# Report for FYS-STK4155: Project 3

Laurent Fontaine, Maziar Kosarifar, Maksym Brilenkov and Knut Hauge Engvik

December 2019

**Abstract**

Datasets in several fields of research are characterized by low numbers of observations in proportion to the number of predictors, as is the case of microbial ecology. In this study, a dataset comprised of 72 observations and 15000 predictors was analyzed using Random Forests, XGboost, Siamese and Feed-Forward Neural Networks. While first first three were used for both regression and classification, the latter was used only for classification. The regression attempts involved either a full dataset as input or a dimensionally-reduced one. In both cases, XGboost performed well in terms of execution time and accuracy, while the feed-forward neural network yielded accurate predictions, although at the cost of greatly increased execution time. In the case of using the full data as input, XGboost outperformed random forests slightly in terms of accuracy and substantially in execution time. To explore classification with random forests and siamese networks, the different sites were grouped into two classes. While the networks certainly showed signs of learning, getting good results on the training data, they displayed little to no ability to generalize. These results suggest XGboost is well suited for datasets where the number of predictors greatly exceeds the number of observations and where dimensionality reduction is undesirable.

## Contents

# 1 Introduction

In science, each discovery starts with the question and Machine Learning (ML) is not an exception. Therefore, this project is dedicated to end-to-end data analysis, where we decided to describe, use and process the freshwater microbial ecology data sampled from 72 lakes of southern Norway and Sweden as a part of COMSAT project.

The dataset composed of two tables - *bacterial amplicon sequence variants* (ASV) and *environmental metadata* - and overall contains more than 15000 columns, with 72 data points each. The main advantage here is that both tables are matched by observation; thus, both can be used as either input or an output. However, the actual amount of data points limits us in terms of predictive algorithms we can use in order to achieve high accuracy scores.

Overall, the dataset seemed to be an excellent candidate from both scientific and educational point of views. Hence, as our research questions, we decided to (1) identify patterns of variation in bacterial community composition along environmental gradients, (2) predict environmental metadata variables from bacterial community composition and (3) predict presence or absence of bacterial ASVs from environmental metadata.

After careful speculation about the nature of the data set, we presumed that *Random Forest Classifier* together with *Extreme Gradient Boosting* (XGBoost) algorithm are viable methods to address these tasks. Among other benefits, the methods greatly reduce the overfitting, and can help highlight the most relevant features of the areas under study. Moreover, we have also implemented the so-called *Siamese Neural Network* (SNN) described by *Koch at el* [1] [Koch, Siamese Neural Networks for One-shot Image Recognition] using the combination of manual coding and `Keras`[1] - the *Deep Learning* (DP) Library developed for fast and easy Neural Network (NN) implementation. In addition, we also used codes for *Feed Forward Neural Network* (FFNN) developed in project 2 [2] and implemented `Keras` analog to compare the results with Random Forests and XGBoost.

The report organized as follows. In section 2, we briefly introduce the concepts of Random Forests, XGBoost and SNN, and how to work with them. The main results of the pipeline run are presented in section 3. Section 4 is reserved for discussion. The codes are listed in the Appendix.

[1] https://keras.io/

# 2 Methods and algorithms

In this section, we briefly describe the theory behind approaches we have chosen to analyze the the aforementioned data set (for more deep description, please refer to [3]).

## 2.1 Decision tree

*Decision Trees* are the main building blocks for *Random Forests* and are essentially the supervised learning algorithm, which can be used for both classification and regression tasks and which uses a tree-like graph structure to make respective predictions. Hence, the term "tree".

As in the real trees, every decision tree starts with the *root node*, which then grows into *interior nodes* (the test on the attribute) with the final *leaf nodes* (the class label), which are connected by the so-called *branches* (the outcome of the test) [3]. The entire path from root to leaf is the classification rule.

The simplified version of training the tree is as follows [3]:

- Present a dataset containing of a number of training instances characterized by a number of descriptive features and a target feature;
- Train the decision tree model by continuously splitting the target feature along the values of the descriptive features using a measure of information gain during the training process;
- Grow the tree until we accomplish a stopping criteria create leaf nodes which represent the predictions we want to make for new query instances;
- Show query instances to the tree and run down the tree until we arrive at leaf nodes.

### 2.1.1 Regression tree

To grow the *regression* tree we first split the predictor space, $x_1$, $x_2$, ..., $x_p$ into $J$ distinct non-overlapping regions, $R_1$, $R_2$, ...,$R_J$ (which are usually represented by high-dimensional rectangles for simplicity), where we make the same prediction for each observation which falls into the same region [3]. Therefore, out goal is to find such $R_1$, $R_2$, ..., $R_J$, which minimize the MSE

$$\text{MSE} = \sum_{j=1}^{J} \sum_{i} \left(y_i - \bar{y}_{R_j}\right)^2, \tag{2.1}$$

where $\bar{y}_{R_j}$ is the mean response for the training observations within the box $j$.

It is not possible, however, to consider all possible combinations to partition feature space into the boxes; thus, we are using the so-called *top-down* approach [3]: starting at the top

of the tree, we will split the predictor space into two new brunches, look into the best result between the two and then continue splitting again further down the tree (see [3] for more details).

This approach is straightforward, but leads often to overfitting and complicated trees. To avoid this, we are using procedure called *pruning* [3]: once the tree $T_0$ has grown, we remove several sub-nodes in order to obtain a sub-tree. Going even further, we can consider the sequence of trees (defined by non-negative tuning parameter, $\alpha$), instead of every possible sub-tree. This process is called *cost complexity tuning* and mathematically means to make

$$\sum_{m=1}^{\bar{T}} \sum_{i} \left(y_i - \bar{y}_{R_m}\right)^2 + \alpha \bar{T}, \tag{2.2}$$

as low a possible. Here $\bar{T}$ is the number of terminal nodes of the tree $T$, $R_m$ is the the subset of predictor space corresponding to the $m$-th terminal node.

When $\alpha = 0$, the sub-tree $T$ will simply equal $T_0$, but, as $\alpha$ increases, the pruning appears to be in a nested and predictable fashion; thus, resulting in an easily obtainable sequence of sub-trees, as a function of [3]. The value of $\alpha$ can be selected via cross-validation.

### 2.1.2 Classification tree

To grow the *classification* tree we use similar approach as in case of regression. However, instead of MSE used for binary splitting we are now using *classification error rate* [3]

$$p_{mk} = \frac{1}{N_m} \sum_{i} I\left(y_i \neq k\right) = 1 - p_{mk}, \tag{2.3}$$

where we define PDF, $p_{mk}$, to represent the number of observations of a class $k$ in a region $R_m$ with $N_m$ observations.

Alternatively, we can also use either *gini index*

$$g = \sum_{k=1}^{K} p_{mk}\left(1 - p_{mk}\right), \tag{2.4}$$

or *information entropy*

$$s = -\sum_{k=1}^{K} p_{mk} \log p_{mk}, \tag{2.5}$$

as they are more sensitive to node purity.

To set up of decision tree, we are using the *Classification and Regression* (CART) algorithm

[3] [Morten], which splits the data set into two subsets via single feature $k$ and some threshold value $t_k$. It then tries to minimize the *cost function* (see e.g. [4] for more details)

$$C(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}, \tag{2.6}$$

in case of classification. Here $G_{\text{left/right}}$ measures the impurity of the left/right subset and $m_{\text{left/right}}$ is the number of instances in the left/right subset.

In case of regression, the cost function is [3]

$$C(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}}, \tag{2.7}$$

with

$$\text{MSE}_{\text{node}} = \frac{1}{m_{node}} \sum_i \left(\bar{y}_{\text{node}} - y_i\right)^2, \quad \bar{y}_{\text{node}} = \frac{1}{m_{node}} \sum_{i \in \text{node}}, \tag{2.8}$$

### 2.1.3 Pros and Cons

Overall, there are a lot of advantages to use decision trees [3]:

- No feature normalization needed;
- Can handle both continuous and categorical data;
- Can model nonlinear relationships;
- Can model interactions between the different descriptive features;
- Can be displayed graphically, and are easily interpreted;

However, there are also disadvantages [3]:

- Generally do not have the same level of predictive accuracy as some other regression and classification approaches;
- If continuous features are used the tree may become quite large and hence less interpratable;
- Are prone to overfit the training data and hence do not well generalize the data;
- Small changes in the data may lead to a completely different tree;
- Unbalanced datasets may lead to biased trees;
- Features with many levels may be preferred over features with less levels.

In order to increase the predictive performance of trees, the simple approach is to stuck together many decision trees, using methods like random forests and boosting.

## 2.2 Random Forest Classifier

As it was mentioned above, the *Random Forest* is a model based on aggregating many decision trees. The name "random" stems from the two main concepts [3]:

- Random sampling of data points - we build a number of decision trees on bootstrapped training samples. In this way, although each individual tree may have high variance, the entire forest will have lower variance (without bias increase). At test time, predictions are made by averaging the predictions of each decision tree.
- Only a subset of all predictors, $m$, is considered for splitting each node in each decision tree from the full set of predictors, $p$. Usually, it is set to [3]: $m \approx \sqrt{p}$ for classification.

Random forest combines hundreds or thousands of decision trees, trains each one on a different set of the observations and then makes a final prediction based on averaging the predictions on each individual tree. The overall algorithm can be described as follows [3] :

1. For $b = 1$ to $B$:
   a) Draw a bootstrap sample of from the training data organized in our $\mathbf{X}$ matrix.;
   b) We grow then a random forest tree, $T_b$, based on the bootstrapped data by repeating the steps outlined till we reach the maximum node size is reached:
      i. Select $m$ variables at random from the $p$ variables;
      ii. Pick the best variable/split-point among the $m$;
      iii. Split the node into two daughter nodes;
2. Output the ensemble of trees $\{T_b\}_1^B$ and make predictions.

## 2.3 Extreme Gradient Boosting

*Gradient Boosting* (GBoost) is an algorithm, based on a decision tree approach. The key difference here is that it combines weak classifiers to create a good classifier via series of iterations [3]. Mathematically speaking, this means that we define a cost function [3]:

$$C(f) = \sum_{i=0}^{n-1} L\big(y_i, f(x_i)\big), \quad \left(\text{e.g., squared-error function: } \sum_{i=0}^{n-1} \big(y_i - f(x_i)\big)^2\right), \tag{2.9}$$

where $y_i$is our target, $f(x_i)$ is a function to model it, and then try to minimize it using the following algorithm [3]:

1. Initialize the zero estimate of $f_0(x)$;
2. For $m = 0$ to $M$, compute:
   a) compute the negative gradient vector $\mathbf{u}_m = -\partial C(\mathbf{y}, \mathbf{f})/\partial \mathbf{f}(x)$ at $f(x) = f_{m-1}(x)$;
   b) fit the so-called *base-learner* to the negative gradient $h_m(u_m, x)$;
   c) update the estimate $f_m(x) = f_{m-1}(x) + \nu h_m(u_m, x)$;
3. Compute the final estimate: $f_M(x) = \sum_{m=1}^{M} \nu h_m(u_m, x)$.

The work "extreme" implies usage of modern multiprocessing algorithm to make the gradient descent as efficient as possible. All this combines into XGBoost library [5], which is highly scalable, efficient, flexible and portable[3].

## 2.4 Siamese Neural Networks

In project 2, we have already described in a detail the FFNN (see, e.g. [6]). In this project, we are using the more advanced approached based on the hybrid NN - *Siamese Neural Network* (SNN) - which is used when faced with a classification problem and small datasets.

SNN have shown good results when employed in image recognition [7, 8]. The basic premise of these networks is that the feature space is mapped to some metric space where the images of input vectors can be compared. Vectors whose images are “close” in the given metric can be thought of as similar. The goal then, is to train the network to map the different categories to different “areas” of the metric space and, in a sense, learn the concept of sameness.

The name "siamese" stems from the structuring of the network. It consists of a base network, which can have any structure, that acts as a function mapping input vectors into an $N$ dimensional space, where $N$ is the number of output nodes for the base network. For a twin network, two (three for triplets) instances of this base network are created with shared weights and biases. The network can thus take two (or three) different vectors as input and the conjoined base networks will output a corresponding number of $N$ dimensional representations. Then follows a merging layer where the distances between the images of the input vectors are calculated. This output is then fed to a layer that outputs a similarity score between each vector.

One method to train a twin network is to first pick one representative, referred to as an anchor, from each class. Then, for each anchor, $a_i$, and datapoint, $x_j$, the pair, ($a_i$,$x_j$), is created. If the pair belongs to the same category they are labeled 1 and if they belong to different

categories the pair receives the label 0. These pairs, along with their labels, will then serve as training data for the network. The network is then updated according to a loss function that pushes inputs from different classes apart while clustering similar inputs. Hadsell *et al* [7] proposes the *Contrastive Loss* function:

$$y(\text{dist}(a,x)^2 + (1-y)(\max(m - \text{dist}(a,x),0))^2, \tag{2.10}$$

which is basically a combination of two different loss functions. If the label $y = 1$, then the distance is minimized, if $y = 0$, it is maximized up to a margin $m$.

For triplet networks, the method is somewhat similar. For each data point $x_i$, pick one representative from the same category, $a_s$, and one representative from a different category, $a_d$. Then, update the shared network according to the *Triplet Loss* function from `FaceNet` [8]:

$$\max(\text{dist}\,(a_s, x_i)^2 - \text{dist}\,(a_d, x_i)^2 + m, 0)\,, \tag{2.11}$$

Again, $m$ is a margin beyond which the network will stop updating.

# 3 Results

In this section, we discuss the data set and results we have obtained by analyzing it.

## 3.1 Pipeline

During the course of this project, we have developed several separate pieces of code (which can be found inside "`code/separately`" folder), which process and analyze the data set, via XGBoost, Random Forest, together with our own code for a Feed-Forward Neural Network as well as Keras implementations of Siamese and Triplet Neural Network. Additionally, all codes have been unified in one pipeline, which is meant to be highly configurable. The resulting effort can be found in folder "`code/unified`". To run the unified code, one needs to run file `main.py`, with prior configuration if the parameter file, `ParameterFile.yaml`.

## 3.2 Data exploration

**Chosen dataset.** This study was performed on freshwater microbial ecology data generated by sampling 72 lakes from southern Norway and Sweden (fig. 3.1). It was part of a project designated COMSAT. The dataset comprises two tables. One consists of counts of amplicon sequence variants (ASV) for bacteria, while the other contains environmental metadata (table 3.2). Each observation in either table corresponds to a lake. Both tables are matched by observation and can thus be used as input and output for each other. Bacterial ASVs can be treated as a proxy for the abundance of bacterial species. These ASV counts can also be converted to binary with 0 equal to 0 and values above 0 set to 1 in order to study presence/absence patterns.


Figure 3.1: Freshwater lakes from southern Norway and Sweden sampled for the COMSAT project. Secchi depth is displayed to provide a general impression of the longitudinal gradient in the dataset.

| Site | ASV1 | ASV2 | ASV3 | ASV4 | ASV5 | ... | Latitude | Longitude | Altitude | Area | Depth | Temperature | Secchi | O2 | CH4 | pH | TIC | SiO2 | KdPAR |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10000_Hurdalsjøen | 18464 | 5231 | 6963 | 7563 | 9516 | ... | 60.37648 | 11.04077 | 176 | 32.81 | 20.0 | 17.03 | 6.50 | 0.9044194 | 11.797343 | 6.870 | 0.82230 | 3312 | 0.62 |
| 10001_Harestuvatnet | 15296 | 58728 | 30659 | 1614 | 17059 | ... | 60.19323 | 10.71212 | 234 | 1.98 | 13.0 | 15.85 | 4.50 | 0.8468347 | 72.674567 | 7.365 | 4.05800 | 3783 | 0.89 |
| 170B_Gjersjøen | 13356 | 52215 | 25810 | 1367 | 14586 | ... | 59.78970 | 10.77485 | 40 | 2.64 | 22.0 | 19.65 | 3.30 | 0.8131012 | 52.953904 | 7.685 | 8.08500 | 3563 | 0.95 |
| 170_Gjersjøen | 16227 | 53747 | 26456 | 2823 | 3119 | ... | 59.78970 | 10.77485 | 40 | 2.64 | 22.0 | 19.65 | 3.30 | 0.8131012 | 52.953904 | 7.685 | 8.08500 | 3563 | 0.95 |
| 180_Øgderen | 52862 | 4887 | 1361 | 14854 | 25616 | ... | 59.71388 | 11.41303 | 133 | 12.66 | 9.5 | 18.61 | 1.10 | 0.8406025 | 85.639780 | 7.225 | 2.66800 | 1125 | 1.60 |
| 189_Krøderen | 18830 | 53461 | 50015 | 12664 | 13253 | ... | 60.13485 | 9.75860 | 133 | 43.91 | 14.0 | 15.44 | 2.80 | 0.8582522 | 29.100059 | 6.695 | 0.81360 | 2499 | 0.82 |
| 191_Rødbyvatnet | 43828 | 7657 | 1836 | 34800 | 20517 | ... | 59.58175 | 10.48715 | 118 | 1.16 | 10.0 | 18.55 | 2.10 | 0.8527711 | 260.596931 | 7.535 | 3.09200 | 2063 | 1.32 |
| 214_Gjesåssjøen | 10532 | 588 | 9275 | 19181 | 5315 | ... | 60.68167 | 11.99235 | 176 | 3.98 | 3.5 | 19.63 | 1.15 | 0.8360833 | 97.561306 | 7.070 | 1.73200 | 2924 | 2.27 |
| 2252_Rotnessjøen | 14088 | 39265 | 35086 | 11061 | 7228 | ... | 60.49690 | 12.34120 | 260 | 1.09 | 26.0 | 16.55 | 1.95 | 0.7350632 | 41.956068 | 6.635 | 0.77310 | 5559 | 1.08 |

Figure 3.2: Subsets of the ASV and metadata tables. The columns to the right show the first 5 ASVs in decreasing order of abundance while the columns to the left show linearly independent metadata variables that can be treated as explanatory to bacterial community composition.

**Research questions.** It was chosen in this study to (1) identify patterns of variation in bacterial community composition along environmental gradients, (2) predict environmental metadata variables from bacterial community composition and (3) predict presence or absence of bacterial ASVs from environmental metadata. The first objective was pursued using pairs of dissolved organic matter water content as input with corresponding pairwise bacterial community Bray-Curtis distances as output. The second objective used the full bacterial community composition – ASV – table as input and single environmental metadata variables as outputs, yielding a separate model for each predicted metadata variable. The third objective used a subset of environmental variables as input to predict a subset of the bacterial community composition table converted to presence/absence values.

## 3.3 Data processing

**Data filtering, formatting and transformation.** The dataset contained only continuous variables. There were missing values in the environmental metadata, which were replaced by intrapolation using Multivariate Imputation by Chained Equations [9]. This decision to keep observations with missing values is motivated by the very small number of observations in the full dataset compared to the number of descriptors which would make it even harder to train models successfully, were the number of observations to be reduced any further. The ASV table was scaled using ranging (set values to interval of 0-1) due to this transformation yielding somewhat better clustering in terms of ecological meaningfulness compared to subtraction of mean and division by standard deviation [10]. Environmental metadata were used with or without scaling, the former being performed by subtracting the mean and dividing by standard deviation.

**Dimensionality reduction.** In order to extract patterns from the ASV data without using all predictors, pairwise distances between observations were used. Since the counts for most ASVs across most observations is 0, it was necessary to use an asymmetrical coefficient in order to avoid inflated similarity between observations. The Bray-Curtis distance was used accordingly. In the case of the environmental metadata, groups of linearly dependent variables had all but one variable kept, while a subsequent pruning was performed in the same manner where groups of variables presented *variance inflation factors* (VIF) above a threshold of 20 [11]. Correlations among environmental variables were visualized using a heatmap (fig. 3.3).

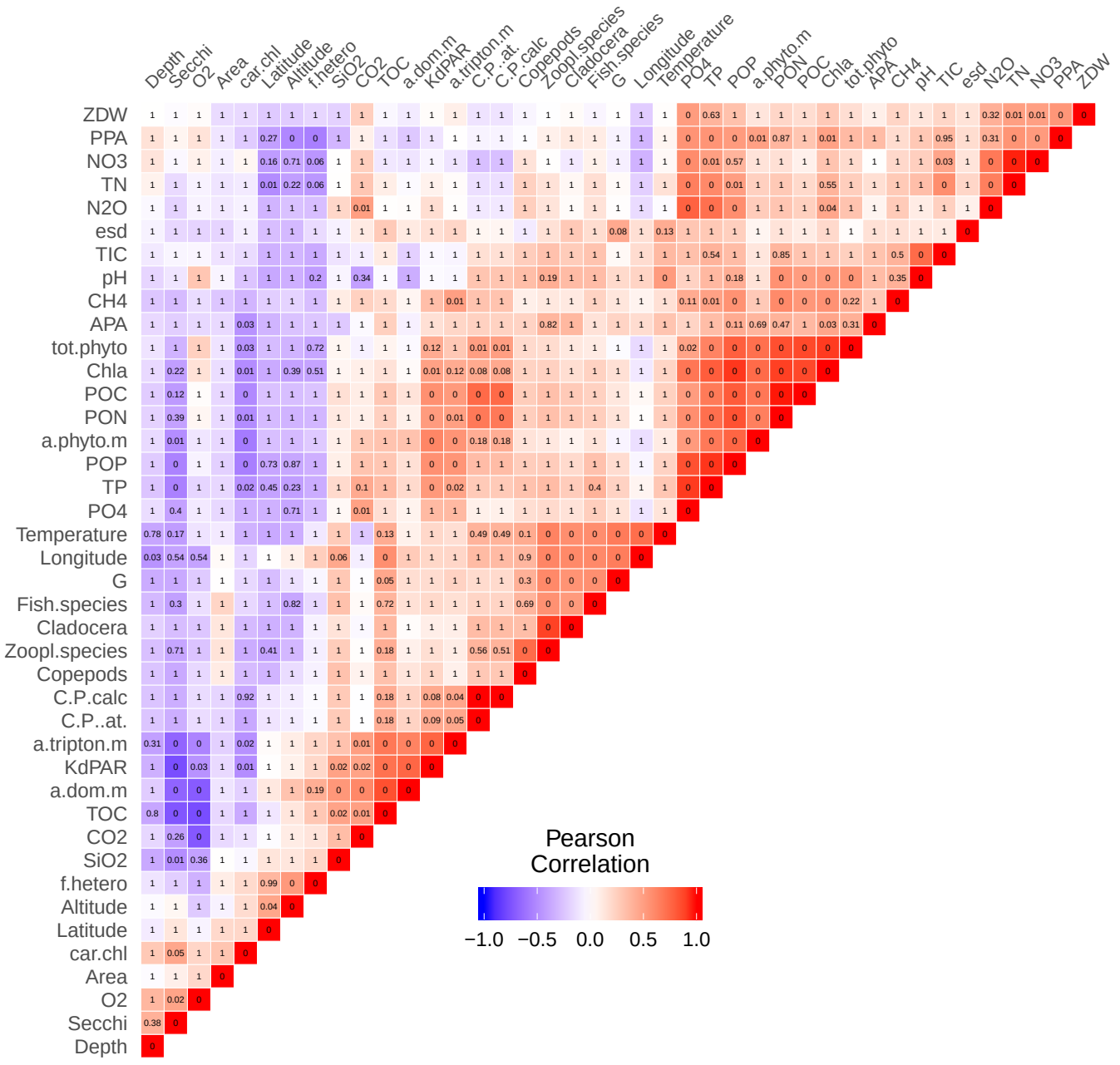


Figure 3.3: Pearson correlations among environmental metadata variables. Correlation values are displayed by color; the values printed in tiles indicate p-values for correlation significance.

## 3.4 XGBoost and Random Forest

**Regression using XGboost.** Regression using `scikit-learn`'s implementation of XGboost was used to predict bacterial community Bray-Curtis distances along the dissolved organic matter (DOM) gradient ("`a.dom.m`" in the environmental metadata table). The model was trained on data split into 80% training and 20% test sets. The same analysis was performed with own code for a feed forward neural network [2]. After training the respective models, response data was generated for a meshgrid of values matching the minimum-maximum range of the DOM gradient. The idea here was to see how well XGboost performed for generating a model used for *intrapolation*. Intrapolated outputs were also generated with ordinary least-squares regression to appreciate how well an XGboost model can contain complex structures in the data where linear models cannot.

**Comparison of XGboost and Random Forest.** A selection of non collinear environmental variables were predicted for an input of the full ASV table using `scikit-learn`'s XGboost and Random Forest libraries. The metadata tables were then scaled by subtracting the mean and dividing by standard deviation. Euclidean distance matrices were computed for true data as well as XGboost and Random Forest predictions, followed by Procrustes tests to determine whether the clusters of observations were significantly similar between predicted and true data. Unweighted pair group method with arithmetic mean (UPGMA) hierarchical clusterings of the distance matrices were visualized as Tanglegrams [12] (see figs. 3.4, 3.5 and 3.6).

### 3.4.1 Results

**Regression using XGboost.** The patterns in both XGboost and project 2 FFNN [2] predicted values are similar. XGboost is however much faster when training the model. The linear model poorly captured the complex structures in data revealed by the XGboost and neural network regressions (fig. 3.7). While the predicted values from the XGboost model seem correct on the grid of values from the original dataset, the model returns some type of step function when predicting values for a meshgrid containing $(x,y)$ values not present in the training dataset.

**Comparison of XGboost and Random Forest.** The tree topologies for true data, XGboost and Random Forest predictions are all significantly similar with $p$-values below 0.001. The correlation is highest between true data and XGboost predictions, with a value of 0.87, while Random Forest returned a correlation of 0.84. The correlation for XGboost and Random Forest was 0.75.

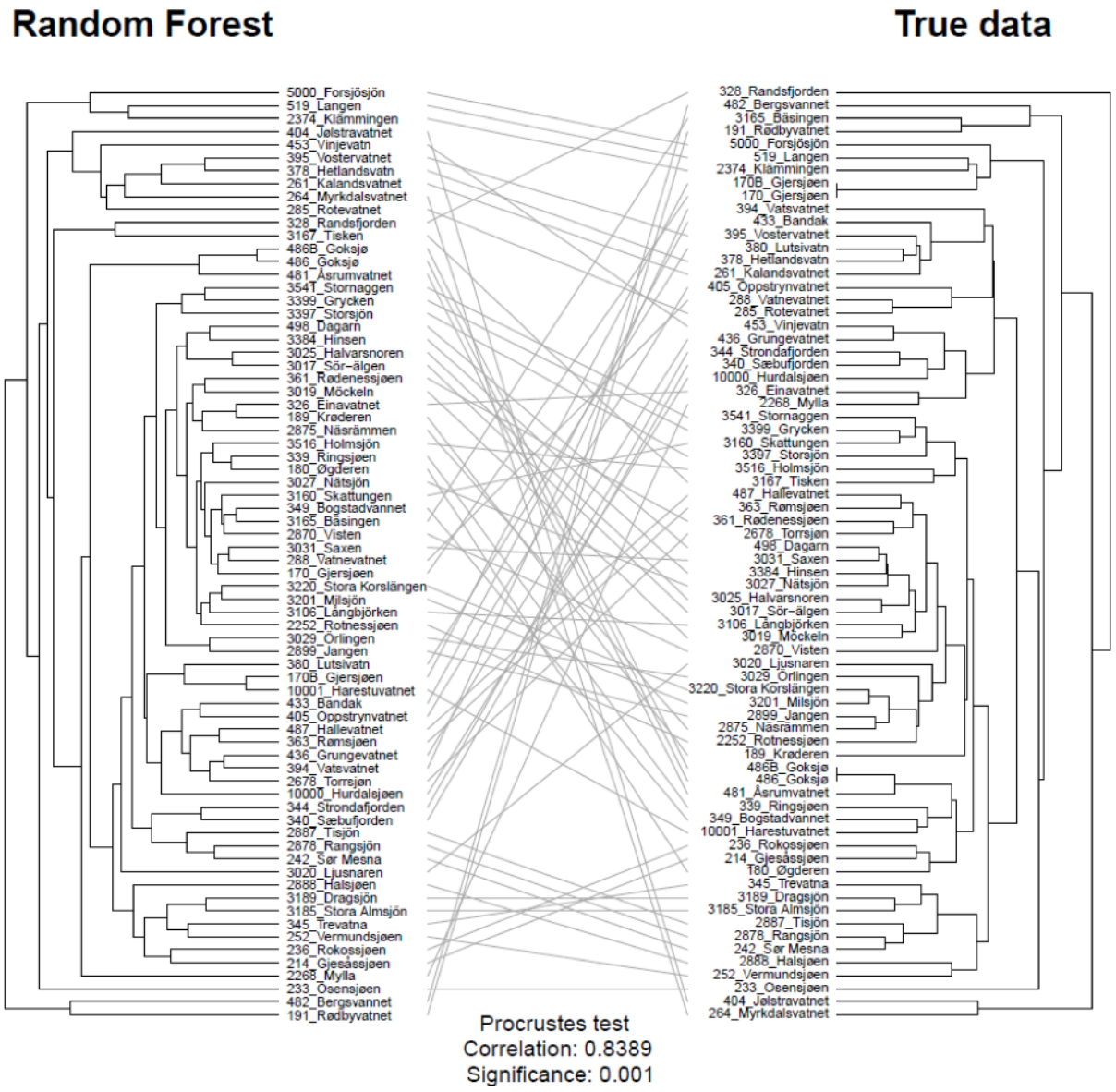


Figure 3.4: Comparison of UPGMA hierarchical clusters of Random Forest-predicted environmental metadata vs. true values of the same variables. Branch length between two leaves or nodes is proportional to the Euclidean distance between said leaves or nodes. Lines between both trees link corresponding leaves on each. The correlation between both trees is 0.8389 and is significant with a $p$-value of 0.001.

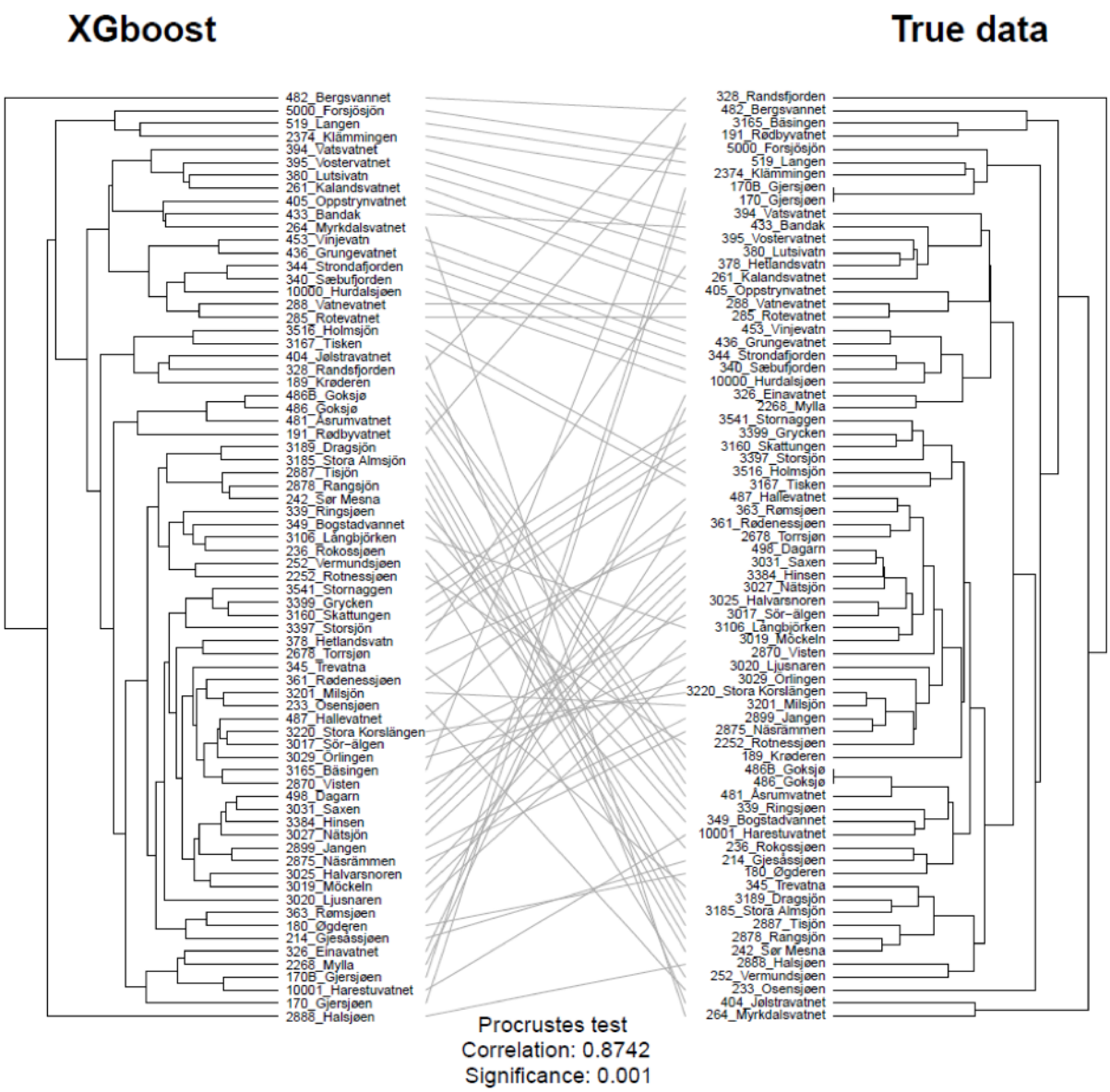


Figure 3.5: Comparison of UPGMA hierarchical clusters of XGboost-predicted environmental metadata vs. true values of the same variables. Branch length between two leaves or nodes is proportional to the Euclidean distance between said leaves or nodes. Lines between both trees link corresponding leaves on each. The correlation between both trees is 0.8742 and is significant with a $p$-value of 0.001.

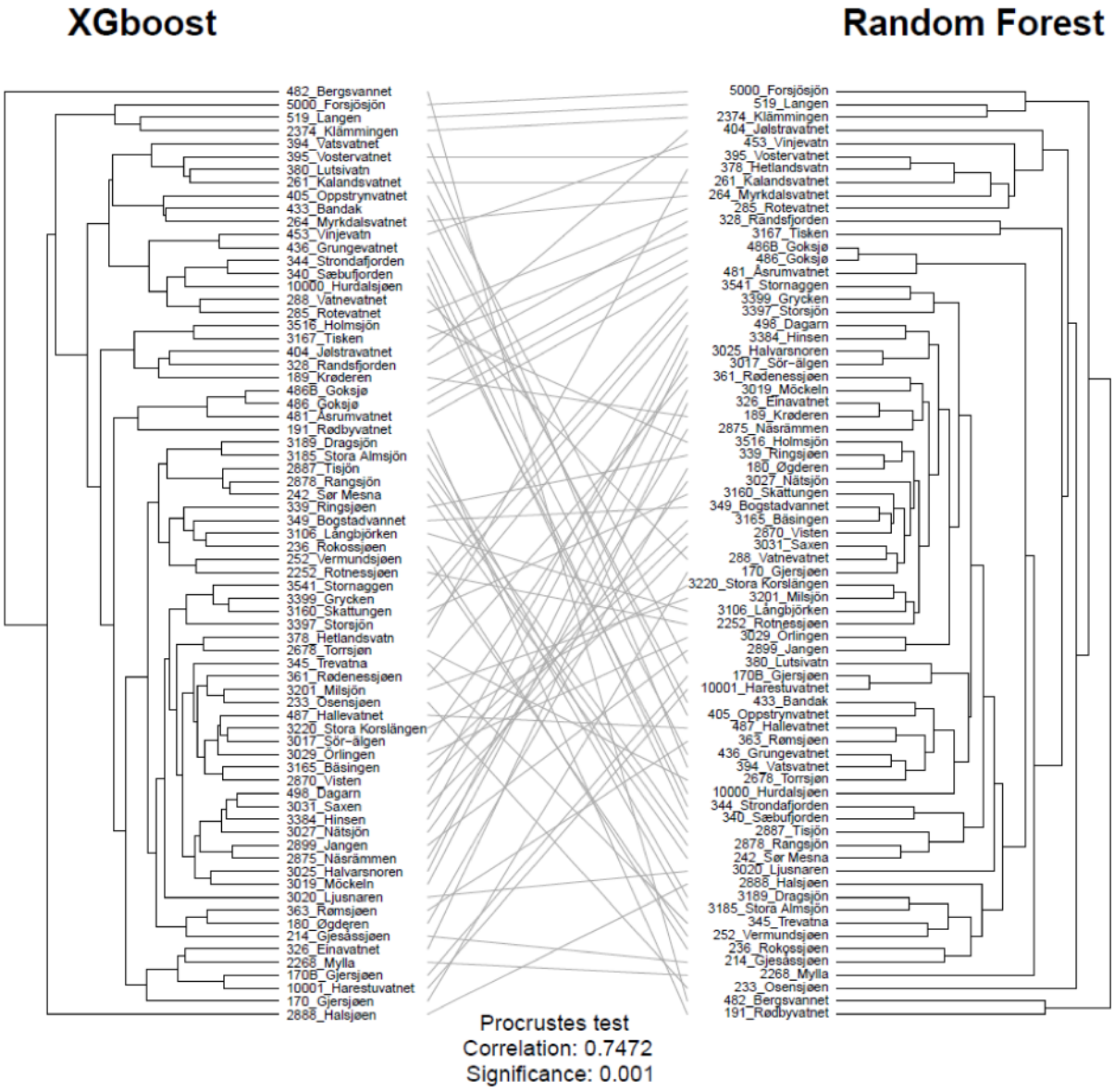


Figure 3.6: Comparison of UPGMA hierarchical clusters of XGboost-predicted environmental metadata vs. Random Forest predictions of the same variables. Branch length between two leaves or nodes is proportional to the Euclidean distance between said leaves or nodes. Lines between both trees link corresponding leaves on each. The correlation between both trees is 0.7472 and is significant with a $p$-value of 0.001.

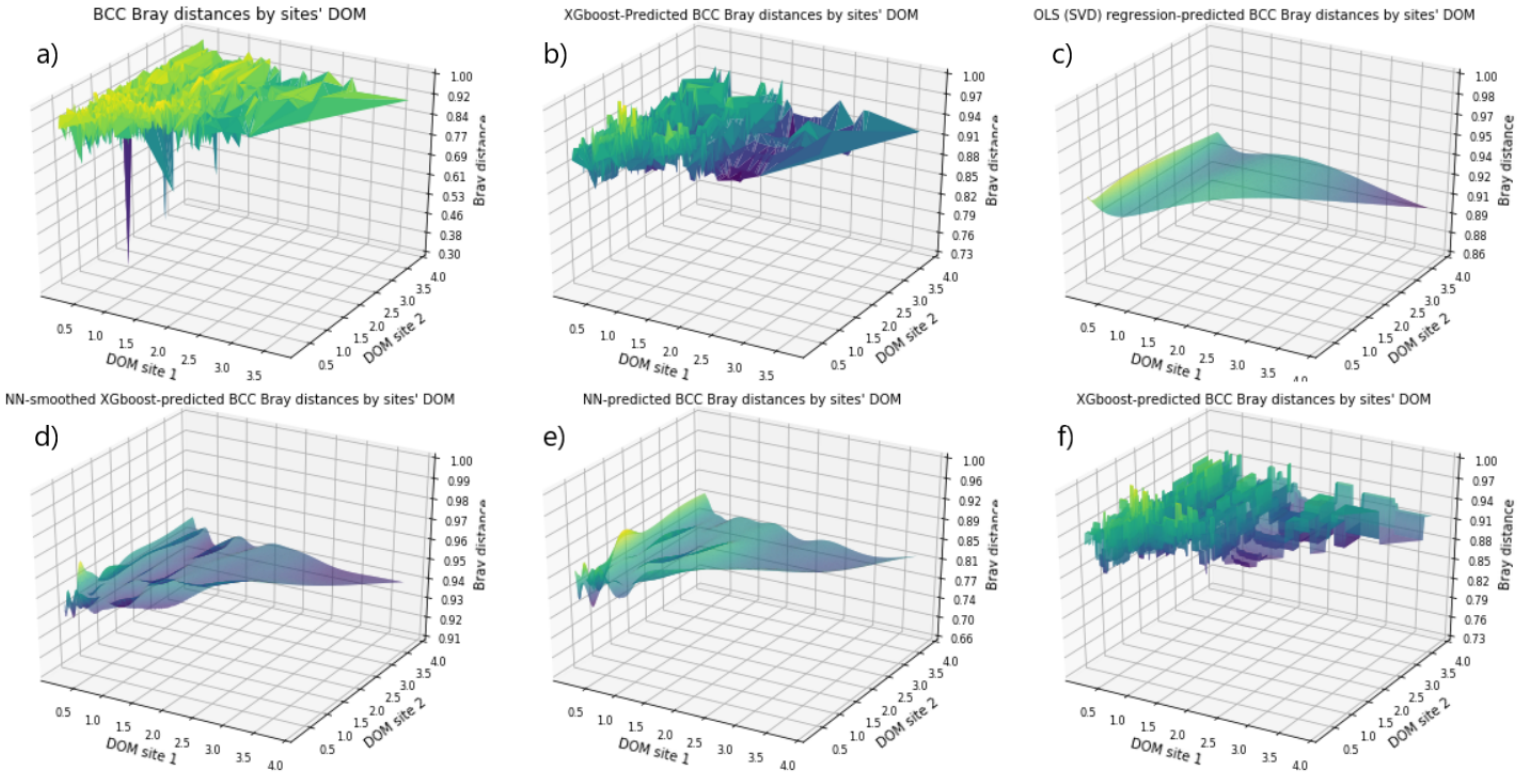


Figure 3.7: Bray-Curtis community composition distances by DOM gradient. a) Bacterial community composition (BCC) Bray-Curtis distances by DOM (raw data). b) XGboost-Predicted BCC Bray distances by sites' DOM. c) OLS (SVD) regression-predicted BCC Bray distances by sites' DOM, DOM 0.01 step meshgrid. d) NN-smoothed XGboost-predicted BCC Bray distances by sites' DOM, DOM 0.01 step meshgrid. e) NN-predicted BCC Bray distances by sites' DOM, DOM 0.01 step meshgrid. f) XGboost-predicted BCC Bray distances by sites' DOM, DOM 0.01 step meshgrid.

## 3.5 Neural Networks

## 3.6 Siamese Implementation

To explore classification with random forests and siamese networks, the different sites were grouped into to classes based on whether the local measurement was above or below the meadian for of all 72 lakes. It is not unreasonable to expect certain variables, such as temperature, pH or phosphate content to correlate with the distribution of microbial species. As

such, attempts were made to predict, for one variable at a time, whether or not sites were above or below the median for the given variable by looking at the species distributions.

Only species present in all lakes were used as predictors and target variables chosen were total phosphorus (TP) and temperature. Three different approaches to classify the lakes into high/low categories were tested. Two siamese network variants and one using random forest. For the siamese networks, anchors, i.e prototypical representatives for the given class, were chosen by inspection. For the twin siamese network, pairs consisting of one of the two anchors and a data point to be evaluated, were created for all combinations of data points and anchors, yielding a total of 144 pairs. For the triplet network, each data point was paired with both anchors, giving a total of 72 triplets. 80% of the pairs/triplets were used for training and the remaining 20% were used to asses predictive ability.

## 3.7 Results

For the siamese network approach, the results were, on the whole, disappointing. While the networks certainly showed signs of learning, getting good results on the training data, they displayed little to no ability to generalize. The results from a typical run can be seen in fig. 3.8 and fig. 3.9.

Attempts to alleviate overfitting by adding dropout layers or adjusting hyper parameters, did not lead to improvements on predictive ability much beyond the level of random guessing.

With our choice of data set, it is not obvious whether this failure is due to the methods employed or a lack of significant correlation between the microbial community composition, and the various target variables, such as phosphate concentrations or temperature at the sample sites. Another possible reason for these results might be the choice of classification method. The choice to classify sites by whether they fell above or below the median of the data set was not inspired by any particular biological theory beyond different environments breeding different biota. The environmental variables which the sites were classified by might have critical points at which the community distribution significantly changes, but there is no a priori reason to assume this would be the median.

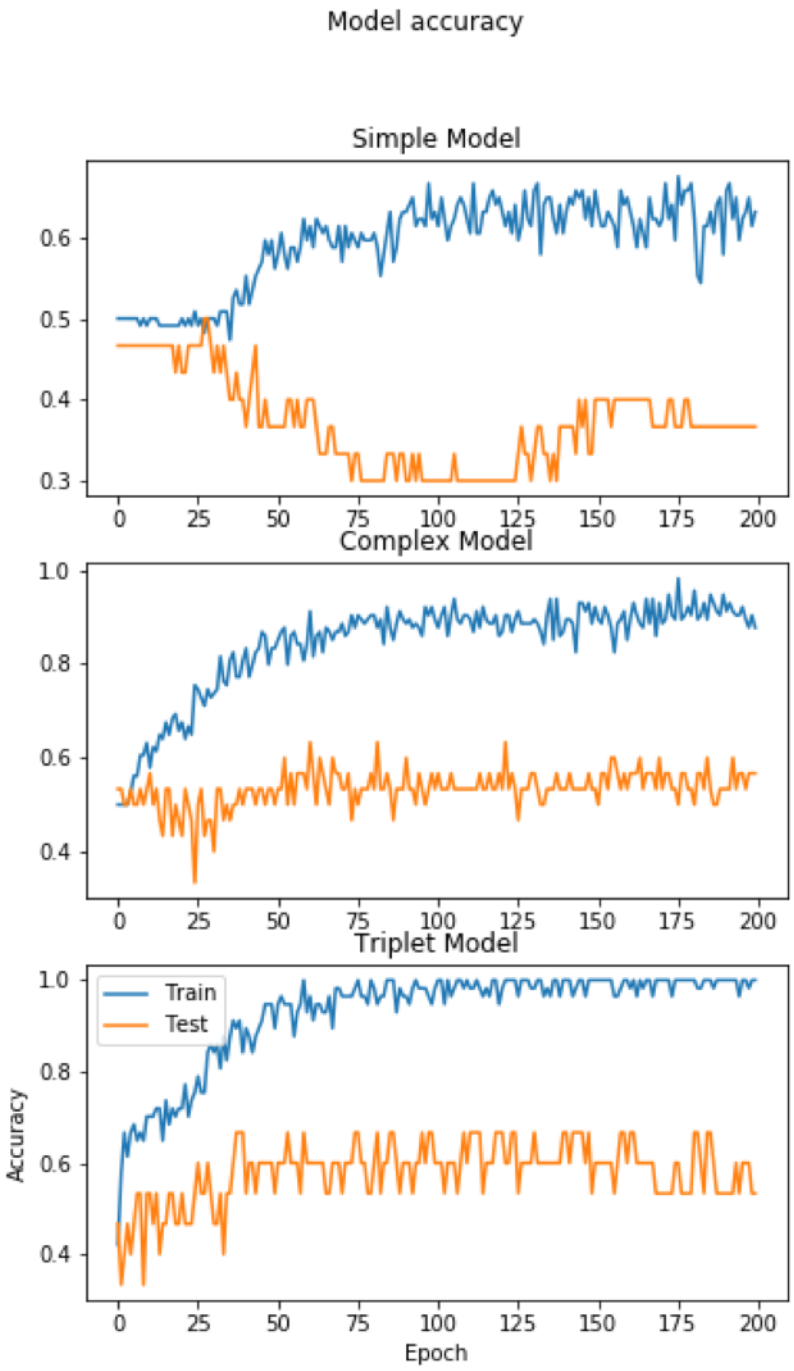


Figure 3.8: Accuracy scores for neural networks predicting high or low total phosphorus content by looking at community composition. Top: Simple siames network with 2 hidden layers of 50 neurons. Middle: Complex siamese network with 3 hidden layers of 100 neurons. Bottom: Triplet network using the complex siamese base network.

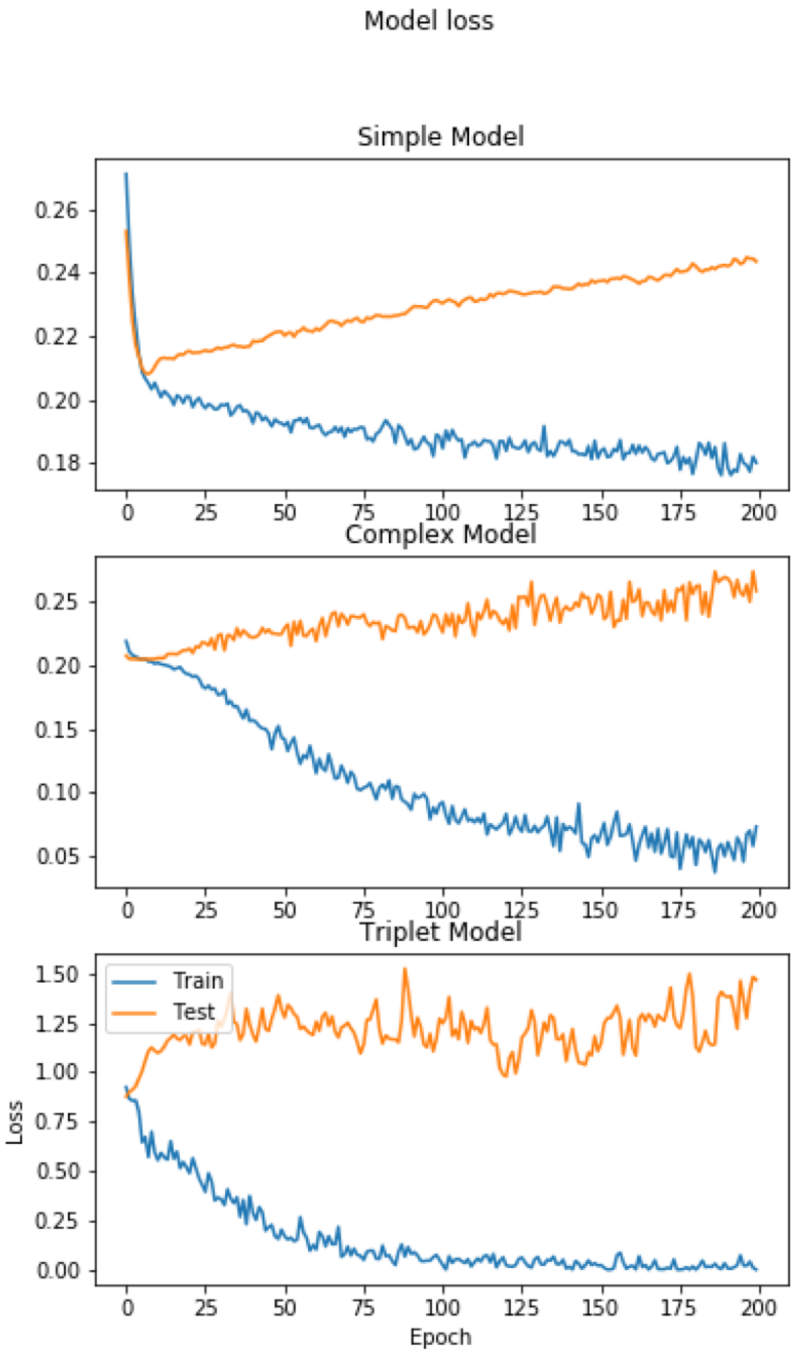


Figure 3.9: Loss values for neural networks predicting high or low total phosphorus content by looking at community composition. Top: Simple siames network with 2 hidden layers of 50 neurons. Middle: Complex siamese network with 3 hidden layers of 100 neurons.

# 4 Discussions

**Regression using XGboost** Performing regression on the bacterial community composition distances by dissolved organic matter content was orders of magnitude faster than the own code for the feed-forward neural network. However, the usefulness of the resulting XGboost model falls somewhat short when an output is predicted for input values absent from the training data. Attempting the latter resulted in a step function pattern when the input was a meshgrid of DOM values found inside the interval of training DOM values. This problem did not occur for the FFNN when predicting the BCC distances for the same meshgrid of DOM values; the output was a smooth surface. Attempting to obtain a smooth surface of XGboost predictions by standard regression methods does not work either when structures in the data are as complex as is the case with this dataset. It is possible to train a feed-forward neural network on the XGboost predictions for training data in order to produce a smooth surface for a higher resolution meshgrid, although it comes at the cost of introducing possible artifacts and bias from slightly different results associated with distinct random seeds. Overall, the FFNN remains preferable when the input data does not follow regular steps along a gradient and when a model is trained for a subsequent intrapolation purpose.

**Comparison of XGboost and Random Forest** When using the same input to generate the same outputs, XGboost was much faster than Random Forest. While it might be possible to get more accurate results with Random Forest using more estimators, the computational requirements to do so quickly grow beyond practical means. The outputs from XGboost and Random forest differed more from each other than they each did from true data. There was no clear indication as to how the predictions diverged to produce this result. Both algorithms could handle the entire ASV table as input and return satisfyingly accurate outputs. This removes the need for performing dimensionality reduction on the inputs. Overall, it can be said the main advantages of using XGboost over Random Forest lies in execution speed as well as predictions that are slightly more accurate. Consequently, XGboost shows great potential for typical microbial ecology datasets such as the one analyzed in this study, since it is usually very difficult to select which predictors to exclude, especially when they may be part of a priori unforeseen but relevant structures in the data.

# References


[1] Koch, G.,Zemel, R., Salakhutdinov, R. *Koch, Siamese Neural Networks for One-shot Image Recognition* (2015), Semantic Scholar, link to pdf version

[2] Fontaine, L., *Report for FYS-STK4155: project 2* (2019).

[3] Hjorth-Jensen, M.. *Data Analysis and Machine Learning: From Decision Trees to Forests and all that* Oct. 2019.

[4] Hjorth-Jensen, M.. *Data Analysis and Machine Learning Lectures: Optimization and Gradient Methods* Oct. 2019.

[5] Chen,T. Guestrin, C., *XGBoost: A Scalable Tree Boosting System*, arXiv: 1603.02754

[6] Brilenkov, M. *Report for FYS-STK4155: project 2* (2019)

[7] Hadsell, R., Chopra, S., LeCun, Y. *Dimensionality Reduction by Learning an Invariant Mapping* (2005), link to pdf

[8] Schroff, F., Kalenichenko, D., Philbin, J., *FaceNet: A Unified Embedding for Face Recognition and Clustering* (2015), arXiv: 1503.03832

[9] Buuren, S. van, Groothuis-Oudshoorn, K. (2011). mice: Multivariate Imputation by Chained Equations inR. Journal of Statistical Software, 45(3). doi: https://doi.org/10.18637/jss.v045.i03

[10] Legendre, P., Legendre, L. F. (2012). Numerical ecology(Vol. 24). Elsevier

[11] ter Braak, C. J. F. P. Smilauer. 2002. *CANOCO reference manual and CanoDraw for Windows user's guide – Software for canonical community ordination* (version 4.5). Microcomputer Power, Ithaca, New York. 500 pp.

[12] Galili, T. (2015). *dendextend: an R package for visualizing, adjusting and comparing trees of hierarchical clustering.* Bioinformatics, 31(22), 3718–3720. https://doi.org/10.1093/bioinformatics/btv428

# Code

Listing 1: "Parameter File"

```
#
    ↪ ##############################################################################################
    ↪
# Parameter File
#
    ↪ ##############################################################################################
    ↪
# Feed Forward NN, Siamese NN, Triplet NN, XGBoost, Random Forest
# ['ffnn_keras', 'snn_keras', 'tnn_keras', 'xgb', 'rf_main', 'rf_side']
type:           'rf_side'
# Path to data
DataPath:       ['data/ASV_table_mod.csv','data/Metadata_table.tsv'] #'data/
    ↪ ASV_table_mod.csv'
# Output Path - where to save all the files (phg's etc.)
OutputPath:     'output/'
# Specify the seed
RandomSeed:     1
# Size of the test sample
TestSize:       0.2
#
    ↪ ##############################################################################################
    ↪
# Neural Network Configuration (ignore it, if you are using xgboost or random
    ↪ forest)
#
    ↪ ##############################################################################################
    ↪
# Loss Functions (most of them are from Keras library) to use. We strongly suggest
    ↪  to use only binary, triplet or contrastive
# because others were not tested
# ['binary', 'contrastive', 'triplet', 'mse', 'mae', 'mape', 'msle', 'hinge', '
    ↪ shinge', 'chinge', 'logcosh',
# 'categorical', 'sparse', 'kullback', 'poisson', 'proximity']
# with siamese network use 'contrastive', with triplet use 'triplet'
Loss:           'binary' #'triplet' #'contrastive' #'binary'
# Weights (as in keras library)
# Random Normal, Xavier Normal, He Normal, Xavier Uniform, He Uniform
# ['norm', 'xnorm','hnorm', 'unif', 'xunif', 'hunif']
Weights:        'norm'
# Choose number of layers (can be any number)
NHiddenLayers:  3
# Activation functions for hidden and output layers
# (['sigmoid', 'tanh', 'relu', 'softmax'])
HiddenFunc:     'relu' #'sigmoid'
OutputFunc:     'sigmoid'
# Number of Neurons for hidden and output layers
NHiddenNeurons: 2000 # 21 # 4  # 30
```

```
NOutputNeurons: 2 # classification = 2, Regression = 1
# Epochs to train the algorithm
epochs:          10
# Optimization Algorithm: choose it wisely :)
#['MBGD', 'Adagrad' 'GD'] <= for minibatch, if you choose 1 you will get just
    ↪ stochstic GD
# if you choose simply GD, then it willl ignore batchSize parameter and will use
    ↪ the whole data set
Optimization:    'adagrad' # please use Adagrad for linear regression (as it may
    ↪ crush overwise
# Batch size for Gradient Descent => if 0, will use simple gradient descent
BatchSize:       10 #16 # 1 #10000 <= increase batch size and it will be good
# Learning rate
alpha:           1e-8 # 0.01 # 0.01 #0.0001 #np.logspace(-5, 1, 7)
# Regularisation - parameter used in Dropout layers
lambda:          0.1
```

Listing 2: "Data Processing"

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib import scale
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from sklearn.model_selection import train_test_split

class DataProcessing:

    def __init__(self):
        self.url                 = "https://github.com/laurent0001/tsv/raw/master/
            ↪ CDOM.gradient.mat.tsv"
        self.url_mesh            = "https://github.com/laurent0001/tsv/raw/master/
            ↪ Bray_distances_by_CDOM_gradient_meshgrid.tsv"
        self.url_CDOM_diag_mesh  = "https://github.com/laurent0001/tsv/blob/master
            ↪ /CDOM.diag.mesh.tsv?raw=true"
        self.url_CDOM_sorted     = "https://github.com/laurent0001/tsv/raw/master/
            ↪ CDOM.tsv"
        # Load bacterial community data
        self.url_ASV             = "https://github.com/laurent0001/Project-3/blob/
            ↪ master/ASV_table.tsv?raw=true"
        self.url_ASV_ranged      = "https://github.com/laurent0001/Project-3/blob/
            ↪ master/ASV_table_ranged.tsv?raw=true"
        self.url_metadata        = "https://github.com/laurent0001/Project-3/raw/
            ↪ master/Metadata_table.tsv?raw=true"
        self.url_metadata_scaled = "https://github.com/laurent0001/Project-3/raw/
            ↪ master/Metadata_table_scaled.tsv?raw=true"

    def GetMainData(self, *args):
        '''
        Data Preparation step
        '''
        CDOM = pd.read_csv(self.url, sep="\t") #BCC pairwise distances with CDOM
            ↪ values for both sites for each row
        #CDOM_sites = pd.read_csv(url_sites, sep="\t") #Sites matching the order
            ↪ of BCC pairwise distances with CDOM values of both sites for each
            ↪ row
        CDOM.mesh = pd.read_csv(self.url_mesh, sep="\t")
        CDOM_diag_mesh = pd.read_csv(self.url_CDOM_diag_mesh, sep="\t")
        CDOM_sorted = pd.read_csv(self.url_CDOM_sorted, sep="\t")
        CDOM_diag_mesh.columns = ["CDOM.x1", "CDOM.x2", "CDOM.mid"]

        ASV = pd.read_csv(self.url_ASV, sep="\t")
        ASV_ranged = pd.read_csv(self.url_ASV_ranged, sep="\t")
        metadata = pd.read_csv(self.url_metadata, sep="\t")
        metadata_scaled = pd.read_csv(self.url_metadata_scaled, sep="\t")
        X_ASV = ASV_ranged
```

```
        X_ASV.columns = [''] * len(X_ASV.columns)
        X_ASV = X_ASV.to_numpy()
        #y_CDOM = metadata.iloc[:, 27][:, np.newaxis]

        # split data into train and test sets
        y_CDOM = metadata.iloc[:, 27] #Requires 1d array

        data = CDOM, CDOM_sorted, CDOM_diag_mesh, \
               ASV, ASV_ranged, \
               metadata, metadata_scaled, \
               X_ASV, y_CDOM
        return data
```

Listing 3: "Entry point to the program"

```
"""
@author: maksymb
"""

# Library imports
import os, sys
import numpy as np
import keras
from sklearn.preprocessing import OneHotEncoder, LabelEncoder, StandardScaler,
    ↪ RobustScaler, MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, mean_squared_error
from xgboost import XGBRegressor

import pandas as pd
import yaml
# libraries for plotting results
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib import scale
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import seaborn as sbn
# to calculate time
import time

# importing manually created libraries
import neural, funclib, regression, xgb, random_forest, data_processing

'''
The main class of the program
'''
class MainPipeline:
    # constructor
    def __init__(self, *args):
        paramFile = args[0]
        # Getting values from Parameter file
        with open(paramFile) as f:
            self.paramData = yaml.load(f, Loader = yaml.FullLoader)
        # creating an output directory
        outputPath = self.paramData['OutputPath']
        if not os.path.exists(outputPath):
            os.makedirs(outputPath)

        # Instantiating object variable from Functions Library
        self.funcs = funclib.Functions(self.paramData['RandomSeed'],
                                       self.paramData['Loss'])

    '''
    Method to preprocess the data set for Side Research Questions
```

```
    '''
    def PreProcessing(self, *args):
        # Random Seed
        seed = self.paramData['RandomSeed']
        # Loss function - we need it to instantiate funclib variable
        loss = self.paramData['Loss']

        '''
        Data preprocessing
        '''
        # getting data from parameter file
        #data = pd.read_csv(self.paramData['dataPath'], delimiter='\s+', encoding
            ↪ ='utf-8')
        #print(self.paramData['dataPath'][0])
        # as in Knut's file
        data = pd.read_csv(self.paramData['DataPath'][0])
        meta = pd.read_csv(self.paramData['DataPath'][1], delimiter=r"\s+")
        dt   = data.replace(0, pd.np.nan).dropna(axis=1, how='any').fillna(0).
            ↪ astype(int)
        data = self.funcs.normalize(dt)
        # our target variable
        target_choice="TP"
        tp = meta[target_choice]
        # Simple Feed Forward Neural Network
        if self.paramData['type'] == 'ffnn_keras':
            self.Y = tp.values.reshape(-1, 1)
            self.X_norm = data.to_numpy()
            # Split into training and testing
            self.X_train, self.X_test, y_train, y_test = train_test_split(self.
                ↪ X_norm,
                                                                          self.Y,
                                                                          random_state
                                                                              ↪ =
                                                                              ↪ seed
                                                                              ↪ ,
                                                                          test_size
                                                                              ↪ =
                                                                              ↪ self
                                                                              ↪ .
                                                                              ↪ paramData
                                                                              ↪ [
                                                                              ↪ '
                                                                              ↪ TestSize
                                                                              ↪ '
                                                                              ↪ ])
                                                                              ↪
            y_train_l, y_test_l = self.funcs.set_category(self.Y, y_train, y_test)
            # doing one hot encoding
            oh = OneHotEncoder(sparse=False, categories="auto")
            self.Y_train_onehot = oh.fit_transform(y_train_l)
            self.Y_test_onehot = oh.fit_transform(y_test_l)
```

```
        elif self.paramData['type'] == 'snn_keras':
            self.Y = tp.values.reshape(-1, 1)
            # normalizing data
            self.X_norm = data.to_numpy()
            # Split into training and testing
            self.X_train, self.X_test, y_train, y_test = train_test_split(self.
                ↪ X_norm,
                                                                          self.Y,
                                                                          random_state
                                                                              ↪ =
                                                                              ↪ seed
                                                                              ↪ ,
                                                                          test_size
                                                                              ↪ =
                                                                              ↪ self
                                                                              ↪ .
                                                                              ↪ paramData
                                                                              ↪ [
                                                                              ↪ '
                                                                              ↪ TestSize
                                                                              ↪ '
                                                                              ↪ ])
                                                                              ↪
            y_train_l, y_test_l = self.funcs.set_category(self.Y, y_train, y_test)
            oh = OneHotEncoder(sparse=False,categories="auto")
            #Anchors selected by inspection. Serves as positive/negatives for
            #comparison in network. First position is a low valued representative
            #second position is a high valued representative.
            anchors = {"TMP":(data.iloc[10],data.iloc[70])}
            anchors[target_choice] = (data.iloc[15],data.iloc[67])

            # Make pairs and labels of "same" or "different"
            self.pairs_train, \
            self.labels_train, \
            self.pairs_test, \
            self.labels_test = self.funcs.make_anchored_pairs(self.X_train,
                                                    y_train_l,
                                                     self.X_test,
                                                     y_test_l,
                                                     anch = anchors[target_choice])
            # doing one hot encoding
            self.Y_train_onehot = oh.fit_transform(self.labels_train.reshape(-1,1)
                ↪ )
            self.Y_test_onehot = oh.fit_transform(self.labels_test.reshape(-1,1))
        elif self.paramData['type'] == 'tnn_keras':
            self.Y = tp.values.reshape(-1, 1)
            # normalizing data
            self.X_norm = data.to_numpy()
            # Split into training and testing
            self.X_train, self.X_test, y_train, y_test = train_test_split(self.
                ↪ X_norm,
```

```
                                                                    self.Y,
                                                                    random_state
                                                                        ↪ =
                                                                        ↪ seed
                                                                        ↪ ,
                                                                    test_size
                                                                        ↪ =
                                                                        ↪ self
                                                                        ↪ .
                                                                        ↪ paramData
                                                                        ↪ [
                                                                        ↪ '
                                                                        ↪ TestSize
                                                                        ↪ '
                                                                        ↪ ])
                                                                        ↪
        y_train_l, y_test_l = self.funcs.set_category(self.Y, y_train, y_test)
        oh = OneHotEncoder(sparse=False,categories="auto")
        #Anchors selected by inspection. Serves as positive/negatives for
        #comparison in network. First position is a low valued representative
        #second position is a high valued representative.
        anchors = {"TMP":(data.iloc[10],data.iloc[70])}
        anchors[target_choice] = (data.iloc[15],data.iloc[67])

        # Make pairs and labels of "same" or "different"
        self.pairs_train, \
        self.labels_train, \
        self.pairs_test, \
        self.labels_test = self.funcs.make_anchored_pairs(self.X_train,
                                                  y_train_l,
                                                  self.X_test,
                                                  y_test_l,
                                                  anch = anchors[
                                                      ↪ target_choice])
        #Create triplets for triplet network
        self.triplets_train = self.funcs.make_training_triplets(anchors[
            ↪ target_choice],self.X_train, y_train_l)
        self.triplets_test  = self.funcs.make_training_triplets(anchors[
            ↪ target_choice],self.X_test, y_test_l)
        # doing one hot encoding
        self.Y_train_onehot = y_train_l#oh.fit_transform(self.labels_train.
            ↪ reshape(-1,1))
        self.Y_test_onehot = y_test_l#oh.fit_transform(self.labels_test.
            ↪ reshape(-1,1))

    elif self.paramData['type'] == 'rf_side':

        #ph = meta["pH"]
        #n2o = meta["N20"]
        #temp = meta["Temperature"]
        tp = meta[target_choice]
```

```
        anchors = {"TMP": (data.iloc[10], data.iloc[70])}
        anchors[target_choice] = (data.iloc[15], data.iloc[67])
        # Some data formating
        # y = make_categories(temp,2)
        #np.random.seed(2)
        self.Y = tp#.values.reshape(-1, 1)
        # normalizing data
        self.X_norm = data.to_numpy()
        # y = to_categorical(y, num_classes=None)
        # Split into training and testing
        self.X_train, self.X_test, y_train, y_test = train_test_split(self.
            ↪ X_norm,
                                                                      self.Y,
                                                                      random_state
                                                                          ↪ =
                                                                          ↪ seed
                                                                          ↪ ,
                                                                      test_size
                                                                          ↪ =
                                                                          ↪ self
                                                                          ↪ .
                                                                          ↪ paramData
                                                                          ↪ [
                                                                          ↪ '
                                                                          ↪ TestSize
                                                                          ↪ '
                                                                          ↪ ])
                                                                          ↪
        self.y_train_l, self.y_test_l = self.funcs.set_category(self.Y,
            ↪ y_train, y_test)

    '''
    Creating Network Architecture
    '''
    # Network Type
    NNType = self.paramData['type']
    # Number of Hidden Layers
    NHiddenLayers = self.paramData['NHiddenLayers']
    # Total Number of Layers
    NTotalLayers = NHiddenLayers + 2
    # No of input data <= amount of data in a single column
    Ndata = self.X_train.shape[0] #self.X_norm.shape[0]
    # No of Input Neurons <= amount of variables
    NInputNeurons = self.X_train.shape[1] #self.X_norm.shape[1] #paramData['N
        ↪ ']
    # No of hidden neurons
    NHiddenNeurons = self.paramData['NHiddenNeurons']
    # No of output neurons
    NOutputNeurons = self.paramData['NOutputNeurons']
    # Activation Functions
    # activation functions
```

```
        aHiddenFunc = self.paramData['HiddenFunc']
        aOutputFunc = self.paramData['OutputFunc']
        # Neural Network Layer Architecture
        NNArch = []
        # Creating NN architecture
        for layer in range(0, NTotalLayers):
            # input layer
            if layer == 0:
                NNArch.append({"LSize": NInputNeurons, "AF": aHiddenFunc})
            # output layer
            elif layer == (NTotalLayers-1):
                NNArch.append({"LSize": NOutputNeurons, "AF": aOutputFunc})
            # hidden layers
            else:
                NNArch.append({"LSize": NHiddenNeurons, "AF": aHiddenFunc})
        # weights
        weights = self.paramData['Weights']
        # epochs to train the algorithm
        epochs = self.paramData['epochs']#1000
        # learning rate
        alpha = self.paramData['alpha'] #0.3
        # regularization parameter
        lmbd = self.paramData['lambda'] #0.001
        # Optimization Algorithm
        optimization = self.paramData['Optimization']
        # batch size
        batchSize = self.paramData['BatchSize']

        NNdata = seed, NNType, NHiddenLayers, NTotalLayers,\
                    Ndata, NInputNeurons, NHiddenNeurons, \
                    NOutputNeurons, aHiddenFunc, aOutputFunc, NNArch,\
                    weights, epochs, alpha, lmbd, optimization, loss, batchSize
        # returning the NN entire data structure
        return NNdata

    # Main Method
    def Run(self, *args):
        # getting data for main problem
        dataproc = data_processing.DataProcessing()
        data = dataproc.GetMainData()
        # getting everything we need
        CDOM, CDOM_sorted, CDOM_diag_mesh, \
        ASV, ASV_ranged, \
        metadata, metadata_scaled, \
        X_ASV, y_CDOM = data
        #XGboost with scikitlearn - data with spatial component (BCC Bray distance
            ↪  by CDOM)
        X_CDOM = CDOM.loc[:,["CDOM.x1", "CDOM.x2"]] #Molten meshgrid CDOM values
            ↪ for real data BCC Bray distances
        X_CDOM_diag_mesh = CDOM_diag_mesh.loc[:,["CDOM.x1", "CDOM.x2"]] #Molten
            ↪ meshgrid CDOM values for generating predicted BCC Bray distances
```

```
        y_CDOM = CDOM.loc[:,"ASV.dist"]

        if self.paramData['type'] == 'ffnn_keras':
            # retrieving network data
            NNdata = self.PreProcessing()
            # passing parameter file
            print(self.paramData)
            '''
            Getting Network Architecture
            '''
            network = neural.NeuralNetwork(NNdata)
            # passing network architecture and create the model
            model = network.BuildModel()
            # training model
            model, history = network.TrainModel(model, self.X_train, self.X_test,
                ↪ self.Y_train_onehot, self.Y_test_onehot)#self.X_norm, self.
                ↪ Y_onehot)
            test_loss, test_acc = model.evaluate(self.X_test, self.Y_test_onehot)
            print('Test accuracy:', test_acc)

            # Plotting results
            self.funcs.PlotResultsKeras(history,
                                        self.paramData['type'],
                                        self.paramData['OutputPath'],
                                        self.paramData['epochs'],
                                        self.paramData['Optimization'],
                                        self.paramData['BatchSize'])

        elif self.paramData['type'] == 'snn_keras':
            # retrieving network data
            NNdata = self.PreProcessing()
            '''
            Getting Network Architecture
            '''
            network = neural.NeuralNetwork(NNdata)
            # passing network architecture and create the model
            model = network.BuildModel()
            # training model
            model, history = network.TrainModel(model, self.pairs_train, self.
                ↪ Y_train_onehot, self.pairs_test, self.Y_test_onehot)
            # Plotting results
            self.funcs.PlotResultsKeras(history,
                                        self.paramData['type'],
                                        self.paramData['OutputPath'],
                                        self.paramData['epochs'],
                                        self.paramData['Optimization'],
                                        self.paramData['BatchSize'])
        elif self.paramData['type'] == 'tnn_keras':
            # retrieving network data
            NNdata = self.PreProcessing()
            '''
```

```
        Getting Network Architecture
        '''
        network = neural.NeuralNetwork(NNdata)
        # passing network architecture and create the model
        model = network.BuildModel()
        # training model
        model, history = network.TrainModel(model, self.triplets_train, self.
            ↪ Y_train_onehot, self.triplets_test, self.Y_test_onehot)
        # Plotting results
        self.funcs.PlotResultsKeras(history,
                                    self.paramData['type'],
                                    self.paramData['OutputPath'],
                                    self.paramData['epochs'],
                                    self.paramData['Optimization'],
                                    self.paramData['BatchSize'])

    elif self.paramData['type'] == 'ffnn_manual':
        # Neural network with multiple layers - regression - BCC Bray
            ↪ distances by CDOM - original data
        X_CDOM = CDOM.loc[:,["CDOM.x1", "CDOM.x2"]].to_numpy()
        y_CDOM = CDOM.loc[:,"ASV.dist"].to_numpy()[:, np.newaxis] #Original
            ↪ data

        '''
        NN_reg_original = neural.NeuralNetworkML(X_CDOM, y_CDOM,
                                                 trainingShare=0.80,
                                                 n_hidden_layers=3,
                                                 n_hidden_neurons=[2000, 1000,
                                                     ↪  500],
                                                 n_categories=1,
                                                 epochs=10, batch_size=10,
                                                 eta=1e-8,
                                                 lmbd=0, fixed_LR=False,
                                                 method="regression",
                                                 activation="sigmoid",
                                                 seed = self.paramData['
                                                     ↪ RandomSeed'])
        '''
        n_hidden_neurons = []
        for layer in range(self.paramData['NHiddenLayers']):
            n_hidden_neurons.append(self.paramData['NHiddenNeurons'])
        for layer in range(1, self.paramData['NHiddenLayers'], 1):
            n_hidden_neurons[layer] = int(n_hidden_neurons[layer-1]/2)
        #print(n_hidden_neurons)

        NN_reg_original = neural.NeuralNetworkML(X_CDOM, y_CDOM,
                                                 trainingShare=1-self.
                                                     ↪ paramData['TestSize'],
                                                 n_hidden_layers=self.
                                                     ↪ paramData['
                                                     ↪ NHiddenLayers'],
```

```
                                                n_hidden_neurons=
                                                    ↪ n_hidden_neurons,
                                                n_categories=1,
                                                epochs=self.paramData['epochs
                                                    ↪ '],
                                                batch_size=self.paramData['
                                                    ↪ BatchSize'],
                                                eta=self.paramData['alpha'],
                                                lmbd=0,
                                                fixed_LR=False,
                                                method="regression",
                                                activation="sigmoid",
                                                seed = self.paramData['
                                                    ↪ RandomSeed'])



        NN_reg_original.train()
        # Plotting results
        self.funcs.PlotResultsManualFFNN(NN_reg_original,
                                         CDOM,
                                         self.paramData['type'],
                                         self.paramData['OutputPath'],
                                         self.paramData['epochs'],
                                         self.paramData['BatchSize'])
    elif self.paramData['type'] == 'xgb':
        X_train, X_test, y_train, y_test = train_test_split(X_CDOM, y_CDOM,
                                                            train_size=1-self.
                                                                ↪ paramData['
                                                                ↪ TestSize'],
                                                            test_size = self.
                                                                ↪ paramData['
                                                                ↪ TestSize'],
                                                            random_state=self.
                                                                ↪ paramData['
                                                                ↪ RandomSeed'
                                                                ↪ ])
        # initialising xgboosting
        xgboosting = xgb.XGBoosting()
        model = xgboosting.RunModel(X_train, X_test,
                                    y_train, y_test,
                                    X_CDOM, X_CDOM_diag_mesh,
                                    CDOM, CDOM_sorted,
                                    self.paramData['OutputPath'])
        #Get best model by test MSE
        XGboost_best_model_index = model.best_iteration
        XGboost_best_iteration = model.get_booster().best_ntree_limit
        MSE_per_epoch = model.evals_result()

        # make predictions for test data
        y_pred = model.predict(X_test, ntree_limit=XGboost_best_iteration)
```

```
        y_pred_train = model.predict(X_train)
        #predictions = [round(value) for value in y_pred]

        best_prediction = model.predict(X_CDOM, ntree_limit=
            ↪ XGboost_best_iteration)
        CDOM_pred = best_prediction.copy()  #CDOM_pred.shape: (2556,)
            ↪ CDOM_pred are the predicted BCC Bray distances for CDOM value
            ↪ pairs
        CDOM_pred_fine_mesh = model.predict(X_CDOM_diag_mesh, ntree_limit=
            ↪ XGboost_best_iteration)
        '''
        y_pred,\
        y_pred_train,\
        MSE_per_epoch,\
        CDOM_pred, \
        CDOM_pred_fine_mesh, \
        XGboost_best_model_index = xgboosting.RunModel(X_train, X_test,
                                                       y_train, y_test,
                                                       X_CDOM, X_CDOM_diag_mesh,
                                                       CDOM, CDOM_sorted,
                                                       self.paramData['OutputPath
                                                           ↪ '])
        '''
        # plotting 3d plots and mse for XGBoost
        self.funcs.PlotResultsXGBoost(CDOM, CDOM_sorted,
                                      X_CDOM_diag_mesh,
                                      CDOM_pred_fine_mesh,
                                      CDOM_pred,
                                      self.paramData['OutputPath'],
                                      y_pred,
                                      y_pred_train,
                                      MSE_per_epoch,
                                      y_train, y_test,
                                      XGboost_best_model_index)
    elif self.paramData['type'] == 'rf_main':
        rf = random_forest.RandomForest()
        # Laurent
        population_size, metadata = rf.read_data(False, False)
        predictions, test_y, ML_ = rf.prepare_data(population_size,
                                                   metadata,
                                                   self.paramData['TestSize'],
                                                   self.paramData['RandomSeed'
                                                       ↪ ])
        all_predictions = rf.predict_all_metadata(population_size, metadata,
            ↪ ML_)
        # we will compare the outcome with xgboost
        def MergeTable(var_list, metadata_variables):
            table = pd.DataFrame(np.concatenate((var_list), axis=1))
            table.columns = metadata_variables
            return table
```

```
        def PredictMetadata(ASV_table, metadata_variables, train_size,
            ↪ test_size, seed):
            X_ASV = ASV_table
            X_ASV.columns = [''] * len(X_ASV.columns)
            X_ASV = X_ASV.to_numpy()
            metadata_list = []
            for i in metadata_variables:
                #y_CDOM = metadata.loc[:, i][:, np.newaxis]

                # split data into train and test sets
                y_meta = metadata.loc[:, i] #Requires 1d array
                X_train, X_test, y_train, y_test = train_test_split(X_ASV,
                    ↪ y_meta,
                                                                    train_size
                                                                        ↪  =
                                                                        ↪ train_size
                                                                        ↪ ,
                                                                    test_size
                                                                        ↪  =
                                                                        ↪ test_size
                                                                        ↪ ,
                                                                    random_state
                                                                        ↪ =
                                                                        ↪ seed
                                                                        ↪ )

                # fit model no training data
                model = XGBRegressor(objective='reg:squarederror')
                model.fit(X_train, y_train, eval_set=[(X_train, y_train), (
                    ↪ X_test, y_test)],
                          eval_metric='rmse', early_stopping_rounds=100,
                              ↪ verbose=False)

                #Get best model by test MSE
                XGboost_best_model_index = model.best_iteration
                XGboost_best_iteration = model.get_booster().best_ntree_limit

                # make predictions for full dataset
                y_pred = model.predict(X_ASV, ntree_limit=
                    ↪ XGboost_best_iteration)
                metadata_list.append(y_pred[:, np.newaxis])
            return MergeTable(metadata_list, metadata_variables)

        var_list = ["Latitude","Longitude","Altitude","Area","Depth","
            ↪ Temperature","Secchi","O2","CH4","pH","TIC","SiO2","KdPAR"]
        train_size = 1-self.paramData['TestSize']
        test_size = self.paramData['TestSize']
        seed = self.paramData['RandomSeed']
        predicted_metadata = PredictMetadata(ASV, var_list, train_size,
            ↪ test_size, seed)
```

```
            with pd.option_context('display.max_rows', None, 'display.max_columns'
                ↪ , None):  # more options can be specified also
                print(predicted_metadata)


        elif self.paramData['type'] == 'rf_side':
            # retrieving network data
            NNdata = self.PreProcessing()
            rf = random_forest.RandomForest()
            seed = self.paramData['RandomSeed']
            clfs, scores_test, scores_train = rf.predict_t(self.X_train, self.
                ↪ X_test, self.y_train_l, self.y_test_l, seed)

        elif self.paramData['type'] == 'all':
            '''
            Neural Network
            '''
            # Neural network with multiple layers - regression - BCC Bray
                ↪ distances by CDOM - original data
            X_CDOM = CDOM.loc[:,["CDOM.x1", "CDOM.x2"]].to_numpy()
            y_CDOM = CDOM.loc[:,"ASV.dist"].to_numpy()[:, np.newaxis] #Original
                ↪ data

            n_hidden_neurons = []
            for layer in range(self.paramData['NHiddenLayers']):
                n_hidden_neurons.append(self.paramData['NHiddenNeurons'])
            for layer in range(1, self.paramData['NHiddenLayers'], 1):
                n_hidden_neurons[layer] = int(n_hidden_neurons[layer-1]/2)
            #print(n_hidden_neurons)

            NN_reg_original = neural.NeuralNetworkML(X_CDOM, y_CDOM,
                                                     trainingShare=1-self.
                                                         ↪ paramData['TestSize'],
                                                     n_hidden_layers=self.
                                                         ↪ paramData['
                                                         ↪ NHiddenLayers'],
                                                     n_hidden_neurons=
                                                         ↪ n_hidden_neurons,
                                                     n_categories=1,
                                                     epochs=self.paramData['epochs
                                                         ↪ '],
                                                     batch_size=self.paramData['
                                                         ↪ BatchSize'],
                                                     eta=self.paramData['alpha'],
                                                     lmbd=0,
                                                     fixed_LR=False,
                                                     method="regression",
                                                     activation="sigmoid",
                                                     seed = self.paramData['
                                                         ↪ RandomSeed'])
```

```
NN_reg_original.train()

x_mesh = np.log10(np.arange(min(CDOM.loc[:,"CDOM.x1"]), max(CDOM.loc
    ↪ [:,"CDOM.x2"]) + 0.01, 0.01)) + 1
y_mesh = x_mesh.copy()
x_mesh, y_mesh = np.meshgrid(x_mesh,y_mesh)
X_CDOM_mesh = self.funcs.pdCat(x_mesh.ravel()[:, np.newaxis], y_mesh.
    ↪ ravel()[:, np.newaxis]).to_numpy()
best_prediction = NN_reg_original.model_prediction(X_CDOM_mesh,
    ↪ NN_reg_original.accuracy_list.index(min(NN_reg_original.
    ↪ accuracy_list)))

x_mesh = np.arange(min(CDOM.loc[:,"CDOM.x1"]), max(CDOM.loc[:,"CDOM.x2
    ↪ "]) + 0.01, 0.01)
y_mesh = x_mesh.copy()
x_mesh, y_mesh = np.meshgrid(x_mesh,y_mesh)

ff_pred_original = best_prediction.copy()
ff_pred_original = np.reshape(ff_pred_original, (363, 363))
ff_pred_original[x_mesh-y_mesh==0] = np.nan
ff_pred_original[x_mesh>y_mesh] = np.nan

'''
XGBoost part
'''
X_CDOM = CDOM.loc[:,["CDOM.x1", "CDOM.x2"]] #Molten meshgrid CDOM
    ↪ values for real data BCC Bray distances
X_CDOM_diag_mesh = CDOM_diag_mesh.loc[:,["CDOM.x1", "CDOM.x2"]] #
    ↪ Molten meshgrid CDOM values for generating predicted BCC Bray
    ↪ distances
y_CDOM = CDOM.loc[:,"ASV.dist"]

X_train, X_test, y_train, y_test = train_test_split(X_CDOM, y_CDOM,
                                                    train_size=1-self.
                                                        ↪ paramData['
                                                        ↪ TestSize'],
                                                    test_size = self.
                                                        ↪ paramData['
                                                        ↪ TestSize'],
                                                    random_state=self.
                                                        ↪ paramData['
                                                        ↪ RandomSeed'
                                                        ↪ ])
# initialising xgboosting
xgboosting = xgb.XGBoosting()
model = xgboosting.RunModel(X_train, X_test,
                            y_train, y_test,
                            X_CDOM, X_CDOM_diag_mesh,
                            CDOM, CDOM_sorted,
                            self.paramData['OutputPath'])
```

```
#Get best model by test MSE
XGboost_best_model_index = model.best_iteration
XGboost_best_iteration = model.get_booster().best_ntree_limit
MSE_per_epoch = model.evals_result()

# make predictions for test data
y_pred = model.predict(X_test, ntree_limit=XGboost_best_iteration)
y_pred_train = model.predict(X_train)
#predictions = [round(value) for value in y_pred]

best_prediction = model.predict(X_CDOM, ntree_limit=
    ↪ XGboost_best_iteration)
CDOM_pred = best_prediction.copy()  #CDOM_pred.shape: (2556,)
    ↪ CDOM_pred are the predicted BCC Bray distances for CDOM value
    ↪ pairs
CDOM_pred_fine_mesh = model.predict(X_CDOM_diag_mesh, ntree_limit=
    ↪ XGboost_best_iteration)

'''
Simple OLS - generating design matrix out of data set etc.
'''
reg = regression.Regression()
X_mesh = reg.GenerateMesh(0.21, 3.83, 0.21, 3.83, 0.01, 0.01,
    ↪ log_transform=True) # The low number of points on the higher
    ↪ end of the gradient causes distortions for linear regression
X_mesh_degree_list = reg.DesignMatrixList(X_mesh[0], X_mesh[1], 12)
    ↪ [1:]
X_degree_list = reg.DesignMatrixList(CDOM.loc[:,"CDOM.x1"], CDOM.loc
    ↪ [:,"CDOM.x2"], 12)[1:]
X_degree_list_subset = []

z = CDOM_pred #XGboost-predicted values
z = CDOM.loc[:,"ASV.dist"] #Original data
#ebv_no_resampling = reg.
    ↪ generate_error_bias_variance_without_resampling(X_degree_list,
    ↪ 1)
#ebv_resampling = reg.generate_error_bias_variance_with_resampling(
    ↪ X_degree_list, 1, 100)
#reg.ebv_by_model_complexity(ebv_resampling)
#reg.training_vs_test(ebv_no_resampling)

CDOM_pred_reg = X_mesh_degree_list[8] @ reg.beta_SVD(X_degree_list[8],
    ↪  CDOM_pred)
#print(pd.DataFrame(X_mesh_degree_list[1]))
#print(CDOM_pred_reg)
#with pd.option_context('display.max_rows', None, 'display.max_columns
    ↪ ', None):  # more options can be specified also
#  print(pd.DataFrame(CDOM_pred_reg))

x_mesh_reg = np.arange(min(CDOM.loc[:,"CDOM.x1"]), max(CDOM.loc[:,"
    ↪ CDOM.x2"]) + 0.01, 0.01)
```

```
y_mesh_reg = x_mesh_reg.copy()
x_mesh_reg, y_mesh_reg = np.meshgrid(x_mesh_reg,y_mesh_reg)
X_CDOM_mesh = self.funcs.pdCat(x_mesh_reg.ravel()[:, np.newaxis],
    ↪ y_mesh_reg.ravel()[:, np.newaxis])
#print(pd.DataFrame(X_CDOM_mesh))
#print("CDOM_pred_reg.shape", CDOM_pred_reg.shape)
z_CDOM_mesh_pred = np.reshape(CDOM_pred_reg, (x_mesh_reg.shape[0],
    ↪ x_mesh_reg.shape[0]))
z_CDOM_mesh_pred[x_mesh_reg-y_mesh_reg==0] = np.nan
z_CDOM_mesh_pred[x_mesh_reg>y_mesh_reg] = np.nan

'''
Neural Network with data from XGBoost
'''
X_CDOM = CDOM.loc[:,["CDOM.x1", "CDOM.x2"]].to_numpy()
y_CDOM = CDOM_pred[:, np.newaxis] #Predicted data from XGboost

n_hidden_neurons = []
for layer in range(self.paramData['NHiddenLayers']):
    n_hidden_neurons.append(self.paramData['NHiddenNeurons'])
for layer in range(1, self.paramData['NHiddenLayers'], 1):
    n_hidden_neurons[layer] = int(n_hidden_neurons[layer-1]/2)
#print(n_hidden_neurons)

NN_reg = neural.NeuralNetworkML(X_CDOM, y_CDOM,
                                trainingShare=1-self.
                                    ↪ paramData['TestSize'],
                                n_hidden_layers=self.
                                    ↪ paramData['
                                    ↪ NHiddenLayers'],
                                n_hidden_neurons=
                                    ↪ n_hidden_neurons,
                                n_categories=1,
                                epochs=self.paramData['epochs
                                    ↪ '],
                                batch_size=self.paramData['
                                    ↪ BatchSize'],
                                eta=self.paramData['alpha'],
                                lmbd=0,
                                fixed_LR=False,
                                method="regression",
                                activation="sigmoid",
                                seed = self.paramData['
                                    ↪ RandomSeed'])

NN_reg.train()
test_predict = NN_reg.predict(NN_reg.XTest)
print(NN_reg.accuracy_list)

#Use log-transformed CDOM values for creating design matrix, then plot
    ↪  on original values
```

```
x_mesh = np.log10(np.arange(min(CDOM.loc[:,"CDOM.x1"]), max(CDOM.loc
    ↪ [:,"CDOM.x2"]) + 0.01, 0.01)) + 1
y_mesh = x_mesh.copy()
x_mesh, y_mesh = np.meshgrid(x_mesh,y_mesh)
X_CDOM_mesh = self.funcs.pdCat(x_mesh.ravel()[:, np.newaxis], y_mesh.
    ↪ ravel()[:, np.newaxis]).to_numpy()
best_prediction = NN_reg.model_prediction(X_CDOM_mesh, NN_reg.
    ↪ accuracy_list.index(min(NN_reg.accuracy_list)))

x_mesh = np.arange(min(CDOM.loc[:,"CDOM.x1"]), max(CDOM.loc[:,"CDOM.x2
    ↪ "]) + 0.01, 0.01)
y_mesh = x_mesh.copy()
x_mesh, y_mesh = np.meshgrid(x_mesh,y_mesh)

ff_pred = best_prediction.copy()
ff_pred = np.reshape(ff_pred, (363, 363))
ff_pred[x_mesh-y_mesh==0] = np.nan
ff_pred[x_mesh>y_mesh] = np.nan

'''
Plotting 3d graphs for all data
'''
fontsize = 6
#Compare raw data to XGboost, neural network predicted data and
    ↪ XGboost predicted data smoothed with neural network
fig = plt.figure(figsize=plt.figaspect(0.5))
ax = fig.add_subplot(2, 3, 1, projection='3d')
ax.set_title("BCC␣Bray␣distances␣by␣sites'␣DOM", fontsize=fontsize)
#plt.subplots_adjust(left=0, bottom=0, right=2, top=2, wspace=0,
    ↪ hspace=0)
ax.view_init(elev=30.0, azim=300.0)
surf = ax.plot_trisurf(CDOM.loc[:,"CDOM.x1"], CDOM.loc[:,"CDOM.x2"],
    ↪ CDOM.loc[:,"ASV.dist"],
                       cmap='viridis', edgecolor='none')
# Customize the z axis.
ax.set_zlim(0.3, 1)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
ax.tick_params(labelsize=8)
ax.set_zlabel(zlabel="Bray␣distance")
ax.set_ylabel(ylabel="DOM␣site␣2")
ax.set_xlabel(xlabel="DOM␣site␣1")

# Set up the axes for the second plot
ax = fig.add_subplot(2, 3, 2, projection='3d')
#ax.set_title("XGboost-Predicted BCC Bray distances by sites' CDOM,
    ↪ dataset CDOM coordinates", fontsize=8)
ax.set_title("XGboost-Predicted␣BCC␣\n␣Bray␣distances␣by␣sites'␣DOM",
    ↪ fontsize=fontsize)
ax.view_init(elev=30.0, azim=300.0)
```

```
# Plot the surface.
ax.plot_trisurf(CDOM.loc[:,"CDOM.x1"], CDOM.loc[:,"CDOM.x2"],
    ↪ CDOM_pred, #197109 datapoints
                cmap='viridis', edgecolor='none')

# Customize the z axis.
z_range =  (np.nanmax(CDOM_pred) - np.nanmin(CDOM_pred))
ax.set_zlim(np.nanmin(CDOM_pred) - z_range, 1)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
ax.tick_params(labelsize=8)
ax.set_zlabel(zlabel="Bray␣distance")
ax.set_ylabel(ylabel="DOM␣site␣2")
ax.set_xlabel(xlabel="DOM␣site␣1")

# Set up the axes for the third plot
ax = fig.add_subplot(2, 3, 3, projection='3d')
#ax.set_title("OLS (SVD) regression-predicted BCC Bray distances by
    ↪ sites' CDOM, CDOM 0.01 step meshgrid", fontsize=6)
ax.set_title("OLS␣(SVD)␣regression-predicted␣\n␣BCC␣Bray␣distances␣by␣
    ↪ sites'␣DOM", fontsize=fontsize)
ax.view_init(elev=30.0, azim=300.0)

# Plot the surface.
ax.plot_trisurf(x_mesh_reg.ravel(), y_mesh_reg.ravel(),
    ↪ z_CDOM_mesh_pred.ravel(), cmap='viridis', #197109 datapoints
                vmin=np.nanmin(z_CDOM_mesh_pred), vmax=np.nanmax(
                    ↪ z_CDOM_mesh_pred),
                edgecolor='none')

# Customize the z axis.
z_range =  (np.nanmax(z_CDOM_mesh_pred) - np.nanmin(z_CDOM_mesh_pred))
ax.set_zlim(np.nanmin(z_CDOM_mesh_pred) - z_range, 1)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
ax.tick_params(labelsize=8)
ax.set_zlabel(zlabel="Bray␣distance")
ax.set_ylabel(ylabel="DOM␣site␣2")
ax.set_xlabel(xlabel="DOM␣site␣1")

# Set up the axes for the fourth plot
ax = fig.add_subplot(2, 3, 4, projection='3d')
#ax.set_title("NN-smoothed XGboost-predicted BCC Bray distances by
    ↪ sites' CDOM, CDOM 0.01 step meshgrid", fontsize=6)
ax.set_title("NN-smoothed␣XGboost-predicted␣\n␣BCC␣Bray␣distances␣by␣
    ↪ sites'␣DOM", fontsize=fontsize)
ax.view_init(elev=30.0, azim=300.0)

# Plot the surface.
ax.plot_trisurf(x_mesh.ravel(), y_mesh.ravel(), ff_pred.ravel(), #
    ↪ 197109 datapoints
```

```
                cmap='viridis', edgecolor='none',
                vmin=np.nanmin(ff_pred), vmax=np.nanmax(ff_pred))

# Customize the z axis.
z_range =  (np.nanmax(ff_pred) - np.nanmin(ff_pred))
ax.set_zlim(np.nanmin(ff_pred) - z_range, 1)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
ax.tick_params(labelsize=8)
ax.set_zlabel(zlabel="Bray distance")
ax.set_ylabel(ylabel="DOM site 2")
ax.set_xlabel(xlabel="DOM site 1")

# Set up the axes for the fifth plot
ax = fig.add_subplot(2, 3, 5, projection='3d')
#ax.set_title("NN-predicted BCC Bray distances by sites' CDOM, CDOM
    ↪ 0.01 step meshgrid", fontsize=8)
ax.set_title("NN-predicted BCC Bray \n distances by sites' DOM",
    ↪ fontsize=fontsize)
ax.view_init(elev=30.0, azim=300.0)

# Plot the surface.
ax.plot_trisurf(x_mesh.ravel(), y_mesh.ravel(), ff_pred_original.ravel
    ↪ (), #197109 datapoints
                cmap='viridis', edgecolor='none',
                vmin=np.nanmin(ff_pred_original), vmax=np.nanmax(
                    ↪ ff_pred_original))

# Customize the z axis.
z_range =  (np.nanmax(ff_pred_original) - np.nanmin(ff_pred_original))
ax.set_zlim(np.nanmin(ff_pred_original) - z_range, 1)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
ax.tick_params(labelsize=8)
ax.set_zlabel(zlabel="Bray distance")
ax.set_ylabel(ylabel="DOM site 2")
ax.set_xlabel(xlabel="DOM site 1")

# Set up the axes for the sixth plot
ax = fig.add_subplot(2, 3, 6, projection='3d')
#ax.set_title("XGboost-predicted BCC Bray distances by sites' CDOM,
    ↪ CDOM 0.01 step meshgrid", fontsize=8)
ax.set_title("XGboost-predicted BCC Bray \n distances by sites' DOM",
    ↪ fontsize=fontsize)
ax.view_init(elev=30.0, azim=300.0)


# Plot the surface.
ax.plot_trisurf(X_CDOM_diag_mesh.loc[:,"CDOM.x1"], X_CDOM_diag_mesh.
    ↪ loc[:,"CDOM.x2"], CDOM_pred_fine_mesh, #197109 datapoints
                cmap='viridis', edgecolor='none')
```

```
            # Customize the z axis.
            z_range =  (np.nanmax(CDOM_pred_fine_mesh) - np.nanmin(
                ↪ CDOM_pred_fine_mesh))
            ax.set_zlim(np.nanmin(CDOM_pred_fine_mesh) - z_range, 1)
            ax.zaxis.set_major_locator(LinearLocator(10))
            ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
            ax.tick_params(labelsize=8)
            ax.set_zlabel(zlabel="Bray␣distance")
            ax.set_ylabel(ylabel="DOM␣site␣2")
            ax.set_xlabel(xlabel="DOM␣site␣1")

            #filename = self.paramData['OutputPath']
            filename = self.paramData['OutputPath'] + '/' + 'everything_3d' + '.
                ↪ png'
            fig.savefig(filename)

            plt.show()



    def Normalize(self, *args):
        x = args[0]
        return (x-np.amin(x))/(np.amax(x)-np.amin(x))

'''
Entry Point of the program
'''
if __name__ == '__main__':
    # Estimate how much time it took for program to work
    startTime = time.time()
    '''
    Configuring Network via Parameter file
    '''
    # Getting parameter file
    paramFile = 'ParameterFile.yaml'
    # Class Object Instantiation - passing
    # configuration from parameter file
    pipe = MainPipeline(paramFile)
    pipe.Run()

    # End time of the program
    endTime = time.time()
    print("--␣Program␣finished␣at␣%s␣sec␣--" % (endTime - startTime))
```

Listing 4: "Neural Network Module"

```
"""
@author: maksymb
"""
```

```
import sys
import numpy as np
import pandas as pd
import random
import keras
from keras.models import Model
from keras.layers import Input, Dense, Dropout, Lambda, Subtract
from sklearn.preprocessing import OneHotEncoder, LabelEncoder, StandardScaler,
    ↪ RobustScaler, MinMaxScaler
from sklearn.model_selection import train_test_split

import funclib

"""
Class which constructs the Neural Network
"""
class NeuralNetwork:
    # constructor
    def __init__(self, *args):
        # Getting NN data
        NNdata = args[0]
        # retrieving Network Architecture
        self.seed, self.NNType, self.NHiddenLayers, self.NTotalLayers, \
        self.Ndata, self.NInputNeurons, self.NHiddenNeurons, \
        self.NOutputNeurons, self.aHiddenFunc, self.aOutputFunc, self.NNArch, \
        self.weights, self.epochs, self.alpha, self.lmbd, self.optimization, \
        self.loss, self.batchSize = NNdata

        # Printing current network configuration
        print((u'''
␣␣␣␣␣␣␣␣==========================================
␣␣␣␣␣␣␣␣␣␣␣␣␣␣Start␣{}␣Neural␣Network
␣␣␣␣␣␣␣␣==========================================
␣␣␣␣␣␣␣␣No.␣of␣hidden␣layers:␣␣␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣No.␣of␣input␣data:␣␣␣␣␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣No.␣of␣input␣neurons:␣␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣No.␣of␣hidden␣neurons:␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣No.␣of␣output␣neurons:␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣Activ.␣Func␣in␣Hidden␣Layer:␣{}
␣␣␣␣␣␣␣␣Activ.␣Func␣in␣Output␣Layer:␣{}
␣␣␣␣␣␣␣␣No.␣of␣epochs␣to␣see:␣␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣Optimization␣Algorithm:␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣Learning␣Rate␣,␣\u03B1:␣␣␣␣␣␣␣␣␣␣␣{}
␣␣␣␣␣␣␣␣Regularization␣param␣,␣\u03BB:␣␣␣␣␣{}
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣'''.format(self.NNType,
                                    self.NHiddenLayers,
                                    self.Ndata,
                                    self.NInputNeurons,
                                    self.NHiddenNeurons,
                                    self.NOutputNeurons,
                                    self.NNArch[1]['AF'],
```

```
                                self.NNArch[self.NTotalLayers-1]['AF'],
                                self.epochs,
                                self.optimization,
                                self.alpha,
                                self.lmbd)))

        # Instantiating object variable from Functions Library
        self.funcs = funclib.Functions(self.seed, self.loss)

    '''
    Getting Layer Architecture
    '''
    def GetDenseArchitecture(self, *args):
        # shape of the inputs
        inputShape = args[0] #(self.NInputNeurons, )
        # customizing our model
        inputs = Input(shape = inputShape)
        '''
        we check if the NN is siamese (or triplet), than the last layer's
        activation function should be the same for each layer,
        because we will use sigmoid (or other) when merging layers.
        If activation function is simple Feed Forward Neural Network (FFNN)
        '''
        if self.NNType == 'snn_keras':
            # simply getting activation function for the layer before this one
            lastFunc = self.NNArch[self.NTotalLayers-2]['AF']
            # the neurons in the last layer
            lastNeurons = self.NHiddenNeurons
        elif self.NNType == 'tnn_keras':
            # simply getting activation function for the layer before this one
            lastFunc = self.NNArch[self.NTotalLayers-2]['AF']
            # the neurons in the last layer
            lastNeurons = self.NHiddenNeurons
        elif self.NNType == 'ffnn_keras':
            lastFunc = self.NNArch[self.NTotalLayers-1]['AF']
            lastNeurons = self.NOutputNeurons
        else:
            print('Check␣Your␣neural.py!')
            sys.exit()
        # Instantiating the Dense Neural Network
        # adding (connecting) layers
        for layer in range(0, self.NTotalLayers):
            # first layer
            if layer == 0:
                # getting activation function for the first layer
                activation = self.funcs.GetActivation(self.NNArch[layer]['AF'])
                weights = self.funcs.GetWeights(self.weights)
                hidden = Dense(self.NInputNeurons, activation = activation,
                               kernel_initializer=weights, bias_initializer='zeros
                                   ↪ ')(inputs)#,
                               #kernel_regularizer=keras.regularizers.l1_l2(self.
```

```
                                ↪ lmbd, self.lmbd))(inputs)
            # adding drop-out layer
            hidden = Dropout(rate=self.lmbd)(hidden)
        # last layer
        elif layer == self.NTotalLayers-1:
            # getting activation function for the last layer
            activation = lastFunc
            weights = self.funcs.GetWeights(self.weights)
            outputs = Dense(lastNeurons, activation = activation,
                            kernel_initializer=weights, bias_initializer='
                                ↪ zeros')(hidden)#,
                            #kernel_regularizer=keras.regularizers.l1_l2(self.
                                ↪ lmbd, self.lmbd))(hidden)
            outputs = Dropout(rate=self.lmbd)(outputs)
        # intermediate layers
        else:
            # getting activation function for the hidden layers
            activation = self.funcs.GetActivation(self.NNArch[layer]['AF'])
            weights = self.funcs.GetWeights(self.weights)
            hidden = Dense(self.NHiddenNeurons, activation=activation,
                           kernel_initializer=weights,  bias_initializer='
                               ↪ zeros')(hidden)#,
                           #kernel_regularizer=keras.regularizers.l1_l2(self.
                               ↪ lmbd, self.lmbd))(hidden)
            # adding drop-out layer
            hidden = Dropout(rate=self.lmbd)(hidden)
    # building the model
    model = Model(inputs = inputs, outputs = outputs)

    return model
'''
Method to Build the Neural Network Model
(based on the configuration)
'''
def BuildModel(self, *args):
    # Checking the type of Network to Build
    if self.NNType == 'ffnn_keras':
        # the shape inputs
        inputShape = (self.NInputNeurons,)
        # getting model architecture
        model = self.GetDenseArchitecture(inputShape)
        #functions = funclib.Functions(self.seed)
        # retrieving loss function
        loss = self.funcs.GetLoss
        # retrieving gradient
        optimizer = self.funcs.GetGradient(self.optimization, self.alpha)
        # compiling model
        model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])
        # returning the constructed model
        return model
```

```
        # siamese neural network
        elif self.NNType == 'snn_keras':
            # shape of the inputs
            inputShape = (self.NInputNeurons, )
            # customizing our model
            input1 = Input(shape=inputShape)
            input2 = Input(shape=inputShape)
            # getting dense layers -  for siamese the first part is the same
            model = self.GetDenseArchitecture(inputShape)
            # run through first part of network
            run1 = model(input1)
            run2 = model(input2)
            # creating merge layer
            mergeLayer = Lambda(self.funcs.euclidean_distance, output_shape=self.
                ↪ funcs.eucl_dist_output_shape)([run1, run2])
            # getting activation function for the output layer
            activation = self.funcs.GetActivation(self.NNArch[self.NTotalLayers
                ↪ -1]['AF'])
            outputLayer = Dense(self.NOutputNeurons, activation=activation)(
                ↪ mergeLayer)
            model = Model(inputs=[input1, input2], outputs=outputLayer)
            # retrieving loss function
            loss = self.funcs.GetLoss
            # retrieving gradient
            optimizer = self.funcs.GetGradient(self.optimization, self.alpha)
            # compiling model
            model.compile(optimizer=optimizer, loss=loss, metrics=[self.funcs.acc
                ↪ ])

            return model

        # triplet neural network
        elif self.NNType == 'tnn_keras':
            # shape of the inputs
            inputShape = (self.NInputNeurons, )
            # customizing our model
            input1 = Input(shape=inputShape)
            input2 = Input(shape=inputShape)
            input3 = Input(shape=inputShape)
            # getting dense layers -  for triplet the first part is the same
            model = self.GetDenseArchitecture(inputShape)
            # run through first part of the network
            pos = model(input1)
            neg = model(input2)
            sam = model(input3)
            # Check distance between positive and sample
            merge_layer1 = Lambda(self.funcs.euclidean_distance, output_shape=self
                ↪ .funcs.eucl_dist_output_shape)([pos,sam])
            # Check distance between negative and sample
            merge_layer2 = Lambda(self.funcs.euclidean_distance, output_shape=self
                ↪ .funcs.eucl_dist_output_shape)([neg,sam])
```

```
            #Compare distances
            loss_layer = Subtract()([merge_layer1, merge_layer2])
            model = Model(inputs=[input1, input2, input3], outputs=loss_layer)
            # retrieving loss function
            loss = self.funcs.GetLoss
            # retrieving gradient
            optimizer = self.funcs.GetGradient(self.optimization, self.alpha)
            model.compile(loss=loss, optimizer=optimizer, metrics=[self.funcs.
                ↪ triplet_acc])

            return model

    '''
    Training the model
    '''
    def TrainModel(self, *args):
        # getting model to train
        model = args[0]
        # train the model using keras
        #logger = MyLogger(n=10)
        if self.NNType == 'ffnn_keras':
            # training data
            X1 = args[1]
            X2 = args[2]
            Y1 = args[3]
            Y2 = args[4]
            history = model.fit(X1, Y1,
                                #validation_split=0.5,
                                validation_data=(X2, Y2),
                                batch_size=self.batchSize,
                                epochs=self.epochs,
                                verbose=0)
        elif self.NNType == 'snn_keras':
            pairs_train  = args[1]
            labels_train = args[2]
            pairs_test   = args[3]
            labels_test  = args[4]
            history = model.fit([pairs_train[:, 0], pairs_train[:, 1]],
                ↪ labels_train[:],
                                validation_data=([pairs_test[:, 0], pairs_test[:,
                                    ↪ 1]], labels_test[:]),
                                batch_size=self.batchSize,
                                epochs=self.epochs,
                                verbose=0)
        elif self.NNType == 'tnn_keras':
            t_train      = args[1]
            train_labels = args[2]
            t_test       = args[3]
            test_labels  = args[4]
            history = model.fit([t_train[:,0], t_train[:,1], t_train[:,2]],
                ↪ train_labels[:],
```

```
                                validation_data=([t_test[:,0],t_test[:,1],t_test
                                    ↪ [:,2]],test_labels[:]),
                                batch_size=self.batchSize,
                                epochs=self.epochs,
                                verbose=0)
        return model, history

'''
Manual code for Neural Network
(the code left almost unchanged from the project 2)
'''
class NeuralNetworkML:  #Multiple hidden layers
    def __init__(
            self,
            X_data,
            Y_data,
            trainingShare=0.5,
            n_hidden_layers=2,
            n_hidden_neurons=[24,12],
            n_categories=10,
            epochs=10,
            batch_size=100,
            eta=0.1,
            lmbd=0.0,
            fixed_LR=False,
            method="classification",
            activation="sigmoid",
            seed = 1):

        self.seed = seed

        self.X_data_full = X_data
        self.Y_data_full = Y_data

        self.trainingShare = trainingShare
        self.method = method
        self.split_data = self.SplitData(self.X_data_full, self.Y_data_full, self.
            ↪ trainingShare)
        if self.method=="classification":
            self.XTrain = self.split_data[0].toarray()
            self.XTest = self.split_data[1].toarray()
        else:
            self.XTrain = self.split_data[0]
            self.XTest = self.split_data[1]
        self.yTrain = self.split_data[2]
        self.yTest = self.split_data[3]

        self.n_inputs = self.XTrain.shape[0]
        self.n_features = self.XTrain.shape[1]
        self.n_hidden_layers = n_hidden_layers
        self.n_hidden_neurons = n_hidden_neurons
```

```
        self.n_categories = n_categories

        self.epochs = epochs
        self.batch_size = batch_size
        self.eta = eta
        self.lmbd = lmbd
        self.fixed_LR = fixed_LR
        self.activation = activation

        self.create_biases_and_weights()
        self.accuracy_list = []
        self.cost_list = []
        self.models = []
        self.current_epoch = 0
        self.w_dict = {}

    def create_biases_and_weights(self):
        for n in range(self.n_hidden_layers):
            exec("self.n_hidden_neurons_" + str(eval("n + 1")) + "=" + str(eval("
                ↪ self.n_hidden_neurons[n]")))
        for n in range(self.n_hidden_layers):
            if n==0:
                exec("self.hidden_weights_1 = np.random.randn(self.n_features, 
                    ↪ self.n_hidden_neurons_1)")
                exec("self.hidden_bias_1 = np.zeros((1, self.n_hidden_neurons_1)) 
                    ↪ + 0.01")
            else:
                exec("self.hidden_weights_" + str(eval("n + 1")) + "= np.random.
                    ↪ randn(self.n_hidden_neurons_" + str(eval("n")) + ", self.
                    ↪ n_hidden_neurons_" + str(eval("n+1")) + ")")
                exec("self.hidden_bias_" + str(eval("n + 1")) + "= np.zeros((1, 
                    ↪ self.n_hidden_neurons_" + str(eval("n + 1")) + ")) + 0.01")
            exec("self.output_weights = np.random.randn(self.n_hidden_neurons_" +
                ↪ str(eval("self.n_hidden_layers")) + ", self.n_categories)")
            exec("self.output_bias = np.zeros((1, self.n_categories)) + 0.01")

    def act(self, x):
        if self.activation=="sigmoid":
            return self.SigmoidFunction(x)
        elif self.activation=="ELU":
            return self.ELU(x, alpha=0.01)
        elif self.activation=="LeakyReLU":
            return self.LeakyReLU(x, alpha=0.01)

    def feed_forward(self):
        for n in range(self.n_hidden_layers):
            if n==0:
                self.z_h_1 = np.matmul(self.XTrain_batch, self.hidden_weights_1) +
                    ↪  self.hidden_bias_1
                self.a_h_1 = self.act(self.z_h_1)
            else:
```

```
                exec("self.z_h_" + str(eval("n␣+␣1")) + "␣=␣np.matmul(self.a_h_" +
                    ↪  str(eval("n")) + ",␣self.hidden_weights_" + str(eval("n␣+␣
                    ↪ 1")) + ")␣+␣self.hidden_bias_" + str(eval("n␣+␣1")))
                exec("self.a_h_" + str(eval("n␣+␣1")) + "␣=␣self.act(self.z_h_" +
                    ↪ str(eval("n␣+␣1")) + ")")
        exec("self.z_o␣=␣np.matmul(self.a_h_" + str(eval("self.n_hidden_layers"))
            ↪ + ",␣self.output_weights)␣+␣self.output_bias")
        self.probabilities = self.LogRegPredict(self.z_o)
        self.a_o = self.act(self.z_o)

    def feed_forward_out(self, X):
        for n in range(self.n_hidden_layers):
            if n==0:
                z_h_1 = np.matmul(X, self.hidden_weights_1) + self.hidden_bias_1
                a_h_1 = self.act(z_h_1)
            # feed-forward for output
            else:
                exec("z_h_" + str(eval("n␣+␣1")) + "␣=␣np.matmul(a_h_" + str(eval(
                    ↪ "n")) + ",␣self.hidden_weights_" + str(eval("n␣+␣1")) + ")␣
                    ↪ +␣self.hidden_bias_" + str(eval("n␣+␣1")))
                exec("a_h_" + str(eval("n␣+␣1")) + "␣=␣self.act(z_h_" + str(eval("
                    ↪ n␣+␣1")) + ")")
        z_o=eval("np.matmul(a_h_" + str(eval("self.n_hidden_layers")) + ",␣self.
            ↪ output_weights)␣+␣self.output_bias")
        a_o = self.act(z_o)
        if self.method=="classification":
            probabilities = self.LogRegPredict(z_o)
        elif self.method=="regression":
            yPred = z_o
            probabilities = z_o
        return probabilities

    def LogRegPredict(self, z_o):
        yPred = self.SigmoidFunction(z_o)
        for i in range(0, yPred.shape[0], 1):
            if yPred[i] <= 0.5:
                yPred[i] = 0
            else:
                yPred[i] = 1
        return yPred

    def backpropagation(self):
        w_list = [self.hidden_weights_1, self.hidden_bias_1]
        for n in range(self.n_hidden_layers - 1, -1, -1):
            if n + 1 == self.n_hidden_layers:
                if self.method=="regression":
                    error_output = (self.a_o - self.yTrain_batch) #Cost function:
                        ↪ derivative of mean squared error
                else:
                    error_output = (self.a_o - self.yTrain_batch) * self.a_o * (1
                        ↪ - self.a_o) #Cost function: derivative of cross-entropy
```

```
                if self.n_hidden_layers==1:
                    self.error_hidden_1 = np.matmul(error_output, self.
                        ↪ output_weights.T) * self.a_h_1 * (1 - self.a_h_1)
                    self.output_weights_gradient = np.matmul(self.a_h_1.T,
                        ↪ error_output)
                    self.output_bias_gradient = np.sum(error_output, axis=0)
                    self.hidden_weights_gradient_1 = np.matmul(self.XTrain_batch.T
                        ↪ , self.error_hidden_1)
                    self.hidden_bias_gradient_1 = np.sum(self.error_hidden_1, axis
                        ↪ =0)
                else:
                    exec("self.error_hidden_" + str(eval("self.n_hidden_layers"))
                        ↪ + "␣=␣np.matmul(error_output,␣self.output_weights.T)␣*␣
                        ↪ self.a_h_" + str(eval("self.n_hidden_layers")) + "␣*␣(1
                        ↪ ␣-␣self.a_h_" + str(eval("self.n_hidden_layers")) + ")"
                        ↪ )
                    exec("self.hidden_weights_gradient_" + str(eval("n␣+␣1")) + "␣
                        ↪ =␣np.matmul(self.a_h_" + str(eval("n")) + ".T,␣self.
                        ↪ error_hidden_" + str(eval("n␣+␣1")) + ")")
                    exec("self.hidden_bias_gradient_" + str(eval("n␣+␣1")) + "␣=␣
                        ↪ np.sum(self.error_hidden_" + str(eval("n␣+␣1")) + ",␣
                        ↪ axis=0)")
                    exec("self.output_weights_gradient␣=␣np.matmul(self.a_h_" +
                        ↪ str(eval("n␣+␣1")) + ".T,␣error_output)")
                self.output_bias_gradient = np.sum(error_output, axis=0)
            elif n > 0:
                exec("self.error_hidden_" + str(eval("n+1")) + "␣=␣np.matmul(self.
                    ↪ error_hidden_" + str(eval("n␣+␣2")) + ",␣self.
                    ↪ hidden_weights_" + str(eval("n␣+␣2")) + ".T)␣*␣self.a_h_" +
                    ↪  str(eval("n+1")) + "␣*␣(1␣-␣self.a_h_" + str(eval("n+1"))
                    ↪ + ")")
                exec("self.hidden_weights_gradient_" + str(eval("n␣+␣1")) + "␣=␣np
                    ↪ .matmul(self.a_h_" + str(eval("n")) + ".T,␣self.
                    ↪ error_hidden_" + str(eval("n␣+␣1")) + ")")
                exec("self.hidden_bias_gradient_" + str(eval("n␣+␣1")) + "␣=␣np.
                    ↪ sum(self.error_hidden_" + str(eval("n␣+␣1")) + ",␣axis=0)")
            else:
                if self.n_hidden_layers == 1:
                    self.error_hidden_1 = np.matmul(error_output, self.
                        ↪ output_weights.T) * self.a_h_1 * (1 - self.a_h_1)
                else:
                    self.error_hidden_1 = np.matmul(self.error_hidden_2, self.
                        ↪ hidden_weights_2.T) * self.a_h_1 * (1 - self.a_h_1)
                    self.hidden_weights_gradient_1 = np.matmul(self.XTrain_batch.T
                        ↪ , self.error_hidden_1)
                    self.hidden_bias_gradient_1 = np.sum(self.error_hidden_1, axis
                        ↪ =0)

        if self.lmbd > 0.0:
            self.output_weights_gradient += self.lmbd * self.output_weights
            for n in range(self.n_hidden_layers - 1, -1, -1):
```

```
                exec("self.hidden_weights_gradient_" + str(eval("n + 1")) + " += "
                    ↪ self.lmbd * self.hidden_weights_" + str(eval("n + 1")))

        self.output_weights -= self.eta * self.output_weights_gradient/self.
            ↪ batch_size
        self.output_bias -= self.eta * self.output_bias_gradient/self.batch_size

        for n in range(self.n_hidden_layers - 1, -1, -1):
            exec("self.hidden_weights_" + str(eval("n + 1")) + " -= self.eta * "
                ↪ self.hidden_weights_gradient_" + str(eval("n + 1")) + "/self.
                ↪ batch_size")
            exec("self.hidden_bias_" + str(eval("n + 1")) + " -= self.eta * self.
                ↪ hidden_bias_gradient_" + str(eval("n + 1")) + "/self.batch_size
                ↪ ")

            w_list.append(eval("self.hidden_weights_" + str(eval("n + 1"))))
            w_list.append(eval("self.hidden_bias_" + str(eval("n + 1"))))

        w_list.append(self.output_weights)
        w_list.append(self.output_bias)
        self.w_dict[self.current_epoch] = w_list

    def RescaleOutputToOriginal(self, z_old, z_new): #z_old is the vector of
        ↪ predicted values to rescale to the original scale of response values (
        ↪ z_new)
        max_old = max(z_old)
        min_old = min(z_old)
        max_new = max(z_new)
        min_new = min(z_new)
        z_rescaled = z_old.copy()
        for i in range(len(z_old)):
            z_rescaled[i] = (max_new - min_new)/(max_old - min_old) * (z_old[i] -
                ↪ max_old) + max_new
            #value_new = (max_new - min_new)/(max_old - min_old) * (value_old -
                ↪ max_old) + max_new
        return z_rescaled

    def model_prediction(self, X, iter):
        m_w = self.w_dict[iter-1] #Model weights
        m_o_w = m_w[-2] #Model output weights
        m_o_b = m_w[-1] #Model output bias
        m_h_w_1 = m_w[0] #Model hidden weights 1
        m_h_b_1 = m_w[1] #Model hidden bias 1
        if self.n_hidden_layers > 1:
            m_h_w_n = m_w[-4:0:-2]#[-1::-1] #Model hidden weights n
            m_h_b_n = m_w[-3:1:-2]#[-1::-1] #Model hidden bias n
            for i in range(self.n_hidden_layers-1):
                exec("m_h_w_" + str(eval("i + 2")) + " = m_h_w_n[i+1]")
                exec("m_h_b_" + str(eval("i + 2")) + " = m_h_b_n[i+1]")

        for n in range(self.n_hidden_layers):
```

```
            if n==0:
                m_z_h_1 = np.matmul(X, m_h_w_1) + m_h_b_1
                m_a_h_1 = self.act(m_z_h_1)
            # feed-forward for output
            else:
                exec("m_z_h_" + str(eval("n␣+␣1")) + "␣=␣np.matmul(m_a_h_" + str(
                    ↪ eval("n")) + ",␣m_h_w_" + str(eval("n␣+␣1")) + ")␣+␣m_h_b_"
                    ↪  + str(eval("n␣+␣1")))
                exec("m_a_h_" + str(eval("n␣+␣1")) + "␣=␣self.act(m_z_h_" + str(
                    ↪ eval("n␣+␣1")) + ")")
        m_z_o=eval("np.matmul(m_a_h_" + str(eval("self.n_hidden_layers")) + ",␣
            ↪ self.output_weights)␣+␣self.output_bias")
        m_a_o = self.act(m_z_o)
        if self.method=="classification":
            probabilities = self.LogRegPredict(m_z_o)
        elif self.method=="regression":
            yPred = self.RescaleOutputToOriginal(m_z_o, self.Y_data_full)
            probabilities = self.RescaleOutputToOriginal(m_z_o, self.Y_data_full)
        return probabilities

    def predict(self, X):
        probabilities = self.feed_forward_out(X)
        #return np.argmax(probabilities, axis=1)
        return probabilities

    def predict_probabilities(self, X):
        probabilities = self.feed_forward_out(X)
        return probabilities

    def train(self):
        t0, t1 = 5, 500
        self.accuracy_list.append(self.accuracy(self.yTest, self.predict(self.
            ↪ XTest)))
        for i in range(self.epochs):
            self.current_epoch = i
            self.shuffled_data = self.shuffle(self.XTrain, self.yTrain) # Rows for
                ↪  XTrain, yTrain are shuffled for each epoch.
            self.XTrain_shuffled = self.shuffled_data[0]
            self.yTrain_shuffled = self.shuffled_data[1]
            for batch in range(int(self.XTrain.shape[0]/self.batch_size)):
                self.XTrain_batch = self.XTrain_shuffled[self.batch_size * batch:
                    ↪ self.batch_size * (batch + 1), :] #Minibatch training data
                self.yTrain_batch = self.yTrain_shuffled[self.batch_size * batch:
                    ↪ self.batch_size * (batch + 1)] #Minibatch training data

                if self.fixed_LR==False:
                    t = i*int(self.XTrain.shape[0]/self.batch_size) + batch #
                        ↪ Variable learning rate
                    self.eta = self.step_length(t, t0, t1) #Variable learning rate
                #print(self.eta)
```

```
            self.feed_forward()
            self.backpropagation()
        self.accuracy_list.append(self.accuracy(self.yTest, self.predict(self.
            ↪ XTest)))
        print("Epoch␣" + str(i + 1) + "␣completed")

def shuffle(self, XTrain, yTrain):
    random.seed(self.seed)
    n_rows = list(range(0, XTrain.shape[0], 1))
    random.shuffle(n_rows)
    XTrain_post_shuffle = XTrain[n_rows,:]
    yTrain_post_shuffle = yTrain[n_rows]
    return XTrain_post_shuffle, yTrain_post_shuffle

def step_length(self,t,t0,t1):
    return t0/(t+t1)

def SplitData(self, X, y, trainingShare=0.5):
    seed  = 1
    if isinstance(X, pd.DataFrame):
        X = X.values
    if isinstance(y, pd.DataFrame):
        y = y.values
    XTrain, XTest, yTrain, yTest = train_test_split(X, y,
                                                    train_size=trainingShare,
                                                    test_size = 1-
                                                        ↪ trainingShare,
                                                    random_state=seed)
    return XTrain, XTest, yTrain, yTest

def accuracy(self, yTest, yPred):
    if self.method=="classification":
        return  (yTest.flatten() == yPred.flatten()).sum()/len(yTest.flatten()
            ↪ )
    if self.method=="regression":
        self.models.append(self.predict(self.X_data_full))
        return np.mean((self.yTest - self.predict(self.XTest))**2)

def SigmoidFunction(self, x):
    sigma_fn = np.vectorize(lambda x: 1/(1+np.exp(-x)))
    return 1/(1+np.exp(-x))

def LogRegPredict(self, z_o):
    yPred = self.SigmoidFunction(z_o)
    for i in range(0, yPred.shape[0], 1):
        if yPred[i] <= 0.5:
            yPred[i] = 0
        else:
            yPred[i] = 1
    return yPred
```

```
    def ELU(self, x, alpha=0.01):
        ao = x
        for i in range(0, x.shape[0], 1):
            for j in range(0, x.shape[1], 1):
                if x[i,j] < 0:
                    ao[i,j] = alpha*(np.exp(x[i,j]) - 1)
        return ao

    def LeakyReLU(self, x, alpha=0.01):
        ao = x
        for i in range(0, x.shape[0], 1):
            for j in range(0, x.shape[1], 1):
                if x[i,j] <= 0:
                    ao[i,j] = alpha*x[i,j]
        return ao
```

Listing 5: "Regression Class"

```
# Use regression on predicted values to get smooth function of BCC Bray distance
    ↪ by CDOM

import numpy as np
import pandas as pd
from sklearn.utils import resample
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib import scale
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from sklearn.model_selection import train_test_split


'''
Simple linear/logistic regression class -  code written as part of project 1, by
    ↪ Laurent et al
'''
class Regression:

    def GenerateMesh(self, x_min, x_max, y_min, y_max, step_x, step_y,
        ↪ log_transform=False):
        import numpy as np
        if log_transform==False:
            x = np.arange(x_min, x_max + step_x, step_x)  #Add range value to
                ↪ range end value due for last element being excluded from range
            y = np.arange(y_min, y_max + step_y, step_y)  #Add range value to
                ↪ range end value due for last element being excluded from range
        else:
            x = np.log10(np.arange(x_min, x_max + step_x, step_x)) + 1  #log1p
                ↪ transformation
            y = np.log10(np.arange(y_min, y_max + step_y, step_y)) + 1  #log1p
                ↪ transformation
```

```
        x, y = np.meshgrid(x,y)
        x = x.ravel()
        y = y.ravel()
        return x, y

    def GenerateDesignMatrix(self, x, y, order):
        X = np.c_[np.ones(len(x))]
        if order==0:
            return X
        elif order>0:
            X_str = "np.c_[np.ones(len(x)),"
            for i in range(1, order+1, 1):
                str_len = (i+1)*(2*i+1)-1
                poly_string = "x" * str_len
                poly_string_list = list(poly_string)
                for j in range(2*i-1, str_len, (2*(i+1)-1)):
                    poly_string_list[j] = ","
                for j in range(2*i, str_len, (2*(i+1)-1)):
                    poly_string_list[j] = "␣"
                for j in range(1, str_len, 2*i+1):
                    if i==2:
                        poly_string_list[j] = "*"
                    elif i>=3:
                        poly_string_list[j:(j+2*(i-1)):2] = (i-1)*"*"
                poly_string_list[-2:2*i-1:-(2*i+1)] = i*"y"
                for j in range(0, i, 1):
                    poly_string_list[-2*(1+j):2*i-1+j*(2*i+1):-(2*i+1)] = (i-j)*"y
                        ↪ "
                poly_string = "".join(poly_string_list)
                X_str = X_str + "␣" + poly_string
            X = list(X_str)
            X[-1] = "]"
            X = eval("".join(X))
            colnames = list(X_str.replace(",", "").replace("np.c_[np.ones(len(x))"
                ↪ , "").split("␣"))
        return X, colnames

    """
    # Print design matrix formatted as pandas dataframe
    DesignMat_out = GenerateDesignMatrix(0.21, 3.83, 0.21, 3.83, 0.01, 0.01, 5)
        ↪ #3.83
    DesignMat = pd.DataFrame(DesignMat_out[0])
    DesignMat.columns = DesignMat_out[1]
    print(DesignMat)
    """

    def DesignMatrixList(self, x, y, order):
        X_degree_list = []
        for i in range(order+1):
            X_degree_list.append(self.GenerateDesignMatrix(x, y, i)[0])
        return X_degree_list
```

```
    def SVDinv(self, A):
        ''' Takes as input a numpy matrix A and returns inv(A) based on singular
            ↪ value decomposition (SVD).
        SVD is numerically more stable than the inversion algorithms provided by
        numpy and scipy.linalg at the cost of being slower.
        '''
        U, s, VT = np.linalg.svd(A)
        D = np.zeros((len(U),len(VT)))
        for i in range(0,len(VT)):
            D[i,i]=s[i]
        UT = np.transpose(U); V = np.transpose(VT); invD = np.linalg.pinv(D) #np.
            ↪ linalg.pinv instead of np.linalg.inv
        return np.matmul(V,np.matmul(invD,UT))

    def beta_SVD(self, X, z):
        A = np.transpose(X) @ X
        beta_SVD = self.SVDinv(A).dot(X.T).dot(z)
        #ztilde_SVD = X @ beta_SVD
        return beta_SVD

    def yPredictedSVD(self, X, z):  #Takes design matrix as X and a column vector
        ↪ of same number of rows as z
        z_pred_SVD = X @ self.beta_SVD(X, z)
        return z_pred_SVD

    def Fig_2_11(self, design_mat, degree, n_bootstraps):
        x_train, x_test, z_train, z_test = train_test_split(design_mat, z,
            ↪ test_size=0.3)
        z_pred = np.empty((z_test.shape[0], n_bootstraps))
        for i in range(n_bootstraps):
            x_, z_ = resample(x_train, z_train)
            z_pred[:,i] = x_test @ self.beta_SVD(x_, z_)
        z_test = z_test[:, np.newaxis]
        error = np.mean( np.mean((z_test - z_pred)**2, axis=1, keepdims=True) )
        bias = np.mean( (z_test - np.mean(z_pred, axis=1, keepdims=True))**2 )
        variance = np.mean( np.var(z_pred, axis=1, keepdims=True) )
        return error,bias,variance

    def Fig_2_11_no_resampling(self, design_mat, degree):  # Using SVD
        x_train, x_test, z_train, z_test = train_test_split(design_mat, z,
            ↪ test_size=0.3)
        z_pred = x_test @ self.beta_SVD(x_train, z_train)
        z_train_pred = x_train @ self.beta_SVD(x_train, z_train)
        error = np.mean((z_test - z_pred)**2)
        train_error = np.mean((z_train - z_train_pred)**2 )
        #print("z_pred")
        #print(pd.DataFrame(z_pred))
        #print("z_train_pred")
        #print(pd.DataFrame(z_train_pred))
        return error, train_error
```

```
    # With resampling
    def generate_error_bias_variance_with_resampling(self, mat_list,
        ↪ starting_degree, bootstrap):
        degree = starting_degree
        degree_list = []
        Fig_2_11_error_list= []
        Fig_2_11_bias_list = []
        Fig_2_11_variance_list = []
        for mat in mat_list:
            #print(pd.DataFrame(mat))
            #print(degree)
            #print("Start", degree, time.clock())
            Fig_2_11_result = self.Fig_2_11(mat, degree, bootstrap)
            Fig_2_11_error_list.append(Fig_2_11_result[0])
            Fig_2_11_bias_list.append(Fig_2_11_result[1])
            Fig_2_11_variance_list.append(Fig_2_11_result[2])
            degree_list.append(degree)
            degree = degree + 1
            #print(Fig_2_11_result)
            #print("End", time.clock())
        return Fig_2_11_error_list, Fig_2_11_bias_list, Fig_2_11_variance_list,
            ↪ degree_list

    # No resampling
    def generate_error_bias_variance_without_resampling(self, mat_list,
        ↪ starting_degree):
        degree = starting_degree
        degree_list = []
        Fig_2_11_error_list= []
        Fig_2_11_train_error_list = []
        for mat in mat_list:
            #print(pd.DataFrame(mat))
            #print(degree)
            #print("Start", degree, time.clock())
            Fig_2_11_result = self.Fig_2_11_no_resampling(mat, degree)
            Fig_2_11_error_list.append(Fig_2_11_result[0])
            Fig_2_11_train_error_list.append(Fig_2_11_result[1])
            degree_list.append(degree)
            degree = degree + 1
            #print(Fig_2_11_result)
            #print("End", time.clock())
        return Fig_2_11_error_list, Fig_2_11_train_error_list, degree_list

    # Plot error, bias and variance as a function of model complexity
    def ebv_by_model_complexity(self, metrics):
        plt.plot(metrics[3], metrics[0])
        plt.plot(metrics[3], metrics[1])
        plt.plot(metrics[3], metrics[2])
        plt.scatter(metrics[3], metrics[0], label='error')
        plt.scatter(metrics[3], metrics[1], label='bias')
```

```
        plt.scatter(metrics[3], metrics[2], label='variance')
        plt.xlabel("Model complexity (polynomial order)")
        plt.xticks(np.arange(1, len(metrics[3])+1, 1))
        #plt.yticks(np.arange(0, 0.2, 0.05))
        plt.yscale('log')
        plt.legend()
        plt.show()

    # Plot training and sample errors as functions of model complexity
    def training_vs_test(self, metrics):
        plt.plot(metrics[2], metrics[0])
        plt.plot(metrics[2], metrics[1])
        plt.scatter(metrics[2], metrics[0], label='Test sample')
        plt.scatter(metrics[2], metrics[1], label='Training sample')
        plt.xlabel("Model complexity (polynomial order)")
        plt.xticks(np.arange(1, len(metrics[2])+1, 1))
        #plt.yticks(np.arange(0, 0.2, 0.05))
        plt.yscale('log')
        plt.legend()
        plt.show()
```

Listing 6: "Functions' Library"

```
import os, sys
import numpy as np
import pandas as pd
import keras
from keras import backend as K
from keras.models import Model
from keras.layers import Input, Dense

from sklearn.metrics import mean_squared_error

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib import scale
from matplotlib.ticker import LinearLocator, FormatStrFormatter

import time

'''
This class iof used to store all functions
necessary to process data and run neural network
and/or XGBoost
'''
class Functions:
    # constructor
    def __init__(self, *args):
        # passing the random seed
        self.seed = args[0]
        self.lossName = args[1]
        np.random.seed(self.seed)

    # Initialising Weights
    def GetWeights(self, *args):
        # the value from parameter file
        name = args[0]
        # random seed
        #seed = args[1]
        if name == 'norm':
            # random normal
            return keras.initializers.RandomNormal(mean=0.0, stddev=0.05, seed=
                ↪ self.seed)
        elif name == 'unif':
            # random uniform
            return keras.initializers.RandomUniform(minval=-0.05, maxval=0.05,
                ↪ seed=self.seed)
        elif name == 'xnorm':
            # Xavier normal
            return keras.initializers.glorot_normal(seed=self.seed)
        elif name == 'xunif':
            # Xavier uniform
```

```
            return keras.initializers.glorot_uniform(seed=self.seed)
        elif name == 'hnorm':
            # He normal
            return keras.initializers.he_normal(seed=self.seed)
        elif name == 'hunif':
            # He uniform
            return keras.initializers.he_uniform(seed=self.seed)
        else:
            print("Check your weights!")
            sys.exit()

    # Activation Functions
    def GetActivation(self, *args):
        # name of activation functions
        name = args[0]
        #
        #x = args[1]
        if name == 'linear':
            # Linear
            return keras.activations.linear#(x)
        elif name == 'exp':
            # Exponential
            return keras.activations.exponential#(x)
        elif name == 'tanh':
            # Tanh
            return keras.activations.tanh#(x)
        elif name == 'sigmoid':
            # Sigmoid
            return keras.activations.sigmoid#(x)
        elif name == 'hsigmoid':
            # Hard Sigmoid
            return keras.activations.hard_sigmoid#(x)
        elif name == 'softmax':
            # Softmax
            return keras.activations.softmax#(x, axis=-1)
        elif name == 'softplus':
            # Softplus
            return keras.activations.softplus#(x)
        elif name == 'softsign':
            # Softsign
            return keras.activations.softsign#(x)
        elif name == 'relu':
            # ReLU
            return keras.activations.relu#(x, alpha=0.0, max_value=None, threshold
                ↪ =0.0)
        elif name == 'elu':
            # eLU
            return keras.activations.elu#(x, alpha=1.0)
        elif name == 'selu':
            # SeLU
            return keras.activations.selu#(x)
```

```
        else:
            print('Check␣activation␣function!')
            sys.exit()

    # Regularization (L1, L2 and combined to avoid overfitting)
    def GetRegularizer(self, *args):
        name = args[0]

        if name == 'l1':
            return keras.regularizers.l1(0.)
        elif name == 'l2':
            return keras.regularizers.l2(0.)
        elif name == 'l1l2':
            return keras.regularizers.l1_l2(l1=0.01, l2=0.01)
        else:
            print('Check␣Regularization')
            sys.exit()

    # Optimization algorithms (to compile and run the model),
    # i.e. various gradients methods
    def GetGradient(self, *args):
        # the name of the method
        name = args[0]
        # learning rate
        alpha = args[1]
        if name == 'sgd':
            # Stochastic Gradient Descent - includes momentum and support for
                ↪ Nesterov momentum
            return keras.optimizers.SGD(lr=alpha, momentum=0.0, nesterov=False)
            # Nesterov
        elif name == 'nesterov':
            return keras.optimizers.SGD(lr=alpha, momentum=0.0, nesterov=True)
        elif name == 'rmsprop':
            # RMSProp
            return keras.optimizers.RMSprop(lr=alpha, rho=0.9)
        elif name == 'adagrad':
            # Adagrad
            return keras.optimizers.Adagrad(lr=alpha)
        elif name == 'adadelta':
            # Adadelta
            return keras.optimizers.Adadelta(lr=alpha, rho=0.95)
        elif name == 'adam':
            # Adam
            return keras.optimizers.Adam(lr=alpha, beta_1=0.9, beta_2=0.999,
                ↪ amsgrad=False)
        elif name == 'adamax':
            # Adamax
            return keras.optimizers.Adamax(lr=alpha, beta_1=0.9, beta_2=0.999)
        elif name == 'nadam':
            # Nadam
            return keras.optimizers.Nadam(lr=alpha, beta_1=0.9, beta_2=0.999)
```

```
        else:
            print('Check␣optimization␣Methods!')
            sys.exit()

    # Getting Loss function
    def GetLoss(self, y_true, y_pred):
        name = self.lossName
        # Getting the correct loss function
        if name == 'mse':
            # Mean Squared Error (MSE)
            return keras.losses.mean_squared_error(y_true, y_pred)
        elif name == 'mae':
            # Mean Absolute Error (MAE)
            return keras.losses.mean_absolute_error(y_true, y_pred)
        elif name == 'mape':
            # Mean Absolute Percentage Error (MAPE)
            return keras.losses.mean_absolute_percentage_error(y_true, y_pred)
        elif name == 'msle':
            # Mean Squared Logarithmic Error (MSLE)
            return keras.losses.mean_squared_logarithmic_error(y_true, y_pred)
        elif name == 'hinge':
            # Hinge
            return keras.losses.hinge(y_true, y_pred)
        elif name == 'shinge':
            # Squared Hinge
            return keras.losses.squared_hinge(y_true, y_pred)
        elif name == 'chinge':
            # Categorical Hinge
            return keras.losses.categorical_hinge(y_true, y_pred)
        elif name == 'logcosh':
            # LogCosh
            return keras.losses.logcosh(y_true, y_pred)
        #elif name == 'huber':
        #    # Huber Loss
        #    return keras.losses.huber_loss(y_true, y_pred, delta=1.0)
        elif name == 'categorical':
            # Categorical Cross Entropy
            return keras.losses.categorical_crossentropy(y_true, y_pred)#,
                ↪ from_logits=False, label_smoothing=0)
        elif name == 'sparse':
            # Sparse Categorical Cross Entropy
            return keras.losses.sparse_categorical_crossentropy(y_true, y_pred)#,
                ↪ from_logits=False, axis=-1)
        elif name == 'binary':
            # Binary Cross Entropy
            return keras.losses.binary_crossentropy(y_true, y_pred)#, from_logits=
                ↪ False, label_smoothing=0)
        elif name == 'kullback':
            # Kullback Leibler Divergence
            return keras.losses.kullback_leibler_divergence(y_true, y_pred)
        elif name == 'poisson':
```

```
            # Poisson
            return keras.losses.poisson(y_true, y_pred)
        elif name == 'proximity':
            # Cosine Proximity
            return keras.losses.cosine_proximity(y_true, y_pred)#, axis=-1)
        elif name == 'contrastive':
            #This is the contrastive loss from Hadsell et al.
            #If y_true= 1, meaning that the inputs come from the same category
            #then the square of y_pred is returned and the network
            #will try to minimize y_pred
            #If y_true=0, meaning different categories, the function
            #returns, if the difference beween margin and ypred > 0,
            #the square of this difference. Thus the network will try to maximize
            #y_pred
            margin = 0.9
            square_pred = K.square(y_pred)
            margin_square = K.square((K.maximum(margin - y_pred, 0)))
            return K.mean(y_true*square_pred + (1 - y_true) * margin_square)
        elif name == 'triplet':
            #Function is FaceNets triplet loss. Y_pred is the difference
            #between the positive distance layer and the negative distance layer
            margin = 0.9
            return K.mean(K.maximum(y_pred + margin, 0))
        else:
            print("Check␣Loss␣function!")
            sys.exit()

    '''
    Method used for plotting (in Neural Networks with Keras)
    '''
    def PlotResultsKeras(self, *args):
        # getting the history to plot
        history = args[0]
        # getting network type
        type = args[1]
        # output path were to save figures
        outputPath = args[2]
        # epochs
        epochs = args[3]
        # optimization algorothm used
        optimizer = args[4]
        # batch size
        batch = args[5]
        #filename = outputPath + '/'+ 'logreg_costs_e' + str(epochs).zfill(4)+'.
            ↪ png'
        if type == 'ffnn_keras':
            name = 'Feed␣Forward␣Neural␣Network'
            metrics = ['acc', 'val_acc']
        elif type == 'snn_keras':
            name = 'Siamese␣Neural␣Network'
            metrics = ['acc', 'val_acc']
```

```
        elif type == 'tnn_keras':
            name = 'Triplet Neural Network'
            metrics = ['triplet_acc', 'val_triplet_acc']
        else:
            print('Something wrong with the plots!')
            sys.exit()
        # list all data in history
        print(history.history.keys())
        # summarizing history for accuracy and loss
        fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
        ax1.plot(history.history[metrics[0]])
        ax1.plot(history.history[metrics[1]])
        #ax1.set_title("Accuracy")
        ax2.plot(history.history['loss'])
        ax2.plot(history.history['val_loss'])
        #ax2.set_title("Loss")
        ax1.set_ylabel('Accuracy')
        ax1.set_xlabel('Epochs')
        ax2.set_ylabel('Loss')
        ax2.set_xlabel('Epochs')
        # plot title
        fig.suptitle(name, fontsize = 16)
        plt.legend(['Train', 'Test'], loc='upper left')
        plt.show()
        '''
        Saving figure
        '''
        filename = outputPath + '/'+ type + '_e' + str(epochs).zfill(4)+ '_l'+
            ↪ self.lossName + '_o' + str(optimizer) + '_b'+str(batch)+'.png'
        #print(filename)
        fig.savefig(filename)

    '''
    Normalizing using MinMaxScaler method
    '''
    def normalize(self, x):
        return (x - np.amin(x)) / (np.amax(x) - np.amin(x))

    # accuracy for siamese neural network
    def acc(self, y_true, y_pred):
        #Accuracy function for use with siames twin network
        ones = K.ones_like(y_pred)
        return K.mean(K.equal(y_true, ones - K.clip(K.round(y_pred), 0, 1)), axis
            ↪ =-1)

    # accuracy for triplet neural network
    def triplet_acc(self, y_true,y_pred):
        return K.mean(K.less(y_pred,0))

    def set_category(self, data, train=None, test=None):
        # The creation of bins/categories  aims to create to categories from
```

```
        # the data set, seperated by the mean. The digitize function
        # of numpy returns an array with 1,2,3....n as labels for each of n
        # bins. The min, max cut off are  chosen to be larger/smaller than
        # max min values of the data
        bins = np.array([0, data.mean(), 100])

        if train is None:
            temp = np.digitize(data, bins)
            return temp - 1
        train_labels = np.digitize(train, bins)
        test_labels = np.digitize(test, bins)
        return train_labels - 1, test_labels - 1

    def make_anchored_pairs(self, data, target,test_data,test_target, anch):
        '''
        This function returns anchored of data points for comparison
        in siamese oneshot twin network
        '''
        #create lsits to store pairs and labels to return
        pairs = []
        labels = []
        test_pairs = []
        test_labels = []
        #Create all possible training pairs where one element is an anchor
        for i, a in enumerate(anch):
            x1 = a
            for index in range(len(data)):
                x2 = data[index]
                labels += [1] if target[index] == i else [0]
                pairs += [[x1,x2]]
                print(len(labels))
            for index in range(len(test_data)):
                x2 = test_data[index]
                test_labels += [1] if test_target[index] == i else [0]
                test_pairs += [[x1,x2]]

        return np.array(pairs),np.array(labels),np.array(test_pairs),np.array(
            test_labels)

    def make_training_triplets(self, anchors,samples,labels):
        #Creates triplets for siamese triplet network.
        triplets = []
        for s,l in zip(samples,labels):
            positive = anchors[0] if l==0 else anchors[1]
            negative = anchors[0] if l==1 else anchors[1]
            triplets += [[positive,negative,s]]
        return np.array(triplets)

    def eucl_dist_output_shape(self, shapes):
        shape1, shape2 = shapes
        return (shape1[0], 1)
```

```
def euclidean_distance(self, vectors):
    x, y = vectors
    sum_square = K.sum(K.square(x - y), axis=1, keepdims=True)
    return K.sqrt(K.maximum(sum_square, K.epsilon()))

'''
Plot Results for XGBoost
'''
def PlotResultsXGBoost(self, *args):
    # getting the inputs
    CDOM = args[0]
    CDOM_sorted = args[1]
    X_CDOM_diag_mesh = args[2]
    CDOM_pred_fine_mesh = args[3]
    CDOM_pred = args[4]
    outputPath = args[5]
    y_pred = args[6]
    y_pred_train = args[7]
    MSE_per_epoch = args[8]
    y_train = args[9]
    y_test = args[10]
    XGboost_best_model_index = args[11]

    # Making 3d plots
    fontsize = 10
    # Plot Bray distance by CDOM
    x1 = (CDOM_sorted.loc[:,"x"])
    x2 = x1.copy()
    x1, x2 = np.meshgrid(x1,x2)
    CDOM.mesh[x1-x2==0] = np.nan

    fig = plt.figure(figsize=plt.figaspect(0.25))#figsize=(15, 5))#figsize=plt
        ↪ .figaspect(0.5))
    ax = fig.add_subplot(1, 3, 1, projection='3d')
    ax.set_title("BCC Bray distances by sites' CDOM", fontsize=fontsize)
    #plt.subplots_adjust(left=0, bottom=0, right=2, top=2, wspace=0, hspace=0)
    ax.view_init(elev=30.0, azim=300.0)
    surf = ax.plot_trisurf(CDOM.loc[:,"CDOM.x1"], CDOM.loc[:,"CDOM.x2"], CDOM.
        ↪ loc[:,"ASV.dist"],
                           cmap='viridis', edgecolor='none')
    # Customize the z axis.
    ax.set_zlim(0.3, 1)
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
    ax.tick_params(labelsize=8)
    ax.set_zlabel(zlabel="Bray distance")
    ax.set_ylabel(ylabel="CDOM site 2")
    ax.set_xlabel(xlabel="CDOM site 1")
```

```
# Set up the axes for the second plot
ax = fig.add_subplot(1, 3, 2, projection='3d')
ax.set_title("Predicted BCC Bray distances by sites' CDOM, \n CDOM 0.01 step meshgrid", fontsize=fontsize)
ax.view_init(elev=30.0, azim=300.0)


# Plot the surface.
ax.plot_trisurf(X_CDOM_diag_mesh.loc[:,"CDOM.x1"], X_CDOM_diag_mesh.loc[:,"CDOM.x2"], CDOM_pred_fine_mesh, #197109 datapoints
                cmap='viridis', edgecolor='none')

# Customize the z axis.
ax.set_zlim(0.3, 1)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
ax.tick_params(labelsize=8)
ax.set_zlabel(zlabel="Bray distance")
ax.set_ylabel(ylabel="CDOM site 2")
ax.set_xlabel(xlabel="CDOM site 1")


# Set up the axes for the fourth plot
ax = fig.add_subplot(1, 3, 3, projection='3d')
ax.set_title("Predicted BCC Bray distances by sites' CDOM, \n dataset CDOM coordinates", fontsize=fontsize)
ax.view_init(elev=30.0, azim=300.0)

# Plot the surface.
ax.plot_trisurf(CDOM.loc[:,"CDOM.x1"], CDOM.loc[:,"CDOM.x2"], CDOM_pred, #197109 datapoints
                cmap='viridis', edgecolor='none')

# Customize the z axis.
ax.set_zlim(0.3, 1)
ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
ax.tick_params(labelsize=8)
ax.set_zlabel(zlabel="Bray distance")
ax.set_ylabel(ylabel="CDOM site 2")
ax.set_xlabel(xlabel="CDOM site 1")

filename = outputPath + 'xgboost'+ '_3d'+'.png'
fig.savefig(filename)

# evaluate predictions
def TruePredictedTable(T,P):
    table = pd.DataFrame(np.concatenate((T, P), axis=1))
    table.columns = ["True values", "Predicted values"]
    return table
```

```
        MSE_XGboost = mean_squared_error(y_test, y_pred)
        MSE_XGboost_train = mean_squared_error(y_train, y_pred_train)
        print("MSE␣on␣test␣data:", MSE_XGboost)
        print(TruePredictedTable(y_test[:, np.newaxis], y_pred[:, np.newaxis]))
        print("MSE␣on␣train␣data:", MSE_XGboost_train)
        print(TruePredictedTable(y_train[:, np.newaxis], y_pred_train[:, np.
            ↪ newaxis]))


        print("Best␣test␣MSE␣at␣epoch", XGboost_best_model_index)
        fig, ax = plt.subplots(1, 1)
        #plt.figure()
        ax.plot(list(range(0, len(MSE_per_epoch['validation_0']['rmse']), 1)),
            ↪ MSE_per_epoch['validation_1']['rmse'], '-', label = 'MSE␣test',
            ↪ linewidth=1)
        ax.plot(list(range(0, len(MSE_per_epoch['validation_0']['rmse']), 1)),
            ↪ MSE_per_epoch['validation_0']['rmse'], '-', label = 'MSE␣train',
            ↪ linewidth=1)
        ax.set_ylabel("MSE")
        ax.set_xlabel("Number␣of␣epochs")
        plt.grid(False)
        plt.legend(fontsize="medium")
        plt.show()

        filename = outputPath + 'xgboost'+ '_best_mse'+'.png'
        fig.savefig(filename)


    '''
    Plotting Results for manual Feed Forward Neural Network
    '''
    def PlotResultsManualFFNN(self, *args):
        # getting inputs
        NN_reg_original = args[0]
        CDOM = args[1]

        # getting network type
        type = args[2]
        # output path were to save figures
        outputPath = args[3]
        # epochs
        epochs = args[4]
        # batch size
        batch = args[5]


        test_predict = NN_reg_original.predict(NN_reg_original.XTest)
        #best_prediction = NN_reg_original.models[NN_reg_original.accuracy_list.
            ↪ index(min(NN_reg_original.accuracy_list))]
        print(NN_reg_original.accuracy_list)
        print("Minimum␣MSE␣:", min(NN_reg_original.accuracy_list), "reached␣at␣
```

```
    ↪ epoch␣", NN_reg_original.accuracy_list.index(min(NN_reg_original.
    ↪ accuracy_list)))

fig, ax = plt.subplots(1, 1)
#plt.figure()
ax.plot(list(range(0, len(NN_reg_original.accuracy_list), 1)),
    ↪ NN_reg_original.accuracy_list, '-', label = 'MSE', linewidth=1)
ax.set_ylabel("MSE")
ax.set_xlabel("Number␣of␣epochs")
plt.grid(False)
plt.legend(fontsize="medium")
plt.legend()
plt.yscale('log')
plt.show()

filename = outputPath + type + '_e' + str(epochs).zfill(4)+ '_l'+ self.
    ↪ lossName + '_b'+str(batch)+'_mse'+'.png'
fig.savefig(filename)


#Use log-transformed CDOM values for creating design matrix, then plot on
    ↪ original values
x_mesh = np.log10(np.arange(min(CDOM.loc[:,"CDOM.x1"]), max(CDOM.loc[:,"
    ↪ CDOM.x2"]) + 0.01, 0.01)) + 1
y_mesh = x_mesh.copy()
x_mesh, y_mesh = np.meshgrid(x_mesh,y_mesh)
X_CDOM_mesh = self.pdCat(x_mesh.ravel()[:, np.newaxis], y_mesh.ravel()[:,
    ↪ np.newaxis]).to_numpy()
best_prediction = NN_reg_original.model_prediction(X_CDOM_mesh,
    ↪ NN_reg_original.accuracy_list.index(min(NN_reg_original.
    ↪ accuracy_list)))

x_mesh = np.arange(min(CDOM.loc[:,"CDOM.x1"]), max(CDOM.loc[:,"CDOM.x2"])
    ↪ + 0.01, 0.01)
y_mesh = x_mesh.copy()
x_mesh, y_mesh = np.meshgrid(x_mesh,y_mesh)

ff_pred_original = best_prediction.copy()
ff_pred_original = np.reshape(ff_pred_original, (363, 363))
ff_pred_original[x_mesh-y_mesh==0] = np.nan
ff_pred_original[x_mesh>y_mesh] = np.nan
#print(pd.DataFrame(ff_pred_original))

# Plot the NN-smoothed surface
fig = plt.figure(figsize=(15, 12))
ax = fig.gca(projection="3d")
ax.set_title("Predicted␣BCC␣Bray␣distances␣by␣sites'␣CDOM", fontsize=12)
ax.view_init(elev=30.0, azim=300.0)
surf = ax.plot_surface(x_mesh, y_mesh, ff_pred_original, cmap='viridis',
                       antialiased=True, vmin=np.nanmin(ff_pred_original),
                           ↪  vmax=np.nanmax(ff_pred_original),
```

```
                                rstride=1, cstride=1)
        # Customize the z axis.
        z_range =  (np.nanmax(ff_pred_original) - np.nanmin(ff_pred_original))
        ax.set_zlim(np.nanmin(ff_pred_original) - z_range, np.nanmax(
            ↪ ff_pred_original) + z_range)
        ax.zaxis.set_major_locator(LinearLocator(10))
        ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
        # Add a color bar which maps values to colors.
        fig.colorbar(surf, shrink=0.5, aspect=5)
        ax.tick_params(labelsize=8)
        plt.show()

        filename = outputPath + type + '_e' + str(epochs).zfill(4)+ '_l'+ self.
            ↪ lossName + '_b'+str(batch)+'_3d'+'.png'
        fig.savefig(filename)


    def pdCat(self, x1, x2):
        table = pd.DataFrame(np.concatenate((x1, x2), axis=1))
        table.columns = ["CDOM.x1", "CDOM.x2"]
        return table
```

Listing 7: "Random Forest Module"

```
import os, sys
import random

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import mean_squared_error as MSE

#from Project3.Code.siamese_good import X_train as X_train_siamese
#from Project3.Code.siamese_good import X_test as X_test_siamese
#from Project3.Code.siamese_good import y_train_l as y_train_siamese
#from Project3.Code.siamese_good import y_test_l as y_test_siamese
#from Project3.Code.siamese_good import set_category

import data_processing

# Trying to set the seed
class RandomForest:
    def __init__(self):
        pass

    def accuracy(self, x_1, x_2):
        conf = [[0, 0], [0, 0]]
        for i, j in zip(x_1, x_2):
            conf[i][j] += 1

        print(conf, (conf[0][0] + conf[1][1]) / len(x_1))


    def get_std(self, pandas_object):
        metadata_std = [x.std() for x in pandas_object.values.T]

        return metadata_std


    def get_corr(self, pandas_object, list_of_features=[]):
        if len(list_of_features) == 0:
            list_of_features = pandas_object.columns

        corr_list = []

        for i in list_of_features:
            corr_list.append([])
            for j in list_of_features:
                corr_list[-1].append(pandas_object[i].corr(pandas_object[j]))
```

```
        return corr_list


    def read_data(self, pop_bool=True, metadata_bool=True):
        data = data_processing.DataProcessing()
        # Reading file into data frame
        #cwd = os.getcwd()
        population_path = data.url_ASV #cwd + '/../Data/ASV_table.tsv'
        metadata_path = data.url_metadata #cwd + '/../Data/Metadata_table.tsv'
        """
        df.columns (identifier)
        df.values (population size)

        population_size.shape -> (72, 14991)
        """
        population_size = pd.read_csv(population_path, delimiter='\s+', encoding='
            ↪ utf-8')

        if pop_bool:
            # find the non-zero bioms
            population_to_drop = [x for x in population_size.columns if
                ↪ population_size.get(x).min() == 0]
            population_size = population_size.drop(population_to_drop, axis=1)
        """
        df.columns (properties)
        df.values (values)

        metadata.shape -> (71, 41)
        """
        metadata = pd.read_csv(metadata_path, delimiter='\s+', encoding='utf-8')

        # l = ["Latitude", "Longitude", "Altitude", "Area",
        if metadata_bool:
            l = ["Temperature", "Secchi", "O2", "CH4", "pH", "TIC",
                 "SiO2", "KdPAR"]

            toDrop = [x for x in metadata.columns if x not in l]
            metadata = metadata.drop(toDrop, axis=1)

        return population_size, metadata


    def prepare_data(self, population_data, metadata, test_size, seed):
        x_values = np.array(population_data.values)
        y_values = np.array(metadata.values)

        train_x, test_x, train_y, test_y = train_test_split(x_values, y_values,
            ↪ test_size=test_size)

        print(train_x.shape, train_y.shape)
        print(test_x.shape, test_y.shape)
```

```
        predictions = np.zeros((test_y.shape[1], test_y.shape[0]))
        ML_ = []
        for i in range(len(metadata.columns)):
            print(i)
            rf = RandomForestRegressor(n_estimators=100, random_state=seed)
            rf.fit(train_x, train_y.T[i])

            ML_.append(rf)
            predictions[i] = rf.predict(test_x)

        print(predictions.shape)

        # errors = abs(pred - test_y.T[i])
        # err = round(np.mean(errors), 2)

        for i in range(len(metadata.columns)):
            print(metadata.columns[i], MSE(test_y.T[i], predictions[i]))

        return predictions, test_y, ML_
        # print("observation", MSE(predictions.T[0], test_y[0]))


    def get_bioms(self):
        """
        To get the bioms that are in the mid 20% (to be or not to be)
        :return:
        """
        #cwd = os.getcwd()
        #population_path = cwd + '/../Data/ASV_table.tsv'
        data = data_processing.DataProcessing()
        population_path = data.url_ASV

        pop_bioms = pd.read_csv(population_path, delimiter='\s+', encoding='utf-8'
            ↪ )

        to_keep = []
        for i in pop_bioms.columns:
            c = 0
            for j in pop_bioms.get(i):
                if j > 2:
                    c += 1
            if 0.6 > c / 72 > 0.4:
                to_keep.append(i)

        to_drop = [x for x in pop_bioms.columns if x not in to_keep]
        pop_bioms = pop_bioms.drop(to_drop, axis=1)
        return pop_bioms


    def predict_exist(self):
```

```
        bioms = self.get_bioms()
        _, metadata = self.read_data()

        bioms_array = (np.array(bioms.values) > 0) * 1
        metadata_array = np.array(metadata.values)

        train_x, test_x, train_y, test_y = train_test_split(metadata_array,
            ↪ bioms_array, test_size=0.30)

        print("train_x size:", train_x.shape, " train_y size:", train_y.shape,
            ↪ train_y.T[0].shape)

        clfs = []
        scores_test = []
        scores_train = []
        for j in range(2, 10):
            clfs.append([])
            scores_test.append([])
            scores_train.append([])
            for i in range(bioms_array.shape[1]):
                print("n_estimators : ", j * 10)
                clf = RandomForestClassifier(max_depth=j, random_state=50,
                    ↪ n_estimators=40)
                clf.fit(train_x, train_y.T[i])
                # rint(clf.predict(test_x))
                scores_test[-1].append(clf.score(test_x, test_y.T[i]))
                scores_train[-1].append(clf.score(train_x, train_y.T[i]))
                # print(scores[-1][-1])
                # print(accuracy(test_y.T[i], clf.predict(test_x)))
                clfs[-1].append(clf)

        return clfs, scores_test, scores_train


    def predict_t(self, X_train, X_test, y_train, y_test, seed):
        _, metadata = self.read_data()
        #y_train_siamese, y_test_siamese = 0, 0
        #X_train_siamese, X_test_siamese = 0, 0
        #if X_train is None or X_test is None:
        #    train_y, test_y = y_train_siamese, y_test_siamese
        #    train_x, test_x = X_train_siamese, X_test_siamese
        #else:
        train_y, test_y = y_train, y_test
        train_x, test_x = X_train, X_test

        print("train_x size:", train_x.shape, " train_y size:", train_y.shape,
            ↪ train_y.T[0].shape)

        clfs = []
        scores_test = []
        scores_train = []
```

```
        for j in range(2, 10):
            print("max_depth␣:␣", j * 10)
            clf = RandomForestClassifier(max_depth=j, random_state=seed,
                ↪ n_estimators=100)
            clf.fit(train_x, train_y.T)
            # rint(clf.predict(test_x))
            scores_test.append(clf.score(test_x, test_y.T))
            scores_train.append(clf.score(train_x, train_y.T))
            # print(scores[-1][-1])
            # print(accuracy(test_y.T[i], clf.predict(test_x)))
            clfs.append(clf)

        return clfs, scores_test, scores_train


    def predict_all_metadata(self, population_size, metadata, ML_):
        x_values = np.array(population_size.values)
        y_value = np.array(metadata.values)

        predictions = np.zeros((y_value.shape[1], y_value.shape[0]))

        for i in range(len(metadata.columns)):
            predictions[i] = ML_[i].predict(x_values)

        print(predictions.shape)

        # errors = abs(pred - test_y.T[i])
        # err = round(np.mean(errors), 2)

        return predictions
```

Listing 8: "XGBoost Module"

```
# XGboost with scikitlearn
# https://machinelearningmastery.com/develop-first-xgboost-model-python-scikit-
    ↪ learn/
# https://xgboost.readthedocs.io/en/latest/python/python_api.html
from numpy import loadtxt
from xgboost import XGBRegressor  #scikit-learn API for XGBoost regression
from xgboost import XGBRFRegressor  #scikit-learn API for XGBoost random forest
    ↪ regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_squared_error

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib import scale
from matplotlib.ticker import LinearLocator, FormatStrFormatter

'''
CLass To implement XGBoost
'''
class XGBoosting:

    def __init__(self, *args):
        pass

    def RunModel(self, *args):
        X_train = args[0]
        X_test  = args[1]
        y_train = args[2]
        y_test  = args[3]
        X_CDOM  = args[4]
        X_CDOM_diag_mesh = args[5]
        CDOM    = args[6]
        CDOM_sorted = args[7]
        outputPath = args[8]
        #!!!Implement Xavier initialization!!!
        # fit model no training data
        model = XGBRegressor(objective='reg:squarederror')
        model.fit(X_train, y_train, eval_set=[(X_train, y_train), (X_test, y_test)
            ↪ ],
                  eval_metric='rmse', early_stopping_rounds=100, verbose=False)
        #print(model)
        return model
```