>> coA_against_penicillin_result.cumulative_score_all(min_score=0.5, max_score=0.9)
224.83239025119866
```

These also take the *interval* parameter if you don't want the default of *[]*.

You may wonder why these two values aren't exactly the same. Addition of floating point numbers isn't associative. You can see that I get still different results if I sum up the values in reverse order:

```
>>> sum(list(row.cumulative_score(min_score=0.5, max_score=0.9)
...                for row in coA_against_penicillin_result)[::-1])
224.83239025119875
```

CHAPTER 5

chemfp API

This chapter contains the docstrings for the public portion of the chemfp API.

CHAPTER 6

chemfp top-level module

The following functions and classes are in the top-level chemfp module.

chemfp.**open**(*source*, *format=None*, *location=None*)

Read fingerprints from a fingerprint file

Read fingerprints from *source*, using the given format. If *source* is a string then it is treated as a filename. If *source* is None then fingerprints are read from stdin. Otherwise, *source* must be a Python file object supporting the `read` and `readline` methods.

If *format* is None then the fingerprint file format and compression type are derived from the source filename, or from the `name` attribute of the source file object. If the source is None then the stdin is assumed to be uncompressed data in "fps" format.

The supported format strings are "fps", "fps.gz" for fingerprints in FPS format and compressed FPS format, respectively.

This version of chemfp does not support the FPB format. Trying to use the "fpb" format will raise a NotImplementedError.

If the chemfp_converters package is available then the "flush" format is also supported.

The optional *location* is a *chemfp.io.Location* instance. It will only be used if the source is in FPS format.

If the source is in FPS format then `open` will return a *chemfp.fps_io.FPSReader*, which will use the *location* if specified.

Here's an example of printing the contents of the file:

```
from chemfp.bitops import hex_encode
reader = chemfp.open("example.fps.gz")
for id, fp in reader:
    print(id, hex_encode(fp))
```

Parameters

- **source** (*A filename string, a file object, or None*) – The fingerprint source.

- **format** (*string, or None*) – The file format and optional compression.

Returns a *chemfp.fps_io.FPSReader*

chemfp.**load_fingerprints**(*reader*, *metadata=None*, *reorder=True*, *alignment=None*, *format=None*)

Load all of the fingerprints into an in-memory FingerprintArena data structure

The FingerprintArena data structure reads all of the fingerprints and identifers from 'reader' and stores them into an in-memory data structure which supports fast similarity searches.

If 'reader' is a string or implements "read" then the contents will be parsed with the 'chemfp.open' function. Otherwise it must support iteration returning (id, fingerprint) pairs. 'metadata' contains the metadata the arena. If not specified then 'reader.metadata' is used.

The loader may reorder the fingerprints for better search performance. To prevent ordering, use reorder=False.

The 'alignment' option specifies the alignment data alignment and padding size for each fingerprint. A value of 8 means that each fingerprint will start on a 8 byte alignment, and use storage space which a multiple of 8 bytes long. The default value of None determines the best alignment based on the fingerprint size and available popcount methods.

Parameters

- **reader** (*a string, file object, or (id, fingerprint) iterator*) – An iterator over (id, fingerprint) pairs
- **metadata** (Metadata) – The metadata for the arena, if other than reader.metadata
- **reorder** (*True or False*) – Specify if fingerprints should be reordered for better performance
- **alignment** (*a positive integer, or None*) – Alignment size in bytes (both data alignment and padding); None autoselects the best alignment.
- **format** (*None, "fps", or "fps.gz". "fpb" will raise a NotImplementedError*) – The file format name if the reader is a string

Returns FingerprintArena

chemfp.**read_structure_fingerprints**(*type*, *source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Deprecated function. Please call read_molecule_fingerprints() instead

The function named changed in chemfp 2.0 to read_molecule_fingerprints() because it was a better fit to the toolkit API. Chemfp-1.3 maintains backwards compatibility with chemfp-1.1, so the function remains. It forwards the call the correct function.

Parameters

- **type** (*string or Metadata*) – information about how to convert the input structure into a fingerprint
- **source** (*A filename (as a string), a file object, or None to read from stdin*) – The structure data source.
- **format** (*string, or None to autodetect based on the source*) – The file format and optional compression. Examples: 'smi' and 'sdf.gz'
- **id_tag** (*string, or None to use the default title for the given format*) – The tag containing the record id. Example: 'ChEBI ID'. Only valid for SD files.

Returns a FingerprintReader

chemfp.**read_molecule_fingerprints**(*type*, *source=None*, *format=None*, *id_tag=None*, *reader_args=None*, *errors="strict"*)

Read structures from 'source' and return the corresponding ids and fingerprints

This returns a FingerprintReader which can be iterated over to get the id and fingerprint for each read structure record. The fingerprint generated depends on the value of 'type'. Structures are read from 'source', which can either be the structure filename, or None to read from stdin.

'type' contains the information about how to turn a structure into a fingerprint. It can be a string or a metadata instance. String values look like "OpenBabel-FP2/1", "OpenEye-Path", and "OpenEye-Path/1 min_bonds=0 max_bonds=5 atype=DefaultAtom btype=DefaultBond". Default values are used for unspecified parameters. Use a Metadata instance with 'type' and 'aromaticity' values set in order to pass aromaticity information to OpenEye.

If 'format' is None then the structure file format and compression are determined by the filename's extension(s), defaulting to uncompressed SMILES if that is not possible. Otherwise 'format' may be "smi" or "sdf" optionally followed by ".gz" or "bz2" to indicate compression. The OpenBabel and OpenEye toolkits also support additional formats.

If 'id_tag' is None, then the record id is based on the title field for the given format. If the input format is "sdf" then 'id_tag' specifies the tag field containing the identifier. (Only the first line is used for multi-line values.) For example, ChEBI omits the title from the SD files and stores the id after the "> <ChEBI ID>" line. In that case, use id_tag = "ChEBI ID".

'aromaticity' specifies the aromaticity model, and is only appropriate for OEChem. It must be a string like "openeye" or "daylight".

Here is an example of using fingerprints generated from structure file:

```
fp_reader = read_molecule_fingerprints("OpenBabel-FP4/1", "example.sdf.gz")
print "Each fingerprint has", fps.metadata.num_bits, "bits"
for (id, fp) in fp_reader:
   print id, fp.encode("hex")
```

Parameters

- **type** (*string or Metadata*) – information about how to convert the input structure into a fingerprint
- **source** (*A filename (as a string), a file object, or None to read from stdin*) – The structure data source.
- **format** (*string, or None to autodetect based on the source*) – The file format and optional compression. Examples: 'smi' and 'sdf.gz'
- **id_tag** (*string, or None to use the default title for the given format*) – The tag containing the record id. Example: 'ChEBI ID'. Only valid for SD files.

Returns a FingerprintReader

chemfp.**open_fingerprint_writer**(*destination*, *metadata=None*, *format=None*, *alignment=8*, *reorder=True*, *tmpdir=None*, *max_spool_size=None*, *errors="strict"*, *location=None*)

Create a fingerprint writer for the given destination

The fingerprint writer is an object with methods to write fingerprints to the given *destination*. The output format is based on the *format*. If that's None then the format depends on the *destination*, or is "fps" if the attempts at format detection fail.

The *metadata*, if given, is a *Metadata* instance, and used to fill the header of an FPS file.

If the output format is "fps" or "fps.gz" then *destination* may be a filename, a file object, or None for stdout. The "fpb" format is not available for this version of chemfp, and function will raise a NotImplementedError in that case.

If the chemfp_converters package is available then the "flush" format is also supported.

The parameters *alignment*, *reorder*, *tmpdir*, and *max_spool_size* are for FPB output and are ignored. The parameters are listed for better forwards-compatibility.

The *errors* specifies how to handle recoverable write errors. The value "strict" raises an exception if there are any detected errors. The value "report" sends an error message to stderr and skips to the next record. The value "ignore" skips to the next record.

The *location* is a `Location` instance. It lets the caller access state information such as the number of records that have been written.

Parameters

- **destination** (*a filename, file object, or None*) – the output destination
- **metadata** (*a Metadata instance, or None*) – the fingerprint metadata
- **format** (*None, "fps", "fps.gz", or "fpb"*) – the output format
- **alignment** (*positive integer*) – arena byte alignment for FPB files
- **reorder** (*True or False*) – True reorders the fingerprints by popcount, False leaves them in input order
- **tmpdir** (*string or None*) – the directory to use for temporary files, when max_spool_size is specified
- **max_spool_size** (*integer, or None*) – number of bytes to store in memory before using a temporary file. If None, use memory for everything.
- **location** (*a Location instance, or None*) – a location object used to access output state information

Returns a *chemfp.FingerprintWriter*

6.1 ChemFPError

class chemfp.**ChemFPError**
Base class for all of the chemfp exceptions

6.2 ParseError

class chemfp.**ParseError**
Exception raised by the molecule and fingerprint parsers and writers

The public attributes are:

msg
a string describing the exception

location
a *chemfp.io.Location* instance, or None

6.3 Metadata

class chemfp.**Metadata**

Store information about a set of fingerprints

The public attributes are:

num_bits
the number of bits in the fingerprint

num_bytes
the number of bytes in the fingerprint

type
the fingerprint type string

aromaticity
aromaticity model (only used with OEChem, and now deprecated)

software
software used to make the fingerprints

sources
list of sources used to make the fingerprint

date
a datetime timestamp of when the fingerprints were made

datestamp
the ISO string representation of the date

__repr__()
Return a string like `Metadata(num_bits=1024, num_bytes=128, type='OpenBabel/FP2', ....)`

__str__()
Show the metadata in FPS header format

copy(*num_bits=None*, *num_bytes=None*, *type=None*, *aromaticity=None*, *software=None*, *sources=None*, *date=None*)
Return a new Metadata instance based on the current attributes and optional new values

When called with no parameter, make a new Metadata instance with the same attributes as the current instance.

If a given call parameter is not None then it will be used instead of the current value. If you want to change a current value to None then you will have to modify the new Metadata after you created it.

Parameters

- **num_bits** (*an integer, or None*) – the number of bits in the fingerprint
- **num_bytes** (*an integer, or None*) – the number of bytes in the fingerprint
- **type** (*string or None*) – the fingerprint type description
- **aromaticity** (*None*) – obsolete
- **software** (*string or None*) – a description of the software
- **sources** (*list of strings, a string (interpreted as a list with one string), or None*) – source filenames

- **date** (*a datetime instance, or None*) – creation or processing date for the contents

Returns a new Metadata instance

6.4 FingerprintReader

class chemfp.**FingerprintReader**

Base class for all chemfp objects holding fingerprint records

All FingerprintReader instances have a `metadata` attribute containing a Metadata and can be iteratated over to get the (id, fingerprint) for each record.

__iter__()
iterate over the (id, fingerprint) pairs

iter_arenas(*arena_size=1000*)
iterate through *arena_size* fingerprints at a time, as subarenas

Iterate through *arena_size* fingerprints at a time, returned as *chemfp.arena.FingerprintArena* instances. The arenas are in input order and not reordered by popcount.

This method helps trade off between performance and memory use. Working with arenas is often faster than processing one fingerprint at a time, but if the file is very large then you might run out of memory, or get bored while waiting to process all of the fingerprint before getting the first answer.

If *arena_size* is None then this makes an iterator which returns a single arena containing all of the fingerprints.

Parameters **arena_size** (*positive integer, or None*) – The number of fingerprints to put into each arena.

Returns an iterator of *chemfp.arena.FingerprintArena* instances

save(*destination*, *format=None*)
Save the fingerprints to a given destination and format

The output format is based on the *format*. If the format is None then the format depends on the *destination* file extension. If the extension isn't recognized then the fingerprints will be saved in "fps" format.

If the output format is "fps" or "fps.gz" then *destination* may be a filename, a file object, or None; None writes to stdout.

If the output format is "fpb" then *destination* must be a filename.

Parameters

- **destination** (*a filename, file object, or None*) – the output destination
- **format** (*None, "fps", "fps.gz", or "fpb"*) – the output format

Returns None

6.5 FingerprintIterator

class chemfp.**FingerprintIterator**

A *chemfp.FingerprintReader* for an iterator of (id, fingerprint) pairs

This is often used as an adapter container to hold the metadata and (id, fingerprint) iterator. It supports an optional location, and can call a close function when the iterator has completed.

A FingerprintIterator is a context manager which will close the underlying iterator if it's given a close handler.

Like all iterators you can use next() to get the next (id, fingerprint) pair.

__init__(*metadata*, *id_fp_iterator*, *location=None*, *close=None*)
Initialize with a Metadata instance and the (id, fingerprint) iterator

The *metadata* is a *Metadata* instance. The *id_fp_iterator* is an iterator which returns (id, fingerprint) pairs.

The optional *location* is a *chemfp.io.Location*. The optional *close* callable is called (as `close()`) whenever `self.close()` is called and when the context manager exits.

__iter__()
Iterate over the (id, fingerprint) pairs

close()
Close the iterator

The call will be forwarded to the `close` callable passed to the constructor. If that `close` is None then this does nothing.

6.6 Fingerprints

class chemfp.**Fingerprints**

A `chemf.FingerprintReader` containing a metadata and a list of (id, fingerprint) pairs.

This is typically used as an adapater when you have a list of (id, fingerprint) pairs and you want to pass it (and the metadata) to the rest of the chemfp API.

This implements a simple list-like collection of fingerprints. It supports:

- for (id, fingerprint) in fingerprints: ...
- id, fingerprint = fingerprints[1]
- len(fingerprints)

More features, like slicing, will be added as needed or when requested.

__init__(*metadata*, *id_fp_pairs*)
Initialize with a Metadata instance and the (id, fingerprint) pair list

The *metadata* is a *Metadata* instance. The *id_fp_iterator* is an iterator which returns (id, fingerprint) pairs.

6.7 FingerprintWriter

class chemfp.**FingerprintWriter**

Base class for the fingerprint writers

The only concrete fingerprint writer class in chemfp 1.x is:

- *chemfp.fps_io.FPSWriter* - write an FPS file

Chemfp 2.0 and later also implement OrderedFPBWriter and InputOrderFPBWriter. If the chemfp_converters package is available then its FlushFingerprintWriter will be used to write fingerprints in flush format.

Use *chemfp.open_fingerprint_writer()* to create a fingerprint writer class; do not create them directly.

All classes have the following attributes:

- metadata - a *chemfp.Metadata* instance
- closed - False when the file is open, else True

Fingerprint writers are also their own context manager, and close the writer on context exit.

write_fingerprint(*id*, *fp*)
Write a single fingerprint record with the given id and fp to the destination

Parameters

- **id** (*string*) – the record identifier
- **fp** (*byte string*) – the fingerprint

write_fingerprints(*id_fp_pairs*)
Write a sequence of (id, fingerprint) pairs to the destination

Parameters **id_fp_pairs** – An iterable of (id, fingerprint) pairs. *id* is a string and *fingerprint* is a byte string.

close()
Close the writer

This will set self.closed to False.

6.8 ChemFPProblem

class chemfp.**ChemFPProblem**
Information about a compatibility problem between a query and target.

Instances are generated by *chemfp.check_fingerprint_problems()* and *chemfp.check_metadata_problems()*.

The public attributes are:

severity
one of “info”, “warning”, or “error”

error_level
5 for “info”, 10 for “warning”, and 20 for “error”

category
a string used as a category name. This string will not change over time.

description
a more detailed description of the error, including details of the mismatch. The description depends on *query_name* and *target_name* and may change over time.

The current category names are:

- “num_bits mismatch” (error)

- "num_bytes_mismatch" (error)
- "type mismatch" (warning)
- "aromaticity mismatch" (info)
- "software mismatch" (info)

chemfp.**check_fingerprint_problems**(*query_fp*, *target_metadata*, *query_name="query"*, *target_name="target"*)

Return a list of compatibility problems between a fingerprint and a metadata

If there are no problems then this returns an empty list. If there is a bit length or byte length mismatch between the *query_fp* byte string and the *target_metadata* then it will return a list containing a *ChemFPProblem* instance, with a severity level "error" and category "num_bytes mismatch".

This function is usually used to check if a query fingerprint is compatible with the target fingerprints. In case of a problem, the default message looks like:

```
>>> problems = check_fingerprint_problems("A"*64, Metadata(num_bytes=128))
>>> problems[0].description
'query contains 64 bytes but target has 128 byte fingerprints'
```

You can change the error message with the *query_name* and *target_name* parameters:

```
>>> import chemfp
>>> problems = check_fingerprint_problems("z"*64, chemfp.Metadata(num_bytes=128),
...       query_name="input", target_name="database")
>>> problems[0].description
'input contains 64 bytes but database has 128 byte fingerprints'
```

Parameters

- **query_fp** (*byte string*) – a fingerprint (usually the query fingerprint)
- **target_metadata** (*Metadata instance*) – the metadata to check against (usually the target metadata)
- **query_name** (*string*) – the text used to describe the fingerprint, in case of problem
- **target_name** (*string*) – the text used to describe the metadata, in case of problem

Returns a list of *ChemFPProblem* instances

chemfp.**check_metadata_problems**(*query_metadata*, *target_metadata*, *query_name="query"*, *target_name="target"*)

Return a list of compatibility problems between two metadata instances.

If there are no probelms then this returns an empty list. Otherwise it returns a list of *ChemFPProblem* instances, with a severity level ranging from "info" to "error".

Bit length and byte length mismatches produce an "error". Fingerprint type and aromaticity mismatches produce a "warning". Software version mismatches produce an "info".

This is usually used to check if the query metadata is incompatible with the target metadata. In case of a problem the messages look like:

```
>>> import chemfp
>>> m1 = chemfp.Metadata(num_bytes=128, type="Example/1")
>>> m2 = chemfp.Metadata(num_bytes=256, type="Counter-Example/1")
>>> problems = chemfp.check_metadata_problems(m1, m2)
```

(continues on next page)

(continued from previous page)

```
>>> len(problems)
2
>>> print(problems[1].description)
query has fingerprints of type 'Example/1' but target has fingerprints of type
↪'Counter-Example/1'
```

You can change the error message with the *query_name* and *target_name* parameters:

```
>>> problems = chemfp.check_metadata_problems(m1, m2, query_name="input", target_
↪name="database")
>>> print(problems[1].description)
input has fingerprints of type 'Example/1' but database has fingerprints of type
↪'Counter-Example/1'
```

Parameters

- **fp** (*byte string*) – a fingerprint
- **metadata** (*Metadata instance*) – the metadata to check against
- **query_name** (*string*) – the text used to describe the fingerprint, in case of problem
- **target_name** (*string*) – the text used to describe the metadata, in case of problem

Returns a list of *ChemFPProblem* instances

chemfp.**count_tanimoto_hits**(*queries*, *targets*, *threshold=0.7*, *arena_size=100*)

Count the number of targets within 'threshold' of each query term

For each query in 'queries', count the number of targets in 'targets' which are at least 'threshold' similar to the query. This function returns an iterator containing the (query_id, count) pairs.

Example:

```
queries = chemfp.open("queries.fps")
targets = chemfp.load_fingerprints("targets.fps.gz")
for (query_id, count) in chemfp.count_tanimoto_hits(queries, targets, threshold=0.
↪9):
    print query_id, "has", count, "neighbors with at least 0.9 similarity"
```

Internally, queries are processed in batches of size 'arena_size'. A small batch size uses less overall memory and has lower processing latency, while a large batch size has better overall performance. Use arena_size=None to process the input as a single batch.

Note: the FPSReader may be used as a target but it can only process one batch, and searching a FingerprintArena is faster if you have more than a few queries.

Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*FingerprintArena or the slower FPSReader*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **arena_size** (*a positive integer, or None*) – The number of queries to process in a batch

Returns An iterator containing (query_id, score) pairs, one for each query

chemfp.**count_tanimoto_hits_symmetric**(*fingerprints*, *threshold=0.7*)

Find the number of other fingerprints within *threshold* of each fingerprint

For each fingerprint in the *fingerprints* arena, find the number of other fingerprints in the same arena which are at least *threshold* similar to it. The arena must have pre-computed popcounts. A fingerprint never matches itself.

This function returns an iterator of (fingerprint_id, count) pairs.

Example:

```
arena = chemfp.load_fingerprints("targets.fps.gz")
for (fp_id, count) in chemfp.count_tanimoto_hits_symmetric(arena, threshold=0.6):
    print fp_id, "has", count, "neighbors with at least 0.6 similarity"
```

Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount_indices*) – The arena containing the fingerprints.
- **threshold** – The minimum score threshold.

Returns An iterator of (fp_id, count) pairs, one for each fingerprint

chemfp.**threshold_tanimoto_search**(*queries*, *targets*, *threshold=0.7*, *arena_size=100*)

Find all targets within 'threshold' of each query term

For each query in 'queries', find all the targets in 'targets' which are at least 'threshold' similar to the query. This function returns an iterator containing the (query_id, hits) pairs. The hits are stored as a list of (target_id, score) pairs.

Example:

```
queries = chemfp.open("queries.fps")
targets = chemfp.load_fingerprints("targets.fps.gz")
for (query_id, hits) in chemfp.id_threshold_tanimoto_search(queries, targets,␣
↪threshold=0.8):
    print query_id, "has", len(hits), "neighbors with at least 0.8 similarity"
    non_identical = [target_id for (target_id, score) in hits if score != 1.0]
    print "  The non-identical hits are:", non_identical
```

Internally, queries are processed in batches of size 'arena_size'. A small batch size uses less overall memory and has lower processing latency, while a large batch size has better overall performance. Use arena_size=None to process the input as a single batch.

Note: the FPSReader may be used as a target but it can only process one batch, and searching a FingerprintArena is faster if you have more than a few queries.

Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (*FingerprintArena or the slower FPSReader*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **arena_size** (*positive integer, or None*) – The number of queries to process in a batch

Returns An iterator containing (query_id, hits) pairs, one for each query. 'hits' contains a list of (target_id, score) pairs.

chemfp.**threshold_tanimoto_search_symmetric**(*fingerprints*, *threshold=0.7*)

Find the other fingerprints within *threshold* of each fingerprint

For each fingerprint in the *fingerprints* arena, find the other fingerprints in the same arena which hare at least *threshold* similar to it. The arena must have pre-computed popcounts. A fingerprint never matches itself.

This function returns an iterator of (fingerprint, SearchResult) pairs. The SearchResult hit order is arbitrary.

Example:

```
arena = chemfp.load_fingerprints("targets.fps.gz")
for (fp_id, hits) in chemfp.threshold_tanimoto_search_symmetric(arena,␣
↪threshold=0.75):
    print fp_id, "has", len(hits), "neighbors:"
    for (other_id, score) in hits.get_ids_and_scores():
        print "   %s  %.2f" % (other_id, score)
```

Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount_indices*) – The arena containing the fingerprints.
- **threshold** – The minimum score threshold.

Returns An iterator of (fp_id, SearchResult) pairs, one for each fingerprint

chemfp.**knearest_tanimoto_search**(*queries*, *targets*, *k=3*, *threshold=0.7*, *arena_size=100*)

Find the 'k'-nearest targets within 'threshold' of each query term

For each query in 'queries', find the 'k'-nearest of all the targets in 'targets' which are at least 'threshold' similar to the query. Ties are broken arbitrarily and hits with scores equal to the smallest value may have been omitted.

This function returns an iterator containing the (query_id, hits) pairs, where hits is a list of (target_id, score) pairs, sorted so that the highest scores are first. The order of ties is arbitrary.

Example:

```
# Use the first 5 fingerprints as the queries
queries = next(chemfp.open("pubchem_subset.fps").iter_arenas(5))
targets = chemfp.load_fingerprints("pubchem_subset.fps")

# Find the 3 nearest hits with a similarity of at least 0.8
for (query_id, hits) in chemfp.id_knearest_tanimoto_search(queries, targets, k=3,␣
↪threshold=0.8):
    print query_id, "has", len(hits), "neighbors with at least 0.8 similarity"
    if hits:
        target_id, score = hits[-1]
        print "    The least similar is", target_id, "with score", score
```

Internally, queries are processed in batches of size 'arena_size'. A small batch size uses less overall memory and has lower processing latency, while a large batch size has better overall performance. Use arena_size=None to process the input as a single batch.

Note: the FPSReader may be used as a target but it can only process one batch, and searching a FingerprintArena is faster if you have more than a few queries.

Parameters

- **queries** (*any fingerprint container*) – The query fingerprints.
- **targets** (FingerprintArena *or the slower FPSReader*) – The target fingerprints.

- **k** (*positive integer*) – The maximum number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **arena_size** (*positive integer, or None*) – The number of queries to process in a batch

Returns An iterator containing (query_id, hits) pairs, one for each query. 'hits' contains a list of (target_id, score) pairs, sorted by score.

chemfp.**knearest_tanimoto_search_symmetric**(*fingerprints*, *k=3*, *threshold=0.7*)

Find the nearest *k* fingerprints within *threshold* of each fingerprint

For each fingerprint in the *fingerprints* arena, find the nearest *k* fingerprints in the same arena which hare at least *threshold* similar to it. The arena must have pre-computed popcounts. A fingerprint never matches itself.

This function returns an iterator of (fingerprint, SearchResult) pairs. The SearchResult hits are ordered from highest score to lowest, with ties broken arbitrarily.

Example:

```
arena = chemfp.load_fingerprints("targets.fps.gz")
for (fp_id, hits) in chemfp.knearest_tanimoto_search_symmetric(arena, k=5,␣
↪threshold=0.5):
    print fp_id, "has", len(hits), "neighbors, with scores",
    print ", ".join("%.2f" % x for x in hits.get_scores())
```

Parameters

- **fingerprints** (*a FingerprintArena with precomputed popcount_indices*) – The arena containing the fingerprints.
- **k** (*positive integer*) – The maximum number of nearest neighbors to find.
- **threshold** – The minimum score threshold.

Returns An iterator of (fp_id, SearchResult) pairs, one for each fingerprint

chemfp.**get_max_threads**()

Return the maximum number of threads available.

If OpenMP is not available then this will return 1. Otherwise it returns the maximum number of threads available, as reported by omp_get_num_threads().

chemfp.**get_num_threads**()

Return the number of OpenMP threads to use in searches

Initially this is the value returned by omp_get_max_threads(), which is generally 4 unless you set the environment variable OMP_NUM_THREADS to some other value.

It may be any value in the range 1 to get_max_threads(), inclusive.

chemfp.**set_num_threads**(*num_threads*)

Set the number of OpenMP threads to use in searches

If *num_threads* is less than one then it is treated as one, and a value greater than get_max_threads() is treated as get_max_threads().

6.9 Open Babel fingerprints

Open Babel implements four fingerprints families and chemfp implements two fingerprint families using the Open Babel toolkit. These are:

- OpenBabel-FP2 - Indexes linear fragments up to 7 atoms.
- OpenBabel-FP3 - SMARTS patterns specified in the file patterns.txt
- OpenBabel-FP4 - SMARTS patterns specified in the file SMARTS_InteLigand.txt
- OpenBabel-MACCS - SMARTS patterns specified in the file MACCS.txt, which implements nearly all of the 166 MACCS keys
- RDMACCS-OpenBabel - a chemfp implementation of nearly all of the MACCS keys
- ChemFP-Substruct-OpenBabel - an experimental chemfp implementation of the PubChem keys

Most people use FP2 and MACCS.

Note: chemfp, starting with version 1.3, implements both RDMACCS-OpenBabel/1 and RDMACCS-OpenBabel/2. Version 1.1 did not have a definition for key 44.

6.10 OpenEye fingerprints

OpenEye's OEGraphSim library implements four bitstring-based fingerprint families, and chemfp implements two fingerprint families based on OEChem. These are:

- OpenEye-Path - exhaustive enumeration of all linear fragments up to a given size
- OpenEye-Circular - exhaustive enumeration of all circular fragments grown radially from each heavy atom up to a given radius
- OpenEye-Tree - exhaustive enumeration of all trees up to a given size
- OpenEye-MACCS166 - an implementation of the 166 MACCS keys
- RDMACCS-OpenEye - a chemfp implementation of the 166 MACCS keys
- ChemFP-Substruct-OpenEye - an experimental chemfp implementation of the PubChem keys

Note: chemfp, starting with version 1.3, implements both RDMACCS-OpenEye/1 and RDMACCS-OpenEye/2. Version 1.1 did not have a definition for key 44.

6.11 RDKit fingerprints

RDKit implements nine fingerprint families, and chemfp implements two fingerprint families based on RDKit. These are:

- RDKit-Fingerprint - exhaustive enumeration of linear and branched trees
- RDKit-MACCS166 - The RDKit implementation of the MACCS keys
- RDKit-Morgan - EFCP-like circular fingerprints
- RDKit-AtomPair - atom pair fingerprints
- RDKit-Torsion - topological-torsion fingerprints
- RDKit-Pattern - substructure screen fingerprint

- RDKit-Avalon - RDKit's interface to the Avalon toolkit fingerprints
- RDMACCS-RDKit - a chemfp implementation of the 166 MACCS keys
- ChemFP-Substruct-RDKit - an experimental chemfp implementation of the PubChem keys

Note: chemfp, starting with version 1.3, implements both RDMACCS-OpenEye/1 and RDMACCS-OpenEye/2. Version 1.1 did not have a definition for key 44.

CHAPTER 7

chemfp.arena module

There should be no reason for you to import this module yourself. It contains the *FingerprintArena* implementation. FingerprintArena instances are returns part of the public API but should not be constructed directly.

7.1 FingerprintArena

class chemfp.arena.**FingerprintArena**

Store fingerprints in a contiguous block of memory for fast searches

A fingerprint arena implements the *chemfp.FingerprintReader* API.

A fingerprint arena stores all of the fingerprints in a continuous block of memory, so the per-molecule overhead is very low.

The fingerprints can be sorted by popcount, so the fingerprints with no bits set come first, followed by those with 1 bit, etc. If `self.popcount_indices` is a non-empty string then the string contains information about the start and end offsets for all the fingerprints with a given popcount. This information is used for the sublinear search methods.

The public attributes are:

metadata
chemfp.Metadata about the fingerprints

ids
list of identifiers, in index order

Other attributes, which might be subject to change, and which I won't fully explain, are:

- arena - a contiguous block of memory, which contains the fingerprints
- start_padding - number of bytes to the first fingerprint in the block
- end_padding - number of bytes after the last fingerprint in the block
- storage_size - number of bytes used to store a fingerprint

- num_bytes - number of bytes in each fingerprint (must be <= storage_size)
- num_bits - number of bits in each fingerprint
- alignment - the fingerprint alignment
- start - the index for the first fingerprint in the arena/subarena
- end - the index for the last fingerprint in the arena/subarena
- arena_ids - all of the identifiers for the parent arena

The FingerprintArena is its own context manager, but it does nothing on context exit.

__len__()
Number of fingerprint records in the FingerprintArena

__getitem__(*i*)
Return the (id, fingerprint) pair at index i

__iter__()
Iterate over the (id, fingerprint) contents of the arena

get_fingerprint(*i*)
Return the fingerprint at index *i*

Raises an IndexError if index *i* is out of range.

get_by_id(*id*)
Given the record identifier, return the (id, fingerprint) pair,

If the *id* is not present then return None.

get_index_by_id(*id*)
Given the record identifier, return the record index

If the *id* is not present then return None.

get_fingerprint_by_id(*id*)
Given the record identifier, return its fingerprint

If the *id* is not present then return None

save(*destination*, *format=None*)
Save the fingerprints to a given destination and format

The output format is based on the *format*. If the format is None then the format depends on the *destination* file extension. If the extension isn't recognized then the fingerprints will be saved in "fps" format.

If the output format is "fps" or "fps.gz" then *destination* may be a filename, a file object, or None; None writes to stdout.

If the output format is "fpb" then *destination* must be a filename.

Parameters

- **destination** (*a filename, file object, or None*) – the output destination
- **format** (*None, "fps", "fps.gz", or "fpb"*) – the output format

Returns None

iter_arenas(*arena_size = 1000*)
Base class for all chemfp objects holding fingerprint records

All FingerprintReader instances have a `metadata` attribute containing a Metadata and can be iteratated over to get the (id, fingerprint) for each record.

copy(*indices=None*, *reorder=None*)

Create a new arena using either all or some of the fingerprints in this arena

By default this create a new arena. The fingerprint data block and ids may be shared with the original arena, which makes this a shallow copy. If the original arena is a slice, or "sub-arena" of an arena, then the copy will allocate new space to store just the fingerprints in the slice and use its own list for the ids.

The *indices* parameter, if not None, is an iterable which contains the indicies of the fingerprint records to copy. Duplicates are allowed, though discouraged.

If *indices* are specified then the default *reorder* value of None, or the value True, will reorder the fingerprints for the new arena by popcount. This improves overall search performance. If *reorder* is False then the new arena will preserve the order given by the indices.

If *indices* are not specified, then the default is to preserve the order type of the original arena. Use `reorder=True` to always reorder the fingerprints in the new arena by popcount, and `reorder=False` to always leave them in the current ordering.

```
>>> import chemfp
>>> arena = chemfp.load_fingerprints("pubchem_queries.fps")
>>> arena.ids[1], arena.ids[5], arena.ids[10], arena.ids[18]
(b'9425031', b'9425015', b'9425040', b'9425033')
>>> len(arena)
19
>>> new_arena = arena.copy(indices=[1, 5, 10, 18])
>>> len(new_arena)
4
>>> new_arena.ids
[b'9425031', b'9425015', b'9425040', b'9425033']
>>> new_arena = arena.copy(indices=[18, 10, 5, 1], reorder=False)
>>> new_arena.ids
[b'9425033', b'9425040', b'9425015', b'9425031']
```

Parameters

- **indices** (*iterable containing integers, or None*) – indicies of the records to copy into the new arena
- **reorder** (*True to reorder, False to leave in input order, None for default action*) – describes how to order the fingerprints

Returns a *FingerprintArena*

sample(*num_samples*, *reorder=True*, *rng=None*)

return a new arena containing *num_samples* randomly selected fingerprints, without replacement

If *num_samples* is an integer then it must be between 0 and the size of the arena. If *num_samples* is a float then it must be between 0.0 and 1.0 and is interpreted as the proportion of the arena to include.

By default the new arena is sorted by popcount. Set *reorder* to *False* to return the fingerprints in random order.

If *rng* is None then use Python's `random.sample()` for the sampling. If *rng* is an integer then use `random.Random(rng).sample()`. Otherwise, use `rng.sample()`.

Added in chemfp 1.6.1.

Parameters

- **num_samples** (*int or float*) – number of fingerprints to select
- **reorder** (*True to reorder, False to leave in the sampling order*) – describes how to order the sampled fingerprints
- **rng** (*None, int, or a random.Random()*) – method to use for random sampling

Returns a *FingerprintArena*

train_test_split(*train_size=None*, *test_size=None*, *reorder=True*, *rng=None*)

return arenas containing *train_size* and *test_size* randomly selected fingerprints, without replacement

If *train_size* is an integer then it must be between 0 and the size of the arena. If *train_size* is a float then it must be between 0.0 and 1.0 and is interpreted as the proportion of the arena to include. If *train_size* is None then it is set to the complement of *test_size*. If both *train_size* and *test_size* are None then the default *train_size* is 0.75.

If *test_size* is an integer then it must be between 0 and the size of the arena. If *test_size* is a float then it must be between 0.0 and 1.0 and is interpreted as the proportion of the arena to include. If *test_size* is None then it is set to the complement of *train_size*. If both *test_size* and *train_size* are None then the default *test_size* is 0.25.

By default the new arena is sorted by popcount. Set *reorder* to *False* to return the fingerprints in random order.

If *rng* is None then use Python's `random.sample()` for the sampling. If *rng* is an integer then use `random.Random(rng).sample()`. Otherwise, use `rng.sample()`.

This method API is modelled on scikit-learn's model_selection.train_test_split() function, described at: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

Added in chemfp 1.6.1.

Parameters

- **train_size** (*int, float, or None*) – number of fingerprints for the training set arena
- **test_size** (*int, float, or None*) – number of fingerprints for the test set arena
- **reorder** (*True to reorder, False to leave in the sampling order*) – describes how to order the sampled fingerprints
- **rng** (*None, int, or a random.Random()*) – method to use for random sampling

Returns a training set *FingerprintArena* and a test set *FingerprintArena*

count_tanimoto_hits_fp(*query_fp*, *threshold=0.7*)

Count the fingerprints which are sufficiently similar to the query fingerprint

Return the number of fingerprints in the arena which are at least *threshold* similar to the query fingerprint *query_fp*.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns integer count

threshold_tanimoto_search_fp(*query_fp*, *threshold=0.7*)
Find the fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this arena which are at least *threshold* similar to the query fingerprint *query_fp*. The hits are returned as a *SearchResult*, in arbitrary order.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

knearest_tanimoto_search_fp(*query_fp*, *k=3*, *threshold=0.7*)
Find the k-nearest fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this arena which are at least *threshold* similar to the query fingerprint, and of those, select the top *k* hits. The hits are returned as a *SearchResult*, sorted from highest score to lowest.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

CHAPTER 8

chemfp.search module

The following functions and classes are in the chemfp.search module.

There are three main classes of functions. The ones ending with `*_fp` use a query fingerprint to search a target arena. The ones ending with `*_arena` use a query arena to search a target arena. The ones ending with `*_symmetric` use arena to search itself, except that a fingerprint is not tested against itself.

These functions share the same name with very similar functions in the top-level *chemfp* module. My apologies for any confusion. The top-level functions are designed to work with both arenas and iterators as the target. They give a simple search API, and automatically process in blocks, to give a balanced trade-off between performance and response time for the first results.

The functions in this module only work with arena as the target. By default it searches the entire arena before returning. If you want to process portions of the arena then you need to specify the range yourself.

chemfp.search.**count_tanimoto_hits_fp**(*query_fp*, *target_arena*, *threshold=0.7*)

Count the number of hits in *target_arena* at least *threshold* similar to the *query_fp*

Example:

```
query_id, query_fp = chemfp.load_fingerprints("queries.fps")[0]
targets = chemfp.load_fingerprints("targets.fps")
print chemfp.search.count_tanimoto_hits_fp(query_fp, targets, threshold=0.1)
```

Parameters

- **query_fp** (*a byte string*) – the query fingerprint
- **target_arena** – the target arena
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns an integer count

chemfp.search.**count_tanimoto_hits_arena**(*query_arena*, *target_arena*, *threshold=0.7*)

For each fingerprint in *query_arena*, count the number of hits in *target_arena* at least *threshold* similar to it

Example:

```
queries = chemfp.load_fingerprints("queries.fps")
targets = chemfp.load_fingerprints("targets.fps")
counts = chemfp.search.count_tanimoto_hits_arena(queries, targets, threshold=0.1)
print counts[:10]
```

The result is implementation specific. You'll always be able to get its length and do an index lookup to get an integer count. Currently it's a ctypes array of longs, but it could be an array.array or Python list in the future.

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – The query fingerprints.
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns an array of counts

chemfp.search.**count_tanimoto_hits_symmetric**(*arena*, *threshold=0.7*, *batch_size=100*)

For each fingerprint in the *arena*, count the number of other fingerprints at least *threshold* similar to it

A fingerprint never matches itself.

The computation can take a long time. Python won't check check for a ^C until the function finishes. This can be irritating. Instead, process only *batch_size* rows at a time before checking for a ^C.

Note: the *batch_size* may disappear in future versions of chemfp. I can't detect any performance difference between the current value and a larger value, so it seems rather pointless to have. Let me know if it's useful to keep as a user-defined parameter.

Example:

```
arena = chemfp.load_fingerprints("targets.fps")
counts = chemfp.search.count_tanimoto_hits_symmetric(arena, threshold=0.2)
print counts[:10]
```

The result object is implementation specific. You'll always be able to get its length and do an index lookup to get an integer count. Currently it's a ctype array of longs, but it could be an array.array or Python list in the future.

Parameters

- **arena** (a *chemfp.arena.FingerprintArena*) – the set of fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **batch_size** (*integer*) – the number of rows to process before checking for a ^C

Returns an array of counts

chemfp.search.**partial_count_tanimoto_hits_symmetric**(*counts*, *arena*, *threshold=0.7*, *query_start=0*, *query_end=None*, *target_start=0*, *target_end=None*)

Compute a portion of the symmetric Tanimoto counts

For most cases, use *chemfp.search.count_tanimoto_hits_symmetric()* instead of this function!

This function is only useful for thread-pool implementations. In that case, set the number of OpenMP threads to 1.

counts is a contiguous array of integers. It should be initialized to zeros, and reused for successive calls.

The function adds counts for counts[*query_start*:*query_end*] based on computing the upper-triangle portion contained in the rectangle *query_start*:*query_end* and *target_start*:target_end* and using symmetry to fill in the lower half.

You know, this is pretty complicated. Here's the bare minimum example of how to use it correctly to process 10 rows at a time using up to 4 threads:

```
import chemfp
import chemfp.search
from chemfp import futures
import array

chemfp.set_num_threads(1)  # Globally disable OpenMP

arena = chemfp.load_fingerprints("targets.fps")  # Load the fingerprints
n = len(arena)
counts = array.array("i", [0]*n)

with futures.ThreadPoolExecutor(max_workers=4) as executor:
    for row in xrange(0, n, 10):
        executor.submit(chemfp.search.partial_count_tanimoto_hits_symmetric,
                        counts, arena, threshold=0.2,
                        query_start=row, query_end=min(row+10, n))

print counts
```

Parameters

- **counts** (*a contiguous block of integer*) – the accumulated Tanimoto counts
- **arena** (a *chemfp.arena.FingerprintArena*) – the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **query_start** (*an integer*) – the query start row
- **query_end** (*an integer, or None to mean the last query row*) – the query end row
- **target_start** (*an integer*) – the target start row
- **target_end** (*an integer, or None to mean the last target row*) – the target end row

Returns None

chemfp.search.**threshold_tanimoto_search_fp**(*query_fp*, *target_arena*, *threshold=0.7*)

Search for fingerprint hits in *target_arena* which are at least *threshold* similar to *query_fp*

The hits in the returned *chemfp.search.SearchResult* are in arbitrary order.

Example:

```
query_id, query_fp = chemfp.load_fingerprints("queries.fps")[0]
targets = chemfp.load_fingerprints("targets.fps")
print list(chemfp.search.threshold_tanimoto_search_fp(query_fp, targets,␣
↪threshold=0.15))
```

Parameters

- **query_fp** (*a byte string*) – the query fingerprint
- **target_arena** (a *chemfp.arena.FingerprintArena*) – the target arena
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResult*

chemfp.search.**threshold_tanimoto_search_arena**(*query_arena*, *target_arena*, *threshold=0.7*)

Search for the hits in the *target_arena* at least *threshold* similar to the fingerprints in *query_arena*

The hits in the returned *chemfp.search.SearchResults* are in arbitrary order.

Example:

```
queries = chemfp.load_fingerprints("queries.fps")
targets = chemfp.load_fingerprints("targets.fps")
results = chemfp.search.threshold_tanimoto_search_arena(queries, targets,␣
↪threshold=0.5)
for query_id, query_hits in zip(queries.ids, results):
    if len(query_hits) > 0:
        print query_id, "->", ", ".join(query_hits.get_ids())
```

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – The query fingerprints.
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResults*

chemfp.search.**threshold_tanimoto_search_symmetric**(*arena*, *threshold=0.7*, *include_lower_triangle=True*, *batch_size=100*)

Search for the hits in the *arena* at least *threshold* similar to the fingerprints in the arena

When *include_lower_triangle* is True, compute the upper-triangle similarities, then copy the results to get the full set of results. When *include_lower_triangle* is False, only compute the upper triangle.

The hits in the returned *chemfp.search.SearchResults* are in arbitrary order.

The computation can take a long time. Python won't check check for a ^C until the function finishes. This can be irritating. Instead, process only *batch_size* rows at a time before checking for a ^C.

Note: the *batch_size* may disappear in future versions of chemfp. Let me know if it really is useful for you to have as a user-defined parameter.

Example:

```
arena = chemfp.load_fingerprints("queries.fps")
full_result = chemfp.search.threshold_tanimoto_search_symmetric(arena,␣
↪threshold=0.2)
upper_triangle = chemfp.search.threshold_tanimoto_search_symmetric(
          arena, threshold=0.2, include_lower_triangle=False)
assert sum(map(len, full_result)) == sum(map(len, upper_triangle))*2
```

Parameters

- **arena** (a *chemfp.arena.FingerprintArena*) – the set of fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **include_lower_triangle** (*boolean*) – if False, compute only the upper triangle, otherwise use symmetry to compute the full matrix
- **batch_size** (*integer*) – the number of rows to process before checking for a ^C

Returns a *chemfp.search.SearchResults*

chemfp.search.**partial_threshold_tanimoto_search_symmetric**(*results*, *arena*, *threshold=0.7*, *query_start=0*, *query_end=None*, *target_start=0*, *target_end=None*, *results_offset=0*)

Compute a portion of the symmetric Tanimoto search results

For most cases, use *chemfp.search.threshold_tanimoto_search_symmetric()* instead of this function!

This function is only useful for thread-pool implementations. In that case, set the number of OpenMP threads to 1.

results is a *chemfp.search.SearchResults* instance which is at least as large as the arena. It should be reused for successive updates.

The function adds hits to results[*query_start*:*query_end*], based on computing the upper-triangle portion contained in the rectangle *query_start*:*query_end* and *target_start*:*target_end*.

It does not fill in the lower triangle. To get the full matrix, call *fill_lower_triangle*.

You know, this is pretty complicated. Here's the bare minimum example of how to use it correctly to process 10 rows at a time using up to 4 threads:

```
import chemfp
import chemfp.search
from chemfp import futures
import array

chemfp.set_num_threads(1)

arena = chemfp.load_fingerprints("targets.fps")
n = len(arena)
results = chemfp.search.SearchResults(n, n, arena.ids)

with futures.ThreadPoolExecutor(max_workers=4) as executor:
    for row in xrange(0, n, 10):
        executor.submit(chemfp.search.partial_threshold_tanimoto_search_symmetric,
                        results, arena, threshold=0.2,
                        query_start=row, query_end=min(row+10, n))

chemfp.search.fill_lower_triangle(results)
```

The hits in the *chemfp.search.SearchResults* are in arbitrary order.

Parameters

- **results** (a *chemfp.search.SearchResults* instance) – the intermediate search results
- **arena** (a *chemfp.arena.FingerprintArena*) – the fingerprints.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **query_start** (*an integer*) – the query start row
- **query_end** (*an integer, or None to mean the last query row*) – the query end row
- **target_start** (*an integer*) – the target start row
- **target_end** (*an integer, or None to mean the last target row*) – the target end row
- **results_offset** – use results[results_offset] as the base for the results
- **results_offset** – an integer

Returns None

chemfp.search.**fill_lower_triangle**(*results*)

Duplicate each entry of *results* to its transpose

This is used after the symmetric threshold search to turn the upper-triangle results into a full matrix.

Parameters **results** (a *chemfp.search.SearchResults*) – search results

chemfp.search.**knearest_tanimoto_search_fp**(*query_fp*, *target_arena*, *k=3*, *threshold=0.7*)

Search for *k*-nearest hits in *target_arena* which are at least *threshold* similar to *query_fp*

The hits in the *chemfp.search.SearchResults* are ordered by decreasing similarity score.

Example:

```
query_id, query_fp = chemfp.load_fingerprints("queries.fps")[0]
targets = chemfp.load_fingerprints("targets.fps")
print list(chemfp.search.knearest_tanimoto_search_fp(query_fp, targets, k=3,␣
↪threshold=0.0))
```

Parameters

- **query_fp** (*a byte string*) – the query fingerprint
- **target_arena** (a *chemfp.arena.FingerprintArena*) – the target arena
- **k** (*positive integer*) – the number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResult*

chemfp.search.**knearest_tanimoto_search_arena**(*query_arena*, *target_arena*, *k=3*, *threshold=0.7*)

Search for the *k* nearest hits in the *target_arena* at least *threshold* similar to the fingerprints in *query_arena*

The hits in the *chemfp.search.SearchResults* are ordered by decreasing similarity score.

Example:

```
queries = chemfp.load_fingerprints("queries.fps")
targets = chemfp.load_fingerprints("targets.fps")
results = chemfp.search.knearest_tanimoto_search_arena(queries, targets, k=3,␣
↪threshold=0.5)
for query_id, query_hits in zip(queries.ids, results):
    if len(query_hits) >= 2:
        print query_id, "->", ", ".join(query_hits.get_ids())
```

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – The query fingerprints.
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.
- **k** (*positive integer*) – the number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.

Returns a *chemfp.search.SearchResults*

chemfp.search.**knearest_tanimoto_search_symmetric**(*arena*, *k=3*, *threshold=0.7*, *batch_size=100*)

Search for the *k*-nearest hits in the *arena* at least *threshold* similar to the fingerprints in the arena

The hits in the *SearchResults* are ordered by decreasing similarity score.

The computation can take a long time. Python won't check check for a ^C until the function finishes. This can be irritating. Instead, process only *batch_size* rows at a time before checking for a ^C.

Note: the *batch_size* may disappear in future versions of chemfp. Let me know if it really is useful for you to keep as a user-defined parameter.

Example:

```
arena = chemfp.load_fingerprints("queries.fps")
results = chemfp.search.knearest_tanimoto_search_symmetric(arena, k=3,␣
↪threshold=0.8)
for (query_id, hits) in zip(arena.ids, results):
    print query_id, "->", ", ".join(("%s %.2f" % hit) for hit in  hits.get_ids_
↪and_scores())
```

Parameters

- **arena** (a *chemfp.arena.FingerprintArena*) – the set of fingerprints
- **k** (*positive integer*) – the number of nearest neighbors to find.
- **threshold** (*float between 0.0 and 1.0, inclusive*) – The minimum score threshold.
- **include_lower_triangle** (*boolean*) – if False, compute only the upper triangle, otherwise use symmetry to compute the full matrix
- **batch_size** (*integer*) – the number of rows to process before checking for a ^C

Returns a *chemfp.search.SearchResults*

chemfp.search.**contains_fp**(*query_fp*, *target_arena*)

Find the target fingerprints which contain the query fingerprint bits as a subset

A target fingerprint contains a query fingerprint if all of the on bits of the query fingerprint are also on bits of the target fingerprint. This function returns a *chemfp.search.SearchResult* containing all of the target fingerprints in *target_arena* that contain the *query_fp*.

The SearchResult scores are all 0.0.

There is currently no direct way to limit the arena search range. Instead create a subarena by using Python's slice notation on the arena then search the subarena.

Parameters

- **query_fp** (*a byte string*) – the query fingerprint
- **target_arena** (a *chemfp.arena.FingerprintArena*) – The target fingerprints.

Returns a SearchResult instance

chemfp.search.**contains_arena**(*query_arena*, *target_arena*)

Find the target fingerprints which contain the query fingerprints as a subset

A target fingerprint contains a query fingerprint if all of the on bits of the query fingerprint are also on bits of the target fingerprint. This function returns a *chemfp.search.SearchResults* where SearchResults[i] contains all of the target fingerprints in *target_arena* that contain the fingerprint for entry *query_arena* [i].

The SearchResult scores are all 0.0.

There is currently no direct way to limit the arena search range, though you can create and search a subarena by using Python's slice notation.

Parameters

- **query_arena** (a *chemfp.arena.FingerprintArena*) – the query fingerprints
- **target_arena** (a *chemfp.arena.FingerprintArena*) – the target fingerprints

Returns a *chemfp.search.SearchResults* instance, of the same size as query_arena

8.1 SearchResults

class chemfp.search.**SearchResults**

Search results for a list of query fingerprints against a target arena

This acts like a list of SearchResult elements, with the ability to iterate over each search results, look them up by index, and get the number of scores.

In addition, there are helper methods to iterate over each hit and to get the hit indicies, scores, and identifiers directly as Python lists, sort the list contents, and more.

__len__()

The number of rows in the SearchResults

__iter__()

Iterate over each SearchResult hit

__getitem__(*i*)

Get the *i*-th SearchResult

shape

Read-only attribute.

the tuple (number of rows, number of columns)

The number of columns is the size of the target arena.

iter_indices()
For each hit, yield the list of target indices

iter_ids()
For each hit, yield the list of target identifiers

iter_scores()
For each hit, yield the list of target scores

iter_indices_and_scores()
For each hit, yield the list of (target index, score) tuples

iter_ids_and_scores()
For each hit, yield the list of (target id, score) tuples

clear_all()
Remove all hits from all of the search results

count_all(*min_score=None*, *max_score=None*, *interval="[]"*)
Count the number of hits with a score between *min_score* and *max_score*

Using the default parameters this returns the number of hits in the result.

The default *min_score* of None is equivalent to -infinity. The default *max_score* of None is equivalent to +infinity.

The *interval* parameter describes the interval end conditions. The default of "[]" uses a closed interval, where min_score <= score <= max_score. The interval "()" uses the open interval where min_score < score < max_score. The half-open/half-closed intervals "(]" and "[)" are also supported.

Parameters

- **min_score** (*a float, or None for -infinity*) – the minimum score in the range.
- **max_score** (*a float, or None for +infinity*) – the maximum score in the range.
- **interval** (*one of "[]", "()", "(]", "[)"*) – specify if the end points are open or closed.

Returns an integer count

cumulative_score_all(*min_score=None*, *max_score=None*, *interval="[]"*)
The sum of all scores in all rows which are between *min_score* and *max_score*

Using the default parameters this returns the sum of all of the scores in all of the results. With a specified range this returns the sum of all of the scores in that range. The cumulative score is also known as the raw score.

The default *min_score* of None is equivalent to -infinity. The default *max_score* of None is equivalent to +infinity.

The *interval* parameter describes the interval end conditions. The default of "[]" uses a closed interval, where min_score <= score <= max_score. The interval "()" uses the open interval where min_score < score < max_score. The half-open/half-closed intervals "(]" and "[)" are also supported.

Parameters

- **min_score** (*a float, or None for -infinity*) – the minimum score in the range.
- **max_score** (*a float, or None for +infinity*) – the maximum score in the range.

- **interval** (*one of "[]", "()", "(]", "[)"*) – specify if the end points are open or closed.

Returns a floating point count

reorder_all(*order="decreasing-score"*)
Reorder the hits for all of the rows based on the requested *order*.

The available orderings are:

- increasing-score - sort by increasing score
- decreasing-score - sort by decreasing score
- increasing-index - sort by increasing target index
- decreasing-index - sort by decreasing target index
- move-closest-first - move the hit with the highest score to the first position
- reverse - reverse the current ordering

Parameters **ordering** – the name of the ordering to use

to_csr(*dtype=None*)
Return the results as a SciPy compressed sparse row matrix.

The returned matrix has the same shape as the SearchResult instance and can be passed into, for example, a scikit-learn clustering algorithm.

By default the scores are stored with the *dtype* is "float64".

This method requires that SciPy (and NumPy) be installed.

Parameters **dtype** (*string or NumPy type*) – a NumPy numeric data type

8.2 SearchResult

class chemfp.search.**SearchResult**

Search results for a query fingerprint against a target arena.

The results contains a list of hits. Hits contain a target index, score, and optional target ids. The hits can be reordered based on score or index.

__len__()
The number of hits

__iter__()
Iterate through the pairs of (target index, score) using the current ordering

clear()
Remove all hits from this result

get_indices()
The list of target indices, in the current ordering.

get_ids()
The list of target identifiers (if available), in the current ordering

iter_ids()
Iterate over target identifiers (if available), in the current ordering

get_scores()
The list of target scores, in the current ordering

get_ids_and_scores()
The list of (target identifier, target score) pairs, in the current ordering

Raises a TypeError if the target IDs are not available.

get_indices_and_scores()
The list of (target index, score) pairs, in the current ordering

reorder(*ordering="decreasing-score"*)
Reorder the hits based on the requested ordering.

The available orderings are:

- increasing-score - sort by increasing score
- decreasing-score - sort by decreasing score
- increasing-index - sort by increasing target index
- decreasing-index - sort by decreasing target index
- move-closest-first - move the hit with the highest score to the first position
- reverse - reverse the current ordering

Parameters **ordering** (*string*) – the name of the ordering to use

count(*min_score=None*, *max_score=None*, *interval="[]"*)
Count the number of hits with a score between *min_score* and *max_score*

Using the default parameters this returns the number of hits in the result.

The default *min_score* of None is equivalent to -infinity. The default *max_score* of None is equivalent to +infinity.

The *interval* parameter describes the interval end conditions. The default of "[]" uses a closed interval, where min_score <= score <= max_score. The interval "()" uses the open interval where min_score < score < max_score. The half-open/half-closed intervals "(]" and "[)" are also supported.

Parameters

- **min_score** (*a float, or None for -infinity*) – the minimum score in the range.
- **max_score** (*a float, or None for +infinity*) – the maximum score in the range.
- **interval** (*one of "[]", "()", "(]", "[)"*) – specify if the end points are open or closed.

Returns an integer count

cumulative_score(*min_score=None*, *max_score=None*, *interval="[]"*)
The sum of the scores which are between *min_score* and *max_score*

Using the default parameters this returns the sum of all of the scores in the result. With a specified range this returns the sum of all of the scores in that range. The cumulative score is also known as the raw score.

The default *min_score* of None is equivalent to -infinity. The default *max_score* of None is equivalent to +infinity.

The *interval* parameter describes the interval end conditions. The default of "[]" uses a closed interval, where min_score <= score <= max_score. The interval "()" uses the open interval where min_score < score < max_score. The half-open/half-closed intervals "(]" and "[)" are also supported.

Parameters

- **min_score** (*a float, or None for -infinity*) – the minimum score in the range.
- **max_score** (*a float, or None for +infinity*) – the maximum score in the range.
- **interval** (*one of "[]", "()", "(]", "[)"*) – specify if the end points are open or closed.

Returns a floating point value

format_ids_and_scores_as_bytes(*ids=None*, *precision=4*)

Format the ids and scores as the byte string needed for simsearch output

If there are no hits then the result is the empty string b"", otherwise it returns a byte string containing the tab-seperated ids and scores, in the order ids[0], scores[0], ids[1], scores[1], ...

If the *ids* is not specified then the ids come from self.get_ids(). If no ids are available, a ValueError is raised. The ids must be a list of Unicode strings.

The *precision* sets the number of decimal digits to use in the score output. It must be an integer value between 1 and 10, inclusive.

This function is 3-4x faster than the Python equivalent, which is roughly:

```
ids = ids if (ids is not None) else self.get_ids()
formatter = ("%s\t%." + str(precision) + "f").encode("ascii")
return b"\t".join(formatter % pair for pair in zip(ids, self.get_scores()))
```

Parameters

- **ids** (*a list of Unicode strings, or None to use the default*) – the identifiers to use for each hit.
- **precision** (*an integer from 1 to 10, inclusive*) – the precision to use for each score

Returns a byte string

CHAPTER 9

chemfp.bitops module

The following functions from the chemfp.bitops module provide low-level bit operations on byte and hex fingerprints.

chemfp.bitops.**byte_contains**(*super_fp*, *sub_fp*)
Return 1 if the on bits of sub_fp are also 1 bits in super_fp

chemfp.bitops.**byte_contains_bit**(*fp*, *bit_index*)
Return True if the the given bit position is on, otherwise False

chemfp.bitops.**byte_difference**(*fp1*, *fp2*)
Return the absolute difference (xor) between the two byte strings, fp1 ^ fp2

chemfp.bitops.**byte_from_bitlist**(*fp*[, *num_bits=1024*])
Convert a list of bit positions into a byte fingerprint, including modulo folding

chemfp.bitops.**byte_hex_tanimoto**(*fp1*, *fp2*)
Compute the Tanimoto similarity between the byte fingerprint *fp1* and the hex fingerprint *fp2*. Return a float between 0.0 and 1.0, or raise a ValueError if *fp2* is not a hex fingerprint

chemfp.bitops.**byte_intersect**(*fp1*, *fp2*)
Return the intersection of the two byte strings, *fp1* & *fp2*

chemfp.bitops.**byte_intersect_popcount**(*fp1*, *fp2*)
Return the number of bits set in the instersection of the two byte fingerprints

chemfp.bitops.**byte_popcount**(*fp*)
Return the number of bits set in a byte fingerprint

chemfp.bitops.**byte_tanimoto**(*fp1*, *fp2*)
Compute the Tanimoto similarity between two byte fingerprints

chemfp.bitops.**byte_tversky**(*fp1*, *fp2*, *alpha=1.0*, *beta=1.0*)
Compute the Tversky index between the two byte fingerprints *fp1* and *fp2*

chemfp.bitops.**byte_hex_tversky**(*fp1*, *fp2*, *alpha=1.0*, *beta=1.0*)
Compute the Tversky index between the byte fingerprint *fp1* and the hex fingerprint *fp2*. Return a float between 0.0 and 1.0, or raise a ValueError if *fp2* is not a hex fingerprint

chemfp.bitops.**byte_to_bitlist**(*bitlist*)
Return a sorted list of the on-bit positions in the byte fingerprint

chemfp.bitops.**byte_union**(*fp1*, *fp2*)
Return the union of the two byte strings, *fp1* | *fp2*

chemfp.bitops.**hex_contains**(*sub_fp*, *super_fp*)
Return 1 if the on bits of sub_fp are also on bits in super_fp, otherwise 0. Return -1 if either string is not a hex fingerprint

chemfp.bitops.**hex_contains_bit**(*fp*, *bit_index*)
Return True if the the given bit position is on, otherwise False.

This function does not validate that the hex fingerprint is actually in hex.

chemfp.bitops.**hex_difference**(*fp1*, *fp2*)
Return the absolute difference (xor) between the two hex strings, *fp1* ^ *fp2*. Raises a ValueError for non-hex fingerprints.

chemfp.bitops.**hex_from_bitlist**(*fp*[, *num_bits=1024*])
Convert a list of bit positions into a hex fingerprint, including modulo folding

chemfp.bitops.**hex_intersect**(*fp1*, *fp2*)
Return the intersection of the two hex strings, *fp1* & *fp2*. Raises a ValueError for non-hex fingerprints.

chemfp.bitops.**hex_intersect_popcount**(*fp1*, *fp2*)
Return the number of bits set in the intersection of the two hex fingerprint, or -1 if either string is a non-hex string

chemfp.bitops.**hex_isvalid**(*s*)
Return 1 if the string is a valid hex fingerprint, otherwise 0

chemfp.bitops.**hex_popcount**(*fp*)
Return the number of bits set in a hex fingerprint, or -1 for non-hex strings

chemfp.bitops.**hex_tanimoto**(*fp1*, *fp2*)
Compute the Tanimoto similarity between two hex fingerprints. Return a float between 0.0 and 1.0, or -1.0 if either string is not a hex fingerprint

chemfp.bitops.**hex_tversky**(*fp1*, *fp2*, *alpha=1.0*, *beta=1.0*)
Compute the Tversky index between two hex fingerprints. Return a float between 0.0 and 1.0, or raise a ValueError if either string is not a hex fingerprint

chemfp.bitops.**hex_to_bitlist**(*bitlist*)
Return a sorted list of the on-bit positions in the hex fingerprint

chemfp.bitops.**hex_union**(*fp1*, *fp2*)
Return the union of the two hex strings, *fp1* | *fp2*. Raises a ValueError for non-hex fingerprints.

chemfp.bitops.**hex_encode**(*s*)
Encode the byte string or ASCII string to hex. Returns a text string.

chemfp.bitops.**hex_encode_as_bytes**(*s*)
Encode the byte string or ASCII string to hex. Returns a byte string.

chemfp.bitops.**hex_decode**(*s*)
Decode the hex-encoded value to a byte string

CHAPTER 10

chemfp.encodings

Decode different fingerprint representations into chemfp form. (Currently only decoders are available. Future released may include encoders.)

The chemfp fingerprints are stored as byte strings, with the bytes in least-significant bit order (bit #0 is stored in the first/left-most byte) and with the bits in most-significant bit order (bit #0 is stored in the first/right-most bit of the first byte).

Other systems use different encodings. These include:

- the '0 and '1' characters, as in '00111101'
- hex encoding, like '3d'
- base64 encoding, like 'SGVsbG8h'
- CACTVS's variation of base64 encoding

plus variations of different LSB and MSB orders.

This module decodes most of the fingerprint encodings I have come across. The fingerprint decoders return a 2-ple of the bit length and the chemfp fingerprint. The bit length is None unless the bit length is known exactly, which currently is only the case for the binary and CACTVS fingerprints. (The hex and other encoders must round the fingerprints up to a multiple of 8 bits.)

chemfp.encodings.**from_binary_lsb**(*text*)

Convert a string like '00010101' (bit 0 here is off) into 'xa8'

The encoding characters '0' and '1' are in LSB order, so bit 0 is the left-most field. The result is a 2-ple of the fingerprint length and the decoded chemfp fingerprint

```
>>> from_binary_lsb('00010101')
(8, '\xa8')
>>> from_binary_lsb('11101')
(5, '\x17')
>>> from_binary_lsb('00000000000000010000000000000')
(29, '\x00\x80\x00\x00')
>>>
```

chemfp.encodings.**from_binary_msb**(*text*)
Convert a string like '10101000' (bit 0 here is off) into 'xa8'

The encoding characters '0' and '1' are in MSB order, so bit 0 is the right-most field.

```
>>> from_binary_msb('10101000')
(8, '\xa8')
>>> from_binary_msb('00010101')
(8, '\x15')
>>> from_binary_msb('00111')
(5, '\x07')
>>> from_binary_msb('00000000000001000000000000000')
(29, '\x00\x80\x00\x00')
>>>
```

chemfp.encodings.**from_base64**(*text*)
Decode a base64 encoded fingerprint string

The encoded fingerprint must be in chemfp form, with the bytes in LSB order and the bits in MSB order.

```
>>> from_base64("SGk=")
(None, 'Hi')
>>> from_base64("SGk=")[1].encode("hex")
'4869'
>>>
```

chemfp.encodings.**from_hex**(*text*)
Decode a hex encoded fingerprint string

The encoded fingerprint must be in chemfp form, with the bytes in LSB order and the bits in MSB order.

```
>>> from_hex('10f2')
(None, '\x10\xf2')
>>>
```

Raises a ValueError if the hex string is not a multiple of 2 bytes long or if it contains a non-hex character.

chemfp.encodings.**from_hex_msb**(*text*)
Decode a hex encoded fingerprint string where the bits and bytes are in MSB order

```
>>> from_hex_msb('10f2')
(None, '\xf2\x10')
>>>
```

Raises a ValueError if the hex string is not a multiple of 2 bytes long or if it contains a non-hex character.

chemfp.encodings.**from_hex_lsb**(*text*)
Decode a hex encoded fingerprint string where the bits and bytes are in LSB order

```
>>> from_hex_lsb('102f')
(None, '\x08\xf4')
>>>
```

Raises a ValueError if the hex string is not a multiple of 2 bytes long or if it contains a non-hex character.

chemfp.encodings.**from_cactvs**(*text*)
Decode a 881-bit CACTVS-encoded fingerprint used by PubChem

```
>>> from_cactvs("AAADceB7sQAEAAAAAAAAAAAAAAAAWAAAAAwAAAAAAAAAABwAAAHwIYAAAADA" +
...             "rBniwygJJqAACqAyVyVACSBAAhhwIa+CC4ZtgIYCLB0/CUpAhgmADIyYcAgAAO" +
...             "AAAAAAABAAAAAAAAAAIAAAAAAAAAAA==")
(881, '\x07\xde\x8d\x00␣
↪\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x80\x06\x00\x00\x00\x0c\x00\x00\x00\x00\x00\x
↪\x00\x84\xe1@X\x1f\x04\x1df\x1b\x10\x06D\x83\xcb\x0f)
↪%\x10\x06\x19\x00\x13\x93\xe1\x00\x01\x00p\x00\x00\x00\x00\x00\x80\x00\x00\x00\x00\x00\x00\x00
↪')
>>>
```

For format details, see ftp://ftp.ncbi.nlm.nih.gov/pubchem/specifications/pubchem_fingerprints.txt

`chemfp.encodings.`**`from_daylight`**(*text*)

Decode a Daylight ASCII fingerprint

```
>>> from_daylight("I5Z2MLZgOKRcR...1")
(None, 'PyDaylight')
```

See the implementation for format details.

`chemfp.encodings.`**`from_on_bit_positions`**(*text*, *num_bits=1024*, *separator=" "*)

Decode from a list of integers describing the location of the on bits

```
>>> from_on_bit_positions("1 4 9 63", num_bits=32)
(32, '\x12\x02\x00\x80')
>>> from_on_bit_positions("1,4,9,63", num_bits=64, separator=",")
(64, '\x12\x02\x00\x00\x00\x00\x00\x80')
```

The text contains a sequence of non-negative integer values separated by the *separator* text. Bit positions are folded modulo num_bits.

This is often used to convert sparse fingerprints into a dense fingerprint.

CHAPTER 11

chemfp.fps_io module

This module is part of the private API. Do not import it directly.

The function *chemfp.open()* returns an FPSReader if the source is an FPS file. The function *chemfp.open_fingerprint_writer()* returns an FPSWriter if the destination is an FPS file.

11.1 FPSReader

class chemfp.fps_io.**FPSReader**

FPS file reader

This class implements the *chemfp.FingerprintReader* API. It is also its own a context manager, which automatically closes the file when the manager exists.

The public attributes are:

metadata
a *chemfp.Metadata* instance with information about the fingerprint type

location
a *chemfp.io.Location* instance with parser location and state information

closed
True if the file is open, else False

The FPSReader.location only tracks the "lineno" variable.

__iter__()
Iterate through the (id, fp) pairs

iter_arenas(*arena_size=1000*)
iterate through *arena_size* fingerprints at a time, as subarenas

Iterate through *arena_size* fingerprints at a time, returned as *chemfp.arena.FingerprintArena* instances. The arenas are in input order and not reordered by popcount.

This method helps trade off between performance and memory use. Working with arenas is often faster than processing one fingerprint at a time, but if the file is very large then you might run out of memory, or get bored while waiting to process all of the fingerprint before getting the first answer.

If *arena_size* is None then this makes an iterator which returns a single arena containing all of the fingerprints.

Parameters **arena_size** (*positive integer, or None*) – The number of fingerprints to put into each arena.

Returns an iterator of *chemfp.arena.FingerprintArena* instances

save(*destination*, *format=None*)

Save the fingerprints to a given destination and format

The output format is based on the *format*. If the format is None then the format depends on the *destination* file extension. If the extension isn't recognized then the fingerprints will be saved in "fps" format.

If the output format is "fps" or "fps.gz" then *destination* may be a filename, a file object, or None; None writes to stdout.

If the output format is "fpb" then *destination* must be a filename.

Parameters

- **destination** (*a filename, file object, or None*) – the output destination
- **format** (*None, "fps", "fps.gz", or "fpb"*) – the output format

Returns None

close()

Close the file

count_tanimoto_hits_fp(*query_fp*, *threshold=0.7*)

Count the fingerprints which are sufficiently similar to the query fingerprint

Return the number of fingerprints in the reader which are at least *threshold* similar to the query fingerprint *query_fp*.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns integer count

count_tanimoto_hits_arena(*queries*, *threshold=0.7*)

Count the fingerprints which are sufficiently similar to each query fingerprint

Returns a list containing a count for each query fingerprint in the *queries* arena. The count is the number of fingerprints in the reader which are at least *threshold* similar to the query fingerprint.

The order of results is the same as the order of the queries.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns list of integer counts, one for each query

threshold_tanimoto_search_fp(*query_fp*, *threshold=0.7*)
Find the fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this reader which are at least *threshold* similar to the query fingerprint *query_fp*. The hits are returned as a *SearchResult*, in arbitrary order.

Parameters

- **query_fp** (*byte string*) – query fingerprint
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

threshold_tanimoto_search_arena(*queries*, *threshold=0.7*)
Find the fingerprints which are sufficiently similar to each of the query fingerprints

For each fingerprint in the *queries* arena, find all of the fingerprints in this arena which are at least *threshold* similar. The hits are returned as a *SearchResults*, where the hits in each *SearchResult* is in arbitrary order.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResults*

knearest_tanimoto_search_fp(*query_fp*, *k=3*, *threshold=0.7*)
Find the k-nearest fingerprints which are sufficiently similar to the query fingerprint

Find all of the fingerprints in this reader which are at least *threshold* similar to the query fingerprint, and of those, select the top *k* hits. The hits are returned as a *SearchResult*, sorted from highest score to lowest.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResult*

knearest_tanimoto_search_arena(*queries*, *k=3*, *threshold=0.7*)
Find the k-nearest fingerprints which are sufficiently similar to each of the query fingerprints

For each fingerprint in the *queries* arena, find the fingerprints in this reader which are at least *threshold* similar to the query fingerprint, and of those, select the top *k* hits. The hits are returned as a *SearchResults*, where the hits in each *SearchResult* are sorted by similarity score.

Parameters

- **queries** (a *FingerprintArena*) – query fingerprints
- **threshold** (*float between 0.0 and 1.0, inclusive*) – minimum similarity threshold (default: 0.7)

Returns a *SearchResults*

11.2 FPSWriter

class chemfp.fps_io.**FPSWriter**

Write fingerprints in FPS format.

This is a subclass of *chemfp.FingerprintWriter*.

Instances have the following attributes:

- metadata - a *chemfp.Metadata* instance
- closed - False when the file is open, else True
- location - a *chemfp.io.Location* instance

An FPSWriter is its own context manager, and will close the output file on context exit.

The Location instance supports the "recno", "output_recno", and "lineno" properties.

write_fingerprint(*id*, *fp*)

Write a single fingerprint record with the given id and fp

Parameters

- **id** (*string*) – the record identifier
- **fp** (*bytes*) – the fingerprint

write_fingerprints(*id_fp_pairs*)

Write a sequence of fingerprint records

Parameters **id_fp_pairs** – An iterable of (id, fingerprint) pairs.

close()

Close the writer

This will set self.closed to False.

CHAPTER 12

chemfp.io module

This module implements a single public class, *Location*, which tracks parser state information, including the location of the current record in the file. The other functions and classes are undocumented, should not be used, and may change in future releases.

12.1 Location

class chemfp.io.**Location**

Get location and other internal reader and writer state information

A Location instance gives a way to access information like the current record number, line number, and molecule object.:

```
>>> import chemfp
>>> with chemfp.read_molecule_fingerprints("RDKit-MACCS166",
...                         "ChEBI_lite.sdf.gz", id_tag="ChEBI ID") as␣
↪reader:
...   for id, fp in reader:
...     if id == "CHEBI:3499":
...         print("Record starts at line", reader.location.lineno)
...         print("Record byte range:", reader.location.offsets)
...         print("Number of atoms:", reader.location.mol.GetNumAtoms())
...         break
...
[08:18:12]  S group MUL ignored on line 103
Record starts at line 3599
Record byte range: (138171, 141791)
Number of atoms: 36
```

The supported properties are:

- filename - a string describing the source or destination
- lineno - the line number for the start of the file

- mol - the toolkit molecule for the current record
- offsets - the (start, end) byte positions for the current record
- output_recno - the number of records written successfully
- recno - the current record number
- record - the record as a text string
- record_format - the record format, like "sdf" or "can"

Most of the readers and writers do not support all of the properties. Unsupported properties return a None. The *filename* is a read/write attribute and the other attributes are read-only.

If you don't pass a location to the readers and writers then they will create a new one based on the source or destination, respectively. You can also pass in your own Location, created as `Location(filename)` if you have an actual filename, or `Location.from_source(source)` or `Location.from_destination(destination)` if you have a more generic source or destination.

__init__ (*filename=None*)
Use *filename* as the location's filename

from_source (*cls*, *source*)
Create a Location instance based on the source

If *source* is a string then it's used as the filename. If *source* is None then the location filename is "<stdin>". If *source* is a file object then its `name` attribute is used as the filename, or None if there is no attribute.

from_destination (*cls*, *destination*)
Create a Location instance based on the destination

If *destination* is a string then it's used as the filename. If *destination* is None then the location filename is "<stdout>". If *destination* is a file object then its `name` attribute is used as the filename, or None if there is no attribute.

__repr__ ()
Return a string like 'Location("<stdout>")'

first_line
Read-only attribute.

The first line of the current record

filename
Read/write attribute.

A string which describes the source or destination. This is usually the source or destination filename but can be a string like "<stdin>" or "<stdout>".

mol
Read-only attribute.

The molecule object for the current record

offsets
Read-only attribute.

The (start, end) byte offsets, starting from 0

start is the record start byte position and *end* is one byte past the last byte of the record.

output_recno
Read-only attribute.

The number of records actually written to the file or string.

The value `recno - output_recno` is the number of records sent to the writer but which had an error and could not be written to the output.

recno
Read-only attribute.

The current record number

For writers this is the number of records sent to the writer, and output_recno is the number of records sucessfully written to the file or string.

record
Read-only attribute.

The current record as an uncompressed text string

record_format
Read-only attribute.

The record format name

where()
Return a human readable description about the current reader or writer state.

The description will contain the filename, line number, record number, and up to the first 40 characters of the first line of the record, if those properties are available.

CHAPTER 13

What's New / CHANGELOG

13.1 What's new in 1.6.1

Released 21 August 2020

This release adds specialized POPCNT implementations for all 8-byte-multiple fingerprint lengths up to 1024 bytes, plus a faster implementation for 8-byte-multiple lengths beyond that. Previously there were only specialized implementations for 24-, 64-, 112-, 128-, and 256-byte fingerprints, which are the most common in cheminformatics.

In one benchmark, small fingerprints (<256 bits) are about 20% faster, medium fingerprints (256 to 1024 bits) are about 10% faster, and larger fingerprints are a few percent faster.

Added two new FingerprintArena methods. *FingerprintArena.sample()* randomly selects a subset of the fingerprints and returns them in a new arena. *FingerprintArena.train_test_split()* returns two randomly selected and disjoint subsets of the area, typically used as a training set and a test set.

BUG FIX: Fixed bug in fpcat where using `--reorder` would write the FPS header twice.

13.2 What's new in 1.6

Released 24 June 2020

The main goal of this release was to improve the no-cost/free software version of chemfp for use as a good baseline for fingerprint similarity search.

Two performance features were backported from chemfp 3:

- The fast integer-based rejection test in chemfp 1.5 was replaced with an even faster popcount rejection test when searching indexed arenas.
- Additional specialized popcount functions were added which use the POPCNT instruction for fingerprints with storage sizes of 24, 64, 112, and 128 bytes, plus additional multiples of 128 bytes up to 1024 bytes. These improve the search performance of 166-bit fingerprints (MACCS keys), 512-bit fingerprints, 881-bit fingerprints (PubChem fingerprints), and 1024-bit, 2048-bit, 3072-bit, ... up to 8192-bit fingerprints.

The overall performance is about 10-20% faster for common fingerprint sizes. (166-bit is about 15% faster, 881-bit is about 20% faster, 1024-bit is about 15% faster, and 2048 is about 10% faster.)

Improved error handling for oe2fps, ob2fps, and rdkit2fps when the underlying toolkit is not installed. Which is growing increasingly common.

Added Tversky functions to chemfp.bitops. Chemfp 1.6 does not support Tversky search, but you may use the Tversky implementation to validate your own code.

Increased the blocksize in the FPS reader by about 10x. The earlier size gave optimal performance in 2010. It's now 2020. The new size gives a ~15% boost to the "scan" performance of an FPS file.

Modified the FPS parsing code to handle ISO timestamps with fractional seconds.

BUG FIX: Fixed bug which prevented reading FPS files using the Windows newline convention.

13.3 What's new in 1.5

Released 16 August 2018

BUG FIX: the k-nearest symmetric Tanimoto search code contained a flaw when there was more than one fingerprint with no bits set and the threshold was 0.0. Since all of the scores will be 0.0, the code uses the first k fingerprints as the matches. However, they put all of the hits into the first search result (item 0), rather than the corresponding result for each given query. This also opened up a race condition for the OpenMP implementation, which could cause chemfp to crash.

The threshold search used a fast integer-based rejection test before computing the exact score. The rejection test is now included in the count and k-nearest algorithms, making them about 10% faster.

Unindexed search (which occurs when the fingerprints are not in popcount order) now uses the fast popcount implementations rather than the generic byte-based one. The result is about 5x faster.

Changed the simsearch `--times` option for more fine-grained reporting. The output (sent to stderr) now looks like:

```
open 0.01 read 0.08 search 0.10 output 0.27 total 0.46
```

where 'open' is the time to open the file and read the metadata, 'read' is the time spent reading the file, 'search' is the time for the actual search, 'output' is the time to write the search results, and 'total' is the total time from when the file is opened to when the last output is written.

Added *SearchResult.format_ids_and_scores_as_bytes()* to improve the simsearch output performance when there are many hits. Turns out the limiting factor in that case is not the search time but output formatting. The old code uses Python calls to convert each score to a double. The new code pushes that code into C. My benchmark used a k=all NxN search of ~2,000 PubChem fingerprints to generate about 4M scores. The output time went from 15.60s to 5.62s. (The search time was only 0.11s on my laptop.)

There is a new option, "report-algorithm" with the corresponding environment variable CHEMFP_REPORT_ALGORITHM. The default does nothing. Set it to "1" to have chemfp print a description of the search algorithm used, including any specialization, and the number of threads. For examples:

```
chemfp search using threshold Tanimoto arena, index, single threaded (generic)
chemfp search using knearest Tanimoto arena symmetric, OpenMP (generic), 8 threads
```

This feature is only available if chemfp is compiled with OpenMP support.

Better error handling in simsearch so that I/O error prints an error message and exit rather than give a full stack trace.

Chemfp 3.3 added the options "use-specialized-algorithms" and "num-column-threads", and the corresponding environment variables CHEMFP_USE_SPECIALIZED_ALGORITHMS and CHEMFP_NUM_COLUMN_THREADS. These are supported for future-compatibility, but will alway be 0 and 1, respectively.

Don't warn about the CHEMFP_LICENSE or CHEMFP_LICENSE_MANAGER variables. These are used by chemfp versions which require a license key.

Fixed bugs in bitops.get_option(). The C API returned an error value and raised an exception on error, and the Python API forgot to return the value.

The setup code now recognizes if you are using clang and will set the OpenMP compiler flags.

13.4 What's new in 1.4

Released 19 March 2018

This version mostly contains bug fixes and internal improvements. The biggest additions are the *fpcat* command-line program, support for Dave Cosgrove's 'flush' fingerprint file format, and support for *fromAtoms* in some of the RDKit fingerprints.

The configuration has changed to use setuptools.

Previously the command-line programs were installed as small scripts. Now they are created and installed using the "console_scripts" entry_point as part of the install process. This is more in line with the modern way of installing command-line tools for Python.

If these scripts are no longer installed correctly, please let me know.

The *fpcat* command-line tools was back-ported from chemfp 3.1. It can be used to merge a set of FPS files together, and to convert to/from the flush file format. This version does not support the FPB file format.

If you have installed the chemfp_converters package then chemfp will use it to read and write fingerprint files in flush format. It can be used as output from the *2fps programs, as input and output to fpcat,

Added *fromAtoms* support for the RDKit hash, torsion, Morgan, and pair fingerprints. This is primarily useful if you want to generate the circular environment around specific atoms of a single molecule, and you know the atom indices. If you pass in multiple molecules then the same indices will be used for all of them. Out-of-range values are ignored.

The command-line option is `--from-atoms`, which takes a comma-separated list of non-negative integer atom indices. For examples:

```
--from-atoms 0
--from-atoms 29,30
```

The corresponding fingerprint type strings have also been updated. If fromAtoms is specified then the string *fromAtoms=i,j,k,...* is added to the string. If it is not specified then the fromAtoms term is not present, in order to maintain compability with older types strings. (The philosophy is that two fingerprint types are equivalent if and only if their type strings are equivalent.)

The `--from-atoms` option is only useful when there's a single query and when you have some other mechanism to determine which subset of the atoms to use. For example, you might parse a SMILES, use a SMARTS pattern to find the subset, get the indices of the SMARTS match, and pass the SMILES and indices to rdk2fps to generate the fingerprint for that substructure.

Be aware that the union of the fingerprint for `--from-atoms` X and the fingerprint for `--from-atoms` Y might not be equal to the fingerprint for `--from-atoms X,Y`. However, if a bit is present in the union of the X and Y fingerprints then it will be present in the X,Y fingerprint.

Why? The fingerprint implementation first generates a sparse count fingerprint, then converts that to a bitstring fingerprint. The conversion is affected by the feature count. If a feature is present in both X and Y then X,Y fingerprint may have additional bits sets over the individual fingerprints.

The ob2fps, rdk2fps, and oe2fps programs now also include the chemfp version information on the software line of the metadata. This improves data provenance because the fingerprint output might be affected by a bug in chemfp.

The *Metadata.date* attribute is now always a datetime instance, and not a string. If you pass a string into the Metadata constructor, like Metadata(date="datestr"), then the date will be converted to a datetime instance. Use "metadata.datestamp" to get the ISO string representation of the Metadata date.

13.4.1 Bug fixes

Fixed a bug where a k=0 similarity search using an FPS file as the targets caused a segfault. The code assumed that k would be at least 1. With the fix, a k=0 search will read the entire file, checking for format errors, and return no hits.

Fixed a bug where only the first ~100 queries against an FPS target search would return the correct ids. (Forgot to include the block offset when extracting the ids.)

Fix a bug where if the query fingerprint had 1 bit set and the threshold was 0.0 then the sublinear bounds for the Tanimoto searches (used when there is a popcount index) failed to check targets with 0 bits set.

13.5 What's new in 1.3

Released 18 September 2017

This release has dropped support for Python 2.5 and Python 2.6. It has been over 7 years since Python 2.7 was released, so if you're using an older Python, perhaps it's time to upgrade?

13.5.1 Toolkit changes

RDKit, OEGraphSim, Open Babel, and CDK did not implement MACCS key 44 ("OTHER") because it wasn't defined. Then Accelrys published a white paper which defined that term. All of the toolkits have updated their implementations. The corresponding chemfp fingerprint types are RDKit-MACCS166/2, OpenEye-MACCS166/3, and OpenBabel-MACCS/2. I have also updated chemfp's own RDMACCS definitions to include key 44, and changed the versions from /1 to /2.

This release supports OEChem v2 and OEGraphSim v2 and drops support for OEGraphSim v1, which OpenEye replaced in 2010. It also drops support for the old OEBinary format.

Several years ago, RDKit changed its hash fingerprint algorithm. The new chemfp fingerprint type is "RDKit-Fingerprint/2".

WARNING! In chemfp 1.1 the default for the RDKit-Fingerprint setting nBitsPerHash was 4. It should have been 2 to match RDKit's own default. I have changed the default to 2, but it means that your fingerprints will likely change.

Chemfp now supports the experimental RDKit substructure fingerprint. The chemfp type name is "RDKit-Pattern". There are four known versions. RDKit-Pattern/1 is many years old, RDKit-Pattern/2 was in place for several years up to 2017, RDKit-Pattern/3 was only in the 2017.3 release, and RDKit-Pattern/4 will be in the 2017.9 release. The corresponding *rdkit2fps* flag is `--pattern`.

RDKit has an adapter to use the third-party Avalon chemistry toolkit to create substructure fingerprints. Avalon support used to require special configuration but it's now part of the standard RDKit build process. Chemfp now supports the Avalon fingerprints, as the type "RDKit-Avalon/1". The corresponding *rdkit2fps* flag is `--avalon`.

Updated the #software line to include "chemfp/1.3" in addition to the toolkit information. This helps distinguish between, say, two different programs which generate RDKit Morgan fingerprints. It's also possible that a chemfp bug can affect the fingerprint output, so the extra term makes it easier to identify a bad dataset.

13.5.2 Performance

The k-nearest arena search, which is used in NxM searches, is now parallelized.

The FPS reader is now much faster. As a result, simsearch for a single query (which uses `--scan` mode) is about 40% faster, and the time for chemfp.load_fingerprints() to create an areana is about 15% faster.

Similarity search performance for the MACCS keys, on a machine which supports the POPCNT instruction, is now about 20-40% faster, depending on the type of search.

13.5.3 Command-line tools

In chemfp 1.1 the default error handler for ob2fps, oe2fps, and rdkit2fps was "strict". If chemfp detected that a toolkit could not parse a structure, it would print an error message and stop processing. This is not what most people wanted. They wanted the processing to keep on going.

This was possible by specifying the `--errors` values "report" or "ignore", but that was extra work, and confusing.

In chemfp 1.3, the default `--errors` value is "ignore", which means chemfp will ignore any problems, not report a problem, and go on to the next record.

However, if the record identifier is missing (for example, if the SD title line is blank), then this will be always be reported to stderr even under the "ignore" option. If `--errors` is "strict" then processing will stop if a record does not contain an identifier.

Added `--version`. (Suggested by Noel O'Boyle.)

The ob2fps `--help` now includes a description of the FP2, FP3, FP4, and MACCS options.

13.5.4 API

Deprecated *read_structure_fingerprints()*. Instead, call the new function *read_molecule_fingerprints()*. Chemfp 2.0 changed the name to better fit its new toolkit API. This change in chemfp 1.3 helps improve forward compatibility.

The chemfp.search module implements two functions to help with substructure fingerprint screening. The function *contains_fp()* takes a query fingerprint and finds all of the target fingerprints which contain it. (A fingerprint x "contains" y if all the on-bits in y are also on-bits in x.) The function *contains_arena()* does the same screening for each fingerprint in a query arena.

The new *SearchResults.shape* attribute is a 2-element tuple where the first is the size of the query arena and the second is the size of the target arena. The new *SearchResults.to_csr()* method converts the similarity scores in the SearchResults to a SciPy compressed sparse row matrix. This can be passed to some of the scikit-learn clustering algorithms.

Backported the FPS reader. This fixed a number of small bugs, like reporting the wrong record line number when there was a missing terminal newline. It also added some new features like a context manager.

Backported the FPS writer from Python 3.0. While it is not hard to write an FPS file yourself, the new API should make it even easier. Among other things, it understands how to write the chemfp *Metadata* as the header and it implements a context manager. Here's an example of using it to find fingerprints with at least 225 of the 881 bits set and save them in another file:

```
import chemfp
from chemfp import bitops
with chemfp.open("pubchem_queries.fps") as reader:
  with chemfp.open_fingerprint_writer(
```

(continues on next page)

(continued from previous page)

```
        "subset.fps", metadata=reader.metadata) as writer:
    for id, fp in reader:
      if bitops.byte_popcount(fp) >= 225:
        writer.write_fingerprint(id, fp)
```

The new FPS reader and writer, along with the chemistry toolkit readers, support the `Location` API as a way to get information about the internal state in the readers or writers. This is another backport from chemfp 3.0.

Backported bitops functions from chemfp 3.0. The new functions are: *hex_contains()*, *hex_contains_bit()*, *hex_intersect()*, *hex_union()*, *hex_difference()*, *byte_hex_tanimoto()*, *byte_contains_bit()*, *byte_to_bitlist()*, *byte_from_bitlist()*, *hex_to_bitlist()*, *hex_from_bitlist()*, *hex_encode()*, *hex_encode_as_bytes()*, *hex_decode()*.

The last three functions related to hex encoding and decoding are important if you want to write code which is forward compatible for Python 3. Under Python 3, the simple fp.encode("hex") is no longer supported. Instead, use bitops.hex_encode("fp").

Note that the chemfp 1.x series will not become Python 3 compatible. For Python 3 support, consider purchasing a copy of chemfp 3.3.

13.5.5 Important bug fixes

Fix: As described above, the RDKit-Fingerprint nBitsPerHash default changed from 4 to 2 to match the RDKit default value.

Fix: Some of the Tanimoto calculations stored intermediate values as a double. As a result of incorrectly ordered operations, some Tanimoto scores were off by 1 ulp (the last bit in the double). They are now exactly correct.

Fix: if the query fingerprint had 1 bit set and the threshold was 0.0 then the sublinear bounds for the Tanimoto searches (used when there is a popcount index) failed to check targets with 0 bits set.

Fix: If a query had 0 bits then the k-nearest code for a symmetric arena returned 0 matches, even when the threshold was 0.0. It now returns the first k targets.

Fix: There was a bug in the sublinear range checks which only occurred in the symmetric searches when the batch_size is larger than the number of records and with a popcount just outside of the expected range.

13.5.6 Configuration

The configuration of the –with-* or –without-* options (for OpenMP and SSSE3) support, can now be specified via environment variables. In the following, the value "0" means disable (same as "–without-*") and "1" means enable (same as "–with-*"):

```
CHEMFP_OPENMP  -  compile for OpenMP (default: "1")
CHEMFP_SSSE3   -  compile SSSE3 popcount support (default: "1")
CHEMFP_AVX2    -  compile AVX2 popcount support (default: "0")
```

This makes it easier to do a "pip install" directly on the tar.gz file or use chemfp under an automated testing system like tox, even when the default options are not appropriate. For example, the default C compiler on Mac OS X doesn't support OpenMP. If you want OpenMP support then install gcc and specify it with the "CC". If you don't want OpenMP support then you can do:

```
CHEMFP_OPENMP=0 pip install chemfp-1.5.tar.gz
```

CHAPTER 14

License and advertisement

This program was developed by Andrew Dalke <dalke@dalkescientific.com>, Andrew Dalke Scientific, AB. It is distributed free of charge under the "MIT" license, shown below.

Further chemfp development depends on funding from people like you. Asking for voluntary contributions almost never works. Instead, starting with chemfp 1.1, there are two development tracks. You can download and use the no-cost version or you can pay money to get access to the commercial version.

This is the no-cost/free software version, available under the MIT license.

The commerical version, currently chemfp 3.4.1, is available under several different licenses, including 1) a no-cost evaluation license for a pre-compiled package for Linux-based OSs, 2) source code-available licenses for internal use, and 3) a full source code license under the MIT license.

I'll stress that: the commercial version of chemfp is available under an open source license, although that is the most expensive option.

The current commercial version is 3.4.1. It can handle more than 4GB of fingerprint data, it supports the FPB binary fingerprint format for fast loading, it has an expanded API designed for web server and web services development (for example, reading and writing from strings, not just files), it supports both Python 2.7 and Python 3.5 or later, and it has faster similarity search performance. Note: chemfp 3.4.x is the last version of the commercial chemfp track to support Python 2.7.

If you pay for the commercial distribution then you will get the most recent version of chemfp, free upgrades for one year, support, and a discount on renewing participation in the incentive program.

If you have questions about or with to purchase the commercial distribution, send an email to sales@dalkescientific.com .

```
Copyright (c) 2010-2020 Andrew Dalke Scientific, AB (Sweden)

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
```

(continues on next page)

(continued from previous page)

```
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Copyright to portions of the code are held by other people or organizations, and may be under a different license. See the specific code for details. These are:

- OpenMP, cpuid, POPCNT, and Lauradoux implementations by Kim Walisch, <kim.walisch@gmail.com>, under the MIT license
- SSSE3.2 popcount implementation by Stanford University (written by Imran S. Haque <ihaque@cs.stanford.edu>) under the BSD license
- heapq by the Python Software Foundation under the Python license
- TimSort code by Christopher Swenson under the MIT License
- tests/unittest2 by Steve Purcell, the Python Software Foundation, and others, under the Python license
- chemfp/rdmaccs.patterns and chemfp/rdmaccs2.patterns by Rational Discovery LLC, Greg Landrum, and Julie Penzotti, under the 3-Clause BSD License
- chemfp/argparse.py by Steven J. Bethard under the Apache License 2.0
- chemfp/progressbar/ by Nilton Volpato under the LGPL 2.1 and/or BSD license
- chemfp/futures/ by Brian Quinlan under the Python license

(Note: the last three modules are not part of the public API and were removed in chemfp 3.1.)

CHAPTER 15

Future

The chemfp code base is solid and in use at many companies, some of whom have paid for the commercial version. It has great support for fingerprint generation, fast similarity search, and multiple cheminformatics toolkits.

There are two tracks for improvements. Most of the new feature development is done in the commerical version of chemfp. I make my living in part by selling software, and few people will pay for software they can get for free.

The chemfp 1.x track is maintained only to provide a good reference baseline for benchmarking other similarity search tools. It only supports Python 2.7.

I will also accept contributions to chemfp. These must be under the MIT license or similarly unrestrictive license so I can include it in both the no-cost and commercial versions of chemfp.

CHAPTER 16

Thanks

In no particular order, the following contributed to chemfp in some way: Noel O'Boyle, Geoff Hutchison, the Open Babel developers, Greg Landrum, OpenEye, Roger Sayle, Phil Evans, Evan Bolton, Wolf-Dietrich Ihlenfeldt, Rajarshi Guha, Dmitry Pavlov, Roche, Kim Walisch, Daniel Lemire, Nathan Kurz, Chris Morely, Jörg Kurt Wegner, Phil Evans, Björn Grüning, Andrew Henry, Brian McClain, Pat Walters, Brian Kelley, Lionel Uran Landaburu, and Sereina Riniker.

Thanks also to my wife, Sara Marie, for her many years of support.

CHAPTER 17

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

W