Workshop

AI Applications in Transcription Factor Binding Site and Metabolic Models Analysis

André Borges Farias[1] **e Maria Carolina Sisco**[1]

[1]Laboratório de Bioinformática - Laboratório Nacional de Computação Científica (LNCC)

Email:

bfarias.andre@gmail.com

carolinasisco@gmail.com

Abstract: *This course aims to provide students with a brief explanation of various machine learning algorithms and their applications within a biological context. Subsequently, we will explore how artificial intelligence (AI) has been applied to the study of protein recognition—specifically transcription factors—binding to their respective DNA binding sites (TFBS). Additionally, we will examine the application of AI in the reconstruction of genome-scale metabolic models. The course is divided into two modules to facilitate a structured and organized approach.*
Module 1: *Predictive Modeling of Transcription Factor Binding Sites Using Supervised Algorithms*
Description - Acquisition and preprocessing of TFBS sequences from public databases. Development of supervised machine learning models for TFBS prediction. Application of Shapley values to assess protein-DNA interaction regions.
Module 2: *Application of Deep Learning for Gap-Filling Enhancement (Inference of Missing Metabolic Reactions) in Genome-Scale Metabolic Models*
Description - Fundamental concepts. Genome-scale metabolic models and their applications. Challenges in metabolic model construction. Deep Neural Network-Guided Inference of Reactomes (DNNGIOR) for gap-filling in incomplete metabolic models.

Key words: TFBS; Explainable AI; Metabolic Models; Gap-filling

Contents

1 Setting up the environment

1.1 Enviroment for TFBS predictions

This tutorial will guide you through setting up a Conda environment named `tfbs_unam` with Python 3.8 and installing the required libraries. Additionally, you will create a directory and clone the repository TFBS-Prediction.

1.1.1 Create a Conda Environment

Open a terminal or command prompt and run the following command to create the environment:

```
conda create --name tfbs_unam python=3.8
```

When prompted, confirm the installation by typing `y` and pressing `Enter`.

1.1.2 Activate the Environment

After creating the environment, activate it using:

```
conda activate tfbs_unam
```

1.1.3 Install the Required Libraries

Install the necessary libraries using `conda`:

```
conda install pandas=1.2.4 scikit-learn=0.24.1 matplotlib=3.3.4 \
shap=0.44.1 numpy=1.20.1 -c conda-forge
```

In some cases, Conda may not be able to install all packages due to dependency conflicts or unavailability in its repositories. If you encounter issues, an alternative is to install the packages using `pip`, as explained in the next section. If Conda installation fails, you can install the packages using pip:

```
pip install pandas==1.2.4 scikit-learn==0.24.1 pickle-mixin==1.0.2 \
matplotlib==3.3.4 shap==0.44.1 numpy==1.20.1
```

1.1.4 Create the `Tutorial_UNAM` Folder

Navigate to the `Documents` folder and create the `Tutorial_UNAM` directory.

```
cd ~/Documents
mkdir Tutorial_TFBS_Pred
cd Tutorial_TFBS_Pred
```

1.1.5 Clone the Repository

Clone the repository using `git`:

```
git clone https://github.com/farias-ab/TFBS-Prediction.git
```

If you don't have *git*, please install it using:

```
sudo apt update && sudo apt install git -y
```

1.1.6 Enter the Cloned Repository

After cloning, navigate into the repository folder:

```
cd TFBS-Prediction
```

1.1.7 Install Jupyter Notebook

To run the tutorial, you need Jupyter Notebook. You can install it using either Conda or pip.

To install Jupyter with Conda (Recommended):

```
conda install -c conda-forge notebook -y
```

Installing Jupyter with pip (Alternative): If the Conda installation fails, you can install Jupyter using pip:

```
pip install notebook
```

1.1.8 Open the Jupyter Notebook

Now, open the Jupyter Notebook to run the tutorial file.

```
jupyter-notebook
```

1.2 Gapfilling a Genome Scale Metabolic Model Using Dnngior

Open a terminal or command prompt and run the following command to create the environment:

```
conda create --name dnngior python=3.10.16
```

When prompted, confirm the installation by typing `y` and pressing `Enter`. After that, let's activate our environment

```
conda activate dnngior
```

First, we must install GUROBI optimizer (for further information, you can check on https://support.gurobi.com/hc/en-us/articles/14799677517585-Getting-Started-with-Gurobi-Optimizer).

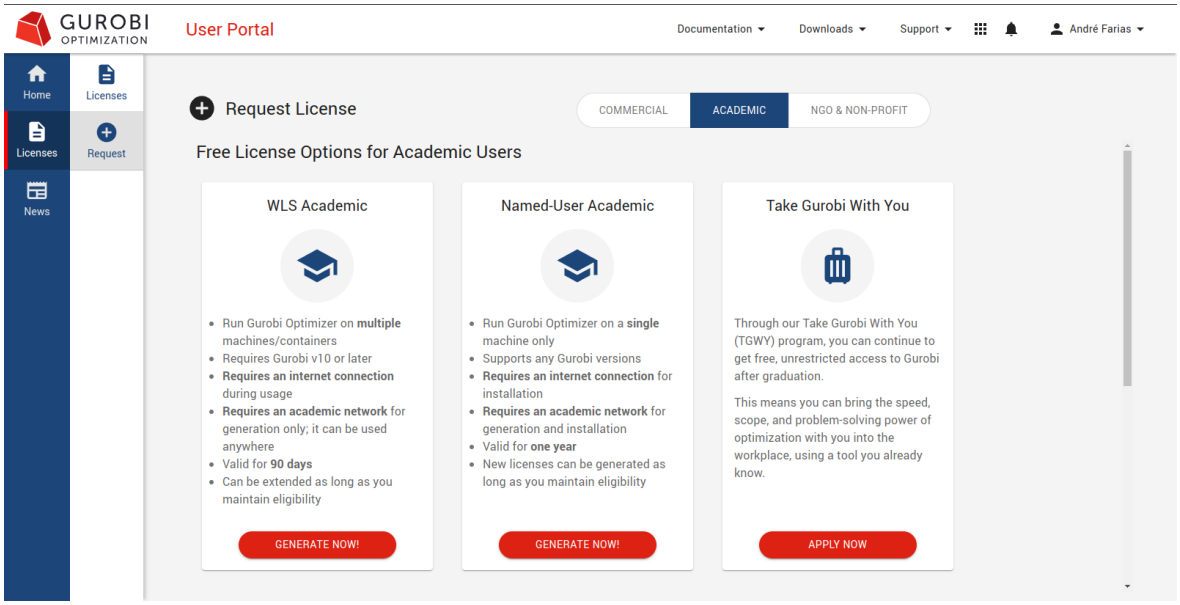
Visit the Download Gurobi Optimizer page (https://www.gurobi.com/downloads/gurobi-software/) and download it. Now, you need to edit the bashrc. Follow the steps bellow:

1. open an terminal;
2. go to root directory using `cd`
3. open the bashrc file. We suggest use gedit, nano or vim;
4. add the following line in the bashrc:

   `export PATH='path-to-gurobi-bin-folder/bin/:$PATH'`

   to locate the `path-to-gurobi-bin-folder`, you have to go to the directory where you have extracted the files from the downloaded tar.gz folder.

Second, we need a Gurobi license, a linear programming solver. You can obtain a free academic named-user-license here https://www.gurobi.com/features/academic-named-user-license/ with your institutional email.



click on named-user-license. You will generate one grbgetkey to your machine. Please, type (changing the X for your respective key):

```
grbgetkey 9f4XXXX-XXXX-XXXX-XXXX-XXXXXXXXXX
```

Now, you will install DNNgior (inside your conda environment) with the following command:

```
pip install dnngior
```

In order to run the dnngior pipeline on a jupyter notebook, you need to install jupyter inside your conda environment with the following command:

```
conda install -c conda-forge notebook -y
or
pip install notebook
```

Open a new notebook typing `jupyter-notebook` on the terminal and test your dnngior installation by typing:

```
import dnngior
```

In some instances, you can find a version inconsistency with numpy, one of the dependencies of dnngior. To fix this, go to the terminal and type:

```
pip install numpy==1.23.5.
```

Then, test your dnngior installation again.

```
Set parameter Username
Set parameter LicenseID to value 2671523
Academic license - for non-commercial use only - expires 2026-05-27
WARNING: To enable the NN_Trainer script, you need to install
 ↪  tensorflow <https://www.tensorflow.org/install>
The rest of dnngior features can be used without it.
```

You should be ready to go!

2 Data availability

The data of models and tutorials are available in github. Please, use the command line to download the entire data for this class:

```
git clone https://github.com/farias-ab/X-meeting_2025.git
git clone https://github.com/farias-ab/TFBS-Prediction.git

git clone https://github.com/carolinasisco/Gapfilling_GSMM_2025.git
```

3 The explainable model for predicting Transcriptor Factor Binding Site (TFBS)

3.1 Predicting Sequences as TFBS or non-TFBS

Activating the Preconfigured Anaconda Environment: First, we will open Anaconda and activate the preconfigured environment. To do this, follow these steps:

```
anaconda
```

Next, activate the environment using:

```
conda activate tfbs_unam
```

If you cannot remember the environment name, simply execute the following command to list all created environments:

```
conda env list
```

Now, open the Jupyter Notebook to run the tutorial file.

```
jupyter-notebook
```

This will open Jupyter Notebook in your web browser. Navigate to the `Tutorial.ipynb` file and open it. Once the Jupyter Notebook is open:

1. Click on **Kernel** in the top menu.
2. Select **Restart & Clear Output**.
3. Confirm by clicking **Restart & Clear All Outputs**.

Now, let's execute each cell one by one:

1. Click on the first cell.
2. Press **Shift + Enter** to execute it.
3. Move to the next cell and repeat until all cells have been executed.

3.2 The case study of unknown sequences

Thus far, we have only been predicting whether a given sequence is a TFBS or non-TFBS. Now, let us consider a scenario where we have two nucleotide sequences of variable length and wish to perform prediction and identification of these sequences as TFBS. How would we approach this?

To predict transcription factor binding sites (TFBS), we need to split sequences into smaller fragments to efficiently scan for binding sites. We will use the k-mer approach.

What is the k-mer strategy?

The k-mer methodology is a way to divide a biological sequence, such as DNA, into small fixed-size segments called **k-mers**.

How does it work?

1. **Choose the k-mer size**: The parameter k defines the fragment size. For example, if $k = 3$, each segment will contain 3 nucleotides.
2. **Slide through the sequence**: Move along the DNA sequence, extracting segments of size k by shifting one nucleotide at a time.

Example with k=3 and sequence "ATGCGT"

- 1st k-mer: `ATG`
- 2nd k-mer: `TGC`
- 3rd k-mer: `GCG`
- 4th k-mer: `CGT`

But first of all, lets create our directory:

```
cd ~/Documents/Tutorial_UNAM
mkdir Predictions
cd Predictions
```

After that, let's create a jupyter notebook to write our codes.

```
jupyter-notebook
```

Generating k-mers from a Sequence and Saving as CSV: The following code demonstrates how to generate k-mers from a DNA sequence and save each k-mer as an individual CSV file.

```
import pandas as pd

# Define the DNA sequence
seq1 = "GTCTACCTCATCATAAATGAATAGTCATGAAGACTTTGGTTG\
CTTTAACGGCGTTGTGCAAGGGGAGATAGCATCAAAAAATCGCCACTT\
TGCGCAGGAATGGAGCGAAAGGGATGAAAAATCAACAAACAGAAAAAAG\
ATCCAAAAAACGCTTGTGCAAAAAAATGGGATCCCTATAATGCGCCTCCA\
TCGACACGGCGGA"

# Lets check the size of this sequence
len(seq1)
```

```
# We know our sequence has 202 nucleotides
seq1 = seq1[:202]

# We decided to work with kmer size of 14
k = 14

kmers = [seq1[i:i+k] for i in range(len(seq1) - k + 1)]

kmer_seq1 = pd.DataFrame(kmers, columns=["Kmer"])

print(kmer_seq1)
```

```
# Save each k-mer as an individual CSV file
for index, row in kmer_seq1.iterrows():
    filename = f"kmer_{index + 1}.csv"
    row_df = pd.DataFrame([row])
    row_df.to_csv(filename, index=False)

print("CSV files successfully generated!")
```

This script:

- Defines a DNA sequence .
- Generates k-mers of size 14.
- Saves each k-mer into an individual CSV file (`kmer_1.csv`, `kmer_2.csv`, etc.).

Let us organize the generated files into a new directory named *kmer*.

```
mkdir kmer && mv kmer* ./kmer
```

Now that we have nucleotide sequences of uniform length, we will convert these sequences into *features* for the model—that is, numerical variables that will enable the model to identify patterns and make predictions.

3.2.1 Converting sequences into DDS values

Now we will process DNA sequences to calculate their **DDS (DNA Duplex Stability)** values. The DDS value for each dinucleotide (two nucleotides) is determined based on the following stability dictionary:

```
# DDS values (DNA Duplex Stability)
stability_values = {
    'AA': -1,    'AT': -0.88, 'TA': -0.58, 'AG': -1.3,
    'GA': -1.3,  'TT': -1,    'AC': -1.45, 'CA': -1.45,
    'TG': -1.44, 'GT': -1.44, 'TC': -1.28, 'CT': -1.28,
    'CC': -1.84, 'CG': -2.24, 'GC': -2.27, 'GG': -1.84
}
```

We will now configure the input and output data directories for our conversion process.

```
import os
# Path for input k-mer sequence files
path = 'kmer/'
# Path for output DDS values
path_out = './kmer/dds'
```

Processing the files and generating the DDS values: This step allows us to transform raw DNA sequences into numerical representations that can be used for further analysis in TFBS prediction.

```
for filename in os.listdir(path):
    if not os.path.isdir(os.path.join(path, filename)):
        with open(path+filename, "r") as file:
            vetor_dados= file.readlines()
            vetor_dados = [line3.strip() for line3 in vetor_dados]
            print(filename)
            for line in vetor_dados:
                if ">" not in line:
                    line_len = len(line)

                    stability_list =
                     ↪  [stability_values.get(line[i:i+2].upper(),
                                   0) for i in range(0,
                                    ↪  len(line)-1)]
                    count = 0
                    with open(path_out+filename, "a+") as out:
                        for item in stability_list:
                            out.write(str(item)+"\t")
                            count+=1
                            print(count)
                            if count == line_len-1:
                                out.write("\n")
                                count = 0
```

Let's analyze the result of a conversion.

```
cat ./kmer/ddskmer_1.csv
0        0        0
-1.44 -1.28 -1.28 -0.58 -1.45 -1.84 -1.28 -1.28 -1.45 -0.88 -1.28
 ↪ -1.45 -0.88
```

As is typical in FASTA format (where headers begin with the '>' symbol), the code will assign zeros to the first line. To simply remove these values, we will use the following command:

```
!sed -i '1d' ./kmer/ddskmer_*.csv
```

```
cat ./kmer/ddskmer* > ./kmer/dds_all.csv
```

Now, let's load the generated file containing all the dds values.

```
# Reading the DDS file
dds1 = pd.read_csv('kmer/dds_all.csv', header=None, sep='\t')

# Removing the 14th column (index 13)
dds1 = dds1.drop(dds1.columns[13], axis=1)

# Displaying the first rows
dds1.head()
```

3.2.2 Selecting the Appropriate Model for our prediction

We need to define the model to be used. In this case, we will load the models for 13 base pairs. You can check the model's expected input dimensions using the following command:

```
print(dds1.shape[1])
13
```

This code prints the number of columns in `dds1`, which corresponds to the features available for the model. Ensure that this aligns with the expected input for the 13-base pair model. Here, you need to change the file path to select the model suitable for your dataset. Use the following code to load the model:

```
import pickle
with open('../../TFBS-Prediction/Models/TFBS_classification/\
13_bps/rf13', 'rb') as f:
    rf13 = pickle.load(f)
```

Make sure to adjust the path to match the location of your model file. This code will load the Random Forest model designed for 13 base pairs, allowing you to use it for classification tasks.

3.2.3 Making Predictions

Now, we will make predictions using the loaded model:

```
# Predicting TFBS classification for sequence 1
y_pred13 = rf13.predict(dds1)

# Storing results in a dictionary
results = {
    "Sequence": [],
    "Classification": []
            }

for i in range(len(y_pred13)):
    results["Sequence"].append(kmer_seq1['Kmer'][i])
    if y_pred13[i] == 0:
        results["Classification"].append("nonTFBS")
    else:
        results["Classification"].append("TFBS")

# Converting dictionary to DataFrame and saving results
results = pd.DataFrame(results)
results.to_csv("tfbs_prediction_seq.csv", index=False)
results
```

The predictions will be saved in a CSV file (`tfbs_prediction_seq1.csv`), containing the sequences and their respective classification as `TFBS` or `nonTFBS`.

Expected Output - After running the classification, we expect to see an output similar to the following:

```
[82]: Sequence          Classification
0     GTCTACCTCATCAT    nonTFBS
1     TCTACCTCATCATA    nonTFBS
2     CTACCTCATCATAA    nonTFBS
3     TACCTCATCATAAA    nonTFBS
4     ACCTCATCATAAAT    nonTFBS
..    ...               ...
184   CTCCATCGACACGG    nonTFBS
185   TCCATCGACACGGC    nonTFBS
186   CCATCGACACGGCG    nonTFBS
187   CATCGACACGGCGG    nonTFBS
188   ATCGACACGGCGGA    TFBS
[189 rows x 2 columns]
```

This output consists of a table with two columns: one for the sequence and another for its classification as either `TFBS` or `nonTFBS`. The classification is performed based on the trained model, and the results are stored in a CSV file for further analysis. If only TFBS results are relevant for this study, we can filter the output as follows:

```
# Filter the df for rows where Classification is TFBS
results[results['Classification'] == 'TFBS']
```

3.2.4 Identifying Associated Genes

Subsequently, we selected only the regions predicted as TFBS and used the RSAT server (http://embnet.ccg.unam.mx/rsat/index.php) to predict the genes associated with these TFBS. By submitting the predicted TFBS sequences to the RSAT server, we can identify potential target genes regulated by these binding sites, providing further insights into their biological significance. Below is a screenshot of the RSAT interface showing the steps to submit the predicted TFBS sequences for gene prediction:

Figure 1 – Step-by-step submission of predicted TFBS sequences on the RSAT website for gene prediction.

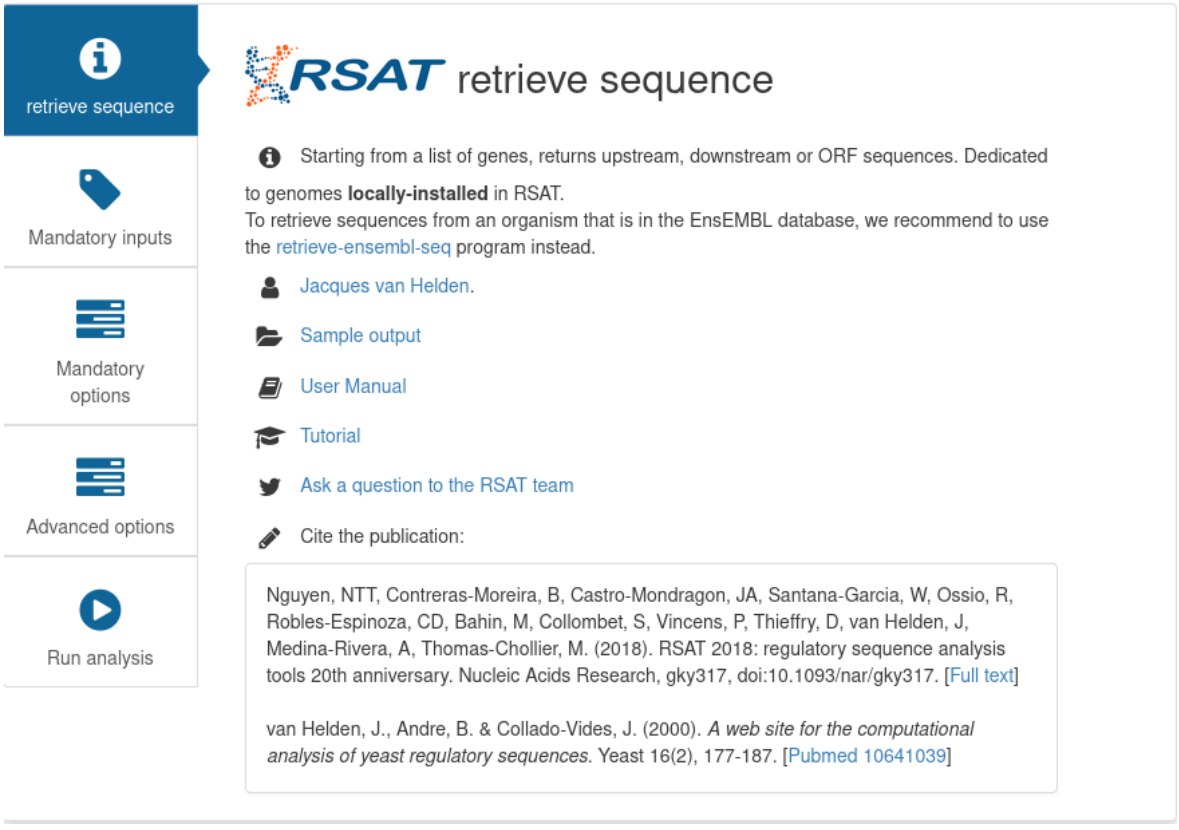


Figure 2 – Step-by-step submission of predicted TFBS sequences on the RSAT website for gene prediction.

3.2.5 Expected Output

We aim to construct a results table containing the predicted TFBS regions and their associated genomic annotations, following the structure presented below:

Table 1 – Table showing predicted TFBS regions and associated information.

PatID	Strand	Pattern	SeqID	Start	End	matching_seq	Score
TCATCATAAATGAA	R	TCATCATAAATGAA	yaaH	-18	-5	TCATCATAATTGAA	0.93
TCATCATAAATGAA	R	TCATCATAAATGAA	STM0015	-24	-11	TCATTATAAATGAA	0.93
ATCATAAATGAATA	R	ATCATAAATGAATA	STM0015	-26	-13	ATTATAAATGAATA	0.93
GGGAGATAGCATCA	R	GGGAGATAGCATCA	imp	-226	-213	GGGAGATAGCCTCA	0.93
GGGAGATAGCATCA	D	GGGAGATAGCATCA	djlA	-30	-17	GGGAGATAGCCTCA	0.93
AATGCGCCTCCATC	D	AATGCGCCTCCATC	rrsH	-304	-291	AATGCGCCTCCATC	1.0
ATCGACACGGCGGA	D	ATCGACACGGCGGA	rrsH	-293	-280	ATCGACACGGCGGA	1.0
AATAGTCATGAAGA	D	AATAGTCATGAAGA	STM0359	-89	-76	AATAGTCTTGAAGA	0.93
ATCAAAAAATCGCC	D	ATCAAAAAATCGCC	prpR	-19	-6	AACAAAAAATCGCC	0.93
CAAAAAATCGCCAC	D	CAAAAAATCGCCAC	prpR	-17	-4	CAAAAAATCGCCCC	0.93
ATCAAAAAATCGCC	R	ATCAAAAAATCGCC	prpB	-231	-218	AACAAAAAATCGCC	0.93
CAAAAAATCGCCAC	R	CAAAAAATCGCCAC	prpB	-233	-220	CAAAAAATCGCCCC	0.93
GGGAGATAGCATCA	R	GGGAGATAGCATCA	STM1331	-101	-88	GGGAGATTGCATCA	0.93
TTGTGCAAAAAAAT	D	TTGTGCAAAAAAAT	STM1527	-64	-51	CTGTGCAAAAAAAT	0.93
TGTGCAAAAAAATG	D	TGTGCAAAAAAATG	STM1527	-63	-50	TGTGCAAAAAAATG	1.0

4 Gapfilling Using DNNgior

Short tutorial (adapted from Boer et al. 2024, see below) by Maria Carolina Sisco.

Improving genome-scale metabolic models of incomplete genomes with deep learning Boer et al. 2024. DOI: 10.1016/j.isci.2024.111349

- DNNgior: Deep Neural Network Guided Imputation of Reactomes
- GSMM: Genome Scale Metabolic Model
- DNNgior uses AI to improve gap-filling by learning from the presence and absence of metabolic reactions across diverse bacterial genomes

4.1 Gapfilling Using a Complete Medium

In this exercise we will gapfill (adding missing reactions) a GSMM of *Blautia*, a genus of anaerobic bacteria with probiotic characteristics.

Open a terminal and activate the dnngior conda environment. Type in:

```
source \{path-to-miniconda3-bin-folder}/activate
conda activate dnngior
jupyter-notebook
```

On the Jupyter home page, click on new and then on Python 3 ipykernel.

The necessary files for this exercise are storaged on the course github. Please, check the section 2.

The files must be on the same folder as the notebook(home folder in this case). If not, you need to write the full path to them.

Let's explore the GSMM with some basic Cobrapy commands!

```
import cobra
from cobra.io import read_sbml_model
draft_reconstruction = read_sbml_model('bh_ungapfilled_model.sbml')
```

```
draft_reconstruction.summary()
```

Notice the objective function (biomass=growth rate). Also, there are no fluxes.

```
draft_reconstruction.optimize()
```

```
draft_reconstruction.medium
```

The exchange reactions (uptake reactions) are set to a unlimited value, there is no constraint regarding what the bacteria can take from the medium, but even so, our model can't simulate growth.

Let's start gapfilling our model! Import the dnngior library and use the Gapfill class to gapfill the reconstruction

```
import os, sys
path_to_blautia_model = ("bh_ungapfilled_model.sbml")
```

```
import dnngior
gapfilled_model_complete = dnngior.Gapfill(draftModel =
 ↪  path_to_blautia_model, medium = None, objectiveName = 'bio1')
```

Make a new object of the gapfilled model

```
gf_model_compl_med = gapfilled_model_complete.gapfilledModel.copy()
```

```
gf_model_compl_med.optimize()
```

It's growing!, the growth rate after optimization is: 146.138 mmol/gDW/ hr (Millimoles per gram dry cell weight per hour), the default flux units used in FBA

Now let's see how many and which reactions DNNgior added in order to simulate growth

```
print("Number of reactions added:", len(gapfilled_model_complete.,
 ↪  added_reactions))
print("~~")
for reaction in gapfilled_model_complete.added_reactions:
        print(gf_model_compl_med.reactions.get_by_id(reaction).name)
```

```
gf_model_compl_med.reactions.get_by_id('EX_cpd15432_e0')
```

```
gf_model_compl_med.reactions.get_by_id('EX_cpd15511_e0')
```

4.2 Gapfilling Using a Defined Medium

First, load the media file containing the composition of the medium

```
medium_file_path = 'Nitrogen-Nitrite_media.tsv'
```

```
import pandas as pd
new_medium = pd.read_csv(medium_file_path, sep="\t")
new_medium.head()
```

Let's gapfill our GSMM so it can growth on this medium

```
gapfill_nitr = dnngior.Gapfill(path_to_blautia_model, medium_file =
 ↪  medium_file_path, objectiveName = 'bio1')
```

Again, make a new object of the gapfilled model and check if it's growing.

```
gf_model_Nit_med = gapfill_nitr.gapfilledModel.copy()
gf_model_Nit_med.optimize()
```

Let's see how many and which reactions DNNgior added in order to simulate growth on the nitrite media

```
print("Number of reactions added:",
 ↪  len(gapfill_nitr.added_reactions))
print("~~")
#for reaction in gapfill_nitr.added_reactions[:5]:
for reaction in gapfill_nitr.added_reactions:
        print(gf_model_Nit_med.reactions.get_by_id(reaction).name)
```

Agradecimentos

Our research is supported by the National Council for Scientific and Technological Development (CNPq) grant no. 305895/2022-2 and by the Rio de Janeiro State Research Support Foundation (FAPERJ) grant FAPERJ-CNE no. E-26/200.555/2023