medusa Documentation

Release 0.2.0

Gregory Medlock

Jul 28, 2020

Getting Started

Medusa is a tool for constraint-based reconstruction and analysis (COBRA) of ensembles. It builds on the cobrapy package (https://github.com/opencobra/cobrapy) by extending most single-model functionality to efficient ensemble-scale analysis. Additionally, Medusa provides novel functions for the analysis of ensembles.

For installation instructions and how to cite Medusa, please see README.rst.

CHAPTER 1

Index

Getting Started

1.1 Why?

1.1.1 Where do ensembles come from?

In the constraint-based reconstruction and analysis (COBRA) field, there are a variety of analyses and approaches that have alternative optimal solutions. Examples of this come up often when analyses involve integer programming, where variables in a problem are switched on/off to maximize or minimize an objective value. When integer programming is applied to fill in gaps in a genome-scale metabolic model (GEM) to fulfill a function, such as production of biomass, there are generally multiple sets of equally likely solutions to the problem (e.g. there is more than one unique set of reactions of the same size that enable growth).

Beyond examples of analyses where multiple solutions with the *exact same* objective value might exist, keeping track of multiple sub-optimal solutions might be valuable because a biologically-correct solution is almost never the exact same as the most likely numerical solution. Instead, we can hedge our bets by maintaining an ensemble of feasible solutions to a problem that all meet some minimum acceptable likelihood.

The primary aim of medusa is to make ensemble analyses more accessible to the COBRA field so that we can start accounting for this uncertainty to improve our predictions and improve our reconstructions more efficiently. The software is written and developed with usability as the top priority, secondary to performance and novelty. Thus, user feedback is essential–please do not hesitate to provide your thoughts or ask questions via an issue on the medusa github repository.

1.2 Installation

Stable releases of `medusa` are available through PyPI for Python 3 (Python 2 is not supported). To download and install the most recent stable release, install pip and run the following at the commandline:

```
$ pip install medusa-cobra
```

You can also download and install the development version of `medusa` via github. To do this, clone the remote github repository with:

```
$ git clone https://github.com/gregmedlock/Medusa.git
```

When working from a cloned repository, you will need to manually install the dependencies for `medusa`. First install tools necessary for setting up the environment:

```
$ pip install --upgrade pip setuptools wheel
```

Now navigate to the directory generated by cloning (e.g. default will create a folder named "Medusa" in the location you performed `git clone`), then install the package requirements:

```
$ pip install -r requirements.txt
```

then checkout the development branch:

```
$ git checkout development
```

the `checkout` command will make your local repository reflect the exact state of the remote repository's development branch. After checking out the development branch, the last step is to setup `medusa` for development:

```
$ python setup.py develop
```

During usage, issues may arise with optional cobrapy dependencies that are not installed by default with `medusa`. For these kinds of issues, please attempt a full installation of cobrapy and consult the cobrapy installation instructions.

Examples

- *Parallelized simulations*
- *Applying machine learning to guide ensemble curation*
- *Statistical testing for ensemble simulations*
- *Assessing ensemble performance via ROC*

1.3 Parallelized simulations

In `medusa`, ensemble Flux Balance Analysis (FBA) can be sped up thanks to the `multiprocessing` Python module. With this approach, each core (or processor) is assigned a subset of ensemble members for which to perform FBA, speeding up the computation in proportion to the number of additional processors allocated.

Let's load a test model to demonstrate parallel ensemble FBA. This ensemble has 1000 members, so each FBA step will return fluxes for each reaction in one of the 1000 members.

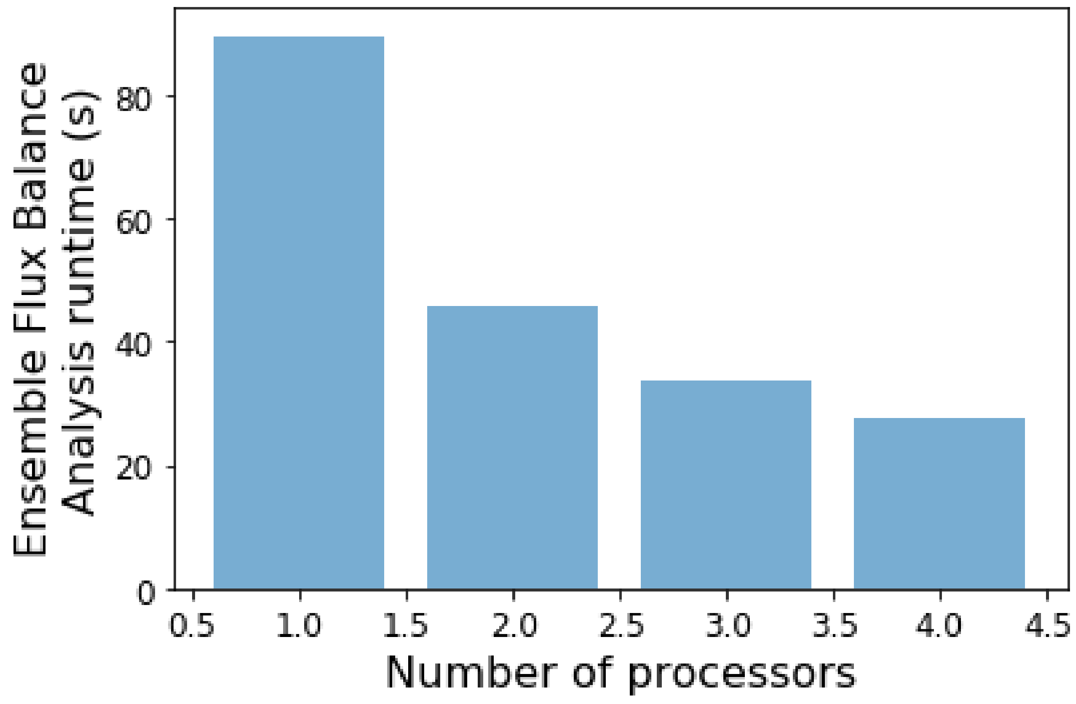
```
In [1]: from medusa.flux_analysis import flux_balance
        from medusa.test import create_test_ensemble
        ensemble = create_test_ensemble("Staphylococcus aureus")
```

Next, perform the actual simulations. To parallelize, just indicate the number of cores you'd like to use with the `num_processes` argument. `medusa` will *not* automatically recognize that you have additional cores available. We'll use the time module to keep track of how long the simulation takes given the number of cores allocated.

```
In [2]: import time

        runtimes = {}
        for num_processes in range(1,5):
            t0 = time.time()
            flux_balance.optimize_ensemble(ensemble, num_processes = num_processes)
            t1 = time.time()
            runtimes[num_processes] = t1-t0
            print(str(num_processes) + ' processors: ' + str(t1-t0) + 's')
```

```
1 processors: 89.6445517539978s
2 processors: 45.74347114562988s
3 processors: 33.75276780128479s
4 processors: 27.72901201248169s
```

```
In [5]: import matplotlib.pylab as plt
        fig,ax = plt.subplots()
        plt.bar(runtimes.keys(), runtimes.values(), align = 'center', alpha = 0.6)
        plt.xlabel('Number of processors')
        plt.ylabel('Runtime (s)')
        ax.set_xlabel('Number of processors',size=16)
        ax.set_ylabel('Ensemble Flux Balance \nAnalysis runtime (s)',size=16)
        ax.tick_params(axis='both', which='major', labelsize=12)
        ax.tick_params(axis='both', which='minor', labelsize=12)
        plt.savefig('parallel_fba.svg')
```



As you can see from the printed output and the plots, a couple of additional cores really speeds things up. However, each core requires an additional copy of the ensemble for its independent simulations. This process (serialization and deserialization) leads to diminishing returns as the number of cores is increased. We are working on improving this, but for now, it is best to choose a modest number of cores (e.g. 2-4).

1.4 Applying machine learning to guide ensemble curation

An ensemble of models can be though of as a set of feasible hypotheses about how a system behaves. From a machine learning perspective, these hypotheses can alternatively be viewed as samples (or observations), each of which has a distinct set of features (i.e. the model components that vary across an ensemble) and can further generate new features by performing simulations. An example of the analyses enabled by this view of ensembles can be found in Medlock & Papin, where ensemble structure and ensemble simulations are used to identify reactions that are high-priority targets for curation.

In this example, we demonstrate how ensembles of genome-scale metabolic models and machine learning can be combined to identify reactions that might strongly influence a single prediction (flux through biomass). We will use an ensemble for *Staphylococcus aureus* that contains 1000 members. The ensemble was generated through iterative gapfilling to enable growth on single C/N media conditions using a draft reconstruction from ModelSEED.

```
In [1]: import medusa
        from medusa.test import create_test_ensemble

        ensemble = create_test_ensemble("Staphylococcus aureus")
```

Using the ensemble, we'll perform flux balance analysis and return flux through the biomass reaction (which has an ID of `"bio1"`). The ensemble already has the media conditions set as "complete", meaning the boundary reactions for all transportable metabolites are open (e.g. the lower bound of all exchange reactions is -1000).

```
In [2]: from medusa.flux_analysis import flux_balance
        biomass_fluxes = flux_balance.optimize_ensemble(ensemble, return_flux="bio1", num_processes =
```

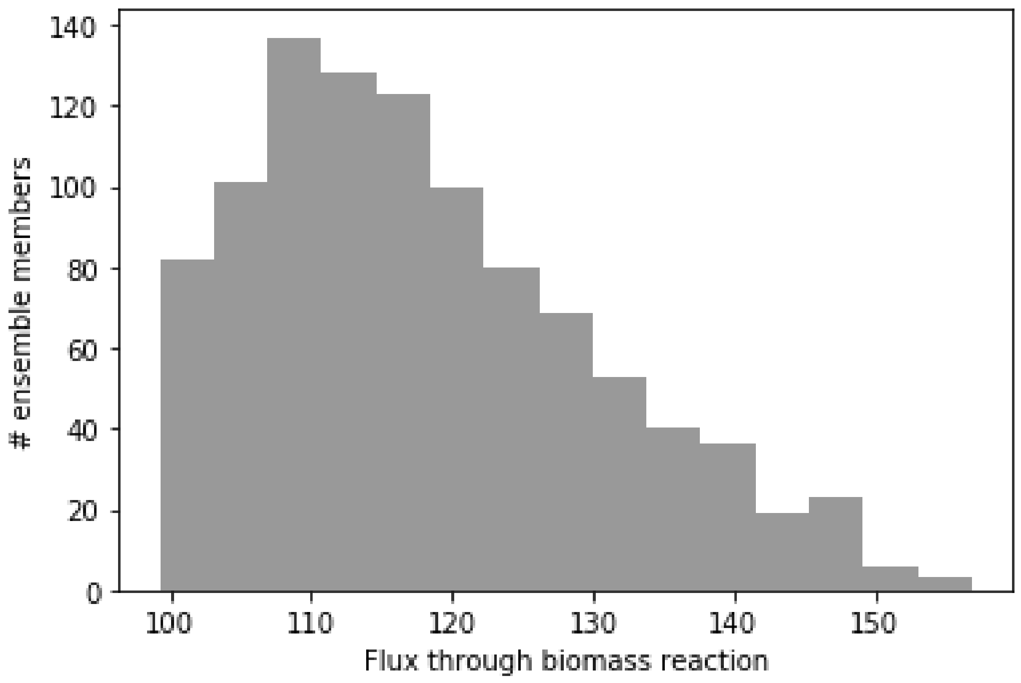
The `optimize_ensemble` function returns a pandas DataFrame, where each column is a reaction and each row is an ensemble member. For illustration, here are the values for the first 10 members of the ensemble:

```
In [3]: biomass_fluxes.head(10)
```

```
Out[3]:                                        bio1
        Staphylococcus aureus_gapfilled_518  118.238182
        Staphylococcus aureus_gapfilled_860  122.523063
        Staphylococcus aureus_gapfilled_900  104.905551
        Staphylococcus aureus_gapfilled_434  148.353976
        Staphylococcus aureus_gapfilled_343  134.100850
        Staphylococcus aureus_gapfilled_706  116.982207
        Staphylococcus aureus_gapfilled_175  137.352545
        Staphylococcus aureus_gapfilled_85   110.488964
        Staphylococcus aureus_gapfilled_345  119.439103
        Staphylococcus aureus_gapfilled_161  118.237318
```

To get a sense for the distribution of biomass flux predictions, we can visualize them with matplotlib:

```
In [4]: import matplotlib.pylab as plt
        fig, ax = plt.subplots()
        plt.hist(biomass_fluxes['bio1'], bins = 15, color = 'black', alpha = 0.4)
        ax.set_ylabel('# ensemble members')
        ax.set_xlabel('Flux through biomass reaction')
        plt.savefig('pre_FBA_curation.svg')
        plt.show()
```



As you can see, there is quite a bit of variation in the maximum flux through biomass! Keep in mind that this is an ensemble of gapfilled reconstructions with no manual curation, and that none of the uptake rates are reallistically constrained, so these predictions are unrealistically high (100 units of flux through biomass is a doubling time of 36 seconds, at least an order of magnitude faster than even the fittest *E. coli* grown *in vitro*).

Our goal now is to identify which features in the ensemble are predictive of flux through biomass. If we can identify these reactions, then turn to the literature or perform an experiment to figure out whether they are really catalyzed by the organism, we can greatly reduce the uncertainty in our predictions of biomass flux!

Given that we have a continous output, our problem can be addressed using regression. We will use the binary presence/absence of each reaction in each member of the ensemble as input to a random forest regressor, implemented in scikit-learn. Many supervised regression models will work for this analysis, but random forest is particularly easy to understand and interpret when the input is binary (i.e. reaction presence/absence).

```
In [5]: import sklearn
        from sklearn.ensemble import RandomForestRegressor
```

We reformat the data here, getting the feature states for each ensemble member and converting them to `True`/`False`, then combine them into a single DataFrame with the biomass flux predictions for matched members.

```
In [6]: # Grab the features and states for the ensemble and convert to a dataframe
        import pandas as pd
        feature_dict = {}
        for feature in ensemble.features:
            feature_dict[feature.id] = feature.states
        feature_frame = pd.DataFrame.from_dict(feature_dict)
        # convert the presence and absence of features to a boolean value
        feature_frame = feature_frame.astype(bool)
        # extract biomass and add it to the dataframe, keeping track of the feature names
        input_cols = feature_frame.columns
        biomass_fluxes.index = [member_id for member_id in biomass_fluxes.index]
        feature_frame['bio1'] = biomass_fluxes['bio1']
```

Now we actually construct and fit the random forest regressor, using 100 total trees in the forest. The `oob_score_` reported here is the coefficient of determination (R2) calculated using the out-of-bag samples for each tree. As a reminder, R2 varies from 0 to 1.0, where 1.0 is a perfect fit.

```
In [7]: # create a regressor to predict biomass flux from reaction presence/absence
        regressor = RandomForestRegressor(n_estimators=1000,oob_score=True)
        fit_regressor = regressor.fit(X=feature_frame[input_cols],y=feature_frame['bio1'])
        fit_regressor.oob_score_
```

```
Out[7]: 0.8684706250117109
```

With a reasonably-performing regressor in hand, we can inspect the important features to identify reactions that contribute to uncertainty in biomass flux predictions.

```
In [8]: imp_frame = pd.DataFrame(fit_regressor.feature_importances_,
                                 index=feature_frame[input_cols].columns).sort_values(
                                 by=0,ascending=False)
        imp_frame.columns = ['importance']
```

```
In [9]: imp_frame.head(10)
```

```
Out[9]:                         importance
        rxn01640_c_upper_bound    0.129785
        rxn01640_c_lower_bound    0.113434
        rxn12585_c_lower_bound    0.044854
        rxn12585_c_upper_bound    0.042830
        rxn15617_c_lower_bound    0.039388
        rxn23244_c_lower_bound    0.039336
        rxn00602_c_lower_bound    0.037124
        rxn15617_c_upper_bound    0.036253
        rxn00602_c_upper_bound    0.032443
        rxn23244_c_upper_bound    0.028802
```

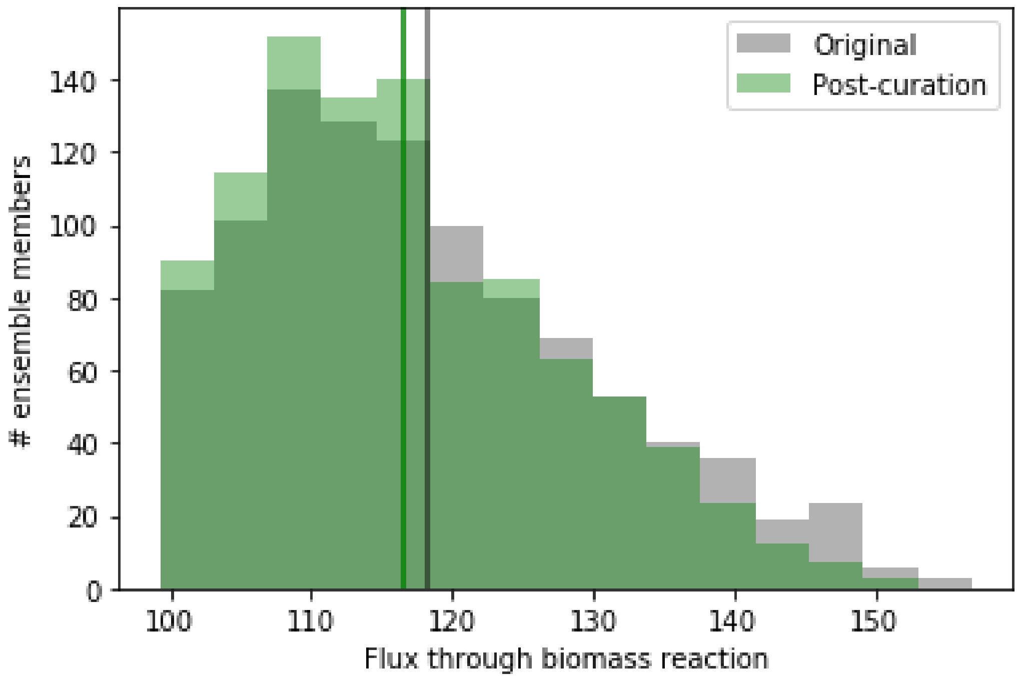
With the list of important features in hand, the first thing we should do is turn to the literature to see if someone else has already figured out whether these reactions are present or absent in *Staphylococcus aureus*. The top reaction, `rxn01640`, is N-Formimino-L-glutamate iminohydrolase, which is part of the histidine utilization pathway. A quick consultation with a review on the regulation of histidine utilization in bacteria suggests that the enzyme for this reaction, encoded by the *hutF* gene, is widespread and conserved amongst bacteria. However, the *hutF* gene is part of a second, less common pathway that branches off of the primary histidine utilization pathway. If we consult PATRIC with a search for the *hutF* gene, we see that, although the gene is widespread, there is no predicted *hutF* gene in any sequenced *Staphylococcus aureus* genome. Although absence of evidence is not evidence of absence, we can be relatively confident that *hutF* is not encoded in the *Staphylococcus aureus* genome, given how well-studied this pathogen is.

What happens if we "correct" this issue in the ensemble? Let's inactivate the lower and upper bound for the reaction in all the members, then perform flux balance analysis again.

```
In [10]: for member in ensemble.features.get_by_id('rxn01640_c_lower_bound').states:
             ensemble.features.get_by_id('rxn01640_c_lower_bound').states[member] = 0
             ensemble.features.get_by_id('rxn01640_c_upper_bound').states[member] = 0

         biomass_fluxes_post_curation = flux_balance.optimize_ensemble(ensemble, return_flux="bio1",
```

```
In [11]: import matplotlib.pylab as plt
         import numpy as np
         fig, ax = plt.subplots()
         # declare specific bins for our histogram
         bins=np.histogram(np.hstack((biomass_fluxes['bio1'],biomass_fluxes_post_curation['bio1'])),
                           bins=15)[1]
         plt.hist(biomass_fluxes['bio1'],
                  bins,
                  label = 'Original',
                  alpha = 0.3,
                  color='black')
```

```
plt.hist(biomass_fluxes_post_curation['bio1'],
         bins,
         label = 'Post-curation',
         alpha = 0.4,
         color = 'green')
plt.axvline(x=biomass_fluxes['bio1'].mean(), c = 'black', alpha = 0.6)
plt.axvline(x=biomass_fluxes_post_curation['bio1'].mean(), c = 'green')
ax.set_ylabel('# ensemble members')
ax.set_xlabel('Flux through biomass reaction')
plt.legend(loc='upper right')
plt.savefig('post_FBA_curation.svg')
plt.savefig('post_FBA_curation.png')
plt.show()
```



```
In [ ]:
```

Here, we show the old distribution in gray and the new distribution in green, with vertical lines at the mean in the same color. As you can see, by resolving the identity of the `hutF`-encoded enzyme, we've reduced the mean and range of predicted flux through biomass. The reduction here is modest, but the process can be repeated for the other important features we identified to continue to refine the distribution and improve the reconstruction in a rational way.

1.5 Statistical testing for ensemble simulations

In traditional COBRA simulations with a single model, most simulations result in a single quantity of interest, thus statistical comparisons usually don't make sense. For example, when simulating growth in two different media conditions, a single model can only output a single predicted growth rate for each condition.

When accounting for uncertainty in model structure using an ensemble, these simulations generate a distribution rather than a single value. Because we are no longer comparing two individual values, proper interpretation requires statistical assessment of the distributions our ensemble simulations generate. In this example, we demonstrate this concept and one statistical option for univariate comparisons (e.g. comparisons between two conditions).

First, let's load an ensemble for *Staphylococcus aureus* and the recipe for biolog growth media, which we'll use to simulate growth in single carbon source media.

```
In [1]: import medusa
        from medusa.test import create_test_ensemble

        ensemble = create_test_ensemble("Staphylococcus aureus")
```

```
In [2]: import pandas as pd
        biolog_base = pd.read_csv("../medusa/test/data/biolog_base_composition.csv", sep=",")
        biolog_base
```

```
Out[2]:          Name          ID
        0         H2O  cpd00001_e
        1          O2  cpd00007_e
        2   Phosphate  cpd00009_e
        3         CO2  cpd00011_e
        4         NH3  cpd00013_e
        5        Mn2+  cpd00030_e
        6        Zn2+  cpd00034_e
        7     Sulfate  cpd00048_e
        8        Cu2+  cpd00058_e
        9        Ca2+  cpd00063_e
        10         H+  cpd00067_e
        11        Cl-  cpd00099_e
        12       Co2+  cpd00149_e
        13         K+  cpd00205_e
        14         Mg  cpd00254_e
        15        Na+  cpd00971_e
        16       Fe2+  cpd10515_e
        17        fe3  cpd10516_e
        18       Heme  cpd00028_e
        19     H2S2O3  cpd00268_e
```

```
In [3]: # convert the biolog base to a dictionary, which we can use to set ensemble.base_model.mediu
        biolog_base = {'EX_'+component:1000 for component in biolog_base['ID']}

        # Double check that the objective is set to the biomass reaction.
        # For this model, 'bio1' is the id of the biomass reaction.
        ensemble.base_model.objective = 'bio1'
```

Let's simulate growth on two different carbon sources, D-glucose (metabolite id: cpd00027) and maltose (metabolite id: cpd00179).

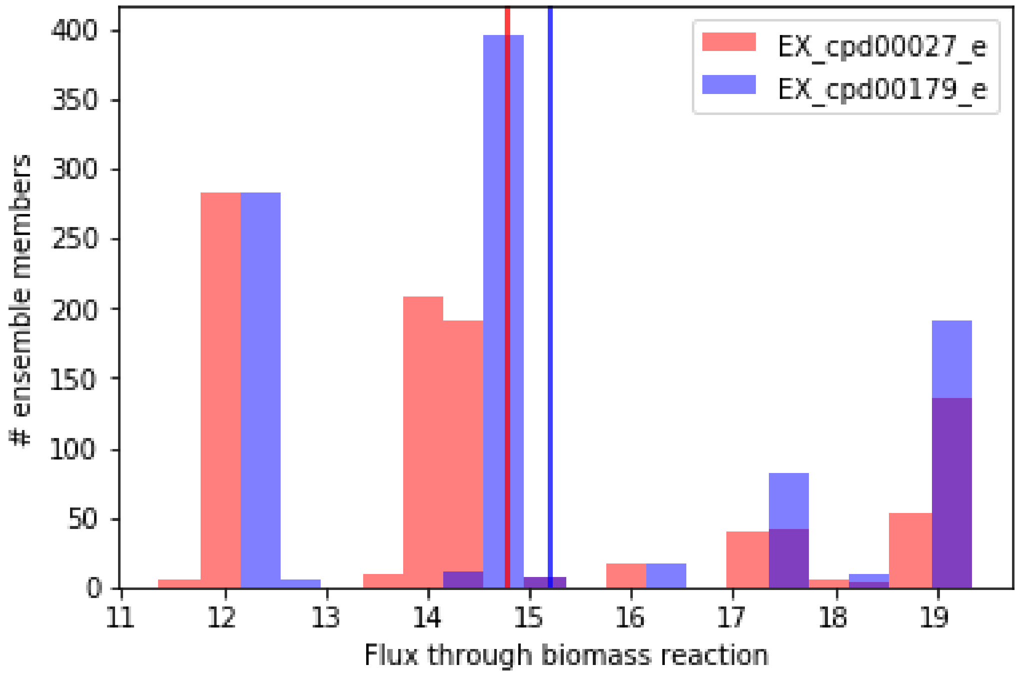
```
In [4]: from medusa.flux_analysis import flux_balance

        carbon_sources = ["EX_cpd00027_e","EX_cpd00179_e"]

        fluxes = {}
        for carbon_source in carbon_sources:
            biolog_base[carbon_source] = 10
            ensemble.base_model.medium = biolog_base
            fluxes[carbon_source] = flux_balance.optimize_ensemble(ensemble,return_flux='bio1', num_p
            biolog_base[carbon_source] = 0
```

Now let's visualize the distributions of predicted flux through biomass using matplotlib. We'll generate a histogram for each condition, and plot the mean using a vertical line:

```
In [5]: import matplotlib.pylab as plt
        import numpy as np
```

```
In [6]: fig, ax = plt.subplots()
```

```
        bins=np.histogram(np.hstack((fluxes[carbon_sources[0]]['bio1'],fluxes[carbon_sources[1]]['bic
        plt.hist(fluxes[carbon_sources[0]]['bio1'],
                 bins = bins,
                 label=carbon_sources[0],
                 color = "red",
                 alpha = 0.5)
        plt.hist(fluxes[carbon_sources[1]]['bio1'],
                 bins = bins,
                 label=carbon_sources[1],
                 color = "blue",
                 alpha = 0.5)
        plt.axvline(x=fluxes[carbon_sources[0]]['bio1'].mean(), c = 'red')
        plt.axvline(x=fluxes[carbon_sources[1]]['bio1'].mean(), c = 'blue')
        ax.set_ylabel('# ensemble members')
        ax.set_xlabel('Flux through biomass reaction')
        ax.legend()
        plt.show()
```



Visually, we can see the mean for D-glucose (cpd00027) is slightly lower than for maltose (cpd00179). To evaluate this statistically, we'll use the Wilcoxon signed-rank test (implemented in SciPy), which tests the null hypothesis that the difference between paired samples (e.g. growth in D-glucose minus growth in maltose for each ensemble member) is symmetrically distributed around zero. Here, we choose a statistical test meant for paired data because each simulation result in one media condition has a related simulation result in the other condition which was generated using the same ensemble member. The Wilcoxon signed-rank test is suitable for paired univariate comparisons regardless of the distribution of data (e.g. when data are non-normally distributed, replace a paired *t*-test with the Wolcoxon signed-rank test).

```
In [7]: from scipy.stats import wilcoxon
        cond1 = fluxes[carbon_sources[0]].copy()
        cond2 = fluxes[carbon_sources[1]].copy()
        cond1.columns = [carbon_sources[0]]
        cond2.columns = [carbon_sources[1]]
        both_conditions = pd.concat([cond1,cond2], axis = 1, join_axes = [cond1.index])
        wilcoxon(x=both_conditions[carbon_sources[0]],y=both_conditions[carbon_sources[1]])
```

```
Out[7]: WilcoxonResult(statistic=0.0, pvalue=3.3257599356529824e-165)
```

The *p* value from the test is well below any reasonable threshold, so we can claim that the predicted flux through biomass with maltose as the sole carbon source is higher than flux through biomass with D-glucose as the sole carbon source.

1.6 Assessing ensemble performance via ROC

This is a placeholder for an example showing how to construct an ROC curve for binary predictions made with an ensemble.

```
In [1]: import medusa
```

User Guide

1.7 Introduction to Medusa

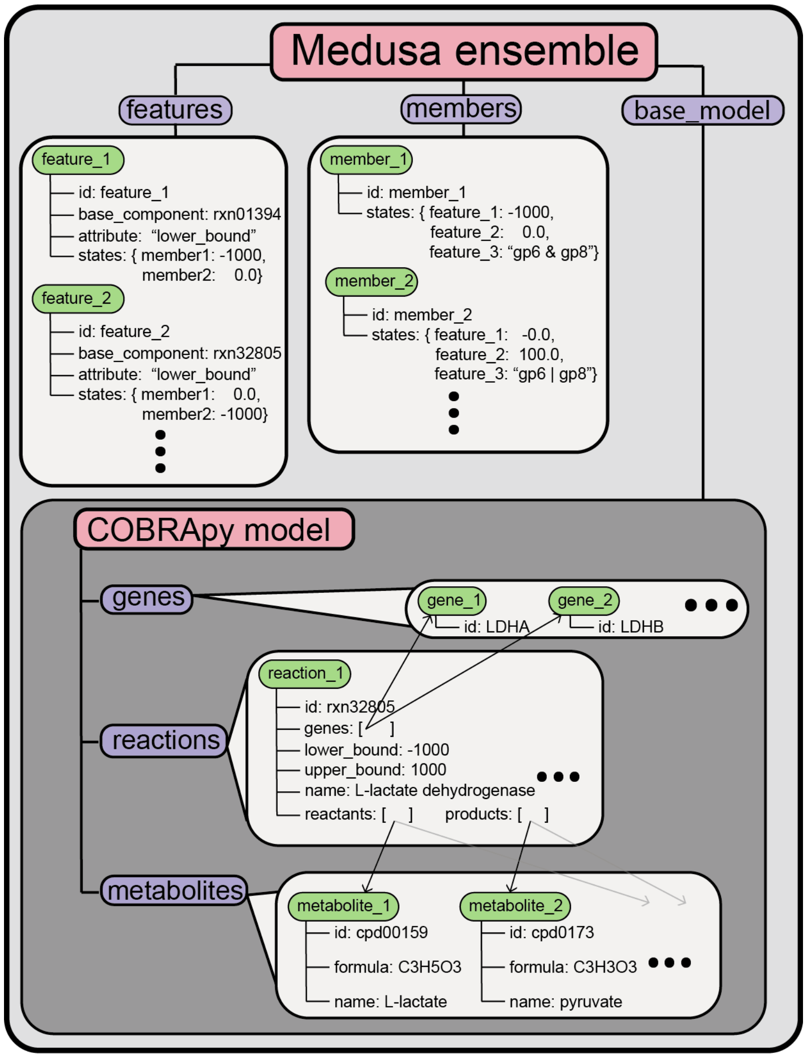
1.7.1 Loading an example ensemble and inspecting its parts

In `medusa`, ensembles of genome-scale metabolic network reconstructions (GENREs) are represented using the `medusa.Ensemble` class. To demonstrate the functionality and attributes of this class, we'll, load a test ensemble. Here, we use a function that takes the *E. coli* core metabolism reconstruction from cobrapy and randomly removes components to generate ensemble members.

```
In [1]: import medusa
        from medusa.test.test_ensemble import construct_textbook_ensemble

        example_ensemble = construct_textbook_ensemble()
```

Each `Ensemble` has three key attributes that specify the structure of the ensemble, which we'll describe below. This schematic also summarizes the structure of `Ensemble` and how each attribute relates to cobrapy objects:

```
In [2]: from IPython.display import Image
        Image(filename='medusa_structure.png', width=500)
```



1.7.2 Components of an ensemble: base_model

The first is the `base_model`, which is a `cobra.Model` object that represents all the possible states of an individual member within the ensemble. Any reaction, metabolite, or gene that is only present in a subset of ensemble members will be present in the `base_model` for an `Ensemble`. You can inspect the `base_model` and manipulate it just like any other `cobra.Model` object.

```
In [3]: extracted_base_model = example_ensemble.base_model
        extracted_base_model
```

```
Out[3]: <Model first_textbook at 0x7efcac754f98>
```

1.7.3 Components of an ensemble: members

The second attribute that each `Ensemble` has is a structure called `members`. `Ensemble.members` maps an identifier for each individual GENRE in the ensemble to a `medusa.Member` object, which holds information about a single member (where a "single member" is an individual GENRE within an ensemble).

`Ensemble.members` is represented by a custom class implemented in cobrapy called a DictList, which is essentially a standard dictionary in python that can also be accessed using integer indices like a list (e.g. dictlist[0] returns the first element in the dictlist).

```
In [4]: # looks like a list when we print it
        example_ensemble.members
```

```
Out[4]: [<Member first_textbook at 0x7efcac983be0>,
         <Member second_textbook at 0x7efcac983ef0>]
```

```
In [5]: # Get the first member with integer indexing
        first_member = example_ensemble.members[0]
```

Each `Member` within the `Ensemble.members` DictList has a handful of attributes as well. You can check the ensemble that the member belongs to, the id of the member, and the network states for that member (we'll discuss states more below).

```
In [6]: print(first_member.ensemble)
        print(first_member.id)
        print(first_member.states)
```

```
textbook_ensemble
first_textbook
{<Feature ACONTb_lower_bound at 0x7efcac761e10>: -1000.0, <Feature ACKr_lower_bound at 0x7efcac91d128
```

1.7.4 Components of an ensemble: features

The states printed above are directly connected to the third attribute that `Ensemble` contains, `Ensemble.features`, which is also a DictList object. `Ensemble.features` contains `medusa.Feature` entries, which specify the components of the `Ensemble.base_model` that vary across the entire ensemble.

```
In [7]: example_ensemble.features
```

```
Out[7]: [<Feature ACALDt_lower_bound at 0x7efcac91d390>,
         <Feature ACALDt_upper_bound at 0x7efcac91d198>,
         <Feature ACKr_lower_bound at 0x7efcac91d128>,
         <Feature ACKr_upper_bound at 0x7efcac91d668>,
         <Feature ACONTb_lower_bound at 0x7efcac761e10>,
         <Feature ACONTb_upper_bound at 0x7efcac91def0>,
         <Feature ACt2r_lower_bound at 0x7efcac91dd68>,
         <Feature ACt2r_upper_bound at 0x7efcac91db38>]
```

Here, we see that this `Ensemble` has 8 features. Each `Feature` object specifies a network component that has a variable parameter value in at least one member of the ensemble (e.g. at least one ensemble member is missing the reaction).

In this case, there are features for 4 reactions, `ACALDt`,`ACKr`,`ACONTb`, and `ACt2r`. There are two `Feature` objects for each reaction, corresponding to the lower and upper bound for that reaction. A feature will be generated for any component of a `cobra.Model` (e.g. `Reaction`, `Gene`) that has an attribute value (e.g. `Reaction.lower_bound`, `Reaction.gene_reaction_rule`) that varies across the ensemble. As you can see from this result, a feature is created at the level of the specific attribute that varies, *not* the model component (e.g. we created a `Feature` for each *bound* of each `Reaction`, not for the `Reaction` objects themselves).

This information can be inferred from feature ID (`medusa.Feature.id`), but each `Feature` also has a set of attributes that encode the information. Some useful attributes, described in the order printed below: getting the `Ensemble` that the `Feature` belongs to, the component in the `Ensemble.base_model` that the `Feature` describes, the attribute of the component in the `Ensemble.base_model` whose value the `Feature` specifies, and the ID of the `Feature`:

```
In [8]: first_feature = example_ensemble.features[0]
        print(first_feature.ensemble)
        print(first_feature.base_component)
        print(first_feature.component_attribute)
        print(first_feature.id)
```

```
textbook_ensemble
ACALDt: acald_e <=> acald_c
lower_bound
ACALDt_lower_bound
```

Just as each `member` has an attribute, `states`, that returns the value of every feature for that `member`, each `feature` has a `states` dictionary that maps each `member.id` to the value of the `feature` in the corresponding `member`, e.g.:

```
In [9]: print(first_feature.states)
```

```
{'second_textbook': -1000.0, 'first_textbook': 0.0}
```

1.7.5 Strategies for getting information about an ensemble and its members

Where possible, we use conventions from cobrapy for accessing information about attributes. In cobrapy, the `Model` object has multiple containers in the form of DictLists: `Model.reactions`,`Model.metabolites`,`Model.genes`. Equivalently in medusa, each `Ensemble` has similarly constructed containers: `Ensemble.members` and `Ensemble.features`.

As such, information about specific `Member` and `Feature` objects can be accessed just like `Reaction`, `Metabolite`, and `Gene` objects in cobrapy:

```
In [10]: # Remember, our Ensemble holds a normal cobrapy Model in base_model
         extracted_base_model = example_ensemble.base_model
         # Accessing object by id is common in cobrapy
         rxn = extracted_base_model.reactions.get_by_id('ACALDt')
         # We can do the same thing for features:
         feat = example_ensemble.features.get_by_id('ACALDt_lower_bound')
         print(rxn)
         print(feat.base_component)
         print(feat.component_attribute)

         # And for members:
         memb = example_ensemble.members.get_by_id('first_textbook')
         print('\nHere are the states for this member:')
         print(memb.states)
```

```
ACALDt: acald_e <=> acald_c
ACALDt: acald_e <=> acald_c
lower_bound

Here are the states for this member:
{<Feature ACONTb_lower_bound at 0x7efcac761e10>: -1000.0, <Feature ACKr_lower_bound at 0x7efcac91d12
```

These DictList objects are all iterables, meaning that any python operation that acts on an iterable can take them as input. This is often convenient when working with either cobrapy `Models` or medusa `Ensembles`. For example, suppose we are interested in getting the list of all components described by features in the `Ensemble`:

```
In [11]: components = []
         for feat in example_ensemble.features:
             components.append(feat.base_component)

         print(components)

         # or, use the one-liner which gives the same result:
         components = [feat.base_component for feat in example_ensemble.features]
```

```
        print(components)
```

```
[<Reaction ACALDt at 0x7efcac66f0b8>, <Reaction ACALDt at 0x7efcac66f0b8>, <Reaction ACKr at 0x7efcac
[<Reaction ACALDt at 0x7efcac66f0b8>, <Reaction ACALDt at 0x7efcac66f0b8>, <Reaction ACKr at 0x7efcac
```

1.8 Creating an ensemble

Medusa has two families of methods for generating ensembles: **expansion** and **degradation**. Expansion approaches currently consist of gapfilling algorithms. Degradation approaches include random degradation (useful for benchmarking new gapfilling methods) and omics integration algorithms that constrain network (e.g. transcriptomics integration; not currently implemented).

1.8.1 Expanding a network

The most common network expansion approach involving metabolic networks is algorithmic gapfilling, where the goal is to identify reactions to add to a network that allow a feasible solution. An example of this is adding a minimal number of reactions to enable biomass production in a model for an organism in a specific condition (e.g. SMILEY [1]). See the gapfilling documentation in cobrapy for the formulation of this problem.

Adding the minimum number of reactions to satisfy a biological function is just one approach to the gapfilling strategy. An alternative approach is to reformulate the problem to add the minimum amount of *flux* through candidate reactions for gapfilling. This has the advantage of being an entirely continuous problem, rather than the integer problem posed by SMILEY, so the time to find a solution is usually 1-2 orders of magnitude shorter.

In `medusa`, implementations of both gapfilling strategies are available, **but we recommend the continuous approach**, which we demonstrate below.

1.8.2 Input for gapfilling

The key inputs for gapfilling are a `cobra.Model` object representing the GENRE you are filling gaps in and a second `cobra.Model` object containing reactions that form a *universal reaction database* (sometimes called a *universal reaction bag*). Additionally, context-specific information, such as the environmental conditions in which a phenotype was observed, may be needed to constrain the model during gapfilling.

Let's use test data available in `medusa` for gapfilling. The approach we'll take to generate multiple solutions involves iteratively gapfilling across multiple media conditions to generate a single gapfilled model. We repeat the process with the original model but with a shuffled version of the media conditions (changing the order in which media conditions are used during gapfilling), each time generating a new solution. You can see examples of this approach in Biggs & Papin [2] and Medlock & Papin [3].

```
In [1]: # Load the test model for Staphylococcus aureus, originally generated with ModelSEED
        import medusa
        from medusa.test import create_test_model
        model = create_test_model('Saureus_seed')

        # Load the biolog data from Plata et al., Nature 2014
        from medusa.test import load_biolog_plata
        biolog_base_composition, biolog_base_dict, biolog_thresholded = load_biolog_plata()
        biolog_base_composition
```

```
Out[1]:          Name           ID
        0         H2O  cpd00001_e
        1          O2  cpd00007_e
```

```
2   Phosphate  cpd00009_e
3         CO2  cpd00011_e
4         NH3  cpd00013_e
5        Mn2+  cpd00030_e
6        Zn2+  cpd00034_e
7     Sulfate  cpd00048_e
8        Cu2+  cpd00058_e
9        Ca2+  cpd00063_e
10         H+  cpd00067_e
11        Cl-  cpd00099_e
12       Co2+  cpd00149_e
13         K+  cpd00205_e
14         Mg  cpd00254_e
15        Na+  cpd00971_e
16       Fe2+  cpd10515_e
17        fe3  cpd10516_e
18       Heme  cpd00028_e
19     H2S2O3  cpd00268_e
```

Here, `biolog_base_composition` describes the media components that are present in every biolog condition (Note: if you are using these data for your own purposes, keep in mind that we added Heme and H2S2O3 due to common issues encountered in models. These are not actually in the biolog medium).

The `biolog_base_dict` is a dictionary version of this, which we'll use as direct input to the models as part of `model.medium`

```
In [2]: biolog_base_dict
```

```
Out[2]: {'cpd00001_e': 1000,
         'cpd00007_e': 1000,
         'cpd00009_e': 1000,
         'cpd00011_e': 1000,
         'cpd00013_e': 1000,
         'cpd00028_e': 1000,
         'cpd00030_e': 1000,
         'cpd00034_e': 1000,
         'cpd00048_e': 1000,
         'cpd00058_e': 1000,
         'cpd00063_e': 1000,
         'cpd00067_e': 1000,
         'cpd00099_e': 1000,
         'cpd00149_e': 1000,
         'cpd00205_e': 1000,
         'cpd00254_e': 1000,
         'cpd00268_e': 1000,
         'cpd00971_e': 1000,
         'cpd10515_e': 1000,
         'cpd10516_e': 1000}
```

The actual growth/no growth data is in `biolog_thresholded`, which is a pandas `DataFrame` with organism species/genus as rows, and biolog media conditions as columns represented by the ModelSEED metabolite ID for the single carbon/nitrogen source present. The original source of these data is [4]; there, you can find the non-thresholded values if curious. Here, we've thresholded the growth data using the same threshold reported in the paper (>=10 relative units of tetrazolium dye).

```
In [3]: # Just inspect the first 5 species
        biolog_thresholded.head(5)
```

```
Out[3]:                              cpd11594_e  cpd00179_e  cpd00794_e  cpd03845_e  \
        Staphylococcus aureus              True        True        True       False
        Ralstonia solanacearum            False       False        True       False
```

```
Staphylococcus haemolyticus        True       True       True      False
Bacillus pumilus                   True      False       True       True
Corynebacterium glutamicum         True       True      False      False

                             cpd05158_e cpd00076_e cpd01133_e cpd00382_e  \
Staphylococcus aureus              True       True      False      False
Ralstonia solanacearum            False       True      False      False
Staphylococcus haemolyticus       False       True      False      False
Bacillus pumilus                   True       True       True       True
Corynebacterium glutamicum        False       True      False      False

                             cpd00208_e cpd03198_e    ...     cpd00024_e  \
Staphylococcus aureus              True      False    ...           True
Ralstonia solanacearum            False      False    ...           True
Staphylococcus haemolyticus        True      False    ...           True
Bacillus pumilus                  False       True    ...           True
Corynebacterium glutamicum        False      False    ...          False

                             cpd00386_e cpd00130_e cpd00281_e cpd03561_e  \
Staphylococcus aureus             False       True      False       True
Ralstonia solanacearum            False       True       True      False
Staphylococcus haemolyticus       False      False      False       True
Bacillus pumilus                   True       True       True       True
Corynebacterium glutamicum        False       True       True       True

                             cpd00094_e cpd00142_e cpd00141_e cpd00029_e  \
Staphylococcus aureus              True       True       True       True
Ralstonia solanacearum            False      False       True       True
Staphylococcus haemolyticus        True       True      False       True
Bacillus pumilus                   True       True       True       True
Corynebacterium glutamicum         True       True       True       True

                             cpd00047_e
Staphylococcus aureus              True
Ralstonia solanacearum             True
Staphylococcus haemolyticus        True
Bacillus pumilus                   True
Corynebacterium glutamicum        False

[5 rows x 60 columns]
```

Now we'll extract the positive growth conditions for the species we're interested in (*Staphylococcus aureus*)

```
In [4]: test_mod_pheno = biolog_thresholded.loc['Staphylococcus aureus']
        test_mod_pheno = list(test_mod_pheno[test_mod_pheno == True].index)
        test_mod_pheno
```

```
Out[4]: ['cpd11594_e',
         'cpd00179_e',
         'cpd00794_e',
         'cpd05158_e',
         'cpd00076_e',
         'cpd00208_e',
         'cpd15584_e',
         'cpd00122_e',
         'cpd00492_e',
         'cpd00232_e',
         'cpd19001_e',
         'cpd00138_e',
         'cpd00082_e',
         'cpd00709_e',
         'cpd00396_e',
         'cpd00246_e',
         'cpd00314_e',
         'cpd01307_e',
         'cpd00100_e',
         'cpd00079_e',
         'cpd00072_e',
         'cpd00320_e',
         'cpd00035_e',
         'cpd00051_e',
         'cpd00041_e',
         'cpd00023_e',
         'cpd00119_e',
         'cpd01293_e',
         'cpd00054_e',
         'cpd00222_e',
         'cpd05264_e',
         'cpd00159_e',
         'cpd00024_e',
         'cpd00130_e',
         'cpd03561_e',
         'cpd00094_e',
         'cpd00142_e',
         'cpd00141_e',
         'cpd00029_e',
         'cpd00047_e']
```

In order to gapfill this model, we have to make sure that the biolog media components are in the model, and that there are exchange reactions for each of these metabolites. To make this process more convenient, we'll load the universal reaction database now, which we will also use later in the process. The universal model is large, and the `load_universal_modelseed` does some extra processing of the model, so loading it may take a few minutes. First we'll check for changes that need to be made:

```
In [5]: # load the universal reaction database
        from medusa.test import load_universal_modelseed
        from cobra.core import Reaction
        universal = load_universal_modelseed()

        # check for biolog base components in the model and record
        # the metabolites/exchanges that need to be added
        add_mets = []
        add_exchanges = []
        for met in list(biolog_base_dict.keys()):
            try:
                model.metabolites.get_by_id(met)
            except:
                print('no '+met)
                add_met = universal.metabolites.get_by_id(met).copy()
                add_mets.append(add_met)

        model.add_metabolites(add_mets)

        for met in list(biolog_base_dict.keys()):
            # Search for exchange reactions
            try:
                model.reactions.get_by_id('EX_'+met)
            except:
                add_met = universal.metabolites.get_by_id(met)
                ex_rxn = Reaction('EX_' + met)
                ex_rxn.name = "Exchange reaction for " + met
                ex_rxn.lower_bound = -1000
                ex_rxn.upper_bound = 1000
                ex_rxn.add_metabolites({add_met:-1})
                add_exchanges.append(ex_rxn)

        model.add_reactions(add_exchanges)
```

```
no cpd00013_e
```

Next, we need to do the same for the single carbon/nitrogen sources in the biolog data. When performing this workflow on your own GENRE, you may want to check that all of the media components that enable growth have suitable transporters in the universal model (or already in the draft model).

```
In [6]: # Find metabolites from the biolog data that are missing in the test model
        # and add them from the universal
        missing_mets = []
        missing_exchanges = []
        media_dicts = {}
        for met_id in test_mod_pheno:
            try:
                model.metabolites.get_by_id(met_id)
            except:
                print(met_id + " was not in model, adding met and exchange reaction")
                met = universal.metabolites.get_by_id(met_id).copy()
                missing_mets.append(met)
                ex_rxn = Reaction('EX_' + met_id)
                ex_rxn.name = "Exchange reaction for " + met_id
                ex_rxn.lower_bound = -1000
                ex_rxn.upper_bound = 1000
                ex_rxn.add_metabolites({met:-1})
                missing_exchanges.append(ex_rxn)
            media_dicts[met_id] = biolog_base_dict.copy()
            media_dicts[met_id] = {'EX_'+k:v for k,v in media_dicts[met_id].items()}
            media_dicts[met_id]['EX_'+met_id] = 1000
        model.add_metabolites(missing_mets)
        model.add_reactions(missing_exchanges)
```

```
cpd11594_e was not in model, adding met and exchange reaction
cpd05158_e was not in model, adding met and exchange reaction
cpd15584_e was not in model, adding met and exchange reaction
cpd00492_e was not in model, adding met and exchange reaction
cpd00232_e was not in model, adding met and exchange reaction
cpd19001_e was not in model, adding met and exchange reaction
cpd00709_e was not in model, adding met and exchange reaction
cpd00396_e was not in model, adding met and exchange reaction
cpd01307_e was not in model, adding met and exchange reaction
cpd00079_e was not in model, adding met and exchange reaction
cpd00072_e was not in model, adding met and exchange reaction
cpd00320_e was not in model, adding met and exchange reaction
cpd01293_e was not in model, adding met and exchange reaction
cpd05264_e was not in model, adding met and exchange reaction
cpd03561_e was not in model, adding met and exchange reaction
cpd00094_e was not in model, adding met and exchange reaction
cpd00142_e was not in model, adding met and exchange reaction
cpd00141_e was not in model, adding met and exchange reaction
cpd00029_e was not in model, adding met and exchange reaction
```

Now, let's fill some gaps using the `iterative_gapfill_from_binary_phenotypes` function. For simplicity, we'll just take the first 5 conditions and perform gapfilling for 10 cycles, which should yield an ensemble with 10 members. We set `lower_bound = 0.05`, which requires that the model produces 0.05 units of flux through the previous objective function (here, biomass production) which is now set as a constraint (i.e. vbm >= 0.05). `inclusion_threshold` is the amount of flux through a reaction required to include it in the gapfill solution, which is necessary because of the limits of numerical precision. Generally a small number (e.g. < 1E-8) is a good choice. However, some gapfill solutions may have reactions with non-zero flux in ranges lower than this; if this occurs, Medusa will raise an error letting you know that it failed to validate the gapfill solution, and that you should try lowering the threshold.

```
In [7]: from medusa.reconstruct.expand import iterative_gapfill_from_binary_phenotypes
        # select a subset of the biolog conditions to perform gapfilling with
        sources = list(media_dicts.keys())
        sub_dict = {sources[0]:media_dicts[sources[0]],
                    sources[1]:media_dicts[sources[1]],
                    sources[2]:media_dicts[sources[2]],
                    sources[3]:media_dicts[sources[3]],
                    sources[4]:media_dicts[sources[4]]}

        num_cycles = 10
        lower_bound = 0.05
        flux_cutoff = 1E-10
        ensemble = iterative_gapfill_from_binary_phenotypes(model,universal,sub_dict,num_cycles,\
                                                             lower_bound=lower_bound,\
                                                             inclusion_threshold=1E-10,\
                                                             exchange_reactions=False,\
                                                             demand_reactions=False,\
                                                             exchange_prefix='EX')
```

```
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
building features...
updating members...
```

```
In [8]: print(len(ensemble.members))
        print(ensemble.members)
```

```
10
[<Member Staphylococcus aureus_gapfilled_1 at 0x7ff9695d2128>, <Member Staphylococcus aureus_gapfille
```

```
In [9]: # Check out the features that vary across the ensemble
        print(len(ensemble.features))
        print([feature.base_component.id for feature in ensemble.features])
```

```
78
['rxn05487_c', 'rxn05487_c', 'rxn13647_c', 'rxn13647_c', 'rxn31356_c', 'rxn31356_c', 'rxn12303_c', '
```

1.8.3 Degrading a network

Coming here soon.

```
In [ ]:
```

References

[1]: Reed et al., "Systems approach to refining genome annotation", *PNAS* 2006

[2]: Biggs & Papin, "Managing uncertainty in metabolic network structure and improving predictions using EnsembleFBA", *PLoS Computational Biology* 2017

[3]: Medlock & Papin, "Guiding the refinement of biochemical knowledgebases with ensembles of metabolic networks and semi-supervised learning", *BioRxiv* 2018

[4]: Plata et al., "Long-term phenotypic evolution of bacteria", *Nature* 2015

1.9 Performing ensemble simulations

With a functional `Ensemble` in hand, you're ready to perform simulations. In `medusa`, most simulations are performed by setting the model structure to represent an individual member, using cobrapy functions for the actual simulation, then repeating for all or many ensemble members.

1.9.1 Ensemble Flux Balance Analysis

Flux balance analysis (FBA) is one of the most widely used techniques in systems biology. See What is flux balance analysis? for an introduction to FBA, and the cobrapy documentation to see how FBA is performed with a single model.

When using `medusa` for FBA, the environmental conditions and objective function should be specified in `ensemble.base_model`, just as if it were a normal cobrapy `Model`:

```
In [1]: import medusa
        from medusa.test import create_test_ensemble

        ensemble = create_test_ensemble("Staphylococcus aureus")
```

```
In [2]: ensemble.base_model.objective.expression
```

```
Out[2]: 1.0*bio1 - 1.0*bio1_reverse_b18f7
```

The current objective function is the biomass reaction (`bio1`)–to change this, just set the objective to another reaction. Let's change the objective to CO2 exchange, then change it back to biomass production:

```
In [3]: ensemble.base_model.objective = 'EX_cpd00011_e'
        print(ensemble.base_model.objective.expression)
        ensemble.base_model.objective = 'bio1'
        print(ensemble.base_model.objective.expression)
```

```
1.0*EX_cpd00011_e - 1.0*EX_cpd00011_e_reverse_896eb
1.0*bio1 - 1.0*bio1_reverse_b18f7
```

Similarly, you can manipulate the environmental conditions as in cobrapy. The base model for this example ensemble is from ModelSEED, so exchange reactions are specified with the `'EX_'` prefix, followed by the metabolite id. Let's take a look at the exchange reactions that are currently open:

```
In [4]: medium = ensemble.base_model.medium
        medium
```

```
Out[4]: {'EX_cpd00001_e': 1000.0,
         'EX_cpd00007_e': 1000.0,
         'EX_cpd00009_e': 1000.0,
         'EX_cpd00010_e': 1000.0,
         'EX_cpd00011_e': 1000.0,
         'EX_cpd00012_e': 1000.0,
         'EX_cpd00013_e': 1000,
         'EX_cpd00023_e': 1000.0,
         'EX_cpd00024_e': 1000.0,
         'EX_cpd00027_e': 1000.0,
         'EX_cpd00028_e': 1000.0,
         'EX_cpd00029_e': 1000,
         'EX_cpd00030_e': 1000.0,
         'EX_cpd00033_e': 1000.0,
         'EX_cpd00034_e': 1000.0,
         'EX_cpd00035_e': 1000.0,
         'EX_cpd00039_e': 1000.0,
         'EX_cpd00041_e': 1000.0,
         'EX_cpd00047_e': 1000.0,
         'EX_cpd00048_e': 1000.0,
         'EX_cpd00051_e': 1000.0,
         'EX_cpd00053_e': 1000.0,
         'EX_cpd00054_e': 1000.0,
         'EX_cpd00058_e': 1000.0,
         'EX_cpd00060_e': 1000.0,
         'EX_cpd00063_e': 1000.0,
         'EX_cpd00064_e': 1000.0,
         'EX_cpd00066_e': 1000.0,
         'EX_cpd00067_e': 1000.0,
         'EX_cpd00069_e': 1000.0,
         'EX_cpd00072_e': 1000,
         'EX_cpd00073_e': 1000.0,
         'EX_cpd00075_e': 1000.0,
         'EX_cpd00076_e': 1000.0,
         'EX_cpd00079_e': 1000,
         'EX_cpd00080_e': 1000.0,
         'EX_cpd00082_e': 1000.0,
         'EX_cpd00092_e': 1000.0,
         'EX_cpd00094_e': 1000,
         'EX_cpd00098_e': 1000.0,
         'EX_cpd00099_e': 1000.0,
         'EX_cpd00100_e': 1000.0,
         'EX_cpd00104_e': 1000.0,
         'EX_cpd00105_e': 1000.0,
         'EX_cpd00117_e': 1000.0,
         'EX_cpd00119_e': 1000.0,
         'EX_cpd00122_e': 1000.0,
         'EX_cpd00129_e': 1000.0,
         'EX_cpd00130_e': 1000.0,
         'EX_cpd00137_e': 1000.0,
         'EX_cpd00138_e': 1000.0,
         'EX_cpd00141_e': 1000,
         'EX_cpd00142_e': 1000,
         'EX_cpd00149_e': 1000.0,
         'EX_cpd00159_e': 1000.0,
         'EX_cpd00179_e': 1000.0,
         'EX_cpd00182_e': 1000.0,
         'EX_cpd00184_e': 1000.0,
         'EX_cpd00205_e': 1000.0,
         'EX_cpd00208_e': 1000.0,
         'EX_cpd00220_e': 1000.0,
         'EX_cpd00222_e': 1000.0,
         'EX_cpd00232_e': 1000,
         'EX_cpd00244_e': 1000.0,
         'EX_cpd00246_e': 1000.0,
         'EX_cpd00249_e': 1000.0,
         'EX_cpd00254_e': 1000.0,
         'EX_cpd00264_e': 1000.0,
         'EX_cpd00268_e': 1000.0,
         'EX_cpd00276_e': 1000.0,
         'EX_cpd00277_e': 1000.0,
         'EX_cpd00305_e': 1000.0,
         'EX_cpd00309_e': 1000.0,
         'EX_cpd00314_e': 1000.0,
         'EX_cpd00320_e': 1000,
         'EX_cpd00322_e': 1000.0,
         'EX_cpd00355_e': 1000.0,
         'EX_cpd00367_e': 1000.0,
         'EX_cpd00393_e': 1000.0,
         'EX_cpd00396_e': 1000,
         'EX_cpd00412_e': 1000.0,
         'EX_cpd00438_e': 1000.0,
         'EX_cpd00492_e': 1000,
         'EX_cpd00531_e': 1000.0,
         'EX_cpd00540_e': 1000.0,
         'EX_cpd00550_e': 1000.0,
         'EX_cpd00588_e': 1000.0,
         'EX_cpd00637_e': 1000.0,
         'EX_cpd00654_e': 1000.0,
         'EX_cpd00681_e': 1000.0,
         'EX_cpd00709_e': 1000,
         'EX_cpd00794_e': 1000.0,
         'EX_cpd00971_e': 1000.0,
         'EX_cpd01012_e': 1000.0,
         'EX_cpd01030_e': 1000.0,
         'EX_cpd01080_e': 1000.0,
         'EX_cpd01171_e': 1000.0,
         'EX_cpd01262_e': 1000.0,
         'EX_cpd01293_e': 1000,
         'EX_cpd01307_e': 1000,
         'EX_cpd01329_e': 1000.0,
         'EX_cpd01914_e': 1000.0,
         'EX_cpd03279_e': 1000.0,
         'EX_cpd03561_e': 1000,
         'EX_cpd03696_e': 1000.0,
         'EX_cpd03724_e': 1000.0,
         'EX_cpd03725_e': 1000.0,
         'EX_cpd04097_e': 1000.0,
         'EX_cpd05158_e': 1000,
         'EX_cpd05264_e': 1000,
         'EX_cpd08305_e': 1000.0,
         'EX_cpd08306_e': 1000.0,
         'EX_cpd10515_e': 1000.0,
         'EX_cpd10516_e': 1000.0,
         'EX_cpd11576_e': 1000.0,
         'EX_cpd11594_e': 1000,
         'EX_cpd11597_e': 1000.0,
         'EX_cpd15584_e': 1000,
         'EX_cpd19001_e': 1000}
```

That's a lot of open exchange reactions! Let's make them a bit more realistic for an *in vitro* situation. We'll load a file specifying the base composition of the media in biolog single C/N growth conditions, and set the media conditions to reflect that. The base composition is missing a carbon source, so we'll enable uptake of glucose. In the medium dictionary, the numbers for each exchange reaction are uptake rates. If you inspect the actual exchange reactions, you will find that the equivalent to an uptake rate of 1000 units is a lower bound of -1000, because our exchange reactions are defined with the boundary metabolite as the reactant, e.g. `cpd00182_e -->`.

```
In [5]: import pandas as pd
        biolog_base = pd.read_csv("../medusa/test/data/biolog_base_composition.csv", sep=",")
        biolog_base
```

```
Out[5]:          Name          ID
        0         H2O  cpd00001_e
        1          O2  cpd00007_e
        2   Phosphate  cpd00009_e
        3         CO2  cpd00011_e
        4         NH3  cpd00013_e
        5        Mn2+  cpd00030_e
        6        Zn2+  cpd00034_e
        7     Sulfate  cpd00048_e
        8        Cu2+  cpd00058_e
        9        Ca2+  cpd00063_e
        10         H+  cpd00067_e
        11        Cl-  cpd00099_e
        12       Co2+  cpd00149_e
        13         K+  cpd00205_e
        14         Mg  cpd00254_e
        15        Na+  cpd00971_e
        16       Fe2+  cpd10515_e
        17        fe3  cpd10516_e
        18       Heme  cpd00028_e
        19     H2S2O3  cpd00268_e
```

```
In [6]: # convert the biolog base to a dictionary, which we can use to set ensemble.base_model.mediu
        biolog_base = {'EX_'+component:1000 for component in biolog_base['ID']}

        # add glucose uptake to the new medium dictionary
        biolog_base['EX_cpd00182_e'] = 10

        # Set the medium on the base model
        ensemble.base_model.medium = biolog_base
        ensemble.base_model.medium
```

```
Out[6]: {'EX_cpd00001_e': 1000,
         'EX_cpd00007_e': 1000,
         'EX_cpd00009_e': 1000,
         'EX_cpd00011_e': 1000,
         'EX_cpd00013_e': 1000,
         'EX_cpd00028_e': 1000,
         'EX_cpd00030_e': 1000,
         'EX_cpd00034_e': 1000,
         'EX_cpd00048_e': 1000,
         'EX_cpd00058_e': 1000,
         'EX_cpd00063_e': 1000,
         'EX_cpd00067_e': 1000,
         'EX_cpd00099_e': 1000,
         'EX_cpd00149_e': 1000,
         'EX_cpd00182_e': 10,
         'EX_cpd00205_e': 1000,
         'EX_cpd00254_e': 1000,
         'EX_cpd00268_e': 1000,
         'EX_cpd00971_e': 1000,
         'EX_cpd10515_e': 1000,
         'EX_cpd10516_e': 1000}
```

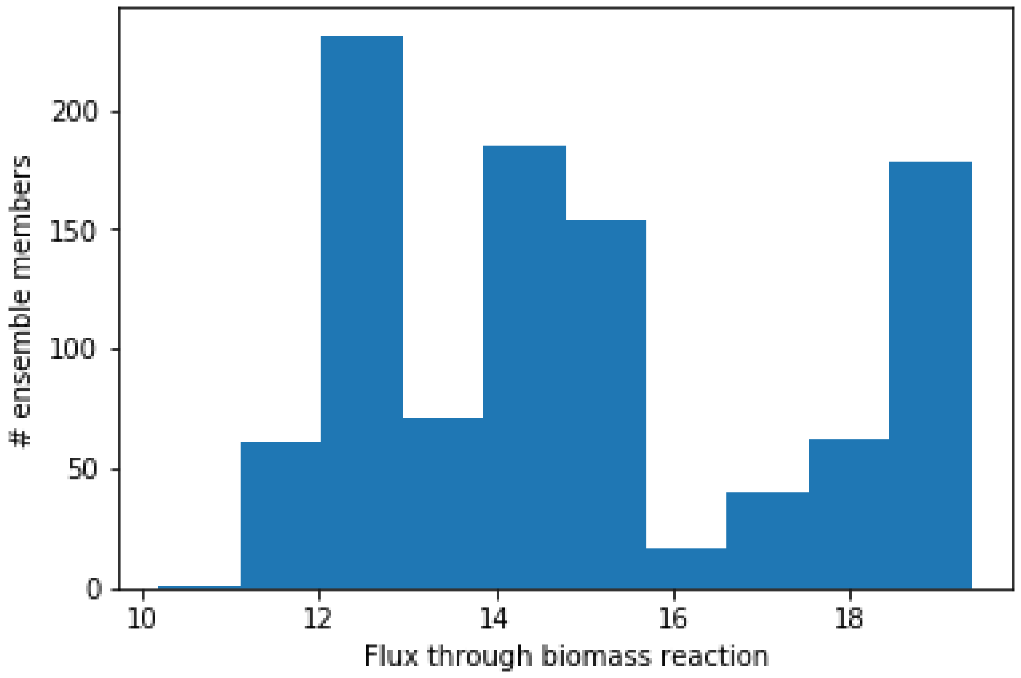
With the medium set, we can now simulate growth in these conditions:

```
In [7]: from medusa.flux_analysis import flux_balance
        fluxes = flux_balance.optimize_ensemble(ensemble,return_flux='bio1')
```

```
In [16]: # get fluxes for the first 10 members
         fluxes.head(10)
```

```
Out[16]:                                           bio1
         Staphylococcus aureus_gapfilled_18   14.890551
         Staphylococcus aureus_gapfilled_477  12.218825
         Staphylococcus aureus_gapfilled_430  19.198765
         Staphylococcus aureus_gapfilled_735  14.875922
         Staphylococcus aureus_gapfilled_916  12.223456
         Staphylococcus aureus_gapfilled_983  19.375070
         Staphylococcus aureus_gapfilled_371  13.113148
         Staphylococcus aureus_gapfilled_255  12.223456
         Staphylococcus aureus_gapfilled_729  14.891239
         Staphylococcus aureus_gapfilled_925  19.198765
```

```
In [10]: import matplotlib.pylab as plt
         fig, ax = plt.subplots()
         plt.hist(fluxes['bio1'])
         ax.set_ylabel('# ensemble members')
         ax.set_xlabel('Flux through biomass reaction')
         plt.show()
```



You may want to perform simulations with only a subset of ensemble members. There are two options for this; either identifying the desired members for simulation with the `specific_models` parameter, or passing a number of random members to perform simulations with the `num_models` parameter.

```
In [14]: # perform FBA with a random set of 10 members:
         subflux = flux_balance.optimize_ensemble(ensemble, num_models = 10, return_flux = "bio1")
         subflux
```

```
Out[14]:                                         bio1
         Staphylococcus aureus_gapfilled_300  18.441010
         Staphylococcus aureus_gapfilled_181  14.875922
         Staphylococcus aureus_gapfilled_667  17.618230
         Staphylococcus aureus_gapfilled_668  14.875922
         Staphylococcus aureus_gapfilled_639  14.186860
         Staphylococcus aureus_gapfilled_636  14.186860
         Staphylococcus aureus_gapfilled_738  14.643953
         Staphylococcus aureus_gapfilled_68   12.223456
         Staphylococcus aureus_gapfilled_87   14.875922
         Staphylococcus aureus_gapfilled_580  12.223456
```

```
In [15]: submembers = [member.id for member in ensemble.members[0:10]]
         print(submembers)
         subflux = flux_balance.optimize_ensemble(ensemble, specific_models = submembers, return_flux
         subflux
```

```
['Staphylococcus aureus_gapfilled_892', 'Staphylococcus aureus_gapfilled_851', 'Staphylococcus aureus
```

```
Out[15]:                                         bio1
         Staphylococcus aureus_gapfilled_372  12.223456
         Staphylococcus aureus_gapfilled_421  13.113148
         Staphylococcus aureus_gapfilled_500  19.198765
         Staphylococcus aureus_gapfilled_501  12.223456
         Staphylococcus aureus_gapfilled_751  14.209162
         Staphylococcus aureus_gapfilled_849  12.224814
         Staphylococcus aureus_gapfilled_851  19.375070
         Staphylococcus aureus_gapfilled_875  17.872504
         Staphylococcus aureus_gapfilled_892  19.375070
         Staphylococcus aureus_gapfilled_927  19.198765
```

1.9.2 Flux Variability Analysis

```
In [ ]:
```

1.9.3 Gene and Reaction Deletions

```
In [ ]:
```

1.10 Input and output

Currently, the only supported approach for loading and saving ensembles in `medusa` is via pickle. `pickle` is the Python module that serializes and de-serializes Python objects (i.e. converts to/from a binary representation). This is an intentional design choice–as `medusa` matures, we will identify a feasible route for standardization through an extension to the Systems Biology Markup Language (SBML), which is the *de facto* standard for sharing genome-scale metabolic network reconstructions.

To load an ensemble, use the `load` function from the `pickle` module:

```
In [1]: import medusa
        from pickle import load

        with open("../medusa/test/data/Staphylococcus_aureus_ensemble.pickle", 'rb') as infile:
            ensemble = load(infile)
```

To save an ensemble, you can pickle it with:

```
In [2]: save_dir = ("../medusa/test/data/Staphylococcus_aureus_repickled.pickle")
        ensemble.to_pickle(save_dir)
```

You can always save the base model for an ensemble using the standard cobrapy I/O functions, but keep in mind the states for each feature will be set statically–the model you save will only represent one of the ensemble members, and will likely have many features shut off (e.g. there will be many closed reactions if the features for those reactions are not present in the ensemble member that the state reflects). When publishing ensembles, we recommend including the pickled `medusa` ensemble, an SBML file for the base model, and a spreadsheet of feature states for each member.

1.11 Ensemble Size and Speed Benchmarking

`Ensembles` are specifically designed for optimal usability, memory usage, and computational speed. In this tutorial we explore the size and speed related characteristics of `Ensembles` compared to using the equivalent individual models. We aim to begin to answer the following questions: - How much memory does an ensemble use when working with it compared to working with the equivalent individual models? - How much disk space is used to store ensembles compared to the equivalent individual models? - How long does it take to run FBA for all members of an ensemble compared to the equivalent individual models?

1.11.1 Ensemble memory requirements during use and when saved

`Ensembles` are structured to minimize the amount of memory required when loaded and when being saved. One of the major challenges when working with ensembles of models is having all of the models readily available in memory while conducting analyses. With efficient packaging of the features that are different between members of an ensemble, we were able to significantly reduce the amount of memory and hard drive space required for working with ensembles of models.

```
In [1]: import sys
        import os
        import psutil
        import medusa
        import numpy
        from medusa.test import create_test_ensemble
```

```
In [2]: # RAM required to load in a 1000 member ensemble

        # Check initial RAM usage
        RAM_before = psutil.Process(os.getpid()).memory_info()[0]/1024**2 # Units = MB

        # Load in test ensemble from file
        ensemble = create_test_ensemble("Staphylococcus aureus")

        # Check RAM usage after loading in ensemble
        RAM_after = psutil.Process(os.getpid()).memory_info()[0]/1024**2 # Units = MB
        RAM_used = RAM_after - RAM_before
        # Print RAM usage increase due to loading ensemble
        print("%.2f" % (RAM_used), "MB")
```

```
57.82 MB
```

```
In [3]: # The test S. aureus model has 1000 members
        print(len(ensemble.members),'Members')
```

```
1000 Members
```

```
In [4]: # RAM required to load a single individual model

        from copy import deepcopy
        # Check initial RAM usage
        RAM_before = psutil.Process(os.getpid()).memory_info()[0]/1024**2 # Units = MB

        # Deepcopy base model to create new instance of model in RAM
        extracted_base_model_copy = deepcopy(ensemble.base_model)

        # Check RAM usage after loading in ensemble
        RAM_after = psutil.Process(os.getpid()).memory_info()[0]/1024**2 # Units = MB
        RAM_used = RAM_after - RAM_before
        # Print RAM usage increase due to loading ensemble
        print("%.2f" % (RAM_used), "MB")
```

```
17.50 MB
```

```
In [5]: # If we were to load the individual base model as 1000 unique
        # model variables we would use 1000x as much RAM:
        RAM_used_for_1000_individual_model_variables = RAM_used * 1000
        print("%.2f" % (RAM_used_for_1000_individual_model_variables), 'MB or')
        print("%.2f" % (RAM_used_for_1000_individual_model_variables/1024.0), 'GB')
```

```
17500.00 MB or
17.09 GB
```

```
In [6]: # Pickle the ensemble and extracted base model
        import pickle
        path = "../medusa/test/data/benchmarking/"
        pickle.dump(ensemble, open(path+"Staphylococcus_aureus_ensemble1000.pickle","wb"))
        pickle.dump(extracted_base_model_copy, open(path+"Staphylococcus_aureus_base_model.pickle","
```

```
In [7]: # Check for file size of ensemble
        file_path = "../medusa/test/data/benchmarking/Staphylococcus_aureus_ensemble1000.pickle"
        if os.path.isfile(file_path):
            file_info = os.stat(file_path)
            mb = file_info.st_size/(1024.0**2) # Convert from bytes to MB
            print("%.2f %s" % (mb, 'MB for a 1000 member ensemble'))
        else:
            print("File path doesn't point to file.")
```

```
6.67 MB for a 1000 member ensemble
```

```
In [8]: # Check for file size of extracted base model
        file_path = "../medusa/test/data/benchmarking/Staphylococcus_aureus_base_model.pickle"
        if os.path.isfile(file_path):
            file_info = os.stat(file_path)
            mb = file_info.st_size/(1024.0**2) # Convert from bytes to MB
            print("%.2f %s" % (mb, 'MB per model'))
        else:
            print("File path doesn't point to file.")

        print("%.2f" % (mb*1000),'MB for 1000 individual model files.')
        print("%.2f" % (mb*1000/1024),'GB for 1000 individual model files.')
```

```
1.07 MB per model
1070.01 MB for 1000 individual model files.
1.04 GB for 1000 individual model files.
```

1.11.2 Flux analysis speed testing

Running FBA requires a relatively short amount of time for a single model, however when working with ensembles of 1000s of models, the simple optimization problems can add up to significant amounts of time. Here we explore the expected timeframes for FBA with an ensemble and how that compares to using the equivalent number of individual models. It is important to note that during this benchmarking, we assume that the computer being used is capable to loading all individual modelings into the RAM; this may not be the case for many modern laptop computers (e.g., ~16GB spare memory required).

```
In [9]: import time
        from medusa.flux_analysis import flux_balance
```

```
In [10]: # Time required to run FBA on a 1000 member ensemble using the innate Medusa functions.
         runtimes = {}
         trials = 5
         for num_processes in [1,2,3,4]:
             runtimes[num_processes] = []
             for trial in range(0,trials):
                 t0 = time.time()
                 flux_balance.optimize_ensemble(ensemble, num_processes = num_processes)
                 t1 = time.time()
                 runtimes[num_processes].append(t1-t0)
             print(str(num_processes) + ' processors: ' + str(numpy.mean(runtimes[num_processes])) +
```

```
1 processors: 87.24728102684021 seconds for entire ensemble
2 processors: 44.09945402145386 seconds for entire ensemble
3 processors: 32.84902577400207 seconds for entire ensemble
4 processors: 27.70060839653015 seconds for entire ensemble
```

```
In [11]: # Time required to run FBA on 1000 individual models using a single processor.
         # This is the equivalent time that would be required if all 1000 models were pre-loaded in

         trial_total = []
         for trial in range(0,trials):
             t_total = 0
             for member in ensemble.members:
                 # Set the member state
                 ensemble.set_state(member.id)
                 # Start the timer to capture only time required to run FBA on each model
                 t0 = time.time()
                 solution = ensemble.base_model.optimize()
                 t1 = time.time()
                 t_total = t1-t0 + t_total
             print("%.2f" % (t_total) ,'seconds for 1000 models')
             trial_total.append(t_total)
         print("%.2f" % (numpy.mean(trial_total)) ,'second average for 1000 models')
```

```
35.06 seconds for 1000 models
34.51 seconds for 1000 models
34.49 seconds for 1000 models
34.62 seconds for 1000 models
34.37 seconds for 1000 models
34.61 second average for 1000 models
```

Using individual models stored in memory is faster than an equivalent ensemble with 1-2 processors, but Medusa is faster with an increasing number of processors. Keep in mind, however, that this comparison doesn't consider the time it takes to load all of the models (~200x faster in Medusa for an ensemble this size), make any modifications to the media conditions for an ensemble (one operation in Medusa; 1000 independent operations with individual models), and that using individual models requires far more memory (~300x in this case).

This comparison also doesn't factor in the time required for the first optimization performed with any COBRApy model. When a model is optimized once, the solver maintains the solution as a starting point for future optimization steps, substantially reducing the time required for future simulations. Medusa intrinsically takes advantage of this by only using one COBRApy model to represent the entire ensemble; the solution is recycled from member to member during ensemble FBA in Medusa. In contrast, the first optimization step for every individual model loaded into memory will be more computationally expensive, as seen by the timing in the cell below.

```
In [13]: # Time required to run FBA on 1000 individual models with a complete solver reset
         # before each optimization problem is solved.

         # Load fresh version of model with blank solver state
         fresh_base_model = pickle.load(open("../medusa/test/data/benchmarking/Staphylococcus_aureus_
         # Determine how long it takes to run FBA on one individual model
         t0 = time.time()
         fresh_base_model.optimize()
         t1 = time.time()
         t_total = t1-t0
         # Calculate how long it would take to run FBA on 1000 unique individual models
         print("%.2f" % (t_total*1000), 'seconds for 1000 models')
```

```
192.96 seconds for 1000 models
```

1.12 Benchmarking ensemble generation

This notebook does some simple benchmarking of ensemble generation in Medusa through iterative gapfilling. You can see the full narrative version of the process being benchmarked in the **Creating an ensemble** section of the User Guide.

```
In [5]: # Load the test model for Staphylococcus aureus, originally generated with ModelSEED
        import medusa
        from medusa.test import create_test_model
        model = create_test_model('Saureus_seed')

        # Load the biolog data from Plata et al., Nature 2014
        from medusa.test import load_biolog_plata
        biolog_base_composition, biolog_base_dict, biolog_thresholded = load_biolog_plata()

        # Extract growth/no growth calls for staph aureus
        test_mod_pheno = biolog_thresholded.loc['Staphylococcus aureus']
        test_mod_pheno = list(test_mod_pheno[test_mod_pheno == True].index)

        # load the universal reaction database
        from medusa.test import load_universal_modelseed
        from cobra.core import Reaction
        universal = load_universal_modelseed()

        # check for biolog base components in the model and record
        # the metabolites/exchanges that need to be added
        add_mets = []
        add_exchanges = []
        for met in list(biolog_base_dict.keys()):
            try:
                model.metabolites.get_by_id(met)
            except:
                print('no '+met)
                add_met = universal.metabolites.get_by_id(met).copy()
                add_mets.append(add_met)

        model.add_metabolites(add_mets)

        for met in list(biolog_base_dict.keys()):
            # Search for exchange reactions
            try:
                model.reactions.get_by_id('EX_'+met)
            except:
                add_met = universal.metabolites.get_by_id(met)
                ex_rxn = Reaction('EX_' + met)
                ex_rxn.name = "Exchange reaction for " + met
                ex_rxn.lower_bound = -1000
                ex_rxn.upper_bound = 1000
                ex_rxn.add_metabolites({add_met:-1})
                add_exchanges.append(ex_rxn)

        model.add_reactions(add_exchanges)

        # Find metabolites from the biolog data that are missing in the test model
        # and add them from the universal
        missing_mets = []
        missing_exchanges = []
        media_dicts = {}
        for met_id in test_mod_pheno:
            try:
                model.metabolites.get_by_id(met_id)
            except:
                print(met_id + " was not in model, adding met and exchange reaction")
                met = universal.metabolites.get_by_id(met_id).copy()
                missing_mets.append(met)
                ex_rxn = Reaction('EX_' + met_id)
                ex_rxn.name = "Exchange reaction for " + met_id
                ex_rxn.lower_bound = -1000
                ex_rxn.upper_bound = 1000
                ex_rxn.add_metabolites({met:-1})
                missing_exchanges.append(ex_rxn)
            media_dicts[met_id] = biolog_base_dict.copy()
            media_dicts[met_id] = {'EX_'+k:v for k,v in media_dicts[met_id].items()}
            media_dicts[met_id]['EX_'+met_id] = 1000
        model.add_metabolites(missing_mets)
        model.add_reactions(missing_exchanges)
```

```
no cpd00013_e
cpd11594_e was not in model, adding met and exchange reaction
cpd05158_e was not in model, adding met and exchange reaction
cpd15584_e was not in model, adding met and exchange reaction
cpd00492_e was not in model, adding met and exchange reaction
cpd00232_e was not in model, adding met and exchange reaction
cpd19001_e was not in model, adding met and exchange reaction
cpd00709_e was not in model, adding met and exchange reaction
cpd00396_e was not in model, adding met and exchange reaction
cpd01307_e was not in model, adding met and exchange reaction
cpd00079_e was not in model, adding met and exchange reaction
cpd00072_e was not in model, adding met and exchange reaction
cpd00320_e was not in model, adding met and exchange reaction
cpd01293_e was not in model, adding met and exchange reaction
cpd05264_e was not in model, adding met and exchange reaction
cpd03561_e was not in model, adding met and exchange reaction
cpd00094_e was not in model, adding met and exchange reaction
cpd00142_e was not in model, adding met and exchange reaction
cpd00141_e was not in model, adding met and exchange reaction
cpd00029_e was not in model, adding met and exchange reaction
```

With the input prepared, let's fill some gaps using the `iterative_gapfill_from_binary_phenotypes` function. We will gapfill each ensemble using 10 media conditions and perform the process three times for target ensemble sizes of 5 members, 25 members, and 100 members. Each ensemble generation step will be repeated 10 times with a different random sampling of 10 media conditions (e.g., 10 ensembles of 5 members, 25 members and 100 members will be generated and the mean/standard deviation of construction time will be reported).

```
In [19]: from medusa.reconstruct.expand import iterative_gapfill_from_binary_phenotypes
         import time
         import random

         num_cycles = 5
         lower_bound = 0.05
         flux_cutoff = 1E-10

         clock_time = {}
         clock_time[5] = []
         for i in range(0,10):
             # sample without replacement
             media_selection = random.sample(list(media_dicts.keys()),10)
             sub_dict = {condition:media_dicts[condition] for condition in media_selection}
             time1 = time.time()
             ensemble = iterative_gapfill_from_binary_phenotypes(model,universal,sub_dict,num_cycles,
                                                  lower_bound=lower_bound,\
                                                  inclusion_threshold=1E-10,\
                                                  exchange_reactions=False,\
                                                  demand_reactions=False,\
                                                  exchange_prefix='EX');
             time2 = time.time()
             clock_time[5].append(time2-time1)
             print("Clock time: " + str(time2-time1))

         num_cycles = 25
         clock_time[25] = []
         for i in range(0,10):
             # sample without replacement
             media_selection = random.sample(list(media_dicts.keys()),10)
             sub_dict = {condition:media_dicts[condition] for condition in media_selection}
             time1 = time.time()
             ensemble = iterative_gapfill_from_binary_phenotypes(model,universal,sub_dict,num_cycles,
                                                  lower_bound=lower_bound,\
                                                  inclusion_threshold=1E-10,\
                                                  exchange_reactions=False,\
                                                  demand_reactions=False,\
                                                  exchange_prefix='EX');
             time2 = time.time()
             clock_time[25].append(time2-time1)
             print("Clock time: " + str(time2-time1))

         num_cycles = 100
         clock_time[100] = []
         for i in range(0,10):
             # sample without replacement
             media_selection = random.sample(list(media_dicts.keys()),10)
             sub_dict = {condition:media_dicts[condition] for condition in media_selection}
             time1 = time.time()
             ensemble = iterative_gapfill_from_binary_phenotypes(model,universal,sub_dict,num_cycles,
                                                  lower_bound=lower_bound,\
                                                  inclusion_threshold=1E-10,\
                                                  exchange_reactions=False,\
                                                  demand_reactions=False,\
                                                  exchange_prefix='EX');
             time2 = time.time()
```

```
            clock_time[100].append(time2-time1)
            print("Clock time: " + str(time2-time1))
```

```
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 97.85026526451111
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 106.4863657951355
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 119.81539511680603
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 97.87515687942505
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 107.44688296318054
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 97.99315094947815
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 110.61738395690918
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 111.50042414665222
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 128.1014850139618
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
building features...
updating members...
Clock time: 95.986576795578
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
building features...
updating members...
Clock time: 267.31880497932434
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
building features...
updating members...
Clock time: 381.626629114151
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
```

```
building features...
updating members...
Clock time: 279.64290976524353
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
building features...
updating members...
Clock time: 266.1675601005554
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
building features...
```

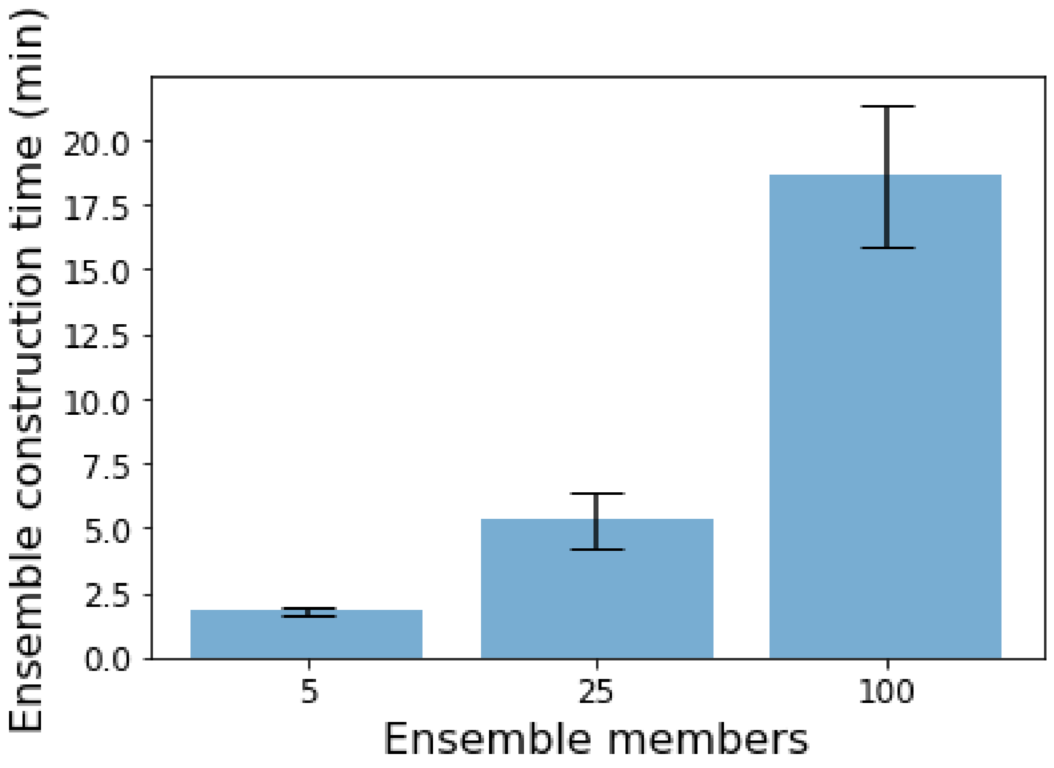
```
updating members...
Clock time: 270.17031383514404
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
building features...
updating members...
Clock time: 253.3940167427063
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
building features...
updating members...
```

```
Clock time: 389.7195827960968
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
building features...
updating members...
Clock time: 380.86448097229004
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
building features...
updating members...
Clock time: 271.53583908081055
```

```
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
building features...
updating members...
Clock time: 435.9781460762024
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
Clock time: 998.9393928050995
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
```

```
Clock time: 1333.905808210373
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
Clock time: 1008.801106929779
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
Clock time: 938.9039621353149
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
Clock time: 1012.624204158783
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
Clock time: 957.0066258907318
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
Clock time: 1158.7237539291382
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
Clock time: 1412.2274470329285
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
Clock time: 1309.229562997818
Constraining lower bound for bio1
starting cycle number 0
starting cycle number 1
starting cycle number 2
starting cycle number 3
starting cycle number 4
starting cycle number 5
starting cycle number 6
starting cycle number 7
starting cycle number 8
starting cycle number 9
starting cycle number 10
starting cycle number 11
starting cycle number 12
starting cycle number 13
starting cycle number 14
starting cycle number 15
starting cycle number 16
starting cycle number 17
starting cycle number 18
starting cycle number 19
starting cycle number 20
starting cycle number 21
starting cycle number 22
starting cycle number 23
starting cycle number 24
starting cycle number 25
starting cycle number 26
starting cycle number 27
starting cycle number 28
starting cycle number 29
starting cycle number 30
starting cycle number 31
starting cycle number 32
starting cycle number 33
starting cycle number 34
starting cycle number 35
starting cycle number 36
starting cycle number 37
starting cycle number 38
starting cycle number 39
starting cycle number 40
starting cycle number 41
starting cycle number 42
starting cycle number 43
starting cycle number 44
starting cycle number 45
starting cycle number 46
starting cycle number 47
starting cycle number 48
starting cycle number 49
starting cycle number 50
starting cycle number 51
starting cycle number 52
starting cycle number 53
starting cycle number 54
starting cycle number 55
starting cycle number 56
starting cycle number 57
starting cycle number 58
starting cycle number 59
starting cycle number 60
starting cycle number 61
starting cycle number 62
starting cycle number 63
starting cycle number 64
starting cycle number 65
starting cycle number 66
starting cycle number 67
starting cycle number 68
starting cycle number 69
starting cycle number 70
starting cycle number 71
starting cycle number 72
starting cycle number 73
starting cycle number 74
starting cycle number 75
starting cycle number 76
starting cycle number 77
starting cycle number 78
starting cycle number 79
starting cycle number 80
starting cycle number 81
starting cycle number 82
starting cycle number 83
starting cycle number 84
starting cycle number 85
starting cycle number 86
starting cycle number 87
starting cycle number 88
starting cycle number 89
starting cycle number 90
starting cycle number 91
starting cycle number 92
starting cycle number 93
starting cycle number 94
starting cycle number 95
starting cycle number 96
starting cycle number 97
starting cycle number 98
starting cycle number 99
building features...
updating members...
Clock time: 1040.6910407543182
```

```
In [61]: import matplotlib.pylab as plt
         import numpy as np
         fig,ax = plt.subplots()
         clock_time_as_min = {t:[ti/60.0 for ti in clock_time[t]] for t in clock_time.keys()}
         x = list(clock_time_as_min.keys())
         x.sort()
         ax.bar(x=[1,2,3],height=[np.mean(clock_time_as_min[time]) for time in x],
                yerr=[np.std(clock_time_as_min[time]) for time in x],
                capsize=10,alpha=0.6)
         ax.set_xlabel('Ensemble members',size=16)
         ax.set_ylabel('Ensemble construction time (min)',size=16)
         ax.tick_params(axis='both', which='major', labelsize=12)
         ax.tick_params(axis='both', which='minor', labelsize=12)
         ax.set(xticks=[1,2,3])
         ax.set_xticklabels(labels=x)
         plt.savefig('benchmark_iter_gapfill.svg')
```



```
In [63]: # what are the mean values?
         [np.mean(clock_time_as_min[time]) for time in x]

Out[63]: [1.789455144802729, 5.327363805770874, 18.61842150807381]
```

As you can see, ensemble size and construction time are fairly proportional. There is a large amount of time spent copying the universal model, so contructing the small 5-member ensemble takes longer than one might expect. A larger universal model will take longer to copy, and each gapfilling step will take longer if more media conditions are included.

1.13 FAQ

```
In [1]: import medusa
```

1.14 Sphinx AutoAPI Index

This page is the top-level of your generated API documentation. Below is a list of all items that are documented here.

1.14.1 `medusa`

Subpackages

`medusa.core`

Submodules

`medusa.core.ensemble`

Module Contents

`medusa.core.ensemble.`**`REACTION_ATTRIBUTES = ['lower_bound', 'upper_bound']`**

`medusa.core.ensemble.`**`MISSING_ATTRIBUTE_DEFAULT`**

`class` `medusa.core.ensemble.`**`Ensemble`** (*list_of_models=[]*, *identifier=None*, *name=None*)

Bases:`cobra.core.object.Object`

Ensemble of metabolic models

Parameters

- **`identifier`** (*`string`*) – The identifier to associate with the ensemble as a string.
- **`list_of_models`** (*`list of cobra.core.model.Model`*) – Either a list of existing Model objects in which case a new Model object is instantiated and an ensemble is constructed using the list of Models, or None/empty list, in which case an ensemble is created with empty attributes.
- **`name`** (*`string`*) – Human-readable name for the ensemble

`base_model`

A cobra.core.Model that contains all variable and invariable components of an ensemble.

Type Model

`members`

A DictList where the key is the member identifier and the value is a medusa.core.member.Member object

Type DictList

features
A DictList where the key is the feature identifier and the value is a medusa.core.feature.Feature object

Type DictList

_populate_features_base (*self*, *list_of_models*)

_populate_members (*self*, *list_of_models*)

set_state (*self*, *member*)
Set the state of the base model to represent a single member.

Sets all features to the state for the provided member. Only reaction states are currently implemented (e.g. GPRs as features will not work)

Parameters **member** (*str or medusa.Member*) – The Member.id, or the Member object itself, to set the state of the Ensemble.base_model to represent.

to_pickle (*self*, *filename*)
Save an ensemble as a pickled object. Pickling is currently the only supported method for saving and loading ensembles.

Parameters **filename** (*String*) – location to save the pickle.

extract_member (*self*, *member*)
Extract an individual member as a cobrapy model (cobra.Model), removing any components associated with features that are inactive in member.

Provided as a more convenient option than medusa.Member.to_model(), but is the exact same.

Parameters **member** (*str or medusa.Member*) – The Member.id, or the Member object itself, to be represented in the cobrapy model output.

Returns **model** – The extracted member as a cobrapy model.

Return type cobra.Model

medusa.core.feature

Module Contents

class medusa.core.feature.**Feature** (*identifier=None*, *name=None*, *ensemble=None*, *base_component=None*, *component_attribute=None*, *states=None*)
Bases:cobra.core.object.Object

Feature describing a network component that varies across an ensemble.

Parameters

- **identifier** (*string*) – The identifier to associate with the feature. Convention is to append the component_attribute to the base_component's id.
- **ensemble** (*medusa.core.ensemble.Ensemble object*) – The ensemble that the feature is associated with.
- **base_component** (*cobra.core.reaction.Reaction*) – Reference to the Reaction object that the feature describes.
- **component_attribute** (*string*) – string indicating the attribute of base_component that the feature describes the modification of (e.g. "lb", "ub")

- **states** (*dictionary of string:component_attribute value*) – dictionary of model ids mapping to the value of the Feature's component_attribute (value type depends on component_attribute type, e.g. float for "lb", string for "_gene_reaction_rule")

get_model_state(*self*, *member_id*)
Get the state of the feature for a particular member

medusa.core.member

Module Contents

class medusa.core.member.**Member**(*ensemble=None*, *identifier=None*, *name=None*, *states=None*)
Bases:`cobra.core.object.Object`

Object representing an individual member (i.e. model) in an ensemble

Parameters

- **identifier** (*string*) – The identifier to associate with the member.
- **ensemble** (*medusa.core.ensemble.Ensemble object*) – The ensemble that the member belongs to.
- **states** (*dictionary of medusa.core.feature.Feature:component_attribute value*) – dictionary of Features mapping to the value of the Feature's component_attribute (value type depends on component_attribute type, e.g. float for "lb", string for "_gene_reaction_rule") for the member.

to_model(*self*)
Generate a cobra.Model object with the exact state of this member.

The resulting cobra.Model does not contain any Metabolites, Genes, or Reactions that were inactive in the member.

Returns **model** – The extracted member as a cobrapy model.

Return type cobra.Model

_set_id_with_model(*self*, *value*)

Package Contents

class medusa.core.**Ensemble**(*list_of_models=[]*, *identifier=None*, *name=None*)
Bases:`cobra.core.object.Object`

Ensemble of metabolic models

Parameters

- **identifier** (*string*) – The identifier to associate with the ensemble as a string.
- **list_of_models** (*list of cobra.core.model.Model*) – Either a list of existing Model objects in which case a new Model object is instantiated and an ensemble is constructed using the list of Models, or None/empty list, in which case an ensemble is created with empty attributes.
- **name** (*string*) – Human-readable name for the ensemble

base_model
A cobra.core.Model that contains all variable and invariable components of an ensemble.

Type Model

members

A DictList where the key is the member identifier and the value is a medusa.core.member.Member object

Type DictList

features

A DictList where the key is the feature identifier and the value is a medusa.core.feature.Feature object

Type DictList

_populate_features_base (*self*, *list_of_models*)

_populate_members (*self*, *list_of_models*)

set_state (*self*, *member*)

Set the state of the base model to represent a single member.

Sets all features to the state for the provided member. Only reaction states are currently implemented (e.g. GPRs as features will not work)

Parameters member (*str or medusa.Member*) – The Member.id, or the Member object itself, to set the state of the Ensemble.base_model to represent.

to_pickle (*self*, *filename*)

Save an ensemble as a pickled object. Pickling is currently the only supported method for saving and loading ensembles.

Parameters filename (*String*) – location to save the pickle.

extract_member (*self*, *member*)

Extract an individual member as a cobrapy model (cobra.Model), removing any components associated with features that are inactive in member.

Provided as a more convenient option than medusa.Member.to_model(), but is the exact same.

Parameters member (*str or medusa.Member*) – The Member.id, or the Member object itself, to be represented in the cobrapy model output.

Returns model – The extracted member as a cobrapy model.

Return type cobra.Model

class medusa.core.**Feature** (*identifier=None*, *name=None*, *ensemble=None*, *base_component=None*, *component_attribute=None*, *states=None*)

Bases:cobra.core.object.Object

Feature describing a network component that varies across an ensemble.

Parameters

- **identifier** (*string*) – The identifier to associate with the feature. Convention is to append the component_attribute to the base_component's id.
- **ensemble** (*medusa.core.ensemble.Ensemble object*) – The ensemble that the feature is associated with.
- **base_component** (*cobra.core.reaction.Reaction*) – Reference to the Reaction object that the feature describes.
- **component_attribute** (*string*) – string indicating the attribute of base_component that the feature describes the modification of (e.g. "lb", "ub")

- **states** (*dictionary of string:component_attribute value*) – dictionary of model ids mapping to the value of the Feature's component_attribute (value type depends on component_attribute type, e.g. float for "lb", string for "_gene_reaction_rule")

get_model_state (*self*, *member_id*)
Get the state of the feature for a particular member

class medusa.core.**Member** (*ensemble=None*, *identifier=None*, *name=None*, *states=None*)
Bases:`cobra.core.object.Object`

Object representing an individual member (i.e. model) in an ensemble

Parameters

- **identifier** (*string*) – The identifier to associate with the member.
- **ensemble** (*medusa.core.ensemble.Ensemble object*) – The ensemble that the member belongs to.
- **states** (*dictionary of medusa.core.feature.Feature:component_attribute value*) – dictionary of Features mapping to the value of the Feature's component_attribute (value type depends on component_attribute type, e.g. float for "lb", string for "_gene_reaction_rule") for the member.

to_model (*self*)
Generate a cobra.Model object with the exact state of this member.

The resulting cobra.Model does not contain any Metabolites, Genes, or Reactions that were inactive in the member.

Returns **model** – The extracted member as a cobrapy model.

Return type cobra.Model

_set_id_with_model (*self*, *value*)

medusa.flux_analysis

Submodules

medusa.flux_analysis.deletion

Module Contents

medusa.flux_analysis.deletion.**ensemble_single_reaction_deletion** (*ensemble*, *num_models=None*, *specific_models=[]*)
Performs single reaction deletions on models within an ensemble and returns the objective value after optimization with each reaction removed.

Parameters

- **ensemble** (`medusa.core.Ensemble`) – The ensemble with which to perform reaction deletions
- **num_models** (*int, optional*) – Number of models for which reaction deletions will be performed. The number of models indicated will be randomly sampled and reaction deletions will be performed on the sampled models. If None, all models will be selected

(default), or the models specified by specific_models will be selected. Cannot be passed concurrently with specific_models.

- **specific_models** (*list of str, optional*) – List of member.id corresponding to the models for which reaction deletions will be performed. If None, all models will be selected (default), or num_models will be randomly sampled and selected. Cannot be passed concurrently with num_models.

Returns A dataframe in which each row (index) represents a model within the ensemble, and each column represents a reaction for which values of objective when the reaction is deleted are returned.

Return type pandas.DataFrame

`medusa.flux_analysis.deletion.`**`ensemble_single_gene_deletion`**(*ensemble*, *num_models=None*, *specific_models=[]*, *specific_genes=[]*)

Performs single reaction deletions on models within an ensemble and returns the objective value after optimization with each reaction removed.

Parameters

- **ensemble** (`medusa.core.Ensemble`) – The ensemble with which to perform reaction deletions
- **num_models** (*int, optional*) – Number of models for which reaction deletions will be performed. The number of models indicated will be randomly sampled and reaction deletions will be performed on the sampled models. If None, all models will be selected (default), or the models specified by specific_models will be selected. Cannot be passed concurrently with specific_models.
- **specific_models** (*list of str, optional*) – List of member.id corresponding to the models for which reaction deletions will be performed. If None, all models will be selected (default), or num_models will be randomly sampled and selected. Cannot be passed concurrently with num_models.
- **specific_genes** (*list of str, optionsl*) – List of gene.id corresponding to the genes for which deletions should be performed. If none, all genes will be selected (default). We recommend identifying genes that are essential in all ensemble members first, then excluding those genes from specific_genes. This will generally speed up computation.

Returns A dataframe in which each row (index) represents a model within the ensemble, and each column represents a reaction for which values of objective when the reaction is deleted are returned.

Return type pandas.DataFrame

medusa.flux_analysis.flux_balance

Module Contents

`medusa.flux_analysis.flux_balance.`**`_optimize_ensemble`**(*ensemble*, *return_flux*, *member_id*, ***kwargs*)

`medusa.flux_analysis.flux_balance.`**`_optimize_ensemble_worker`**(*member_id*)

`medusa.flux_analysis.flux_balance.`**`_init_worker`**(*ensemble*, *return_flux*) `medusa.flux_analysis.flux_balance.`**`optimize_ensemble`**(*ensemble*, *return_flux=None*, *num_models=None*, *specific_models=None*, *num_processes=None*, ***kwargs*)

Performs flux balance analysis (FBA) on models within an ensemble.

Parameters

- **`ensemble`** (`medusa.core.Ensemble`) – The ensemble on which FBA is to be performed.
- **`return_flux`** (*`str or list of str, optional`*) – List of reaction ids (cobra.core.reaction.id), or a single reaction id, for which to return flux values. If None, all reaction fluxes are returned (default).
- **`num_models`** (*`int, optional`*) – Number of models for which FBA will be performed. The number of models indicated will be randomly sampled and FBA will be performed on the sampled models. If None, all models will be selected (default), or the models specified by specific_models will be selected. Cannot be passed concurrently with specific_models.
- **`specific_models`** (*`list of str, optional`*) – List of ensemble_member.id corresponding to the models for which FBA will be performed. If None, all models will be selected (default), or num_models will be randomly sampled and selected. Cannot be passed concurrently with num_models.
- **`num_processes`** (*`int, optional`*) – An integer corresponding to the number of processes (i.e. cores) to use. Using more cores will speed up computation, but will have a larger memory footprint because the ensemble object must be temporarily copied for each additional core used. If None, one core is used.

Returns A dataframe in which each row (index) represents a model within the ensemble, and each column represents a reaction for which flux values are returned.

Return type pandas.DataFrame

`medusa.flux_analysis.variability`

Module Contents

`medusa.flux_analysis.variability.`**`ensemble_fva`**(*ensemble*, *reaction_list*, *num_models=[]*, *specific_models=None*, *fraction_of_optimum=1.0*, *loopless=False*, ***solver_args*)

Performs FVA on num_models. If num_models is not passed, performs FVA on every model in the ensemble. If the model is a community model, num_models must be passed.

Performs flux variability analysis (FVA) on models within an ensemble.

Parameters

- **`ensemble`** (`medusa.core.Ensemble`) – The ensemble on which FVA is to be performed.
- **`reaction_list`** (*`str or list of str, optional`*) – List of reaction ids (cobra.core.reaction.id), or a single reaction id, for which to return flux ranges. If None, all reaction fluxes are returned (default).

- **num_models** (*int, optional*) – Number of models for which FVA will be performed. The number of models indicated will be randomly sampled and FVA will be performed on the sampled models. If None, all models will be selected (default), or the models specified by specific_models will be selected. Cannot be passed concurrently with specific_models.
- **specific_models** (*list of str, optional*) – List of ensemble_member.id corresponding to the models for which FVA will be performed. If None, all models will be selected (default), or num_models will be randomly sampled and selected. Cannot be passed concurrently with num_models.
- **fraction_of_optimum** (*float, optional*) – fraction of the optimum objective value, set as a constraint such that the objective never falls below the provided fraction when assessing variability of each reaction.
- **loopless** (*boolean, optional*) – Whether or not to perform loopless FVA. This is much slower. See cobrapy.flux_analysis.variability for details.

Returns A dataframe in which each row (index) represents a model within the ensemble and the lower or upper value of flux ranges, and each column represents a reaction and its lower or upper value of its flux range. Based on this formatting, each model is present in two rows, one of which contains the lower flux value and the other of which contains the upper flux value.

Return type pandas.DataFrame

medusa.quality

Submodules

medusa.quality.mass_balance

Module Contents

`medusa.quality.mass_balance.`**`leak_test`**(*ensemble*, *metabolites_to_test=[]*, *exchange_prefix='EX_'*, *verbose=False*, *num_models=[]*, ***kwargs*)

Checks for leaky metabolites in every member of the ensemble by opening and optimizing a demand reaction while all exchange reactions are closed.

By default, checks for leaks for every metabolite for all models.

medusa.reconstruct

Submodules

medusa.reconstruct.degrade

Module Contents

`medusa.reconstruct.degrade.`**`degrade_reactions`**(*base_model*, *num_reactions*, *num_models=10*)

Removes reactions from an existing COBRA model to generate an ensemble.

Parameters

- **`base_model`** (*`cobra.Model`*) – Model from which reactions will be removed to create an ensemble.
- **`num_reactions`** (*`int`*) – The number of reactions to remove to generate each ensemble member. Must be smaller than the total number of reactions in the model
- **`num_models`** (*`int`*) – The number of models to generate by randomly removing num_reactions from the base_model. Reactions are removed with replacement.

Returns An ensemble

Return type Medusa.core.ensemble

`medusa.reconstruct.expand`

Module Contents

`medusa.reconstruct.expand.`**`REACTION_ATTRIBUTES = ['lower_bound', 'upper_bound']`**

`medusa.reconstruct.expand.`**`MISSING_ATTRIBUTE_DEFAULT`**

`medusa.reconstruct.expand.`**`gapfill_to_ensemble`**(*model, iterations=1, universal=None, lower_bound=0.05, penalties=None, exchange_reactions=False, demand_reactions=False, integer_threshold=1e-06*)

Performs gapfilling on model, pulling reactions from universal. Any existing constraints on base_model are maintained during gapfilling, so these should be set before calling gapfill_to_ensemble (e.g. secretion of metabolites, choice of objective function etc.).

Currently, only iterative solutions are supported with accumulating penalties (i.e. after each iteration, the penalty for each reaction doubles).

Parameters

- **`model`** (*`cobra.Model`*) – The model to perform gap filling on.
- **`universal`** (*`cobra.Model`*) – A universal model with reactions that can be used to complete the model.
- **`lower_bound`** (*`float, 0.05`*) – The minimally accepted flux for the objective in the filled model.
- **`penalties`** (*`dict, None`*) – A dictionary with keys being 'universal' (all reactions included in the universal model), 'exchange' and 'demand' (all additionally added exchange and demand reactions) for the three reaction types. Can also have reaction identifiers for reaction specific costs. Defaults are 1, 100 and 1 respectively.
- **`integer_threshold`** (*`float, 1e-6`*) – The threshold at which a value is considered non-zero (aka integrality threshold). If gapfilled models fail to validate, you may want to lower this value. However, picking a threshold that is too low may also result in reactions being added that are not essential to meet the imposed constraints.
- **`exchange_reactions`** (*`bool, False`*) – Consider adding exchange (uptake) reactions for all metabolites in the model.
- **`demand_reactions`** (*`bool, False`*) – Consider adding demand reactions for all metabolites.

Returns **ensemble** – The ensemble object created from the gapfill solutions.

Return type *medusa.core.Ensemble*

`medusa.reconstruct.expand.`**`iterative_gapfill_from_binary_phenotypes`**(*model*, *universal*, *phenotype_dict*, *output_ensemble_size*, *gapfill_type='continuous'*, *iterations_per_condition=1*, *lower_bound=0.05*, *penalties=None*, *exchange_reactions=False*, *demand_reactions=False*, *inclusion_threshold=1e-06*, *exchange_prefix='EX_'*)

Performs gapfilling on model, pulling reactions from universal. Any existing constraints on base_model are maintained during gapfilling, so these should be set before calling gapfill_to_ensemble (e.g. secretion of metabolites, choice of objective function etc.).

Cycles through each key:value pair in phenotype_dict, iterating over every condition and performing gapfilling on that condition until the number of cycles over all conditions is equal to output_ensemble_size. For each cycle, the order of conditions is randomized, which generally leads to unique sets of solutions for each cycle.

Currently only supports a single iteration for each condition within each cycle (i.e. for each gapfill in a single condition, only one solution is returned). Currently only supports gapfilling to positive growth conditions.

Generally, solutions are easier to find and more likely to exist if users ensure that transporters for metabolites exist in the model already or at minimum are present in the universal model.

Parameters

- **`model`** (*`cobra.Model`*) – The model to perform gap filling on.
- **`universal`** (*`cobra.Model`*) – A universal model with reactions that can be used to complete the model.
- **`phenotype_dict`** (*`dict`*) – A dictionary of condition_name:media_dict, where condition name is a unique string describing the condition (such as the name of a single carbon source) and media_dict is a dictionary of exchange reactions to bounds, as set in cobra.core.model.medium. Exchange reactions are provided with reaction.id, not cobra.core.reaction objects.
- **`output_ensemble_size`** (*`int`*) – Number of cycles over all conditions provided to perform. Equal to the number of lists returned in 'solutions'. When the ensemble is constructed, the number of members may be lower than output_ensemble_size if any duplicate solutions were found across cycles.
- **`iterations_per_condition`** (*`int, 1`*) – The number of gapfill solutions to return in each condition within each cycle. Currently only supports returning a single solution.
- **`lower_bound`** (*`float, 0.05`*) – The minimally accepted flux for the objective in the filled model.

- **penalties** (*dict, None*) – A dictionary with keys being 'universal' (all reactions included in the universal model), 'exchange' and 'demand' (all additionally added exchange and demand reactions) for the three reaction types. Can also have reaction identifiers for reaction specific costs. Defaults are 1, 100 and 1 respectively.
- **inclusion_threshold** (*float, 1e-6*) – The threshold at which a value is considered non-zero (aka integrality threshold in the integer formulation, or the flux threshold in the continuous formulation). If gapfilled models fail to validate, you may want to lower this valu. However, picking a threshold that is too low may also result in reactions being added that are not essential to meet the imposed constraints.
- **exchange_reactions** (*bool, False*) – Consider adding exchange (uptake) reactions for all metabolites in the model.
- **demand_reactions** (*bool, False*) – Consider adding demand reactions for all metabolites.
- **exchange_prefix** (string, "**EX_**") – the default reaction ID prefix to search for when identifying exchange reactions. "**EX_**" is standard for modelSEED models. This will be updated to be more database-agnostic when cobrapy boundary determination is finalized for cobrapy version 1.0.

Returns **solutions** – list of lists; each list contains a gapfill solution for a single cycle. Number of lists is equal to output_ensemble_size.

Return type list

`medusa.reconstruct.expand.`**`_continuous_iterative_binary_gapfill`**(*model, phenotype_dict, cycle_order, universal=None, output_ensemble_size=1, lower_bound=0.05, penalties=None, demand_reactions=False, exchange_reactions=False, flux_cutoff=1e-08, exchange_prefix='EX_'*)

`medusa.reconstruct.expand.`**`_integer_iterative_binary_gapfill`**(*model, phenotype_dict, cycle_order, universal=None, output_ensemble_size=0, lower_bound=0.05, penalties=False, demand_reactions=False, exchange_reactions=False, integer_threshold=1e-06*)

`medusa.reconstruct.expand.`**`_build_ensemble_from_gapfill_solutions`**(*model, solutions, universal=None*) `medusa.reconstruct.expand.`**`validate`**`(`*original_model*, *reactions*, *lower_bound*`)`

`medusa.reconstruct.load_from_file`

Module Contents

`medusa.reconstruct.load_from_file.`**`parent_attr_of_base_component`**`(`*base_comp*`)`

Output a string to indicate the parent attribute of the cobra.core object.

Parameters **`base_comp`** (*`cobra.core object`*) – Ensemble base_component of feature. i.e. cobra reaction, metabolite, or gene

`medusa.reconstruct.load_from_file.`**`batch_load_from_files`**`(`*model_file_names*, *identifier='ensemble'*, *batchsize=5*, *verbose=False*`)`

Loads a list of models my file name in batches to and generates an ensemble object. This function is meant to be used to limit how much flash memory is required to generate very large ensembles.

Parameters

- **`model_jsons`** (*`List`*) – List of json cobra.model file names.
- **`batchsize`** (*`Integer`*) – Total number of models loaded into memory.

`medusa.reconstruct.load_from_file.`**`add_ensembles`**`(`*e1*, *e2*, *verbose=False*`)`

Adds two ensemble objects together.

Parameters **`& e2`** (*`e1`*) – Generated using medusa.core.Ensemble()

`medusa.test`

Submodules

`medusa.test.test_ensemble`

Module Contents

`medusa.test.test_ensemble.`**`REACTION_ATTRIBUTES = ['lower_bound', 'upper_bound']`**

`medusa.test.test_ensemble.`**`MISSING_ATTRIBUTE_DEFAULT`**

`medusa.test.test_ensemble.`**`construct_textbook_ensemble`**`()`

`medusa.test.test_ensemble.`**`construct_mixed_ensemble`**`()`

`medusa.test.test_ensemble.`**`test_ensemble_creation`**`()`

`medusa.test.test_ensemble.`**`test_mixed_ensemble_creation`**`()`

`medusa.test.test_ensemble.`**`test_extract_member`**`()`

`medusa.test.test_ensemble.`**`test_update_member_id`**`()`

`medusa.test.test_ensemble.`**`test_pickle`**`()`

`medusa.test.test_flux_balance`

Module Contents

medusa.test.test_flux_balance.**construct_textbook_ensemble**()

medusa.test.test_flux_balance.**construct_mixed_ensemble**()

medusa.test.test_flux_balance.**test_fba_return_dims**()

medusa.test.test_flux_balance.**test_fba_multiprocessing**()

medusa.test.test_flux_balance.**test_fba_single_return**()

medusa.test.test_flux_balance.**test_fba_random_sample**()

medusa.test.test_flux_balance.**test_fba_specific_models**()

`medusa.test.test_load_from_file`

Module Contents

medusa.test.test_load_from_file.**REACTION_ATTRIBUTES = ['lower_bound', 'upper_bound']**

medusa.test.test_load_from_file.**MISSING_ATTRIBUTE_DEFAULT**

medusa.test.test_load_from_file.**construct_mixed_ensemble_2**()

medusa.test.test_load_from_file.**construct_mixed_batch_ensemble**()

medusa.test.test_load_from_file.**test_batch_load_vs_innate**()

medusa.test.test_load_from_file.**test_all_attributes_in_batch_load_model**()

`medusa.test.test_reconstruct`

Module Contents

medusa.test.test_reconstruct.**REACTION_ATTRIBUTES = ['lower_bound', 'upper_bound']**

medusa.test.test_reconstruct.**MISSING_ATTRIBUTE_DEFAULT**

medusa.test.test_reconstruct.**test_iterative_gapfill_from_binary_phenotypes**()

medusa.test.test_reconstruct.**load_universal_modelseed**()

medusa.test.test_reconstruct.**load_modelseed_model**(*model_name*)

`medusa.test.test_variability`

Module Contents

medusa.test.test_variability.**construct_textbook_ensemble**()

medusa.test.test_variability.**test_fva_return_dims**()

Package Contents

class medusa.test.**Ensemble**(*list_of_models=[]*, *identifier=None*, *name=None*)
Bases:`cobra.core.object.Object`

Ensemble of metabolic models

Parameters

- **identifier** (*string*) – The identifier to associate with the ensemble as a string.
- **list_of_models** (*list of cobra.core.model.Model*) – Either a list of existing Model objects in which case a new Model object is instantiated and an ensemble is constructed using the list of Models, or None/empty list, in which case an ensemble is created with empty attributes.
- **name** (*string*) – Human-readable name for the ensemble

base_model
A cobra.core.Model that contains all variable and invariable components of an ensemble.

Type Model

members
A DictList where the key is the member identifier and the value is a medusa.core.member.Member object

Type DictList

features
A DictList where the key is the feature identifier and the value is a medusa.core.feature.Feature object

Type DictList

_populate_features_base(*self*, *list_of_models*)

_populate_members(*self*, *list_of_models*)

set_state(*self*, *member*)
Set the state of the base model to represent a single member.

Sets all features to the state for the provided member. Only reaction states are currently implemented (e.g. GPRs as features will not work)

Parameters **member** (*str or medusa.Member*) – The Member.id, or the Member object itself, to set the state of the Ensemble.base_model to represent.

to_pickle(*self*, *filename*)
Save an ensemble as a pickled object. Pickling is currently the only supported method for saving and loading ensembles.

Parameters **filename** (*String*) – location to save the pickle.

extract_member(*self*, *member*)
Extract an individual member as a cobrapy model (cobra.Model), removing any components associated with features that are inactive in member.

Provided as a more convenient option than medusa.Member.to_model(), but is the exact same.

Parameters **member** (*str or medusa.Member*) – The Member.id, or the Member object itself, to be represented in the cobrapy model output.

Returns **model** – The extracted member as a cobrapy model.

Return type cobra.Model

medusa.test.**medusa_directory**

`medusa.test.`**`data_dir`**

`medusa.test.`**`create_test_ensemble`**`(`*ensemble_name='Staphylococcus aureus'*`)`

Returns a previously-generated ensemble for testing model_name: str

> One of 'Staphylococcus_aureus_ensemble'

`medusa.test.`**`create_test_model`**`(`*model_name='textbook'*`)`

Returns a cobra.Model for testing model_name: str

> One of ['Staphylococcus aureus'] or any models in cobra.test

`medusa.test.`**`load_biolog_plata`**`()`

`medusa.test.`**`load_universal_modelseed`**`()`

Package Contents

`medusa.`**`__version__ = 0.1`**

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index