moca

multivariate organization of combinatorial alterations

0.75
User's manual

Contents

Please contact David Masica (david.masica@gmail.com) with questions, comments, or bugs. Please read the manual in its entirety and try one or two of the Examples before contacting the "help desk" ☺. If you find potential bugs or runtime errors, please cut and paste the python error into an email, and attach your Arguments and Paths files.

`Courier` font denotes a MOCA command, and is defined in the Commands section.

What is MOCA?

In the broadest sense, the Multivariate Organization of Combinatorial Alterations (MOCA) algorithm can find highly correlated features. Typically, a user wishes to find features that are good predictors of a response. At the time of this writing, MOCA has been most extensively employed to find clinical and genomic features that predict phenotype (e.g., disease, drug response, cancer subtype).

MOCA is flexible. MOCA normalizes and binarizes all data, so diverse data types can be combined with little inconvenience to the user. For example, say you have a patient cohort that exhibits variable response in a drug trial. You have also catalogued some clinical and genomic features across the cohort, and want to know which of these features predict response to the drug. MOCA would easily facilitate the inclusion of any data you might have from this trial into a single calculation. Gene mutation and expression could be easily combined with patient age, smoking habits, and occurrence of radiation therapy, even though some of these data types are inherently continuous in nature and others inherently binary.

Installation

The recommended installation follows. This is a simple installation with no automated installer; therefore, advanced users can tailor most steps to their own liking. This assumes a Linux environment, so if you're using a Mac or any flavor of Linux you are ready to proceed. If you are using windows, install a VirtualBox running Linux (I recommend Ubuntu). There are many great tutorials for accomplishing this, simply google "virtualbox ubuntu windows".

Download MOCA (http://karchinlab.org/apps/appMoca.html) to your home directory and unzip it. To unzip do: unzip MOCA.zip -d MOCA.

Add the MOCA directory to your python path. On Linux this will likely be adding "export PYTHONPATH=/home/user/MOCA/:" to your ".bashrc". On a Mac you'll probably be adding "export PYTHONPATH=/Users/user/MOCA/:" to your ".profile". Of course, "user" would be your username on your machine.

Install NumPy (http://numpy.org), RPy2 (http://rpy.sourceforge.net/rpy2.html), and Fisher's exact test for python (https://pypi.python.org/pypi/fisher/). RPy2 requires that you have R installed. Your R installation will need access to certain non-default R libraries; if you are missing any required libraries for your desired MOCA implementation the runtime error will tell which library to install.

MOCA is written for Python2.7. If your machine's default python is not Python2.7, you need to add the correct interpreter to the moca.py file in the MOCA directory. As an example, the first line of your copy of moca.py could be "#!/home/user/Python-2.7/python", if you wanted to point MOCA to a local copy of Python2.7. Once you've done this, all MOCA code in the Source directory will know where to find the correct python installation.

Input Data

Data you provide as input to MOCA must be feature-by-label matrices. Labels could be samples, cell lines, patients, etc. Features will typically be specific gene alterations (e.g., mutation, expression, copy number) to specific genes or transcripts, or they could be clinical observables such as blood pressure or whether or not a patient smoked. The variates that populate the matrix can be either binary or continuous, though MOCA will binarize continuous-valued data. Binarization is accomplished by converting every feature vector to a vector of feature-specific Z-scores, and applying a Z-score threshold (set by the `ZMin` command) to convert every variate to a "1" (label is positive for feature) or "0" (label is negative for feature). MOCA currently recognizes missing values designated by "NA"; no other designation is currently recognized by MOCA. See the data matrices in the MOCA/Database/Examples for some examples of correctly formatted data for MOCA input.

Here is the correct matrix format, for a matrix comprising different alteration types for four genes across five samples.

```
Sample1 Sample2 Sample3 Sample4 Sample5
TP53:Mutation 0 0 1 0 1
KRAS:Mutation 1 0 1 0 0
EGFR:Expression 2.3 9.6 1.8 1.7 0.9
MYC:Methylation 0.8 1.9 0.8 0.7 2.1
```

Notice, there five columns in the header, one for each sample, and six columns in every other row: one for the feature name and the corresponding variates across the five samples. This is the only correct format for a matrix fed into MOCA. Below is a common matrix format for genomics data that will cause problems with MOCA.

```
HugoSymbol:Alteration Sample1 Sample2 Sample3 Sample4 Sample5
TP53:Mutation 0 0 1 0 1
KRAS:Mutation 1 0 1 0 0
EGFR:Expression 2.3 9.6 1.8 1.7 0.9
MYC:Methylation 0.8 1.9 0.8 0.7 2.1
```

This matrix is identical to the last one, except that the header contains a feature descriptor before the list of labels.

Running MOCA

Once your data is formatted correctly, you just need to tell MOCA you want to use it, where it is, select your MOCA mode (what type of MOCA calculation you want to do) and hit "go"! Below I outline the recommended method for running a MOCA calculation, but those familiar with Linux will quickly recognize that this customizable. If python knows where MOCA/Source is, you have a copy of the MOCA executable (moca.py), and MOCA knows where the data is via the Paths file, then you are ready to run MOCA.

Recommended method: All examples in the Examples section assume you are using this approach.

1. The MOCA directory should be in your home directory, and python needs to know where it is (see Installation). This step is quite arbitrary and can be tailored to your liking with no impact on MOCA's ease of use; however, the Examples will assume this is where MOCA is.
2. Keep your MOCA input data in the MOCA/Databases folder. This way you only need one copy of any data file, and it can remain stationary in the Databases folder (the Paths file will tell MOCA where it is).
3. Run your calculations from the MOCA/Projects folder, making a new folder in MOCA/Projects for each project.
4. Use the Arguments file, rather than the command line, to execute MOCA; this way, when you come back to a project at a later date, you will have detailed record of how you arrived at any result. If you really want to be organized, and your going to be running multiple types of MOCA calculations for a single project, use the -a (or --argumentsfile) method to provide MOCA with a descriptive Arguments filename (rather than using the default filename). For instance, you could have three Arguments files in a project named Arguments.Calc1, Arguments.Calc2, and Arguments.Calc3. Then, when you come

back to this projects months later you will have a record of all the calculations you did and their order.

If you follow 2, 3 and 4, and use the MOCA paths file, you will easily be able to reproduce any calculation at a later date and make slight modifications if needed.

5. Leave the moca.py executable in the MOCA directory. When you start a new project, create a symbolic link to that executable (ln -s ~/MOCA/moca.py moca) rather than copying that executable to each project directory. That way, if you update your MOCA installation at a later date you can rerun any projects you want without having to remember to copy the new executable to all the old projects, because the symbolic links will point to the new executable.

Running on multiple processors (`MultiProcessMode`):

For large calculations, the power user can run MOCA in parallel across multiple processors (cores, nodes, cpu's, etc.). This is achieved using the `MultiProcessMode` command and giving each processor its own Arguments file. The `MultiProcessMode` command takes two arguments: an integer telling MOCA which processor it is being executed on and an integer telling MOCA the total number of processors used for the calculation. The default is `MultiProcessMode = 0 1`, meaning MOCA is running on the zeroth processor and that is the only processor in use. Multi-process mode works by taking a list of items to compute and uses python's built-in extended-slicing functionality.
Here is a toy example, from the python terminal, illustrating how MOCA would farm out a computation based on ten Features:

```
>>> Features = ["F0", "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9"]
>>> TotalNodes = 3
>>> for Node in range(TotalNodes):
...     print Node, Features[Node::TotalNodes]
...
0 ['F0', 'F3', 'F6', 'F9']
1 ['F1', 'F4', 'F7']
2 ['F2', 'F5', 'F8']
```

In this example, Node 0 is taking care of Features F0, F3, F6, and F9. Node 1 is taking care of Features F1, F4, and F7…etc. Ideally, the user will write a bash script (or something similar) to utilize a cluster resource manager. In the MOCA folder, there is an example of such a script, written to utilize the Torque queuing system. To use this script, you'd do: ./MOCA.MultiProcessorMode integer string, where the integer is the number of processors you want to use and string is the name extension of the directories you want the results from each node to be saved.

Customizing MOCA (`MyMOCA`):

Advanced users and python coders can easily create and extend MOCA functionality. This will typically amount to writing one or more functions in the MyMoca.py script, and these functions will call existing MOCA functions that you will be able to tweak and combine in you own way. See the directions in MyMOCA.py to get started.

Examples

The following examples all assume you are running MOCA using the recommended method, as described in Running MOCA. Each example has its own directory, which can be found in the MOCA/Projects/Examples directory. These are simple, trivial examples that are meant to familiarize you with MOCA and only take a few minutes each. The data that each example uses was downloaded from the Cancer Cell Line Encyclopedia (http://www.broadinstitute.org/ccle/home) and formatted as described in Input Data.

For all examples you will begin by going into MOCA/Projects/Examples/$Dir, where $Dir is the directory named for the specific example. From inside the example directory, create a symbolic link to the MOCA executable (ln -s ~/MOCA/moca.py moca). Run MOCA using "python moca" (if your system python is not 2.7, add the interpreter as explained in Installation and run using "./moca").

In each example, open the provided Arguments and Paths file, and familiarize yourself with some commands. Probably the most important and common similarity among all three Examples (and every calculation you can do with MOCA) is `Data = $datatype` in the Arguments file, and the corresponding tags in the Paths file (i.e., `$datatype = PathToData`). As an example, your Arguments file could have the command:

```
Data = Mutation Expression
```

Then your Paths needs to have:

```
Mutation = SomePath/SomeFile
Expression = SomePath/SomeOtherFile
```

Where SomeFile is the mutation matrix and SomeOtherFile would be the expression matrix. Anything else in your Paths file will be ignored, but those two tags need to be there.

Example 1: Find genes whose mutation status predicts drug response by running `Pairwise` MOCA.

Approximately one minute after executing MOCA you will get a file called DrugMutation.pairs. Each line of this file shows a significant pairwise interaction and the corresponding FDR-corrected P-value. For instance, the first line in that file is "AZD6244.MEK:DrugPositive NRAS:Mutation 1.140e-02", meaning that NRAS mutation is significantly correlated with sensitivity to the MEK inhibitor AZD6244, having an FDR-corrected P-Value of 0.01.

The provided Arguments file has three commands: `Data`, `Pairwise`, and `Phenotype`. As previously described, the `Data` command is telling MOCA which data to load, via the Paths File. Here, `Pairwise` is set to `True`, because we are running pairwise MOCA. The `Phenotype` command is used to tell MOCA which data type contains the response vectors (i.e., the thing we are trying to predict). In this example, we have specifically told MOCA we are trying to select genes whose mutation status predicts drug sensitivity. If we run MOCA without the `Phenotype` command, every possible pairwise interaction will be tested, including drug-drug, drug-mutation, and mutation-mutation interactions. This approach is used if you want to find the full network of correlated interactions, rather than biomarkers of a phenotype.

Next, we are going to turn the .pairs file into a more information-rich format, which can also be loaded directly into Cytoscape to get a network view. First, either set `Pairwise` to `False`, or delete that line from the Arguments file entirely. Next, add the line `PostProcess = MakeFilteredNetwork`. Now, run MOCA.

After less than a minute you should get a file called DrugMutation.sif. Open this file and look for our NRAS biomarker from before. You'll know notice the interaction type of co-occurring is now provided (all significant interactions must be either mutually exclusive or co-occurring). The original, uncorrected Fisher's exact P-value of 1.52 X 10^{-5} is also now included. And the last two rows are the statistical sensitivity and specificity of the biomarker, with respect to predicting the phenotype.

Example 2: Find multi-feature Boolean set networks (`Setworks`) of gene expression that predict response to the EGFR inhibitor erlotinib.

A few seconds after running MOCA you get the .setworks file. The first line should read:

```
(('EGFR:ExpressionPositive', 'ERBB2:ExpressionPositive'), (), ('ERBB3:ExpressionNegative',))
8.83e-11 1.42e-08 0.8 0.75 Co-occurring
```

Where the genetic features in the first parentheses are combined via the union Boolean set operation, the second parentheses is the intersection, and the third is the difference. So, this marker says that cell lines with either EGFR overexpression or ERBB2 overexpression are sensitive to erlotinib, provided ERBB3 is not under expressed. This marker is very significantly correlated with erlotinib sensitivity, having a P-value of 8.8×10^{-11} (1.14×10^{-8} corrected), and a statistical sensitivity and specificity of 0.8 and 0.75 respectively.

Now, look in the Arguments file. You can see this Arguments file specifies the use of the Expression and Drug data. Unlike the last example, however, you'll notice the Paths file points specifically to the drug erlotinib, rather that the whole drug matrix. If we had just left the Paths file pointing to the full drug matrix, then setworks would have been selected for each drug, which would have taken just a little longer. `Setworks` is set to `True`, because that is the mode we are running. `Phenotype` is still telling MOCA that we are selecting markers for whatever the Paths file delivers for Drug. We've overridden the MOCA defaults for `Optimization` and `BooleanSets` (see Commands) to make the calculation faster. The `Seed` is set, because selecting features to build the setworks has a random component, and we want the results to reproducible for this tutorial. And finally, we are telling MOCA that we don't want any biomarkers that have a statistical sensitivity and specificity of less than 0.7 for predicting response to erlotinib.

Example 3: Select and validate markers of erlotinib response using a `LeaveSomeOut` cross-validation approach.

In less than a minute after running this example you will get 10 files called DrugExpresssion.Validation.$, where $ is a number 0-9. Each of these files resulted from a unique leave-some-out cross-validation, and each is a python pickle, so we have to do a little more to make them human readable.

Looking at the Arguments file, you'll notice the same Arguments as before, except that we replaced `Setworks` with `LeaveSomeOut`. You can see we did a Leave-42-out cross-validation. There are 416 labels (cell lines) in the example input data, so doing a leave-42-out cross-validation is the same as a 10-fold cross validation.

Next, we are going to do biomarker selection. This is a strict process of only returning biomarkers that were selected regardless of data split (i.e., they appear in all 10 cross-validations). Any biomarker that fits this criterion will be returned with its corresponding overall statistical sensitivity and specificity. To execute this MOCA calculation simply set `LeaveSomeOut` to `False` (or delete the line entirely), add the line `PostProcess = ValidateBiomarkers`, and run MOCA. In about one second MOCA will write the file DrugExpression.Biomarkers. Inside this file there is a single marker, the marker from the previous example. Thus, this biomarker was the only marker that held up to the rigorous 10-fold cross-validation; the marker sensitivity and specificity of 0.8 and 0.75 is the same as it was during the standard biomarker selection as in the `Setworks` example.

Finally, we are going to assess predictive performance of all of the markers in the validation files together, by letting each of those markers vote on whether a cell line is sensitive to erlotinib or not. To do this, set `PostProcess` to `ValidateVote`, and run MOCA. This time, you get the result printed to screen, rather then a file. The result is simply the sensitivity and specificity. As you can see, when put to a vote, the markers selected from the 10-fold cross-validation do not perform as well as the previous marker (sensitivity = 0.68, specificity = 0.76). This will not always be the case, and sometimes the collective vote of the markers will be a better predictor than any single marker that was selected from all testing (hold-out) data.

Commands

Use any of the following commands to override the MOCA defaults. From the Arguments file use the "`Command =`" notation. Use the command syntax shown in parentheses to change MOCA defaults from the command line (short- or long-form). Note: there is no `Arguments` command for the Argument file (chicken-and-egg problem; see below).

`(—a; --argumentfile)`: If you are using an Arguments file, you specify the filename with this command. It is not necessary to use the command if you are using an Arguments file with default path/filename (./Arguments). This is the only command that should never be in the Arguments file, as it could serve no purpose.

- Options: Any user-defined string.
- Default: `False`, but checks for the default path/file (./Arguments)

`PathFile` `(-p; --pathfile)`: A Paths file must be used so that MOCA knows where to find the `Data`; use this command to tell MOCA the name of that file.

- Options: Any user-defined name.

- Default: ./Paths

Data `(-d; --data)`: What is the name(s) of the data being supplied to MOCA.

- Options: Any string(s)
- Default: `None`

Phenotype `(-k; --phenotype)`: Tell MOCA which of your Data types, if any, is the phenotype for which you wish to select predictive biomarkers. For `Setworks`, you must set this. For `Pairwise` calculations you want to set this if you are selecting predictive biomarkers, rather than making a network of all correlated interactions (see Example 1, for instance).

- Options: `True/False`
- Default: `False`

Pairwise `(-x; --pairwise)`: Run MOCA's pairwise mode (see Example 1).

- Options: `True/False`
- Default: `False`

Setworks `(-s; --setworks)`: Run MOCA's multi-feature Boolean set network mode (see Example 2).

- Options: `True/False`
- Default: `False`

Optimization `(-o; --optimization)`: The number of trials (or cycles) during which MOCA will randomly select features to combine using Boolean set operations; The frequency with which to repopulate the feature pool with top-predicting features; The percentage of top-predicting features to put in the feature pool. For the MOCA defaults (below), MOCA would make 1000 random feature selections, then every 100 times individual features from the most predictive 1% of setworks would repopulate their respective feature pools (union features go to the union pool, intersection to the intersection pool, etc.). The number of possible combinations that can be made from even 10 features is very large, so 1% of 100 random samples can be large. By default, "top predicting" is defined as the balanced accuracy (arithmetic mean of specificity and sensitivity). The defaults are only sufficient when the number of features is small.

- Options: `Two integers and a float, separated by`
- Default: `1000 100 0.01`

BooleanSets `(-e; --booleansets)`: Max number of features to be combined using the union, intersection, and difference Boolean set operations. Every possible combination will be tested. So if you specify 3 2 1, the total setworks per trial (cycle) would be 62. For 3 3 3 it would be 504. And for 5 4 3 the total number of setorks sampled per trial would be 4088. Multiply this by the number of trials specified by the `Optimizations` command and you can see the number of calculations can explode quickly!

- Options: Three integers, separated by commas and/or spaces.
- Default: `0 0 0`

`LeaveSomeOut` `(-l; --leavesomeout)`: Performs leave-some-out cross-validation; you specify the integer number to leave out. For the common leave-one-out cross-validation, you'd just specify "1". If you want to do, say, 10-fold cross-validation, divide the total number of labels (e.g., samples, cell lines, tumors, patients, etc.) by 10 and round up to the nearest integer. So if you have 416 labels, you'd do a leave-42-out cross-validation. Beholden to your label imbalance (number of cases vs. number of controls), MOCA will try to keep the same number of cases and controls in the testing (hold-out) group. And of course, each label appears in the testing group only a single time.

- Options: Any integer.
- Default: `False`

`PostProcess` `(-n; --postprocess)`: A few different methods for analyzing MOCA output.

- Options: `MakeFilteredNetwork`, `ValidateBiomarkers`, `ValidateVote`, `MakePriors`
  - `MakeFilteredNetwork`: Method for post-processing output from MOCA's `Pairwise` mode (see Example 1).
  - `ValidateBiomarkers`: Method for post-processing output from MOCA's `LeaveSomeOut` mode (see Example 3).
  - `ValidateVote`: Method for post-processing output from MOCA's `LeaveSomeOut` mode (see Example 3).
  - `MakePriors`: If using priors, you have the option of extracting those priors from previous MOCA `Setworks` or `ValidateBiomarkers` runs. This is useful if you really want to focus the search around some features previously determined to be good markers. When running this command, make sure your Paths file's "Priors" tag points to either the .setworks or .Biomarkers file from which you wish to extract the priors for a subsequent `Setworks` or `LeaveSomeOut` run. See `Priors` for more details.
- Default: `False`

`ZMin` `(-z; --zmin)`: Specifies the Z-score threshold used for converting continuous-valued input data to its binary MOCA representation.

- Options: Any floating-point number.
- Default: `0.8`

`ZData` `(-i; --zdata)`: Tell MOCA if the data your giving it has already been normalized in anyway. MOCA will still threshold the data in order to binarize it, but it will skip the initial step of converting all data to row-specific Z-scores.

- Options: `True/False`
- Default: `False`

`MakeBinaryMatrix` `(-b; --binarymatrix)`: Bypasses all other MOCA modes and simply converts your continuous-valued input data into binary output. This is a huge timesaver when dealing with big data matrices. For instance, if you had a large data matrix, say expression microarray data with 15,000 genes, and you planned on using it in more than one MOCA calculation. Then, you might want to just binarize it once so that subsequent MOCA calculations did not have to reprocess this lengthy binarization calculation.

- Options: `True/False`
- Default: `False`

`TissueSpecific` `(-t; --tissuespecific)`: You might be using data from a pan-cancer study, but wish to analyze one or more of the tissues individually. This requires that the sample headers in the input matrices have the Tissue:Sample naming convention (see for instance, the data provided in the Database/Examples folder that came with the MOCA installation).

- Options: Any string indicating the tissue name.
- Default: `False`

`FileName` `(-j; --filename)`: Specify the prefix name (i.e., anything before the ".") for some of the MOCA input or output.

- Options: Any string indicating the name you want to use for input/output files.
- Default: Joins all data names specified in the `Data` command (e.g., `Data` = Expression Mutation Drug; ExpressionMutationDrug.out)

`Seed` `(-c; --constantseed)`: For any calculation that is too large to be exhaustive, there will be some random sampling of features. Use this command to control these stochastic elements of the calculation and make your calculation reproducible. Note: this might only guarantee reproducibility on a single machine or operating system, but will probably be consistent across operating systems.

- Options: Any integer.
- Default: `False`

`MultiProcessMode` `(-u; --multiprocessmode)`: Use this command to parallelize calculations across multiple processors. This is useful for large calculations.

- Options: Two integers. The first specifies the node that that particular job is running on and the second specifies the total number of nodes to be used. See Running MOCA for more details.
- Default: `False`

`MinimumPerformance` `(-r; --minimumperformance)`: Sets the minimum predictive performance a marker can have to be returned to the user or for post processing. Useful to make output less verbose or use only the most predictive markers when post-processing cross-validation output. See Example 2 and 3.

- Options: `Sensitivity (or sens), specificity (or spec), PPV (`positive predictive value`), or NPV (`negative predictive value`).`
- Default: `False`

`FeatureMin` `(-f; --featuremin)`: Set the minimum number of positive elements ("1"s) a binary feature vector must have to be considered in the MOCA calculation. Particularly useful as a timesaver if you have many vectors too sparse to participate in a significant interaction (e.g., your matrix has 100 labels and some vectors have 2 or fewer positive elements); this threshold can be set to eliminate those vectors from the MOCA calculation *a priori*.

- Options: Any integer.
- Default: `3`

`FDR` `(-q; --fdr)`: Set the False-discovery-rate threshold for accepting interactions as significant. Choice of correction method set using the `CorrectionMethod` command.

- Options: Any floating-point number.
- Default: `0.05`

`CorrectionMethod` `(-v; --correctionmethod)`: Select the method to correct the Fisher's exact P-value and estimate the false-discovery rate (`FDR`).

- Options: `BH` (Benjammini & Hochberg), `BY` (Benjammini & Yekutieli), `bonferroni`, `holm`, `hochberg`, `hommel`, `None` (bypass method).
- Default: `BH`

`MyMOCA` `(-m; --mymoca)`: For the experienced user/Python programmer that wishes to create and expand upon existing MOCA functionality. See Running MOCA for more detail.

- Options: Any user-defined function.
- Default: `False`

`Continuous` `(-g; --continuous)`: Forces MOCA to skip binarization step and leave data in original format, which may include continuous-valued data. Currently, this option will not work with any mode except for MyMOCA. Therefore, its only use is for those wishing to use the raw input data in their own custom functions.

- Options: `True/False`
- Default: `False`

`Priors` `(-w; --priors)`: Specify any priors (features), to include for each Boolean set operation, when making setworks. If `Priors` is turned on, MOCA looks for a "Priors" tag in the Paths file, so that MOCA know the path/name to the priors file. Example from the Paths file:

```
Priors  = ${HOME}/MOCA/Projects/ThisProject/ThesePriors.txt
```

Then in "ThesePriors.txt", you'd have something like:

```
Union = Feature1:ExpressionPositive Feature20:Mutation
Difference = Feature13:MethylationPositive
```

- Options: `Strict, weighted, or stochastic`
  - `Strict`: You specify exactly which features to add for each Boolean set operation. For the above example, every setwork would include Feature1 and Feature20 in the union and Feature13 in the difference, in addition to any other features selected.
  - `Weighted`: If you use `PostProcess = MakePriors,` to make the Priors file from a previous `Setworks` or `ValidateBiomarkers` run, you will likely have many copies of some features. Using this command will add those features, to their respective Boolean set operations, in a fashion that is biased by the relative differences in the Priors file. Here, priors are added at random, but will be biased (weighted) by their relative counts. So, if you have 50 copies of one Feature1 and 10 copies of Feature2, there is a five-fold likelihood that Feature1 will be randomly selected during any cycle of `Setworks`, relative to Feature2. `Weighted` and `stochastic` options add anywhere between one and 10 of you priors, to setwork; that number is chosen at random during each cycle, but is biased towards lower numbers.
  - `Stochastic`: Like the `weighted` command, except that prior selection is truly random because the list of priors is uniquified before selection. So in the "weighted" example, Feature1 and Feature2 would have an equal probability of getting selected each time. `Weighted` and `stochastic` options add anywhere between one and 10 of you priors, to setwork; that number is chosen at random during each cycle, but is biased towards lower numbers.
- Default: `False`

References

Masica, David L., and Rachel Karchin. "Correlation of somatic mutation and expression identifies genes important in human glioblastoma progression and survival." *Cancer research* 71.13 (2011): 4550-4561.

Masica, David L., and Rachel Karchin. "Collections of simultaneously altered genes as biomarkers of cancer cell drug response." *Cancer research* 73.6 (2013): 1699-1708.