GOMER:
The User Manual

October 10, 2005

Table of Contents

List of Tables

Installation

Requirements

Python, version 2.2.1 or higher.

http://python.org/download/

Numeric (numpy NOT numarray, at least not yet)

http://sourceforge.net/project/showfiles.php?group_id=1369&release_id=144555

Optparse (Optik).

This is part of the standard python distribution as of version 2.3, so if you have version 2.3 or higher, you don't need to download it.

http://sourceforge.net/project/showfiles.php?group_id=38019

If in doubt, run it (see “Rough Test”); GOMER should complain if one or more of the above is missing. If GOMER complains, install what you need, if it doesn't, you should be golden.

Recomendations

psyco http://psyco.sourceforge.net/

Psyco does some optimization magic and can significantly speed up GOMER. I have found a three to four-fold reduction in run time when I use psyco. GOMER will use psyco if it is available. At time of writing, psyco only works on intel processors (or intel “clones”, like AMD). Psyco is processor dependent, but not Operating System dependent, I have found speed-ups in

Linux and Windows. To test if GOMER will be able to use psyco (once you have installed it), run the **test_psyco.py** script.

python optimization option

If you run GOMER by invoking python directly (i.e. typing "`python gomer.py`" at a command prompt), giving python the "-O" or "-OO" option should speed up execution somewhat. Alternatively, you can set the "PYTHONOPTIMIZE" environment variable to 1 or 2, either in your shell's configuration file (e.g. .bashrc, .cshrc, etc.) or each time you invoke a shell in which you will run GOMER. Another option is invoke GOMER using **gomer.sh**. This wrapper, which is included with the GOMER code, should behave the same as invoking GOMER directly (i.e. it should accept the same command-line options), but it automatically uses the python optimization. You can confirm that optimized byte-code is being generated, by checking to see if *.pyo files, instead of *.pyc files, are generated in the directory where the *.py files are found.

Installing

Unpacking

1. cd into the directory where you want GOMER
2. run "tar -zxvf *path_to_tarfiles*/GOMER.tar.gz
3. cd into the directory where you want the genome
4. run "tar -zxvf *path_to_tarfiles*/genomes.tar.gz

Cache Directory

GOMER's cache files (cache files are discussed in the "Caching" section) are all stored in a cache directory. Cache files can be quite large, so it is a good idea to keep an eye on the cache directory and consider deleting cache files that are no longer useful. It is entirely safe to delete the GOMER cache files (as long as GOMER is not running at the time). GOMER recognizes when cache files have been deleted, and handles it gracefully (it even cleans up the relevant entry in the **cache_table** file), the only thing you lose is the time spent regenerating the cache file if the same combination of probability matrix and sequence is used again. It is also OK to delete

the **cache_table** file found in the cache directory. However deleting this file makes all existing cache files useless (since the **cache_table** file contains the information GOMER uses to quickly recognize if a relevant cache file exists), so only delete the **cache_table** if you are also deleting all of the other files in the cache directory. As part of the installation process, you must create a cache directory. This can easily be done with the command **mkdir /home/bob/stuff/cache_directory**. This assumes that the path /home/bob/stuff/ already exists. Of course, the name of the cache directory can be anything you want. You must also supply the cache directory's path to GOMER in the config file (for details see the section "GOMER Configuration File"). Technically, you can use any directory you want for the cache directory, you do not necessarily need to create a new directory for the cache directory, and you could have other files in the cache directory, but this might be untidy, and I wouldn't recommend it.

GOMER Configuration File

Location

You will need to create a GOMER configuration file. See the file format description in "GOMER Configuration File." A sample can be found in the examples directory of the GOMER distribution. When GOMER is run, if it is given the `-c/--config FILE`, option, FILE is used as the configuration file, otherwise, GOMER searches for a file named "gomer_config" in the following locations, in order (see discussion of "GOMER_HOME"):

```
$GOMER_HOME/
$GOMER_HOME/input_files/config/
$GOMER_HOME/input_files/
$GOMER_HOME/../input_files/config/
$GOMER_HOME/../input_files/
```

GOMER_HOME

The GOMER_HOME environment variable can be set by the user to tell GOMER where to find the GOMER directory. If it is not found in the environment, GOMER internally sets GOMER_HOME to the directory from which GOMER was run. Regardless of how it is set,

GOMER_HOME is used as a starting point when looking for files (i.e. it becomes part of the included in the search path), for an example, see the discussion on the GOMER config file format.

To set GOMER_HOME, at a Bourne shell prompt (sh or bash), or in your .bashrc file, type `export GOMER_HOME=`"*path*", where path is the path to the GOMER home directory. At a C shell prompt (csh or tcsh) , or in .cshrc, .tcshrc type `setenv GOMER_HOME` *path*.

Genome Sequences

Unless you are only going to use "sequence_feature" mode, you will need genome sequences. Currently GOMER is only capable of parsing the feature files of *Saccharomyces cerevisiae* available from the *Saccharomyces* Genome Database (SGD™) at http://www.yeastgenome.org/.

At the time of writing, the feature file for *Saccharomyces cerevisiae* could be found at (files named chrXX.fsa):

ftp://genome-ftp.stanford.edu/pub/yeast/data_download/chromosomal_feature/chromosomal_feature.tab

And the sequences could be found at:

ftp://genome-ftp.stanford.edu/pub/yeast/data_download/sequence/genomic_sequence/chromosomes/fasta/

Genome Installation

To "install" a genome, first download the chromosomal sequences and the feature file into a single directory. It is then necessary to create a "chromosome table" file. There is an example in the example directory of the GOMER distribution.

Rough Test

cd into "GOMER/python_code" Run[12]:

[1] In sample command lines provided here, the backslash ('\') indicates a line break where a command line is too long to fit

```
1. ./gomer.py -s ../test_short_output      \
   ../examples/cerevisiae_chromosome_table \
   ../examples/paper_gold.probs                        \
   ../examples/a1_alpha2_regulated.list                \
   -r kappa_modules/single_square_reg_region_model_gomer.py \
   "regulatory_five_prime=600;regulatory_three_prime=0"  >
   ../test_run_std_output
2. diff -s ../test_short_output ../examples/test_short_output
3. diff -s ../test_run_std_output ../examples/test_run_std_output
```

The results of the diff's shouldn't be too large (I know that this isn't very helpful), if you're lucky, the files will be identical. However, if the first command runs without errors, then everything is probably OK.

---

on a single line in this document. When running these sample commands, one can put everything on one line, and should not include the backslashes.

[2] Remember that in the arguments to the -r/--reg_region_kappa option, the parameters in the parameter string (here "regulatory_five_prime=600;regulatory_three_prime=0") are separated by a *semicolon*, anything else will give you an error!

GOMER Command Line

Below is a brief description of the command line options that GOMER accepts. These options and the information supplied with them are described in more detail in the appropriate places in this manual. In the standard mode, GOMER expects three command line arguments.

- **Chromosome Table** – The information in this file provides the location of the chromosome sequence and annotation files, and a mapping between a chromosome's label and the chromosome's sequence file.
- **Probability Matrix** – This file contains the description of the binding specificity of the primary transcription factor.
- **Regulated Features** – This file lists the genes that are regulated (and where appropriate the genes which are unregulated, or should be excluded from analysis) by the primary transcription factor.

In other modes, different command line arguments are required.

Alternative Modes

Using a "mode" option changes the command line arguments that GOMER expects.

Coordinate Feature Mode

--coordinate_feature=CoordinateFeature

In this mode, GOMER expects, as an argument to the "--coordinate_feature" option, the name of the coordinate feature file. The coordinate feature file consists of a list of features and their genome coordinates. This mode allows a user to supply GOMER with genomic features not described in the genome annotation file. This option can be given more than once, so that multiple coordinate feature files can be used in a single run. In addition to the coordinate feature file, when this mode is used, GOMER expects the standard command line arguments

(Chromosome Table, Probability Matrix, and Regulated Features).

When run with coordinate style features, instead of using features defined in the standard annotation file, a "coordinate feature file" is supplied that (at minimum) provides a feature name, chromosome, and start and stop base pair. These coordinate features often define a regulatory region, or an equivalent feature. For this reason, distinct coordinate feature weight functions are generally needed (for example, a regulatory weight function that defines a regulatory region that is upstream of genome features would be inappropriate for coordinate features, which are themselves regulatory sequences). A good example of a use for this mode (and the reason this option was provided) is to enable scoring "intergenic regions" that are spotted for ChIP on chip microarrays, in this case, the coordinate feature file would give the name and coordinates of the PCR products which are spotted.

The coordinate feature file can either be tab-separated or comma-separated. The columns of the file are: Name, Chromosome, StartCoord, StopCoord, Alias, Corresponding, Description, but values are only required Name, Chromosome, StartCoord, and StopCoord. If there is a fifth column it is assumed to be an alias list[3], if there is a sixth column, it is assumed to be a "corresponding" list[4], if there is a seventh column, it is assumed to be a Description[5]. Because of this, if you desire to supply a description, but not an alias list or corresponding list, you must still supply at least empty values for alias and corresponding. For comma separated files, this would look like:

```
ACH1,2,193484,194084,,,acetyl CoA hydrolase
```

For tab separated files:

```
ACH1 2 193484       194084        acetyl CoA hydrolase
```

This illustrates a situation where the comma separated format can be useful - if one is generating or editing a coordinate file by hand, it can be difficult (without special tools) to distinguish tab characters from space characters, and how many tabs are present.

[3] If more than one alias is given, the aliases in the list should be separated by ``|'' characters, *not* by tabs or commas

[4] Also separated by ``|''

[5] It is important to keep in mind that the separator (comma or tab) should not appear *within* the column, i.e. if using a comma-separated format, the data in the Description column should not contain any commas

Sequence Feature

--sequence_feature=SequenceFeatureFile

In this mode, GOMER expects, as an argument to the “--sequence_feature” option, the name of the sequence feature file. The sequence feature file should be a multi-sequence FASTA file (with minor restrictions). In this mode, the sequences themselves are the features. This mode allows GOMER to be used with sequences that cannot be found in the genome sequence and annotation formats that GOMER expects. This option can be given more than once, so that multiple sequence feature files can be used in a single run. In addition to the sequence feature file, when this mode is used, GOMER expects the Probability Matrix and Regulated Features command line arguments (of course, a genome sequence is irrelevant in this mode).

Instead of using the standard feature file, a "sequence feature file" is supplied. The sequence feature file is essentially a multi-sequence FASTA file, where the feature name is the first "word" on the FASTA header (comment) line. These sequence features are then used like regulatory regions "features." Because of this a sequence feature kappa function is needed. This mode is similar in spirit to the coordinate feature mode, but where the coordinate feature mode uses feature coordinates on the chromosome sequences to determine the "feature"(regulatory region) sequence, the sequence feature (multi-FASTA) file provides all the sequences, no chromosome sequence is used. Again, a good example of a use for this mode (and the reason this option was provided) is to enable scoring "intergenic regions" that are spotted for ChIP on chip microarrays, in this case, the sequence feature file would give the name and sequence of the PCR products which are spotted.

Potential Differences Between Results for Coordinate and Sequence Features

In many situations, coordinate features will receive the same score as the sequence features generated from those coordinates (as long as one uses equivalent regulatory region weight functions, such as **coordinate_single_square_reg_region_model_gomer.py** and **sequencefeature_single_square_reg_region_model_gomer.py**). However, this “equality” will not hold when a model depends on sequences that at all fall outside the features themselves,

since for the sequence feature, there is no sequence defined outside of each of the features. This inequality is most likely to be a problem when cooperative or competitive models are used, since cooperative and competitive distances are relative to the primary transcription factor binding site. In scoring a coordinate feature, secondary binding sites outside of the feature can be taken into consideration (since the whole chromosome sequence is available). Since only the sequence for the feature itself is available, only the supplied sequence can be taken into consideration when looking for cooperative binding in a sequence feature.

Top sites

-tNUM, --top=NUM

GOMER finds and reports the "NUM" top scoring binding sites in genome, and the feature closest to each site. In addition to the number of top sites, when this mode is used, GOMER expects the Chromosome Table and Probability Matix command line arguments. If two feature are equidistant, both will be printed, (the second feature will be printed on a line by itself, without "hit" information), for example:

```
3947.764 4 1206689 1206708 W YDRWdelta27 1206692 1207025
                            YDRWTy1-5    1206692 1212609
```

Informational Modes

Base Frequencies

--frequency=ChromTable

In this mode GOMER calculates and reports the base frequencies of the sequences in the Chromosome Table supplied as an argument.

Ambiguous names

-aFILE, --ambiguous=FILE

GOMER determines and reports what names in the supplied feature file are ambiguous.

List feature types

--feature_types=FEATURE FILE

Report the types of features found in FEATURE FILE

kappa function help

-kKAPPA_FILE, --kappa_help=KAPPA_FILE

List the params required by KAPPA_FILE, an example, and a description. This option is not as useful as the kappa_help.py utility supplied in the GOMER suite.

GOMER run parameters

Configuration file

-cFILE, --config=FILE

Use the configuration file name supplied instead of the default configuration file.

Free concentration ratio

--free_conc_ratio=CONC_RATIO

Use the free concentration supplied (as a ratio of the default) for the primary transcription factor, instead of the default free concentration.

Feature type selection

-fFEATURE, --feature=FEATURE

Score genome features of the type supplied. This option can be supplied more than once, if multiple feature types should be scored. Feature type names are case-sensitive

Filter cutoff

--filter_cutoff_ratio=CUTOFF

A CUTOFF value less than one turns off filtering.

Kappa Functions

Regulatory region kappa function

-rKAPPA_FILE PARAM_STRING

--reg_region_kappa=KAPPA_FILE PARAM_STRING

Use the regulatory kappa function supplied in KAPPA_FILE, with the parameters in the PARAM_STRING, instead of the default regulatory kappa function (defined in the GOMER configuration file). The PARAM_STRING argument should be a string consisting of parameter names and parameter values, separated by an equal sign (e.g. “mean=100”). If a kappa function requires more than one parameter, the name/value pairs should be separated by semi-colons (e.g. “mean=100;std_dev=100;cutoff=0.001”).

Cooperative kappa function

--coop_kappa=

KAPPA_FILE

PARAM_STRING

PROB_MATRIX

FREE_CONC_RATIO

K_DIMER_RATIO

Account for cooperative interactions between the primary transcription factor and the transcription factor represented by PROB_MATRIX when calculating the GOMER score. This option can be supplied multiple times to account for many different transcription factors interacting with the primary.

KAPPA_FILE
The file containing the cooperative kappa module.

PARAM_STRING
This is the string containing initialization parameters for the coop kappa module. See Regulatory region kappa function for a description of the format.

PROB_MATRIX
The binding site description for the secondary transcription factor of interest.

FREE_CONC_RATIO
The free concentration ratio. If FREE_CONC_RATIO is '1' or 'default', the default value is used ($1/K_a^{max}$, where K_a^{max} is the best possible K_a given by the PROB_MATRIX), otherwise, the free

concentration will be (FREE_CONC_RATIO x $1/K_a^{max}$)

K_DIMER_RATIO

K_DIMER_RATIO is used to calculate the K_{dimer} (dimerization constant) between the primary transcription factor and the secondary transcription factor (the transcription factor supplied as PROB_MATRIX). If K_DIMER_RATIO is "1" or "default," the value of K_{dimer} is K_a^{max} (essentially the default value). Otherwise, the K_{dimer} will be (K_DIMER_RATIO x K_a^{max}). In both of these cases, K_a^{max} refers to the K_a^{max} of the secondary transcription factor.

Competition kappa function

--comp_kappa=

KAPPA_FILE

PARAM_STRING

PROB_MATRIX

FREE_CONC_RATIO

Account for competitive interactions between the primary transcription factor and the transcription factor represented by PROB_MATRIX when calculating the GOMER score. This option can be supplied multiple times to account for many different transcription factors interacting with the primary.

KAPPA_FILE

The file containing the competitive kappa module.

PARAM_STRING

The string containing initialization parameters for the coop kappa module. See Regulatory region kappa function for a description of the format.

PROB_MATRIX

The binding site description for the secondary transcription factor of interest.

FREE_CONC_RATIO

The free concentration ratio. If FREE_CONC_RATIO is '1' or 'default', the default value is used ($1/K_a^{max}$, where K_a^{max} is the best possible K_a given by the PROB_MATRIX), otherwise, the free concentration will be (FREE_CONC_RATIO x $1/K_a^{max}$)

Output

The ranks of features run from one to the number of features. In both the output generated by -s/-short_output and, the regular output, the best scoring feature has a rank of one.

Short ouput

-sSHORT_OUTPUT_FILE, --short_output=SHORT_OUTPUT_FILE

Save a short version of the output to the SHORT_OUTPUT_FILE. This option has no effect on whether or not long output is generated. If the file name supplied to the -s/-short_output option is "-", then the short output is printed to standard out.

Long output

-oOUTPUT_FILE, --output=OUTPUT_FILE

Save the output of the run to OUTPUT_FILE. This option takes precedence over the no long output option.

No long output

-n, --no_long

Do not generate long output.

Cache

Fast cache

--fast_cache=DIR

Use the supplied directory path as a fast cache directory, overriding any fast cache value supplied in the configuration file.

Delete cache

-D, --delete_cache

Delete the cache file when execution is complete. This option only works when the cache is originally generated, it will not delete a preexisting cache file (this must be done by hand), since, in reality, this option causes a newly generated cache to not be saved. The delete cache option takes precedence over the compression option, so if both options are supplied, the cache file will be deleted.

Cache compression

-CLEVEL, --compression=LEVEL

Use compression on the cache file. LEVEL is an integer from 1 to 9 setting the level of compression; 1 is fastest but compresses the least, 9 is slowest but compresses maximally. Any value not between 1 and 9 defaults to 9

Cache directory

--cache_directory=DIR

Use the supplied directory path as the cache directory, instead of the directory supplied in the GOMER configuration file.

File Formats

In general, GOMER will accept (and appropriately expand) pathnames that contain environment variables and “~” characters.

GOMER Configuration File

Entries

Those entries that are required in the configuration file are shown in Table 1. For **regulatory_region**, try **single_square_reg_region_model_gomer.py**. A reasonable default value for **regulatory_region_params**, for this kappa function is “regulatory_five_prime=600; regulatory_three_prime=1”, this considers the regulatory region for a feature to be the 600bp immediately 5' to it, ending with the last base before the start of the feature. These optional configuration file entries are shown in Table 2.

Format

Python's ConfigParser module is used to parse the GOMER configuration file, so a description of the general format can be found in the documentation for that module.

In the configuration file, a section name is enclosed in square brackets ([Section]), and occurs on its own line, and entries are found (also one per line) in the format "name=value" or "name: value" (Table 3).

Chromosome Table File

The chromosome table file (Table 4) tells GOMER where to find the chromosome sequence files, the genome annotation file(s), and (optionally) the base frequencies of the genome.

Table 1: Required GOMER configuration file entries

Section	Parameter	Description
directories	cache_directory	Directory for storing cache files
kappas	regulatory_region	Default regulatory region kappa module
kappas	regulatory_region_params	Default parameters for default regulatory region kappa

Table 2: Optional GOMER configuration file entries

Section	Parameter	Description
directories	fast_cache	A fast drive or ramdisk for storing cache files while in use
directories	gomer_home	The home directory for GOMER
kappas	coordinate_regulatory_region	Default regulatory region weight function module in coordinate feature mode
kappas	coordinate_regulatory_region_params	Default parameters for default coordinate regulatory region kappa
kappas	sequence_regulatory_region	Default regulatory region weight function module in sequence feature mode
kappas	sequence_regulatory_region_params	Default parameters for default sequence regulatory region kappa
parameters	feature_types	A comma separated list of feature

Table 3: Sample of GOMER configuration file.

```
[directories]
cache_directory = ~/GOMER_data/window_cache
gomer_home = ~/GOMER

[kappas]
regulatory_region_params = regulatory_five_prime=600;
regulatory_three_prime=1
```

Table 4: Sample Chromosome Table File

```
Directory: ~/genomes/sacch_cere/oct_23_2002/
Feature file: chromosomal_feature.tab

Frequency: A 0.30851345595763963
Frequency: C 0.19148654404236037
Frequency: T 0.30851345595763963
Frequency: G 0.19148654404236037

Chromosome: 1 chr01.fsa
Chromosome: 2 chr02.fsa
Chromosome: 3 chr03.fsa
. . .
```

At first, the chromosome table should have the following information:

"Directory:" statement:

The absolute path to the directory where the sequences and feature file can be found

"Feature file:" statement:

The name of the feature file (no path, the feature file must be in the same directory as the sequences[6])

"Chromosome:" statement:

One for each chromosome, this provides a mapping from chromosome names (as used in the feature file)[7] to chromosome sequence files.

[6] This requirement ensures that even if different versions of the same genome exist on the computer, it takes some effort to use the wrong feature file.

[7] In the *Saccharomyces cerevisiae* feature file, all chromosomes are refered to by (arabic) numbers - 1 through 16, and 17 for the mitochodrial genome, these arabic numbers should be used as the chromosome names. Note that in the sequences provided by SGD™, the header contains roman numeral versions of the chromosome number, except the mitochodrial genome which simply gives the modifier ``[location=mitochondrion]"

Once you have created the chromosome table, it is a good idea to determine the base frequencies for the sequences in the genome[8]. Run GOMER using the command

`gomer.py --frequency` *`ChromsomeTable`*[9]

The “Frequency:” statements that follow the “Calculated Base Frequencies:” line can be directly copied to the Chromsome Table file you have just created.

Binding Matrix

The binding matrix file (Table 5) consists of a header that declares the name and pseudo-temperature of the matrix, and ends with a whitespace separated base header. Following the header are lines of whitespace separated base probabilities. Lines beginning with a “#” are considered comments, but they are only allowed in the header (above the “base header”). The probabilities for a position (i.e. the probabilities on a line) must add to 1, and since probabilities of 0 are meaningless, they are not allowed. You can see a sample binding matrix in the example directory, a file named paper_gold.probs.

Table 5: Sample Binding Matrix File

```
# Comment line
%PSEUDO_TEMP          300
%NAME                 a1_alpha2_paper_gold

A         C         G         T
0.1       0.2       0.4       0.3
0.2       0.3       0.3       0.2
0.6       0.1       0.2       0.1
```

Generating a Binding Matrix

If you have a probability matrix where the positions don't add up to 1 or with probabilities of

[8] This step is not required, but if you do not include the base frequencies in the chromsome table, they will be computed on the fly every time you run GOMER

[9] Depending on how your PATH environment variable is configured, you might have to prepend ``./'' to commands discussed in these instructions, so ``test_controller_gomer.py'' would become ``./test_controller_gomer.py''. How can you tell? It never hurts to prepend ``./'', but you can always try it without first, if you get a message like ``bash: test_controller_gomer.py: command not found'', this means you need to prepend ``./''

zero, you can use the script `prob_to_pwm.py`, with either the `-p/--probs` or `--gomer` option. Both modes replace zeroes with the smallest of:

- Half of the smallest non-zero probability in the matrix
- 0.01

The `--gomer` option generates a file which is acceptable as a binding matrix for input to GOMER, by adding a `%PSEUDO_TEMP 300` line and a `%NAME` line, using the name of the input file for the name.

If this is not satisfactory, one can make the necessary changes by hand. Actually, the probabilities must add to 1 ± 10^{-10}, so changing zero positions to a value sufficiently smaller than 10^{-10}, for example "1e-20", will have virtually no effect on the scores generated by the matrix, but GOMER won't complain.

Regulated Feature File

The basic file format for regulated feature files (Table 6) had three types of lines:

- Empty Lines: Lines containing only white space, or nothing are ignored.
- Comment Lines: Lines where a "#" precedes any text (other than whitespace) is considered a comment line. These are ignored.
- Feature Line: A feature name (feature names are case-insensitive) followed by (tab-separated - because there are a few feature names in the yeast genome which have spaces in their names) a "status" tag.

There are three status types[10]:

- **Excluded:** These features are excluded (i.e. they are neither regulated nor unregulated) from the statistical analysis. This status overrides.
- **Unregulated:** These features are considered to be unregulated. At the moment, this tag is superfluous, since all features that are included in an analysis, but not explicitly named in the regulated feature file are considered unregulated.
- **Regulated:** These features are considered regulated. The regulated tag is not required,

if a feature line consists of a feature on a line, with no status tag, it is considered to be regulated. The above statuses are listed in order of precedence, in other words, if a feature is listed twice (or three times) in a regulated feature file as Excluded and Regulated, the Excluded status takes precedence. In this case, where a feature name is listed with more than one status, GOMER will print an error message, but it will continue to run. There is a similar situation possible, which is slightly more complicated. Since features can have multiple names, in this case, GOMER will print an error message and *quit*!

Any feature that is not explicitly named in the regulated feature file is considered unregulated.

Features named in the regulated feature file, for which only the feature name is given, (no other information, including status), are considered regulated. In other words, the regulated feature file can be a simple list consisting solely of feature names (Table 7).

The regulated feature file should have one feature per line. For the sake of thoroughness, it is encouraged but not required regulated features be labeled with a “regulated” tag. In other words, for the feature files shown in Table 7 YBL016W, YCR040W, YDR103W, YGR044C, YHR005C, YPR122W are all regulated, even though some do not have status tags.

[10] Status types names are case-insensitive

Table 6: Complex Regulated Feature File

```
YDL227C excluded
YBR080C excluded

# This line is a comment!
YGR044C regulated
YHR005C regulated
YPR122W regulated

# The following are considered Regulated, since they
# have no status tag
YBL016W
YCR040W
YDR103W

# Normally the following would be considered Regulated
# since it has no status tag, but it is also listed as
# excluded, so the excluded status takes precedence
YDL227C
```

Table 7: Simple regulated feature file

```
YBL016W
YCR040W
YDL227C
YDR103W
YGR044C
YHR005C
YPR122W
```

Comment lines are allowed, they must begin with a pound sign ("#"). If MNCP and ROC AUC scores are calculated (this depends on the output type chosen), these features will be used as the "regulated" set, and their scores and rank will be output. If a name is given for which there is no known feature (e.g. if the name is misspelled), it will be ignored and not considered in calculating statistics. Unknown "regulated" features are reported in the output. The names given in the Regulated Feature File must be unambiguous[11], if an ambiguous name is given, GOMER will complain. A detailed discussion of testing for ambiguous file names can be found in the section Checking for Ambiguous Feature Names.[12]

End Of Line

GOMER expects input files to have lines terminated with a UNIX style (linefeed) end-of-line character. End-of-line characters can be checked and fixed by using the script **check_end_of_line.py**, which is included in the GOMER suite.

Sequences

GOMER expects sequences to consist of the standard bases, A,C,G, or T. There are two exceptions allowed:

- X: The null-symbol. Any binding site containing this symbol is assigned a K_a of 0.
- N: The no-information symbol. This symbol carries no information, so it doesn't

[11] The ambiguity test is case-insensitive, in other words, if one feature has a name "MY_FEATURE1" and another feature has a name "my_feature1", the ambiguity test will consider both names (and any case variation of this name, such as "My_FeATure1") to be ambiguous.

[12] Keep in mind that if, for example, "snR47" is listed as regulated in a feature file which is used for an analysis that only includes ORFs, it is essential ignored, since it is a snoRNA, not an ORF, so it will not be scored, and therefore not considered in calculating the MNCP and ROC AUC statistics.

contribute to the K_a for any binding site containing it.

These non-standard bases are “self-complementing.” In other words, the complement of an “N” base is “N.”

Sequence Feature Files

Sequence feature files (Table 8), are essentially multi-sequence FASTA files. Sequences are separated by headers. Headers in sequence feature files are essentially the same as “headers” in FASTA files, they begin with a “>“, followed by the name of the feature described by the sequence that follows it. Lines of whitespace are allowed between the end of one sequence, and the header for the next sequence, but they are not required. Empty lines are, however, not allowed within a sequence body! Whitespace between the “>“ and text is allowed, but not required.

Additional optional information is allowed in the header, it must be separated from the feature name by a tab character, and tabs are used to separate each additional piece of information. The tab separated information *must* be provided in the format “LABEL=VALUE”. The VALUEs will be stored in a dictionary, keyed by the LABELs. Currently this information is parsed and the dictionary is returned by the parser, but nothing is done with this dictionary

Table 8: Sample Sequence Feature File

```
> FEATURE_NAME1 other information file=myfile.txt
acgtaaaaccgtgc
acgtggtctccgta

> FEATURE_NAME2 other information file=myfile2.txt
acgtaaaacccgtg
ttacc

> FEATURE_NAME3 other information file=myfile.txt
acgtaaaaacgttg
aaacgttaccgtt
```

Kappa Weight Functions

At runtime, GOMER loads and compiles the kappa weight function files, which are supplied as parameters of the run. This feature allows the user to create and utilize novel kappa functions of all types. In order for GOMER to utilize a user-supplied kappa function, it must supply a prescribed application programming interface (API) described below for each of the three types of kappa functions. Also described below are tools that can be helpful in designing and testing novel kappa functions.

Kappa Search Path

When a kappa file name is supplied (either in the GOMER configuration file, or on the command line), GOMER searches (in this order) for the named kappa file relative to the

1. Current Working Directory
2. kappa_directory (if defined in the GOMER configuration file)
3. GOMER home directory.

Module Data Attributes

Every kappa module should have the following data attributes. This data is used by GOMER and kappa_help.py to provide information to the user:

- DESCRIPTION: A description of the module itself and its usage.
- NAME
- PARAMETERS: A dictionary of the parameters required to initialize this kappa model. The keys are the parameter names, and the values are the data types of the parameters. For example:

```
PARAMETERS = {'mean':types.IntType,
              'std_dev'  : types.FloatType,
              'max_dist' : types.IntType,
```

```
            'cutoff'   : types.FloatType}
```

- PARAMETER_STRING_EXAMPLE: A sample parameter. It is helpful if the values given here are reasonable starting values for a normal run. For example (note that this module, can be initialized with a *max_dist* or *cutoff* parameter):

```
PARAMETER_STRING_EXAMPLE = \
     ("mean=300; std_dev=50; max_dist=250",
      "mean=300; std_dev=50; cutoff=0.0001")
```

- REQUIRES_STRAND: Whether or not this model requires strand information from features. Models that require strand information cannot be used with features that are "strandless" (e.g. sequence features). This value must be true or false.
- TYPE: The type of kappa model in this module. Valid values are 'CompetitiveModel', 'CooperativityModel', 'CoordinateRegulatoryRegionModel', 'RegulatoryRegionModel', and 'SequenceFeatureRegulatoryRegionModel'

Regulatory Region Kappa Modules

4. `class RegulatoryRegionModel`
   a. `__init__(self, probability_matrix[, other_param1[, other_param2…]])`
   b. `GetRegulatoryRegionRanges(self, feature)`
   c. `GetSiteKappa(self, feature, site_index, site_strand)`

RegulatoryRegionModel

A regulatory region kappa module must supply GOMER with a `RegulatoryRegionModel` class. This `RegulatoryRegionModel` must have the following methods.

__init__(*probability_matrix [, other_param1[, other_param2…]]*)

The __init__ method must accept a *probability_matrix* instance (in most cases, the kappa module will need information from this *probability_matrix*). Depending on the needs of the kappa module, the __init__ method may require an unlimited number of additional parameters. These

parameters are supplied to GOMER on the command line as a "parameter string" (described in Regulatory region kappa function on page 14). GOMER parses the parameter string, and initializes the `RegulatoryRegionModel` with these parameters.

GetRegulatoryRegionRanges(*feature*)

In theory, this method is unnecessary, however, it is important in enabling GOMER to run efficiently. Theoretically, all binding sites on the same chromosome as a genomic feature can contribute to the regulation of that feature. In practice, most binding sites at significant distance from a feature have little or no effect on that feature. The regulatory region weights computed by a regulatory region kappa function are calculated with respect to a given feature. For a given feature, any site with a weight of zero does not contribute to the GOMER score for that site. Therefore, GOMER can save time by ignoring all sites with weights of zero. GetRegulatoryRegionRanges returns a list of range 2-tuples (or 2 element lists) of binding site (window) indices. The first element of each tuple is the start index, and the second element is the stop index: `[(start1_index, stop1_index), (start2_index, stop2_index), ...]`. This list of tuples tells GOMER the ranges of binding site windows that will have non-zero weights, and therefore should be considered in calculating the GOMER score.[13] Since the range is defined by a list of tuples, it is possible for the range to consist of multiple, discontinuous regions. It is important to keep in mind that these are indices and not base pair coordinates, therefore, the first possible site index is zero, and for each range tuple the stop index is the index of the last window considered, not the number of the last base to fall within the regulatory region. It should also be noted that the range tuple is inclusive, in other words, the windows at both the start index and the stop index are included in the analysis[14]. In theory, a lazy module implementer could write a GetRegulatoryRegionRanges method that returns a list of a single range tuple consisting of the indices of the first and last binding site windows on the chromosome of the *feature*, however this would result in a very slow execution.

[13] As often happens, theory and practice diverge here. There are many mathematical functions, which are reasonable to use for weighting, that will never produce a weight of zero (e.g logarithmic, Gaussian distribution). In these cases it is up to the module implementer to determine the appropriate ranges for the regulatory region.
[14] This is different from the format for common format for the Python range function, and for slices of lists, where the element at the "stop" index is excluded

GetSiteKappa(*feature, site_index, site_strand***)**

This method is straightforward. Based on the *feature* being examined, the *site_index* (i.e. location) and *site_strand*, GetSiteKappa returns the weight for the site (a float). The large amount of information contained within these three parameters allows for very complicated weighting functions. Because the *feature* instance itself is supplied to this method, it is possible to use any of the information contained within the feature instance. The availability of this information allows for great flexibility in defining the regulatory region. For example regulatory regions can be defined to be upstream, downstream, or within a feature; slightly more complex would be a weight function that applied differently to different feature types. It is important to note that the different modes generally require different Regulatory Region Kappa Modules, because they carry different information. For example, coordinate features do not have a strand (so "upstream of" is meaningless for a coordinate feature), whereas most standard genome annotated features do. Similarly, since sequence features are defined by their sequence, binding sites are only meaningful within a sequence of the feature.

Cooperativity Kappa Modules

The Cooperative Kappa Module defines the cooperative interaction of a secondary transcription factor with the primary transcription factor by generating weights on the secondary factor binding sites. These weights are used in calculating the GOMER score for the primary transcription factor (as modified by the secondary). The higher the weight on a secondary site, the stronger the cooperative interaction of a secondary factor bound there with the primary factor bound at the primary site.

```
class CoopModel
 d. __init__(self, primary_prob_matrix, secondary_prob_matrix[,
    other_param1[, other_param2…]])
   GetCoopRanges(self, primary_site_index, primary_site_strand,
    feature)
   GetSiteKappa(self, primary_site_index, primary_site_strand,
```

```
    secondary_site_index, secondary_site_strand, feature)
GetSlices(self, primary_site_index, primary_site_strand, feature)
```

CoopModel

A cooperativity kappa module must supply GOMER with a `CoopModel` class. This `CoopModel` must have the following methods.

__init__(*primary_prob_matrix, secondary_prob_matrix, [, other_param1[, other_param2…]]*)

The __init__ method is quite similar to the __init__ method of the RegulatoryRegionModel, the only change being that for the CoopModel, the __init__ method must accept a *secondary_prob_matrix* (the matrix for the cooperatively binding transcription factor) in addition to the *primary_prob_matrix*.

GetSlices(*primary_site_index, primary_site_strand, feature*)

This method is the real workhorse of the CoopModel. Functionally it integrates the information returned by GetCoopRanges and GetSiteKappa. In fact, it is possible implement GetSlices as simply a wrapper around GetCoopRanges and GetSiteKappa. However, implementing it in this way would largely lose the efficiency gains that were the motivation for creating the GetSlices method. GetSlices was created to make calculations of cooperative interactions faster, by reducing the number function calls necessary in the calculation.

This method returns a list of 4-tuples. This list of 4-tuples is similar to the list of 2-tuples returned by GetRegulatoryRegionRanges. However in addition to the start and stop indices of each range, this 4-tuple includes lists of weights for secondary sites within this range on the plus and minus strands: `[(start1_index, stop1_index, plus_strand_weights_1, minus_strand_weights_1), (start2_index, stop2_index, plus_strand_weights_2, minus_strand_weights_2), ...]`. I will explain with two examples: the zeroeth element of `plus_strand_weights_1` is the weight for the secondary site at the index `start1_index` on the plus strand; the last element of `minus_strand_weights_1` is the weight for the secondary site at the index `stop1_index` on

the minus strand.

Just as GetCoopRanges returns a list of 2-tuples giving the start and stop indices of regulatory ranges with non-zero values (i.e. there can be more than one regulatory regions), GetSlices returns a list of 4-tuples, the elements of the 4-tuples are:

- start_index: The site index of the start of the regulatory region.
- stop_index + 1: The site index of the stop (last index) of the regulatory region, plus one. The value "stop_index + 1" is returned, instead of stop, because it makes it more convenient to grab the slice of the window scores needed to be scored (since taking a slice of a list my_list[a:b] returns elements a through b-1).
- plus strand weight slice: The plus strand weights corresponding to the windows in the slice windows[start:stop+1]
- minus strand weight slice: The minus strand weights corresponding to the windows in the slice windows[start:stop+1]

Here is an example of how GOMER uses the data returned by GetSlices:

```
coop_slice_tuple = coop_model.GetSlices(i_index, i_strand, feature)
for start_coop_index, stop__coop_index, plus_coop_weights,
   minus_coop_weights in coop_slice_tuple:
      plus_second_Ka_slice =
      plus_second_Kas[start_coop_index:stop_coop_index]
      minus_second_Ka_slice =
      minus_second_Kas[start_coop_index:stop_coop_index]
```

GetSlices has almost made GetCoopRanges and GetSiteKappa obsolete. GOMER itself does not use these methods of CoopModel, however the **kappa_help.py** utility, which is useful in testing kappa modules, uses them because it has not yet been updated to use GetSlices. Another utility **test_kappa_slices.py** can be used to ensure that the return values of GetSlices correspond with the values generated by GetCoopRanges and GetSiteKappa.

GetCoopRanges(*primary_site_index, primary_site_strand, feature*)

This method is roughly equivalent to the GetRegulatoryRegionRanges method of the RegulatoryRegionModel, the difference being that this method must accept the *primary_site_index* and *primary_site_strand* parameters, the position and strand of the primary

binding site under consideration. These values are supplied because in designing the Cooperativity Kappa Module it was assumed that the cooperative weight on a secondary binding site would depend on its location and orientation with respect to the primary binding site under consideration, and it would be not be dependent where the secondary site is located with respect to the feature. There is, however, nothing preventing the cooperative weight from partially or entirely depending on the feature. As with GetRegulatoryRegionRanges, this method returns a list of range 2-tuples (or 2 element lists) of binding site (window) indices.

GetSiteKappa(*primary_site_index, primary_site_strand, secondary_site_index, secondary_site_strand, feature***)**

This method is roughly equivalent to the GetSiteKappa method of the RegulatoryRegionModel, the difference being that this method must accept the *secondary_site_index* and *secondary_site_strand* parameters. As with the eponymous method of RegulatoryRegionModel, this method returns the cooperative weight on the secondary site.

Competition Kappa Modules

The Competition Kappa Module defines the competitive interaction of a secondary transcription factor with the primary transcription factor by generating weights on the secondary factor binding sites. These weights are used in calculating the GOMER score for the primary transcription factor (as modified by the secondary). The higher the weight on a secondary site, the stronger the competitive interaction of a secondary factor bound there with the primary factor bound at the primary site.

5. class CompModel
   e. __init__(self, primary_prob_matrix, secondary_prob_matrix[, other_param1[, other_param2…]])
   f. GetCompetitiveRegionRanges(self, primary_site_index, primary_site_strand, feature)
   g. GetSiteKappa(self, primary_site_index, primary_site_strand, secondary_site_index, secondary_site_strand, feature)

```
h. GetSlices(self, primary_site_index, primary_site_strand, feature)
```

CompModel

A competitive kappa module must supply GOMER with a `CompModel` class. This `CompModel` must have the following methods.

__init__(*primary_prob_matrix, secondary_prob_matrix, [, other_param1[, other_param2…]]***)**

The interface of this method is identical to __init__ method of the CoopModel.

GetSlices(*primary_site_index, primary_site_strand, feature***)**

The interface of this method is identical to GetSlices method of the CoopModel. As with the CoopModel GetSlices, GetSlices has almost made GetCompRanges and GetSiteKappa obsolete, however these methods are used by the **kappa_help.py** utility, and again **test_kappa_slices.py** can be used to ensure that the return values of GetSlices correspond with the values generated by GetCompRanges and GetSiteKappa.

GetCompRanges(*primary_site_index, primary_site_strand, feature***)**

The interface of this method is identical to the GetCoopRanges method of the CoopModel.

GetSiteKappa(*primary_site_index, primary_site_strand, secondary_site_index, secondary_site_strand, feature***)**

The interface of this method is identical to the GetSiteKappa method of the CoopModel.

Kappa Function Testing Tools

kappa_help

This utility is useful both for kappa module implementers and for GOMER users. When run with the name of a file containing a kappa module, it extracts and prints a description of the model within the kappa module, based on the data attributes described in Module Data Attributes. When run with a kappa module filename and a parameter string appropriate for the module, it generates

a kappagraph, a visualization of the weight function provided by the module.[15] This information should make it clear to a GOMER user what a given kappa module does, and how to use it.

test_kappa_slices.py

This utility tests to be sure that, for CoopModel and CompModel modules, the return values of GetSlices correspond with the values generated by GetCoopRanges and GetSiteKappa.

[15] In order to generate kappagraphs, the biggles module must be installed. This module is available from http://biggles.sourceforge.net/.

Genome Weighting

A genome weight applies to all probablility matrices.

GOMER Output

Unknown Feature Names

Unknown Feature Names are features named in the Regulated Feature File that cannot be found (by any alias) in the feature table, therefore these unknown features are ignored.

Output Feature Name

Many features defined in the feature file have more than one name, when there is a “common name” specified, GOMER uses it preferentially when outputting results. Because of this, a different name might be used for a feature in GOMER output than the name supplied in an input file, such as the regulated feature file.

Troubleshooting

No ROCAUC or MNCP scores?

- You didn't specify any features in your regulated feature file.
- None of the features in your regulated feature file match a known feature (e.g. you misspelled them).
- You only specified features as “excluded” of “unregulated” in the regulated feature file.
- You specified features of type X (e.g. ORF) in the feature file, but the feature type(s) scored were of type Y and Z (e.g. tRNA and snRNA)

Caching and Filtering

Caching

When the whole genome is scored using a probability matrix, the scores for each window are by default saved to a binary cache file. This caching has two advantages:

Reduced RAM usage

Using python floats (double precision - 8 bytes) the score using a single probability matrix, for every window on both strands of the yeast genome uses about 184MB. This sort of memory usage would either require massive amounts of RAM, or a lot of time (due to swapping to virtual memory) for multiple matrices (multiple transcription factors), or even for single transcription factors for larger genomes. GOMER uses cache files like virtual memory, but in an application specific manner, significantly reducing the amount of RAM consumed by window scores, without causing a significant speed penalty due to accessing the hard drive.

Faster run times

Since the cache files are persistent, subsequent runs of GOMER can use the window scores generated in previous runs (if the cached data is appropriate - same matrix, same pseudo temperature, equal or less stringent filtering (see discussion of Filtering below), without incurring the cost of recalculating the window (site) scores.

GOMER handles the cache files transparently, checking for the availability of appropriate cache files, using cryptographic hashing,[16] and using them if available, before embarking on calculating, from scratch, K_a scores for all sites in the genome. The only issue a user might need

[16] GOMER could be tricked into using inappropriate cache files (thereby producing spurious results) without too much effort by a malicious user intent on wrecking havoc, but the assumption is that this will not generally be a concern. And any malicious user who committed such misdeeds could just as easily alter the GOMER source code to generate false data.

to consider concerning GOMER's caching mechanism is the use of disk space. The cache files take up about as much disk space as they would take up in RAM, which is roughly equal to: bases in genome $\times$ 2 strands $\times$ 8 bytes/float , so it might be necessary for the user to clean out old cache files from time to time. This is made easier by the fact that the cache files have headers that contain a fair amount of information on the values used to generate the scores in the cache. Additionally, GOMER detects when a cache file has been deleted, and automatically removes it from the cache table, so the user only needs to delete cache files that are no longer needed, GOMER takes care of the rest.

Speed Ups

Fast Cache

The fast cache command line option was designed to speed up GOMER by taking advantage of a ramdisk or fast hard drive, if available. The fast cache command line option is supplied a directory path. When this option is used, during a GOMER run, the cache files is stored in this fast cache directory, to allow for faster access to the cache file. If the fast cache directory is used when a novel cache is generated, the cache file is written to the fast cache directory, and then copied to the standard cache directory. If the fast cache directory is used when a preexisting cache is available, the cache file is copied to the fast cache directory during the run. The cache files are only stored in the fast cache directory during a run, and are automatically removed once the run is completed. Because the fast cache option adds a copying step, it is possible for the use of this option to make a run slower if the fast cache directory is on a slow hard drive.

Filtering

Filtering is an option, which essentially ignores all windows (binding sites) with K_a values below a certain threshold. In practice, filtering resets the value of sites below the threshold to zero. Since, by the definition of occupancy, sites with K_a values of zero don't contribute to the occupancy score, the feature scoring functions (RegulatoryRegionScoreOrfSlice,

CooperativeScoreFeature, and FullModelScoreFeature) detect K_a values of zero, and skip the computation (which would simply result in multiplying a value of 1 to the cumulative product), saving significant computer time. Since most of the sites in the genome receive very low scores, it should be acceptable to reset the site scores that fall below this threshold to zero without having a serious adverse effect on the feature score.

Filtering removes information, but if a reasonable threshold is chosen, it is safe to assume that the information removed is no different from random. This is because for a low scoring site, the difference between the site's PWM determined score and a score of zero, or the difference between two low scoring sites is likely to be experimentally indistinguishable (if not biologically meaningless), and smaller than the error that went into determining the PWM used to generate the site scores to begin with.

The K_a threshold filtering is determined from the filter_cutoff_ratio value supplied by the user: $K_a^{cutoff} = K_a^{max} / \text{cutoff ratio}$. Since values of filter_cutoff_ratio below one result in filtering all possible K_a values falling below the threshold, it is defined that a value of filter_cutoff_ratio less than one turns filtering off. Note that a filter_cutoff_ratio value of one means that all sites with a score not equal to K_a^{max} will be filtered.

Filtering provides two advantages:

Faster running time:

Computations involving sites with K_a values below the threshold can be skipped.

Compressibility of Cache Files:

GOMER offers the option of save cache files in a compressed format to save storage space. Unfiltered cache files are not very compressible, since the scores of windows in the genome appear close to random. Even when filtered at a high ratio, most of the sites in the genome will fall below the threshold, and therefore will be set to zero, the larger the number of sites that fall below the threshold, the less random, and therefore the more compressible the cache file (Table 9).

Table 9: Compressibility of a filtered cache file

For the probability matrix used in this analysis, $K_a^{max} = 2.61 \times 10^{8}$, $K_a^{min} = 3.25 \times 10^{-25}$.

Filter Cutoff Ratio	Original	Compressed	Compress Time (user sec)	Decompress Time (user sec)
0	184	177 MB	43.8	7.2
100000	184	192 KB	5.8	2.0

Refiltering

When filtering is used, and an appropriate cache file exists, GOMER automatically uses refiltering to generate the cache file. Refiltering generates a more rigorously filtered (i.e. containing less information) cache file from a preexisting cache file. Refiltering is implemented in such a way that it is much faster than regenerating a cache file from scratch, so when it can be utilized, it speeds up GOMER runs. In order for a cache file to be refiltered, it must be less rigorously filtered (filtered at a lower filter_cutoff_ratio) than the filter_cutoff_ratio used for the current run.

Checking for Ambiguous Feature Names

Running GOMER with the `-a\--ambiguous FEATURE_FILE` option will provide a list of all the ambiguous names in the feature file, and all the names (including the ambiguous name) associated with each of the features to which the ambiguous name refers. For example, if I have several features I am interested in, “SSM1”, “SFA1”, “HO”, “FUS3” and I would like to determine if any of these names are ambiguous, one would run the following command:

```
> gomer.py -a chromosomal_feature.tab > ambiguous_names
```

This outputs the results to a file called “ambiguous_names”. One can then “grep” the “ambiguous_names” file with the feature names of interest. Using the “-i” option so that the match is case-insensitive (since the ambiguity test is case-insensitive), and using the `--context 2` option to show me the two lines above and below the match, so I can see all the features to which an ambiguous name refers. Irrelevant lines have removed been removed from the output of **grep** shown below, but it is important that one uses an argument to the `--context` option that is sufficiently large so that I see all the features referred to by an ambiguous name.

```
> grep --context 2 -i ssm1 ambiguous_names
SSM1  7  78855    YGL224C, SDT1, SSM1, S0003192, L0003406
      16 135789   YPL220W, RPL1A, SSM1, S0006141, L0002657
```

“SSM1” is clearly ambiguous. If the feature of interest is the one on chromosome 7, and which starts at 78855bp, one could probably use any of “YGL224C, SDT1, S0003192, L0003406” to refer to it, although it is important to run whichever is chosen against the ambiguous list to confirm that it is unique.

```
> grep --context 2 -i sfa1 ambiguous_names
ADH5 2     533720   YBR145W, ADH5, S0000349, L0000045
     4     159605   YDL168W, SFA1, ADH5, S0002327, L0001868
```

“SFA1” does refer to a feature that has an ambiguous name, but “SFA1” is not itself

ambiguous, so it can be safely used (however one would not want to use the name “ADH5”)

```
> grep --context 1 -i ho ambiguous_names
DSS1  4  1202117 YDR363W-A, SEM1, DSS1, HOD1, S0007235, L0003539,
L0004647
      13 845345  YMR287C, MSU1, DSS1, S0004900, L0001208
```

grep finds a line with “HO”, because it is a substring of “HOD1”, but since the ambiguity match depends on matching the *whole* name, the name “HO” itself is not ambiguous, and is therefore safe to use.[17]

```
> grep --context 1 -i fus3 ambiguous_names
```

Grep finds no matches for “FUS3,” and we are therefore insured that it is an unambiguous name.

[17] “HOD1” is also unambiguous, but its not the gene of interest here.