eiR

Kevin Horan, Yiqun Cao, Thomas Girke

April 30, 2013

Contents

1 Introduction

EiR provides an index for chemical compound databases allowing one to quickly find similar compounds in a very large database. To create this index, r reference compounds are selected to represent the database. Then each compound in the database is embedded into d-dimensional space based on their distance to each reference compound. This requires time linear in the size of the database, but only needs to be done once for a database. Within this space, Locality Sensitive Hashing (LSH) (Dong et al., 2008a,b) is employed to allow sub-linear time nearest neighbor lookups. This means that nearest neighbors can be found without doing a linear scan through the entire compound database. Additional compounds can be added to the database in time linear to the number of new compounds. No time is spend processing existing compounds, as long as the set of reference compounds remains the same. Given the ability to quickly find nearest neighbors, this method enables fast clustering with the Jarvis-Pattrick algorithm as well (Jarvis and Patrick, 1973). For details on the whole process see Cao et al. (2010).

This library uses an SQL back-end (SQLite by default) to store chemical compound definitions, either in SDF or SMILE format, as well as descriptors. Several different kinds of descriptors can be stored for each compound, for example, one could compute and store atom-pair and fingerprint descriptors for each compound. The SQLite database, if used, is stored in a directory called "data". Also in this directory is a file called "Main.iddb", which stores the id of each compound in the current "logical" database. This allows you to have compounds in the SQL database that are not being used, or to create various subsets of one large SQL database. The eiInit function is used to create a new database, it can import data from SDF or SMILE formated files, or an SDFset object.

Once a database has been created, an embedding must also be created (Agrafiotis et al., 2001). In this step the reference compounds are chosen and each compound is embedded into the new space. This step creates a new directory called "run-r-d", where "r" and "d" are the corresponding values. This is the most costly step of the process and is handled by the eiMakeDb function. This step can be parallelized by providing a SNOW cluster to the eiMakeDb function.

Given an embedded database, queries can be run against it with the eiQuery function. Additional compounds can also be added to an existing database and embedding using eiAdd. Performance tests can be run using the eiPerformanceTest function, and Jarvis-Patrick clustering can be done with the eiCluster function.

EiR also provides some mechanisms to allow the user to extend the set of descriptor formats used and to define new distance functions. See Section 7 for details.

2 Initialization

An initial compound database must be created with the following command:

```
> library(eiR)
> data(sdfsample)
> eiInit(sdfsample[1:99])

[1] "createing db"
[1] "99  loaded by eiInit"
 [1] 252 201 231 270 214 233 209 278 226 287 205 277 260 237 203 222 273 217 256
[20] 239 225 235 266 208 211 206 299 296 258 263 281 280 291 243 249 248 212 254
[39] 234 241 245 279 297 261 216 293 207 224 230 295 232 272 219 223 255 238 210
[58] 262 218 286 228 269 276 268 289 202 265 246 213 288 285 229 294 221 236 253
[77] 282 244 242 259 284 264 274 250 271 215 204 267 240 292 298 257 283 275 220
[96] 227 247 290 251
```

EiInit can take either an SDFset, or a filename. If a filename is given it must be in either SDF or SMILE format and the format must be specified with the `format` paramter. It might complain if your SDF file does not follow the SDF specification. If this happens, you can create an SDFset with the `read.SDFset` command and then use that instead of the filename.

Descriptors will also be computed at this time. The default descriptor type is atompair. Other types can be used by setting the `descriptorType` parameter. Currently available types are "ap" for atompair, and "fp" for fingerprint. The set of available descriptors can be extended, see Section 7. EiInit will create a folder called 'data'. Commands should always be executed in the folder containing this directory (ie, the parent directory of "data"), or else specify the location of that directory with the `dir` option.

eiInit will return a list of compound id numbers which represent the compounds just inserted. These numbers can be used to issue queries later.

3 Creating a Searchable Database

In this step the compounds in the data directory will be embedded in another space which allows for more efficient searching. The main two parameters are r and d. r is the number of reference compounds to use and d is the dimension of the embedding space. We have found in practice that setting d to around 100 works well. r should be large enough to "represent" the full compound database.

To help tune these values, `eiMakeDb` will pick `numSamples` non-reference samples which can later be used by the `eiPerformanceTest` function. Since this is the longest running step, a SNOW cluster can be provided to parallelize the task.

`eiMakdDb` does its job in a job folder, named after the number of reference compounds and the number of embedding dimensions. For example, using 300 reference compounds to generate a 300-dimensional embedding ($r = 300, d = 100$) will result in a job folder called run-300-100. The embedding result is the file matrix.<r>.<d>. In the above example, the output would be run-300-100/matrix.300.100.

Since more than one type of descriptor can be stored for each compound, the desired descriptor type must be given to this function with the `descriptorType` parameter. The default value is "ap", for atompair. You can also specify a custom distance function that must be able to take two descriptors in the format specified and return a distance value. The default distance method used is $1 - \text{Tanimoto}(d1, d2)$.

The return value is the path of the refIddb file, which is a file containing list of referenice id values. This file is needed by other functions.

```
> r<- 60
> d<- 40
> refIddb <- eiMakeDb(r,d)
```

4 Queries

Queries can be given in several formats, defined by the `format` parameter. The default format is "sdf". The `queries` parameter can be either an sdf file or and SDFset under this format. Other valid values for `format` are "name" and "compound_id". Under these two formats the `queries` parameter is expected to be a list of compound names (as returned by sdfid on an SDFset), or a list of id numbers from the database, such as what is returned by the eiInit function.

The `r` and `d` parameters are required, as well as the `refIddb` file, which is returned by the `eiMakeDb` function. These three values determine which embedded database to use. As with `eiMakeDb`, the `descriptorType` and `distance` parameters may be given if desired. They will default to atompair and the Tanimoto Coefficient, respectively. Finally, the parameter `K` is the number of results that will be returned. In some cases, particularly if `K` is small, you may need to set it to a larger value and then trim down the result set yourself. This is because LSH is not an exact algorithm. Internally, it actually searches for xK neighbors, where x is referred to as the expansion ratio, generally set to 2. This allows it to pick the best K matches, according to the true distance function, out of a larger set of candidates. When K is small though, sometimes that expansion ratio is not quite enough.

Note also that this function returns distance values and not similarities. Similarities can be computed as shown in the example below.

Then you can perform a query as follows:

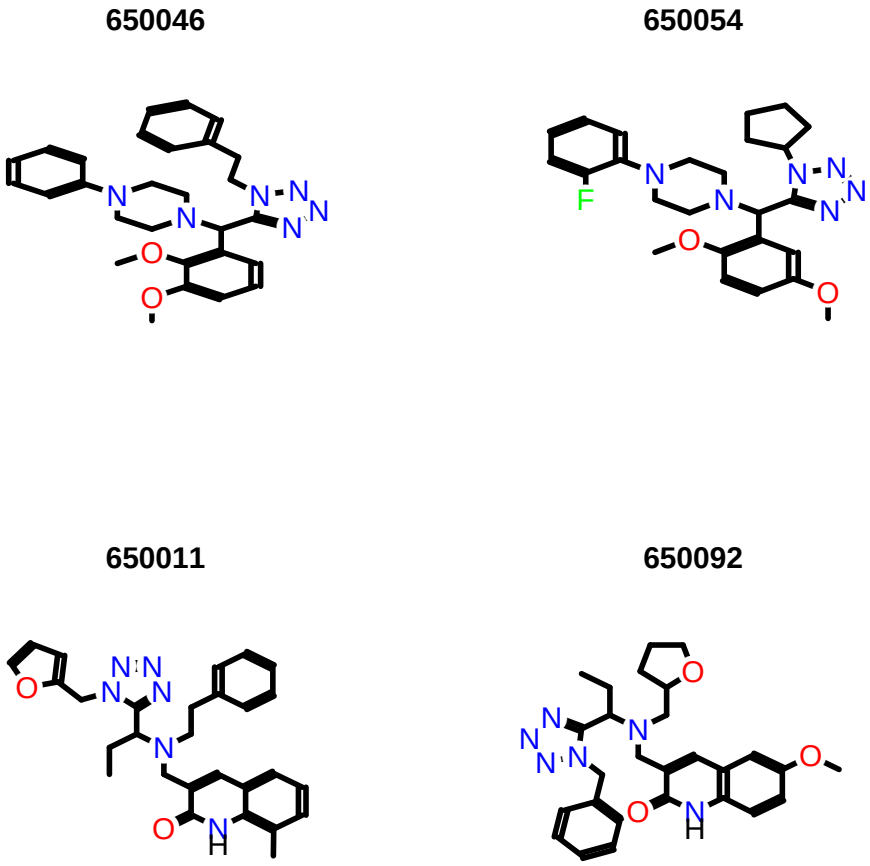
```
> #find compounds similar to each query
> result=eiQuery(r,d,refIddb,sdfsample[45],K=10)
> #compute similarities from distance values
> result$similarities=1-result$distance
> print(result[1:4,])

   query target  distance target_ids similarities
1 650046 650046 0.0000000        245    1.0000000
2 650046 650054 0.4866582        251    0.5133418
3 650046 650011 0.5348837        211    0.4651163
4 650046 650092 0.6077348        286    0.3922652

> #Compare to traditional similarity search:
> data(apset)
> print(cmp.search(apset,apset[45],type=3,cutoff=4,quiet=TRUE))
```

```
  index    cid    scores
1    45 650046 1.0000000
2    51 650054 0.5133418
3    11 650011 0.4651163
4    86 650092 0.3922652
```

```
> cid(sdfsample)=sdfid(sdfsample)
> plot(sdfsample[result$target[1:4]])
```



The result will be a data frame with four columns. The first is a query id, the second is a target, or hit, id, the third is the id number of the target, and the fourth is the distance between the query and target. Lsh parameters can be passed in as well, see Section 6 for more details.

5 Adding New Compounds

New Compounds can be added to an existing database, however, the reference compounds cannot be changed. To add new compounds, use the eiAdd function. This function is very similar to the eiQuery function, except instead of a `queiries` parameter, there is an `additions` parameter, defining the compounds to be added. The format of the value of this parameter works the same as in the eiQuery function. For example, to add one compound from an SDFset you would do:

```
> eiAdd(r,d,refIddb,sdfsample[100])
```

```
[1] "1  loaded by eiInit"
```

6 Performance Tests

The eiPerforamceTest function will run several tests using some sample data to evaluate the performace of the current embedding. It takes the usual r and d parameters, as well as on option distance function and descriptor type, to choose which set of descriptors to use. It also takes several LSH parameters, though the defaults are usually fine. To evaluate the performance you can run:

```
> eiPerformanceTest(r,d,K=22)
```

This will perform two different tests. The first tests the embedding results in similarity search. The way this works is by approximating 1,000 random similarity searches (determined by data/test_queries.iddb) by nearest neighbor search using the coordinates from the embedding results. The search results are then compared to the reference search results (chemical-search.results.gz).

The comparison results are summarized in two types of files. The first type lists the recall for different k values, k being the number of numbers to retrieve. These files are named as "recall-ratio-k". For example, if the recall is 70% for top-100 compound search (70 of the 100 results are among the real top-100 compounds) then the value at line 100 is 0.7. Several relaxation ratios are used, each generating a file in this form. For instance, recall.ratio-10 is the file listing the recalls when relaxation ratio is 10. The other file, recall.csv, lists recalls of different relaxation ratios in one file by limiting to selected k value. In this CSV file, the rows correspond to different relaxation ratios, and the columns are different k values. You will be able to pick an appropriate relaxation ratio for the k values you are interested in.

The second test measures the performance of the Locality Sensitive Hash (LSH). The results for lsh-assisted search will be in run-r-d/indexed.performance. It's a 1,000-line file of recall values. Each line corresponds to one test query. LSH search performance is highly sensitive to your LSH parameters (K, W, M, L, T). The default parameters are listed in the man page for `eiPerformanceTest`. When you have your embedding result in a matrix file, you should follow instruction on `http://lshkit.sourceforge.net/dd/d2a/mplsh-tune_8cpp.html` to find the best values for these parameters.

7 Customization

EiR can be extended to understand new descriptor types and new distance functions. New distance functions can be set in two different ways. Any function that takes a distance parameter can be given a new distance function that will be used for just that call. If no distance function is given, it will fetch a default distance function that has been defined for the given descriptor type. This default value can be changed using the `setDefaultDistance` function, which takes the descriptor type and a distance function. Once this function has been called, the new distance function will be used for that descriptor type by all functions using a distance function. The built-in defaults are defined as follows:

```
> setDefaultDistance("ap", function(d1,d2) 1-cmp.similarity(d1,d2) )
> setDefaultDistance("fp", function(d1,d2) 1-fpSim(d1,d2) )
```

New descriptor types can also be added using the `addTransform` function. These transforms are basically just ways to read descriptors from compound definitions, and to convert descriptors between string and object form. This conversion is required because descriptors are stored as strings in the SQL database, but are used by the rest of the program as objects.

There are two main components that need to be added. The `addTransform` function takes the name of the transform and two functions, `toString`, and `toObject`. These have slightly different meanings depending on the component you are adding. The first component to add is a transform from a chemical compound format, such as SDF, to a descriptor format, such as atom pair (AP), in either string or object form. The toString function should take any kind of chemical compound source, such an SDF file, an SDF object or an SDFset, and output a string representation of the descriptors. Since this function can be written in terms of other functions that will be defined, you can usually accept the default value of this function. The toObject function should take the same kind of input, but output the descriptors as an object. The actual return value is a list containing the names of the compounds (in the names field), and the actual descriptor objects (in the descriptors field).

The second component to add is a transform that converts between string and object representations of descriptors. In this case the toString function takes descriptors in object form and returns a string representation for each. The toObject function performs the inverse operation. It takes descriptors in string form and returns them as objects.

The objects returned by this function will be exactly what is handed to the distance function, so you need to make sure that the two match each other.

For example, to allow atom pair descriptors to be extracted from and SDF source we would make the following call:

```
> addTransform("ap","sdf",
+         # Any sdf source -> APset
+         toObject = function(input,dir="."){
+                 sdfset=if(is.character(input) && file.exists(input)){
+                         read.SDFset(input)
+                 }else if(inherits(input,"SDFset")){
+                         input
+                 }else{
+                         stop(paste("unknown type for 'input',
+                                 or filename does not exist. type found:",class(input)))
+                 }
+                 list(names=sdfid(sdfset),descriptors=sdf2ap(sdfset))
+         }
+ )
> addTransform("ap",
+         # APset -> string,
+         toString = function(apset,dir="."){
+                 unlist(lapply(ap(apset), function(x) paste(x,collapse=", ")))
+         },
+         # string or list -> AP set list
+         toObject= function(v,dir="."){
+                 if(inherits(v,"list") || length(v)==0)
+                         return(v)
+
+                 as( if(!inherits(v,"APset")){
+                                 names(v)=as.character(1:length(v));
+                                 read.AP(v,type="ap",isFile=FALSE)
+                         } else v,
+                         "list")
+         }
+ )
```

References

Dimitris K Agrafiotis, Dmitrii N Rassokhin, and Victor S Lobanov. Multidimensional scaling and visualization of large molecular similarity tables. *Journal of Computational Chemistry*, 22(5):488–500, 2001.

Yiqun Cao, Tao Jiang, and Thomas Girke. Accelerated similarity searching and clustering of large compound sets by geometric embedding and locality sensitive hashing. *Bioinformatics*, 26(7):953–959, 2010.

Wei Dong, Zhe Wang, Moses Charikar, and Kai Li. Efficiently matching sets of features with random histograms. In *Proceedings of the 16th ACM international conference on Multimedia*, pages 179–188. ACM, 2008a.

Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling lsh for performance tuning. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 669–678. ACM, 2008b.

Raymond A Jarvis and Edward A Patrick. Clustering using a similarity measure based on shared near neighbors. *Computers, IEEE Transactions on*, 100(11):1025–1034, 1973.