Technical University of Crete

School of Electrical and Computer Engineering

Diploma Thesis

Improving the Efficiency and Enhancing the Capacity of the PyPLT (Python Preference Learning Toolbox) Software Tool

Chaviara Antonia Chrysovalanto

Chania, April 2022

Acknowledgements

I would like to thank Professors Michail G. Lagoudakis, Georgios N. Yannakakis and Georgios Chalkiadakis for the opportunity they gave me along with their support and guidance. I would also like to give special thanks to David Melhart for his thoughtful comments and recommendations on this dissertation. David was always willing to assist in any way he could throughout the whole project. Finally, I cannot forget to thank my friends and colleagues for their participation in the tests, as well as for their continuous support and patience.

Abstract

Research has demonstrated that ordinal approaches to the analysis of subjective values, such as emotions, lead to more reliable predictive models. Preference learning is the machine learning subfield, which deals with datasets including ordinal relations. Preference learning algorithms have proven to be powerful in creating efficient computational models from ordinal data. The Python Preference Learning Toolbox facilitates ordinal data processing and preference learning. The software is open source, available to a wide range of researchers and includes popular algorithms and data processing methods. At first, the toolbox is tested with synthetic datasets in order to identify possible malfunctions during the stages of data processing and modelling. An optimization of the current features, along with the addition of evaluation metrics and preference learning techniques are performed in order to augment the functionality of the software. A user survey follows, in order to test the usability of the toolbox. The results confirm that PyPLT is simple, easy to use, for both novice and experienced researchers. Furthermore, it is capable of producing reliable predictive models provided the necessary data processing and algorithm parameterization which is offered by the toolbox.

Contents

List of Figures

1

Introduction

1.1 Problem formulation

The field concerning the study and development of systems capable of processing human opinions and emotions is known as Emotional Artificial Intelligence or Affective computing. An important task in affective computing is the assignment of labels to human assessments of emotions deriving from records of emotional events. The diversity of the brain operations, along with the limited ability to express preferences directly in terms of a specific value function, [4] complicate the reproduction of human assessments by an application.

In the early stages of affective computing, two ways of conceiving emotions dominated: categorical and numerical. The former represented language and emotion theories, while the latter reflected dimensional theories. In terms of measurement, research in affective computing initially focused in nominal and interval description. In nominal measurement, a sample is associated with an emotion class or label [5]. In realistic conditions though, it is not always possible to assign a single category to an emotion [6]. Interval description applies to attributes of emotion and is useful for change tracking within a record [7]. However, in cases of substantial changes, there is a lack of reliability. Also, each individual, values differently the change of an emotion over a period of time.

Multiple disciplines, such as philosophy, psychology, neuroscience and artificial intelligence, supported through theoretical arguments and empirical evidence the exploration of other approaches to emotion, in order to explore the grounds between nominal and interval. Ordinal annotation and analysis of emotions, is an approach lying somewhere in between. Processing ordinal data as ranks and translating the affect modelling task as a preference learning task respects the nature of the data. It also yields more reliable, valid and general predictive models. Preference learning algorithms have shown an advantage in creating computational models from ordinal data [2].

Despite the numerous case studies in literature that favour ordinal labels for representing and annotating emotion there are still some objections against the use of ordinal labels. A common statement is the lack of data analysis tools and statistical methods available for the processing of ordinal data. Another concern is whether the intensity of an emotion can be captured through relative descriptions, as ranking procedures require at least one reference point.

The existing applications available for data processing are limited and usually focus on a single preference learning method. Furthermore, they are often outdated due to the rise of modern deep learning algorithms in combination with the limited audience of researchers. Also, most applications are addressed to researchers and offer only a command-line interface, making their utilization difficult to users without prior experience. Most of them are not designed to handle datasets with ordinal relations [2].

Preference learning toolbox (PLT) was introduced for the first time in 2015, at the Institute of Digital Games at the University of Malta. PLT is an open-source tool, available under the GNU Lesser General Public License and allows the addition of new algorithms and methods at all phases of preference learning. In 2019, a later version was developed using python, while the initial release had been in java. The toolbox includes a set of popular data pre-processing, feature selection, preference learning methods and model training. PLT is easily accessible by everyone without major experience with preference algorithms, as it includes a graphical user interface along with detailed and comprehensive documentation [1].

Users have reported some errors while working with PLT. The graphical user interface freezes during the loading of big datasets and as a result there is no information on the progress of the loading procedure. Also, a memory error appears in cases of preference modeling with large datasets. They also noticed unreasonably high accuracies, when evaluating models deriving from some preference learning algorithms of the tool.

The current thesis aims to update the toolbox framework by investigating the errors noted by the users and extend it with the addition of further evaluation metrics and preference learning algorithms.

1.2 Thesis Research Questions

An important question is whether the conventional preference learning algorithms of the tool offer sufficient solutions by performing computations, or we should investigate algorithms of other approaches. Another subject is whether it is possible to assess the performance of the derived computational models by combining multiple means of evaluation, instead of using only the accuracy metric. Accuracy is measured as the difference between the data observations and the objective defined reference values or ground truth values. Another important way of measuring the performance of the annotator is precision, which estimates the degree of repeatability of the data observations [1]. In order to produce more valid models, a specific evaluation mean or a combination of metrics could be taken into account, depending on the nature of the data and the use of the inferred model. Isolation and integrity are important factors, when processing human data. Can the tool offer the ability to isolate and process multiple participant's data without the risk of mixing them up? These are some of the questions that the current thesis investigates and addresses.

1.3 Approach

The tool was tested with artificially created datasets of various sizes, in order to generate and investigate possible existing errors during dataset loading, pre-processing or during the training and evaluation of the inferred model. In addition, it has been updated with a new algorithm: Neuroevolution. It is a genetic algorithm, different from the conventional preference learning algorithms, which trains multiple models and selects out the best by using evolutionary strategies. The core of the algorithm is based on the initial version of preference learning toolbox, which was implemented in java. Also, a few performance metrics have been added in order to assess more efficiently the performance of the model during training and testing. Another significant feature is the addition of an extra column in the dataset, containing a unique number for every group of data, which offers the ability to perform several processing steps in parts of the data, without affecting the whole dataset.

1.4 Contributions

The objective of the PLT development is to ease and widen the access of ordinal data processing through preference learning algorithms. People with different backgrounds, from novice users to machine learning researchers and developers, have the ability to experiment with multiple ordinal datasets and create accurate predictive models. As it is an open source tool, new features and methods may be added by everyone. PLT contributes not only in label processing and preference handling, but in the wider section of human computer interaction and emotional retrieval. Rank-based emotion retrieval can be performed from image, video, speech or physiology based applications for health, educational or entertaining purposes [3].

1.5 Summary of Thesis

In the **first chapter**, an introduction to the thesis is performed containing the problem formulation, research questions, thesis approach and summary.

The **second chapter** includes the theoretical background related to the PLT. Important notions are described, in the fields of Artificial Intelligence, Machine, Supervised and Preference Learning. Also, it contains necessary information about Artificial Neural Networks, optimization algorithms and techniques that are relevant to the tool, along with the basis of Neuroevolution and genetic algorithms.

In the **third chapter**, the current functionality and features of PLT are briefly described and portrayed through the graphical user interface.

In the following chapters, there is an overview of all the modifications performed in the tool, based on identified issues in order to augment its functionality. More specifically, in the **fourth chapter** the reported problems are analysed, along with the actions that were performed in order to address them. The **fifth chapter** presents the additions that updated and extended the tool's framework.

The **sixth chapter** contains the results of the usability testing study of the tool.

To conclude, the **seventh and last chapter** presents the limitations of PLT along with further suggestions for improvement.

2

Related Work

2.1 Introduction to Artificial Intelligence and Machine Learning

2.1.1 Artificial Intelligence

Artificial intelligence enables a machine or computer to learn from past experience, adapt to new data and perform tasks similar to human intellectual processes by working intelligently and most of all independently. Modern machine capabilities generally classified as AI, are designed to complement and enhance human capabilities including voice and handwriting recognition, performing medical diagnosis, operating autonomous cars [8]. In order to implement these technologies machines are trained to perform specific tasks by processing large amounts of data and identifying data forms through algorithms.

With the increase of memory capacity and computer processing speed, many programs have reached the levels of human experts in performing certain tasks. On the other hand, programs are not entirely capable of matching human flexibility over wider domains demanding common awareness. Human brain consists of a combination of many diverse abilities, not a single trait, thus research in AI has focused on some components of human intelligence such as learning.

Intelligence can be described as the ability to acknowledge and preserve information in order to adapt within an environment. “Artificial” is a term used to describe an object or behavior produced by human beings rather than by nature [9].

Term artificial intelligence was introduced in the mid-1950s, at a university campus with its primary scope being problem solving and symbolic methods. What followed was computer training to mimic functions associated with the human mind, such as problem solving. For the next two decades AI thrived, as computers were now able to store information, became faster, cheaper, and more accessible. Despite that, computers were incapable of adjusting to AI needs in storing enough information and fast processing, resulting in a ten year slow roll in research. In the 1980’s an increase of funds reestablished the goals for improving artificial intelligence, but soon AI research lost its power of interest. In 1997, a chess playing computer program won the reigning world chess champion, a fact that was proved to have been a significant step forward in the development of an artificially intelligent decision making program. [8].

2.1.2 Machine Learning

Machine learning is a branch of artificial intelligence that enables systems to learn from data and improve from past experience, without human intervention [10]. Nowadays, machine learning algorithms (ML) are widely used in a variety of applications all around us. From digital assistants searching the web and playing music after our vocal commands to medical image analysis systems that help a doctor to spot tumors. A model is built based on sample data, often described as "training data". ML algorithms are trained to find patterns and features between huge amounts of training data in order to make predictions or decisions of unknown data. They self-improve their accuracy and performance through training, by being exposed to more data.

Machine learning consists of four basic steps:

1. Selection and preparation of the training dataset (an object that holds the data and some metadata about them): The input data has to be prepared and checked in order to assure that it won't impact the training in case of imbalances. The data is divided into two subsets equivalent to the phase it will be processed. For example the evaluation subset will be used in the phase of testing while training data is used in the phase of application training. Sometimes the algorithm has to identify and extract features and classifications on its own. In those cases the data is called unlabeled, otherwise it is described as labeled.

2. Selection of the proper algorithm: The type of algorithm depends on the problem to be solved in combination with the type and amount of training data. In case of labeled data, common types of ML algorithms are Regression algorithms, Decision trees and Instance-based algorithms. While in unlabeled data we can use Clustering and Association algorithms [11]. Neural Networks lie somewhere in between, as they are usually used for labeled data, but can be used in cases of unlabeled data as well, depending on the training method and the desired output.

3. Algorithm training to create the model: Algorithm training is a repetitive process. The output is compared with the expected results. After that, parameters like weights and biases within the algorithm are adjusted in order to produce a more accurate result. The variables through the algorithm run until it returns the correct result most of the time. The accurate algorithm deriving from the training, is the requested machine learning model.

4. Use and improvement of the model: The final step is the exposure of the model in new data. For example, a robot vacuum cleaner which is based on a machine learning model will ingest data resulting from interaction with new objects in the room [12]. The accuracy and effectiveness of the machine learning model are being improved over time, depending on how the model is deployed and whether it can learn online. By learning online, the model ingests data in a sequential order between timeframes and thus can dynamically update its parameters at each step.

2.2 Machine Learning Methods

2.2.1 Supervised Machine Learning

Supervised machine learning algorithms can predict future events by applying their knowledge from past experience to the new data. Based on a known dataset, the algorithm creates a function to perform predictions about the output values. By comparing the output with the intended, the algorithm is capable of finding errors and modifying the model accordingly. After sufficient training, the system is able to classify data or predict outcomes accurately for an unseen input.

Supervised algorithms are divided in two categories:

- Classification: It can be described as a discrete form of supervised learning where the output is a category or class. For example handwritten digit recognition.
- Regression: In this category, the output of the algorithm consists of one or more continuous variables. For instance, the prediction of a kid's height as a function of its age and weight [13].

Supervised learning is necessary in solving real-world problems, such as detection and prevention of fraud in banking, stock market forecasting, medical imaging diagnosis. Preference learning toolbox contains supervised methods and thus is a supervised machine learning tool.

2.2.2 Unsupervised Machine Learning

In contrast, unsupervised learning algorithms are less accurate and they are used when the input data is neither classified nor labeled and there are no corresponding target values. No training is provided, which means the algorithm doesn't train on a known dataset. The task may be to group unsorted input according to patterns. The system doesn't always find the right output, but is able to discover hidden structures [14].

Unsupervised algorithms can be grouped as:

- Clustering: Where the aim is to discover natural groupings in data based on features or characteristics.
- Association: Where we aim to discover the dependency of an item to another. For example people that buy product A also tend to buy product B [13].

2.2.3 Reinforcement Learning

Reinforcement learning is often concerned with agents. While they interact with a dynamic environment, they learn to make decisions through trial-and-error. Reinforcement machine learning methods are based on rewarding and punishing. When the optimal behavior is chosen by the agent, a maximum reward is received. A less appropriate decision results in a smaller reward. When the agent performs an undesired selection a penalty is received. The worst behavior results in the highest penalty. By trial error search and reward, the agent learns the best action and maximizes its performance [15].

2.3 Approaches to Supervised Learning

2.3.1 Classification

Classification refers to a predictive modelling problem where the task is to classify the input data as one of the predefined labelled classes. A well-known application of classification is the categorization of emails as “spam” or “not spam”.

There are many different types of classification algorithms for modeling classification predictive modeling problems. The main types of classification algorithms are Binary, Multi-Class and Multi-Label Classification [16]. A good practice, in order to decide which is more suitable in every case, is to use controlled experiments and discover which algorithm results in the best performance for a given classification task. There is a variety of metrics in order to evaluate the performance of a model based on the predicted class labels, with the most popular being the accuracy metric. In later chapters, evaluation metrics will be described in more detail.

2.3.2 Regression

Regression is a supervised machine learning algorithm suitable for cases where the output variable is a continuous value. The aim is to predict a dependent variable based on independent predictors [17].

In rudimental regression analysis, there is a linear connection between the dependent and an independent variable. The aim is to plot a line that best models the given points. The line (prediction or model output) is modelled based on the linear equation $y^{\wedge}=a + b*x$ where x is the input vector. The aim is to find the best values for a and b, for the model to produce as closest predictions as the real values from the examples. The search for parameters a and b turns to a minimization problem, where we want to minimize the error between the predicted and the actual value. The error is calculated by the difference between the predicted values and the ground truth. Then it is squared and sum over all data points. We divide that value by the total number of data points, in order to receive the average squared error [17]. This function is often referred as the Mean Squared Error (MSE) function. Now the target is to minimize the MSE value by adjusting the values a and b.

In order to reduce the cost function MSE, we train the parameters a and b by using another method named Gradient descent. Gradient descent algorithm starts at a point taking steps in the nearest downhill direction. Then, by repeatedly updating the parameters a and b of the model, the algorithm proceeds in small steps until the cost function is reduced. The number of steps we take in order to reach the target (minimum) is the learning rate. This decides on how fast the algorithm converges to the minima. With a small learning rate we could take more time in order to reach the minima, but get closer to it. On the contrary, with a higher learning rate there is a chance of overshooting the minima. The cost function must be a convex function in order to have a single global minimum [18].

Logistic regression is a machine learning technique suitable for classification problems. The cost function used in linear regression cannot be applied in logistic regression as it is a non-convex function with multiple local minimums. The cost function used in logistic regression is the sigmoid function or logistic function. The Gradient Descent algorithm is also used in order to train parameters a and b of the model. The cost function calculates how well the predictions are performed while the Gradient Descent algorithm minimizes the cost function [19].

Simple networks as logistic regression are unable to solve more complex classification problems. As a result we need networks with multiple levels, capable of learning complex functions for data classification. Those networks are ANNs.

2.3.3 Preference Learning

Preference learning is a subset of machine learning that focuses on developing predictive models based on observed preference information. Human preferences are a really important part in AI research and applications. While the concept of preference learning has been emerging for some time in many fields such as economics, it's a relatively new topic in AI research. Preference information appears in various fields of application such as autonomous agents, adaptive user interfaces, decision theory, planning, non-monotonic reasoning and qualitative decision theory [20]. A famous application is recommender systems. For instance, online stores may analyze a customer's purchase record to learn a preference model and then recommend similar products to customers.

There are two practical representations of the preference information: Utility functions and Preference Relations. If it is possible to observe a mapping from data to numbers, then data ranking can be achieved by ranking the numbers. This mapping is called utility function and is a regression learning problem. However, a utility function cannot represent more general relations, such as a partial order. On the other hand, preference relation is the binary representation of preference information. Since preference relation is not transitive, it implies that there may be multiple solutions of ranking. A usual strategy is to discover a ranking solution that is maximally consistent with the preference relations. [21].

Studies have demonstrated that rank-based analysis have multiple benefits over ratings. In rank based questionnaires, the participant specifies the preferred option among two, under a given statement (pairwise preference). Provided more options, the user is requested to choose a ranking of some or all the options. In ratings, annotation biases are noticed on subjective matters, such as experience and emotion based on different cultural or personal criteria, or inconsistencies. On the contrary, ranks offer a clearer comparison between participant’s responses and lead to the creation of more efficient models. In addition, ratings can be easily transformed to ranks. They are compared to one another and a pairwise preference is created for every pair [22].

In order to create computational models that predict those ranks, numerous algorithms are currently available in the field of preference learning. From linear statistical models to non-linear approaches, such as artificial neural networks, support vector machines etc. Preference learning algorithms train on a set of items which have preferences toward labels or other items and predict the preferences for all items. During the training phase, preference learning algorithms have access to examples for which the sought order relation is partially known.

Despite the variety of available methods in the field of preference learning, there is a lack of tools for processing ordinal labels. Pyplt addresses this problem by offering access to a number of ordinal data processing methods and popular preference learning algorithms.

2.4 Artificial Neural Networks

Artificial Neural Networks draw inspiration from humans, more specifically from the natural neural network of their nervous system. The creator of the first neurocomputer, Dr. Robert Hecht-Nielsen, describes a neural network as "...a computing system composed of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs” [23].

ANNs are comprised of numerous nodes. These nodes take input data, perform simple operations and pass the result to other neurons. The result is called activation or node value. Links connect these neurons, using an integer called weight, which controls the signal between them. ANNs learn by altering these weight values.

There are two types of Artificial Neural Networks:

- Feedforward: The ANN has fixed inputs and outputs. The information flows in a single direction: from the input to the output nodes, passing through the hidden nodes. These networks only consider the current input, thus they do not have any memory about what happened in the past. As a result, they have trouble predicting what comes next in a sequence. They are used in image recognition and classification.

- Feedback or Recurrent: Here, feedback loops are allowed. A unit sends and receives information to/from other units. These networks retain information about the input previously received by allowing information to persist, like a short-term memory. This sequential memory is preserved in the network's hidden state vector and represents the context based on the prior inputs and outputs. Unlike a feed-forward network, the same input may produce different outputs depending on the preceding inputs. They are applied to a wide variety of problems where text, audio, video, and time series data is present. For example in speech recognition, analysis of DNA sequences, image captioning, and more [24].

ANNs are widely used in multiple areas such as electronics, economy, industry, medicine. Their learning ability makes them very flexible. Their multi-layered structure and architecture offers fast processing, accurate application of the model to unseen data and adaptability according to the changing environment. A real estate agent uses ANNs to predict market prices, while a doctor classifies whether a tumor is malignant or not. From real estate appraisal (economy) to cancer cell analysis (medicine), they are applied in a variety of fields and contribute to a significant range of applications.

2.4.1 Activation Functions

ANNs are comprised of multiple layers: The input layer, where the information (features) from the environment passes to the hidden layer. The hidden layer where several computations are performed and the result is transferred to the output layer. And finally the output layer which exposes the learnt information to the environment. In order to choose the appropriate neurons that need to be activated or deactivated to manipulate the input/output of the network, activation functions are used. More complicated, high dimensional and non-linear calculations are performed with the help of activation functions [25].

There is a variety of activation functions such as Linear, ReLu, Sigmoid, Softmax, Tanh. We will discuss further about ReLu and Sigmoid which are relevant to the tool.

2.4.1.1 **Sigmoid**

The sigmoid function is a nonlinear function with the formula σ(z) = $\frac{1}{(1+e^{-z})}$, $0 < \sigma(z) < 1$ and the derivative function is formula σ′(z) = $\frac{e^{-z}}{(e^{-z}+1)^2}$. As the range of the sigmoid function is between 0 and 1 it is really useful in binary classification problems in the output layer. However, when z value is too large, the derivative tends to 0, so the gradient is vanished and the learning procedure is slowed down. The graph has the shape of an ‘S’, as shown in figure 3.

2.4.1.2 **ReLU**

The Rectified Linear Unit function offers better performance and generalization compared to the sigmoid. It is a function containing the properties of linear models and as a result enables the optimization of models with gradient-descent methods. For each input where all the values are less than zero, they are set to zero. Thus, the ReLU is represented as: $f(x) = \max(0, x) = \begin{cases} 0, & x_i < 0 \\ x_i, & x_i \geq 0 \end{cases}$. The gradient problem observed in the sigmoid function is eliminated. Furthermore, a boost in the overall computation speed is observed, as it does not compute exponentials and divisions.

Despite the advantages compared to the sigmoid function, the ReLu function has a significant drawback. If too many activations get below zero, then most of the neurons will simply output zero and as a result prohibit learning. In order to solve this problem, LeakyRelu is used.

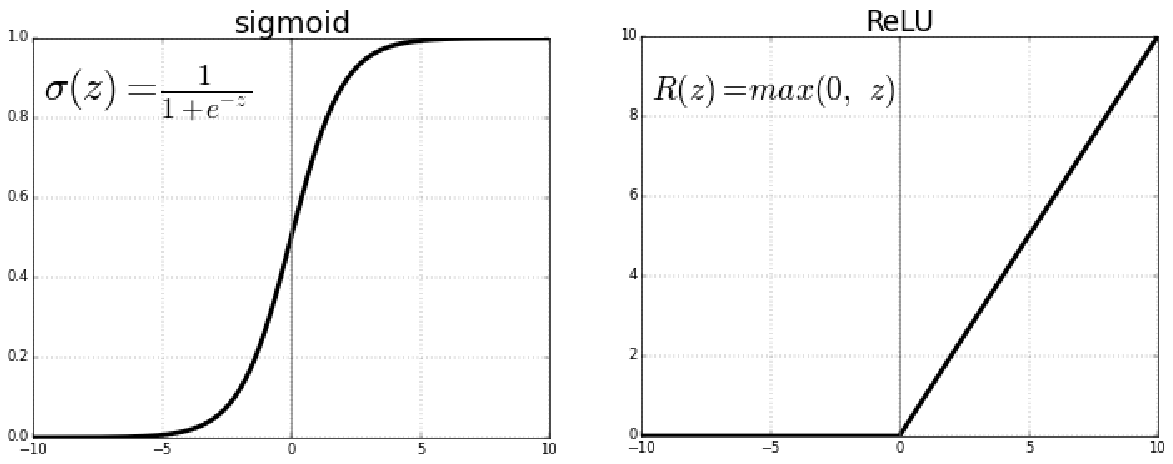


Figure 1: Sigmoid vs ReLU activation function

2.4.2 Optimization algorithms

Optimization algorithms are methods used to change the attributes of a neural network in order to minimize the losses, such as weights and learning rate [26]. There are several optimization methods: Gradient Descent algorithm, Stochastic Gradient Descent, Momentum, Adam Optimization algorithm. The most relevant to our tool are further discussed below.

2.4.2.1 Gradient Descent

Gradient Descent algorithm is an optimization method which was briefly described in linear regression. The algorithm explores the way of modifying the weights, for the loss function to reach a minimum. The weights are altered depending on the losses, which transfer through the layers. Gradient descent is easily implemented and comprehensible. On the other hand, in cases of large datasets, it might take a lot of time and memory to calculate gradient on the whole dataset and change weights to reach the minimum. Another problem of Gradient Descent is that if the loss function is non-convex there are multiple local minimums and the algorithm may be trapped at a local minima.

The main update formula for Gradient Descent is the following:

$$w = w - \eta * \frac{d(L(w))}{dw}$$

Where:

- w: weight vector
- η: learning rate
- L(w): loss function

We minimize the loss function by computing its derivative and the learning rate helps us to set the step size. As we approach the global minimum the loss function gets minimum and we have reached the target in the best possible set of steps.

2.4.2.2 Momentum Based Gradient Descent

As previously described, the current gradient descent is used to update the previous weight values. In Momentum based gradient descent we can calculate momentum based on previous gradients in order to update the weights with a combination of the previous and the current gradient.

The update formula for Momentum Based Gradient Descent is the following:

$$w_{t+1} = w_t - \eta * \frac{d(L(w_t))}{dw_t} + \gamma * \sum_{time=1}^{t} \left(\eta * \frac{d(L(w_{time}))}{dw_{time}} \right)$$

Where:

- time: all the past epochs
- γ: hyper parameter to control the history usually equal to 0.9

Compared to the simple Gradient Descent algorithm, in the momentum case, there is a faster convergence to the minima, as information about the previous steps is preserved. Also, the oscillations are reduced as well as the high variance of parameters. On the other hand, there is still a chance of transcending the global minimum in cases of big parameter changes.

2.4.2.3 **RMS (Root Mean Square)**

RMS algorithm is similar to the Momentum, with the main difference being a small change in the adaptation of parameter h (learning rate) in every step, by dividing it by an exponentially decaying average of squared gradients [26].

The update formula for RMS algorithm is:

$$w_{t+1} = w_t - \frac{d(\eta)}{dE[g^2]_t} * \frac{d(L(w_t))}{dw_t} \;\; Where \;\; E[g^2]_t = (1-\gamma)g^2 + \gamma * \; E[g^2]_{t-1} \; and \; g = \frac{d(L(w_t))}{dw_t}$$

RMS algorithm is better at detecting the global minimum than Momentum.

2.4.2.4 **Adam Optimization**

Adam or Adaptive Moment Estimation is a combination of RMS and Momentum.

The update formula for Adam algorithm is:

$$w = w - \eta * \frac{V_{dw}}{\sqrt{S_{dw}}} \text{ Where } S_{dw} = \frac{S_{dw}}{(1-{\gamma_1}^t)} \text{ and } \frac{V_{dw}}{(1-{\gamma_2}^t)}$$

The hyper parameters γ1, γ2 are defined as in RMS and Momentum respectively:

$S_{dw} = E[g^2]_t = (1 - \gamma_1)g^2 + \gamma_1 * E[g^2]_{t-1}$ And

$$V_{dw} = \gamma_2 * \sum_{time=1}^{t} \left(\eta * \frac{d(L(w_{time}))}{dw_{time}} \right)$$

In conclusion, Adam optimizer is the fastest and converges rapidly to the minimum. Also it rectifies the problems of vanishing learning rate and high variance. The drawback of the method is that it is computationally costly.

2.4.3 Error Functions

As previously mentioned, optimization algorithms are useful in altering the attributes of a neural network, aiming at the minimization of the losses. Error functions help us identify the losses/errors, which are the difference between the predicted and the ground truth values. The error functions used in pyPLT, are Mean Square, Rank Margin and Binary Cross Entropy.

2.4.3.1 Mean Square Error

Mean Square Error function was previously described in linear regression where we wanted to minimize the error between the predicted and the actual value. The Mean Square Error function calculates the sum of the square of the difference between the predicted and ground truth values, divided by the number of all data points.

The MSE formula is the following:

$$\text{MSE } L(y, y^{\wedge}) = \sum_{i=1}^{n} \frac{(y^{\wedge}{}_i - y_i)^2}{n}$$

2.4.3.2 Binary Cross Entropy

Cross-entropy is the most used loss function to use for binary classification problems, where the target values are {0, 1}. Binary cross-entropy determines the average difference between the actual and predicted probability distributions for predicting class 1.

The Binary Cross Entropy formula is the following:

$$\text{BCE } L(y, y^{\wedge}) = -\sum_{i=1}^{n} \frac{y_i * \log y^{\wedge}{}_i + (1-y_i) * \log(1-y^{\wedge}{}_i)}{n}$$

The activation function used with Binary Cross Entropy is the Sigmoid function, as the logarithms exist only if $y^{\wedge}{}_i$ is between 0 and 1.

In PLT the binary_crossentropy () function from the Keras package is used in order to calculate the cross-entropy loss.

2.4.3.3 Rank Margin

In Mean Square Error and Binary Cross Entropy functions we calculate the error depending on the differences between the predicted and actual values. In Rank Margin or Hinge loss as it is often called, the objective is to calculate relative distances between inputs. We only need a similarity score between our data points. For example, in a face recognition we can train a network to choose whether two images belong to the same person or not. Rank Margin is really useful in SVM algorithms, which will be further described later.

Rank Margin Function formula: for a given pair of data samples where x_1 is preferred over x_2:

$$\text{Rank Margin } L(y, y^{\wedge}) = \begin{cases} 0, & y^{\wedge}{}_1 > 1 - y^{\wedge}{}_2 \\ 1 - \left(y^{\wedge}{}_1 - y^{\wedge}{}_2\right), & else \end{cases}$$

2.5 *Neuroevolution*

Neuroevolution is the evolution of neural networks with the use of genetic algorithms [27].

It is more general, in comparison with other learning methods for neural networks, as it enables learning with random network structures. Neuroevolution is a machine learning technique, inspired from the evolution of biological nervous systems in nature. Mostly used in solving reinforcement learning tasks such as game playing, vehicle control, and robotics. Similar to natural selection in nature, which is driven only by feedback from reproductive success, neuroevolution is guided by some measure of overall performance [28].

For instance, in strategic games, it is not always possible to know the best actions at each step, but evolution is capable of observing the performance of a series of actions e.g. resulting in a win or loss. Neuroevolution makes it possible to optimize networks without direct information about what exactly they should be doing, as opposed to the most artificial neural network learning algorithms that operate through supervised learning and are based on labeled input-outputs.

In contrast to most neural learning methods, neuroevolution can optimize the structure of the network, not only the weights of the connections. Furthermore, it can explore ways of altering the network during computation or evaluation and result in learning from experience.

2.5.1 Evolutionary algorithms - Genetic algorithms

Evolutionary algorithms, are an optimization method to solving complex problems that cannot be easily solved in polynomial time. A genetic algorithm, which is the most popular type of Evolutionary Algorithms, is a heuristic search method for solving optimization problems inspired by the theory of natural evolution. Given a population, the best individuals are selected in order to create offspring for the next generation. The better fitness the parents have, the best chances their offspring has at surviving [29].

The procedure starts with a population, meaning a set of individuals with a known size. Each of the individuals is a solution to the problem and is characterized by its genes. Those characteristics (genes) form a chromosome. Two pairs of individuals are chosen based on their fitness scores, calculated by a fitness function, in order to provide new descendants.

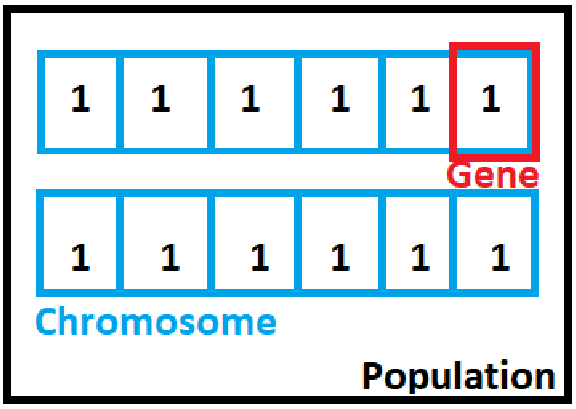


Figure 2: Population, Chromosomes and Genes

For each pair of individuals that are chosen for reproduction, a crossover point is selected within the genes. The parent's genes are exchanged among themselves until the selected point. The individuals with the least fitness die in each creation of a new generation and are replaced by the new descendants.

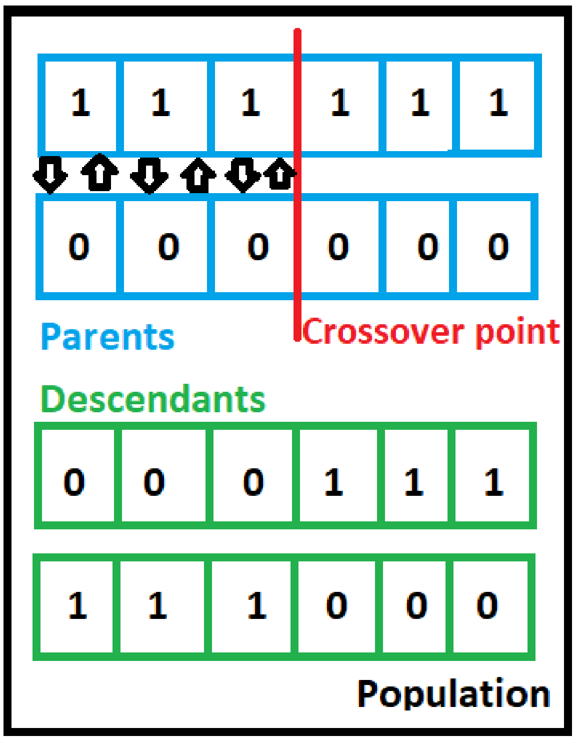


Figure 3: Crossover - Exchanging genes among parents

If the process continues, there is a change of premature convergence. In order to avoid this fact, mutation is performed. By using a low random probability, a few of the descendant's genes are exchanged in order to maintain the population's diversity. When there are no significantly different offspring from the previous generation, the process terminates. A set of solutions to our problem has been found.

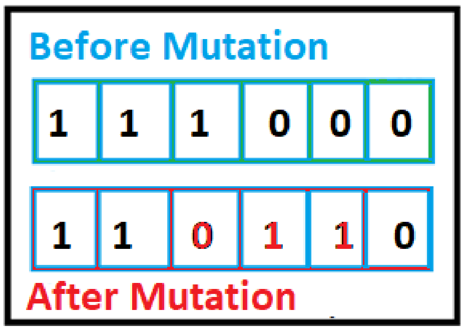


Figure 4: Mutation

Genetic algorithms have many advantages over traditional optimization algorithms. First of all, their ability of dealing with complex problems, irrelevant to the type of the fitness function (nonlinear, discontinuous etc). In addition, as different individuals can be processed simultaneously, genetic algorithms are ideal for parallel implementation. On the other hand, they have a drawback. It is important to explore and select the optimal parameters (eg population size, rate of mutation and crossover) in order to create efficient computational models. Despite this, genetic algorithms are still between the most common optimization methods.

2.5.2 NEAT algorithm

NEAT is the shortcut for NeuroEvolution of Augmenting Topologies. It is a genetic algorithm for the generation of evolving artificial neural networks [29]. Neuroevolution is inspired by natural evolution, thus it heavily mirrors biology. In biology, a genotype is the genetic representation of a creature and the phenotype is its actual, physical representation. In order to represent the individuals of the population genetically in our algorithm a form of encoding is required which can be direct or indirect.

Direct encoding includes an obvious connection between genotype and phenotype. For an individual representing a neural network, each gene is linked to a connection of the network. On the other hand, in indirect encoding there are rules or parameters for creating an individual. Thus, it is harder to create and can result in biases [30].

NEAT algorithm uses a direct encoding methodology. This includes two node gene lists, along with their connections. Between the input and output nodes which are not evolved, multiple hidden nodes can be added. Connection nodes specify the direction of the connection, it's weight and activation, along with an innovation number.

2.5.2.1 **Mutation**

During mutation, NEAT can either mutate existing connections or can add new structure to a network. For every new connection between a start and an end node, a random weight is assigned. A new node must be placed between two nodes that are already connected. For every new node, the previous connection gets disabled and the previous start node is linked to the new node with the weight of the old connection. The link of the new node with the previous end node is assigned a weight of 1.

2.5.2.2 **Crossover**

Randomly crossing the genomes of two neural networks, could result in badly muted or non-functional networks. Genomes could be of different sizes, thus non obviously compatible. In biology, homology is responsible of solving this issue. Homology is the alignment of chromosomes based on matching genes for a specific characteristic. As a result, crossover has fewer chances of error, than if chromosomes were randomly mixed. In order to reduce the chances of creating non-functional individuals, NEAT marks new evolutions with a historical number. Every time a new connection or node is added, a historical marking is assigned, allowing easy alignment when two individuals are to be breed.

2.5.2.3 Speciation

The process of mutation is involved prior to optimization of weights, thus can result in lower performing individuals. In order to allow structures to optimize prior to their entirely elimination from the population, a technique called speciation is performed. This technique divides the population into groups based on the similarity of topology with the aid of historical markings. As a result, individuals compete only with others of the same species and the structure is optimized without the fear of elimination before it is fully explored. Furthermore, through explicit fitness sharing, individuals share their performance across the species and they evolve to even better species. In pyplt though, a simpler form of Neuroevolution algorithm has been applied and will be further discussed in the last chapters.

3

pyPLT

3.1 Python Preference Learning Toolbox pyPLT

Preference Learning Toolbox offers two modes of operation for beginners and advanced users.

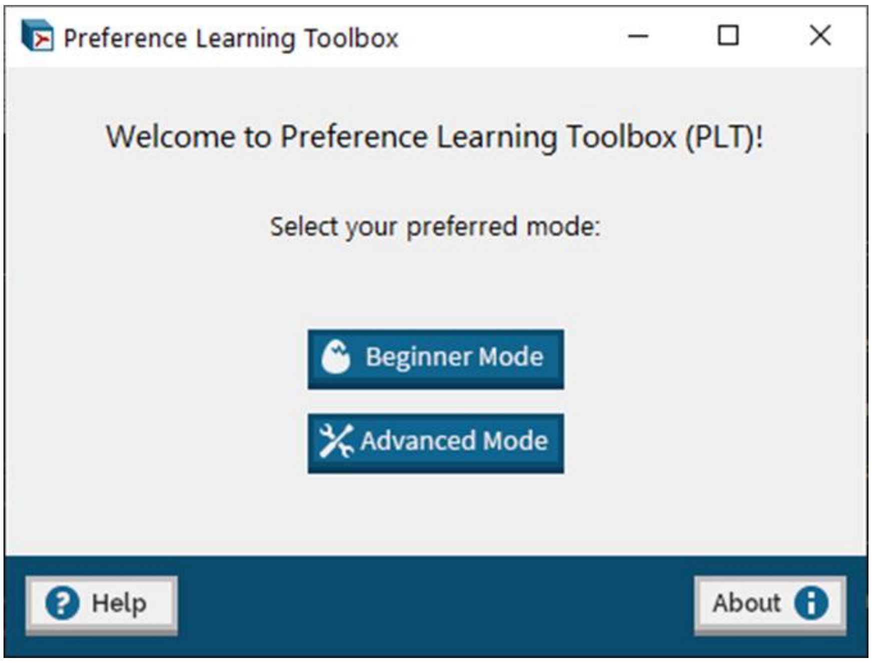

Preference Learning Toolbox
Welcome to Preference Learning Toolbox (PLT)!
Select your preferred mode:
Beginner Mode
Advanced Mode
Help
About

Figure 5: PLT welcome screen

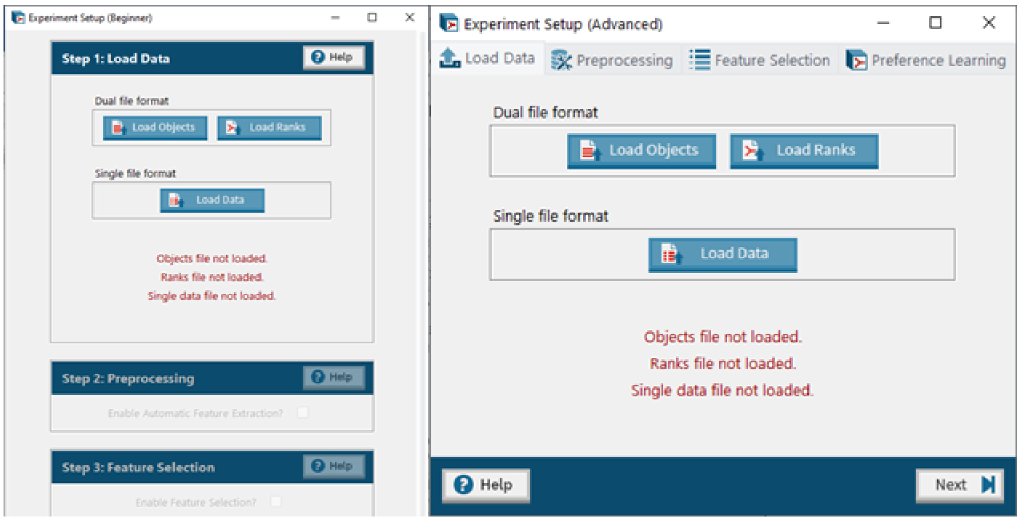
In both modes, the user performs the experiment setup in five steps: loading the dataset, data pre-processing, feature selection, preference learning and running the experiment.

In beginner mode, once the dataset is correctly loaded, there are a few parameters to tune. The preprocessing step consists of specifying whether and how many features are to be automatically extracted from the data set, choosing whether or not to apply feature selection, choosing a preference learning algorithm, and finally running the experiment.

In advanced mode the user is offered a more detailed setup. In preprocessing there is a normalization feature and in the remaining steps the setup may be tuned through a set of options or parameters.

3.1.1 Dataset Loading

The dataset may be loaded in single format for problems where a total order of objects exists, or in dual format where a partial order of objects is known. All the remaining steps are available only after the successful loading of the dataset.



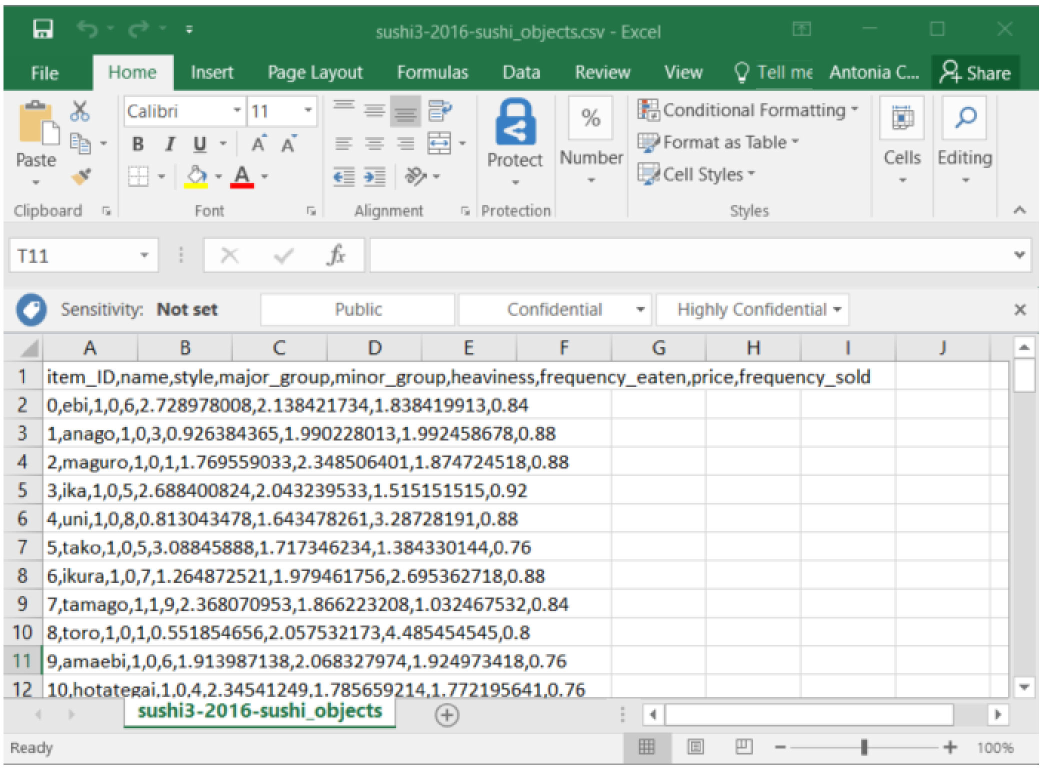
Objects file not loaded.
Ranks file not loaded.
Single data file not loaded.

Objects file not loaded.
Ranks file not loaded.
Single data file not loaded.

Figure 6: Dataset Loading Beginner Vs Advanced mode

In the single file format, a comma-separated-value file must be uploaded containing the objects with their individual ratings. Each row contains the numeric feature values of an object separated by comma and the last column contains the relation.

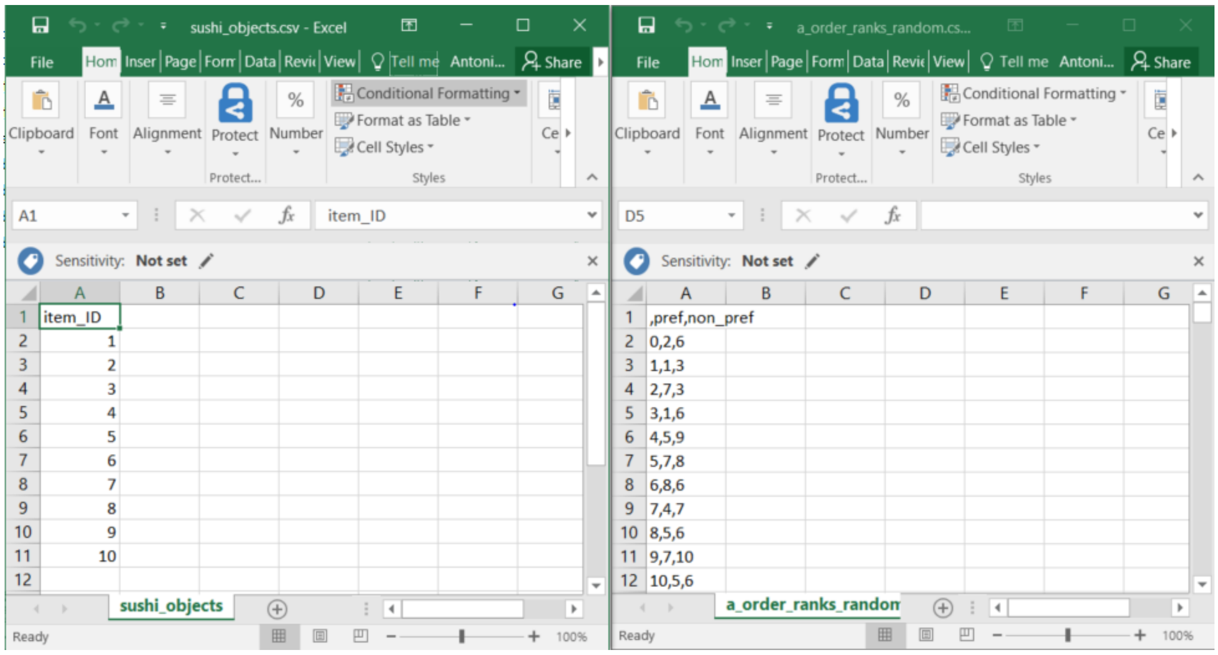
A useful dataset for preference learning experiments is the Sushi dataset. It includes the findings of a series of surveys released by Toshihiro Kamishima, in which 5000 participants responded regarding their preferences about various kinds of sushi.



```
item_ID,name,style,major_group,minor_group,heaviness,frequency_eaten,price,frequency_sold
0,ebi,1,0,6,2.728978008,2.138421734,1.838419913,0.84
1,anago,1,0,3,0.926384365,1.990228013,1.992458678,0.88
2,maguro,1,0,1,1.769559033,2.348506401,1.874724518,0.88
3,ika,1,0,5,2.688400824,2.043239533,1.515151515,0.92
4,uni,1,0,8,0.813043478,1.643478261,3.28728191,0.88
5,tako,1,0,5,3.08845888,1.717346234,1.384330144,0.76
6,ikura,1,0,7,1.264872521,1.979461756,2.695362718,0.88
7,tamago,1,1,9,2.368070953,1.866223208,1.032467532,0.84
8,toro,1,0,1,0.551854656,2.057532173,4.485454545,0.8
9,amaebi,1,0,6,1.913987138,2.068327974,1.924973418,0.76
10,hotategai,1,0,4,2.34541249,1.785659214,1.772195641,0.76
```

Figure 7: Single file format dataset with ratings

In dual file format, two csv files must be uploaded: a file containing the objects and a file containing the pairwise preferences for a number of objects in the dataset (ranks file). Each row of the ranks file contains a pair of object IDs, the first being that of the preferred object in the pair and the second being that of the non-preferred object in the pair. If the object file does not contain object IDs, the row number is used as ID.



sushi_objects.csv - Excel

A1 | item_ID

	A
1	item_ID
2	1
3	2
4	3
5	4
6	5
7	6
8	7
9	8
10	9
11	10
12	

sushi_objects

a_order_ranks_random.cs...

D5

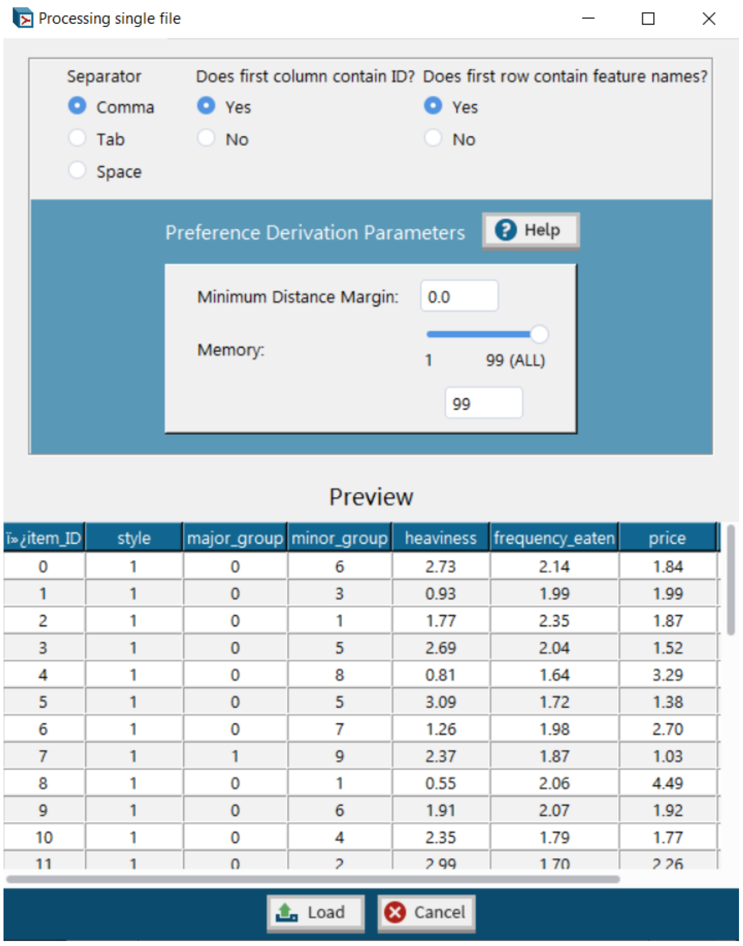
	A
1	,pref,non_pref
2	0,2,6
3	1,1,3
4	2,7,3
5	3,1,6
6	4,5,9
7	5,7,8
8	6,8,6
9	7,4,7
10	8,5,6
11	9,7,10
12	10,5,6

a_order_ranks_randon

Figure 8: Dual file format objects with ranks

3.1.2 Dataset Pre-processing

In the single file format the user is given the option to control how the pairwise preferences (ranks) are derived from the ratings-based data through the minimum distance margin and memory parameters. The minimum distance margin is the minimum required difference between the ratings of a given pair of objects for the object pair to be considered a valid preference. During the creation of pairwise ranks, the memory parameter indicates how many nearby items should be compared with a given object.



Separator: [✓ Comma] Tab Space

Does first column contain ID? [✓ Yes] No

Does first row contain feature names? [✓ Yes] No

Preference Derivation Parameters

Minimum Distance Margin: 0.0

Memory: 1 99 (ALL)

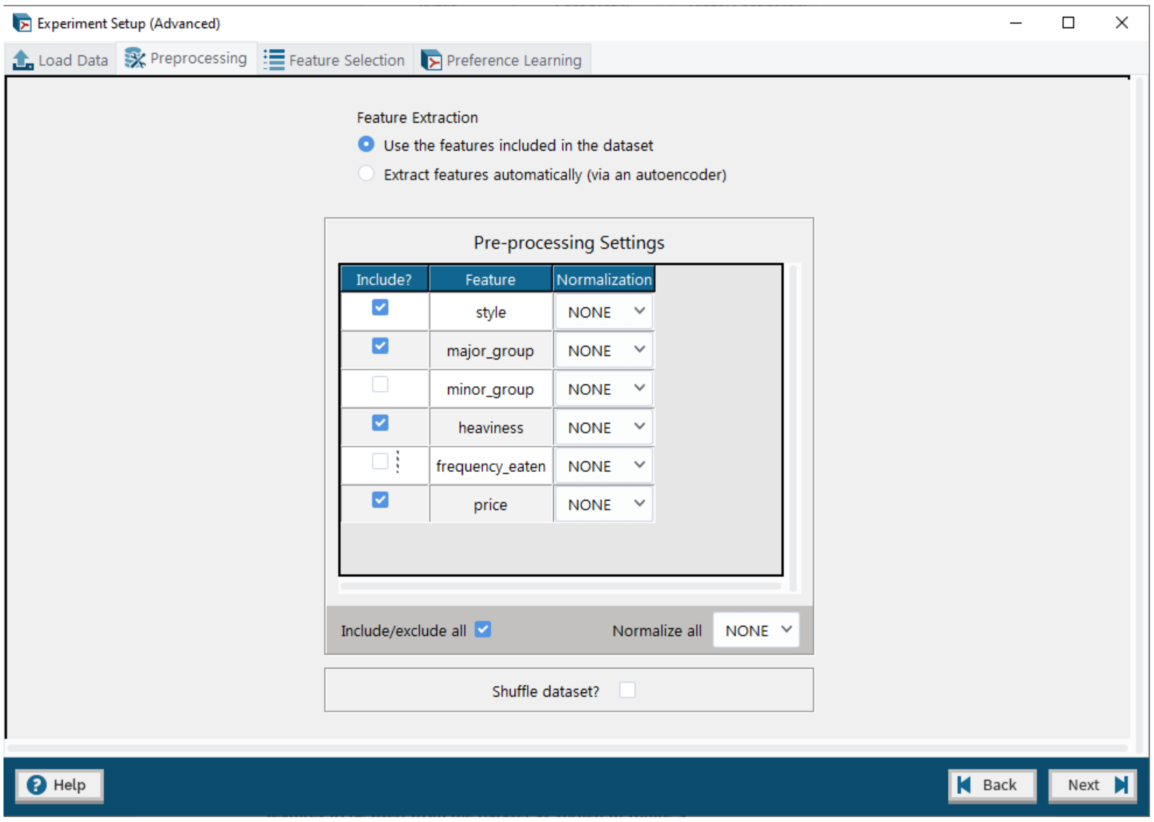
99

Preview

ï»¿item_ID	style	major_group	minor_group	heaviness	frequency_eaten	price
0	1	0	6	2.73	2.14	1.84
1	1	0	3	0.93	1.99	1.99
2	1	0	1	1.77	2.35	1.87
3	1	0	5	2.69	2.04	1.52
4	1	0	8	0.81	1.64	3.29
5	1	0	5	3.09	1.72	1.38
6	1	0	7	1.26	1.98	2.70
7	1	1	9	2.37	1.87	1.03
8	1	0	1	0.55	2.06	4.49
9	1	0	6	1.91	2.07	1.92
10	1	0	4	2.35	1.79	1.77
11	1	0	2	2.99	1.70	2.26

Figure 9: Processing a single file

PLT offers a small preview of the object features within a loaded dataset as shown in figure 3. The user selects several parameters such as the symbol used to separate entries, whether the dataset contains specific IDs, or feature names. In advanced mode the user can specify particular features to be used from the dataset as shown in figure 10.


Experiment Setup (Advanced)

Load Data | Preprocessing | Feature Selection | Preference Learning

Feature Extraction
- [✓] Use the features included in the dataset
- Extract features automatically (via an autoencoder)

Pre-processing Settings

Include?	Feature	Normalization
✓	style	NONE
✓	major_group	NONE
	minor_group	NONE
✓	heaviness	NONE
	frequency_eaten	NONE
✓	price	NONE

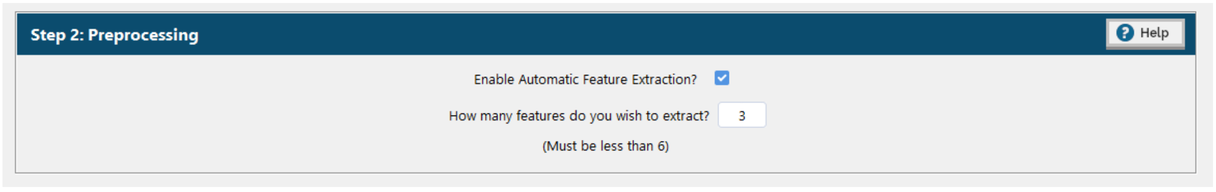
Include/exclude all ✓   Normalize all NONE

Shuffle dataset? ☐

Figure 10: Pre-processing | Advanced User

3.1.2.1 **Feature Extraction**

If the dataset does not include features, the tool offers an automatic feature extraction in order to extract features from the data using an auto encoder. The auto encoder consists of multiple layers: an encoder, the code layer and a decoder. The encoder compresses the input data and the decoder decompresses them in order to create an accurate reconstruction of the input. The auto encoder is optimized using the Adam Optimizer and its performance is determined via the Mean Squared Error function.


Step 2: Preprocessing

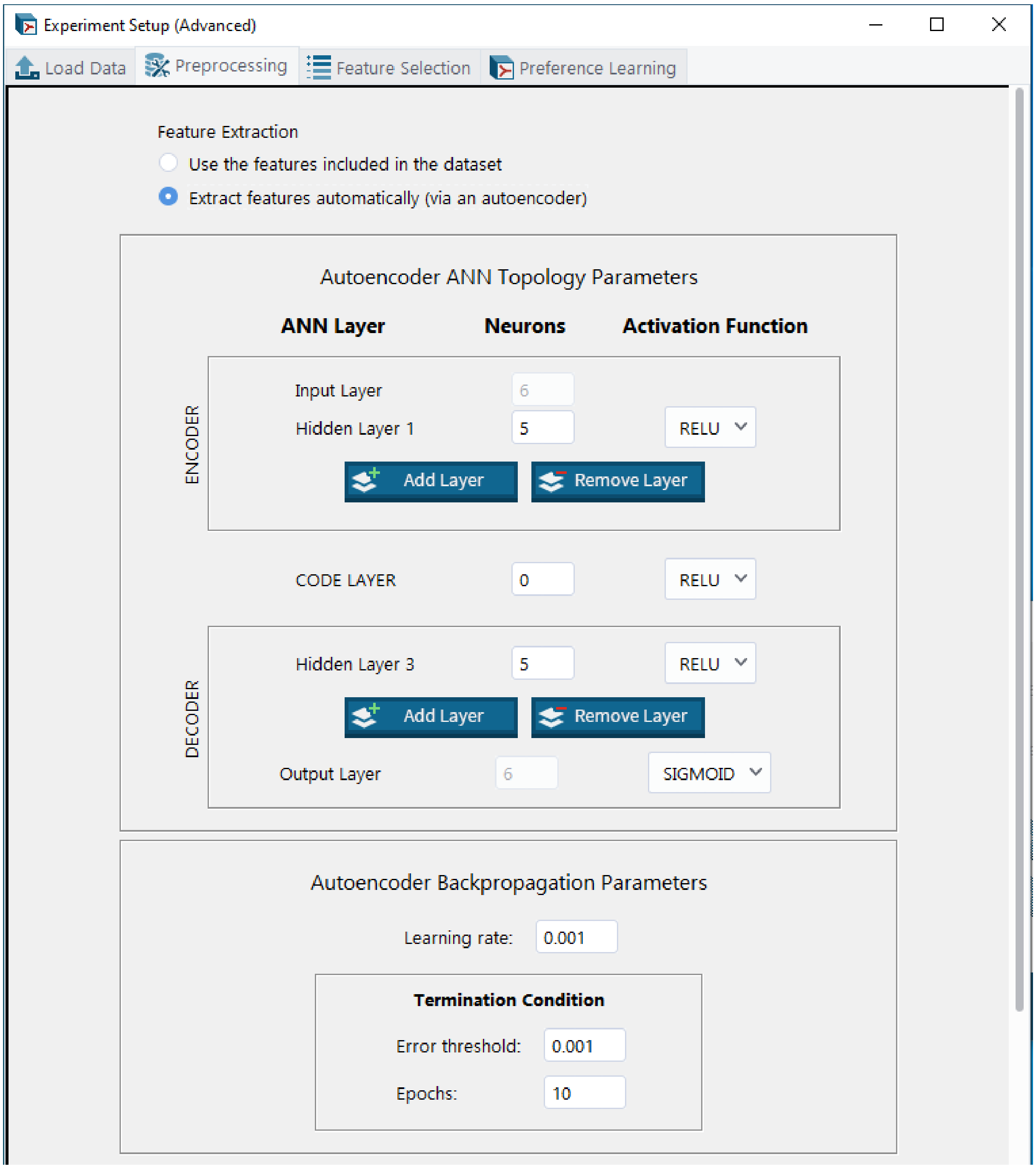
Enable Automatic Feature Extraction? ✓

How many features do you wish to extract? 3

(Must be less than 6)

Figure 11: Feature Extraction | Beginner

In advanced mode the user can specify the parameters of the backpropagation algorithm which trains the auto encoder.



Feature Extraction

( ) Use the features included in the dataset

(✓) Extract features automatically (via an autoencoder)

Autoencoder ANN Topology Parameters

ANN Layer	Neurons	Activation Function
Input Layer	6	
Hidden Layer 1	5	RELU
CODE LAYER	0	RELU
Hidden Layer 3	5	RELU
Output Layer	6	SIGMOID

Autoencoder Backpropagation Parameters

Learning rate: 0.001

Termination Condition

Error threshold: 0.001

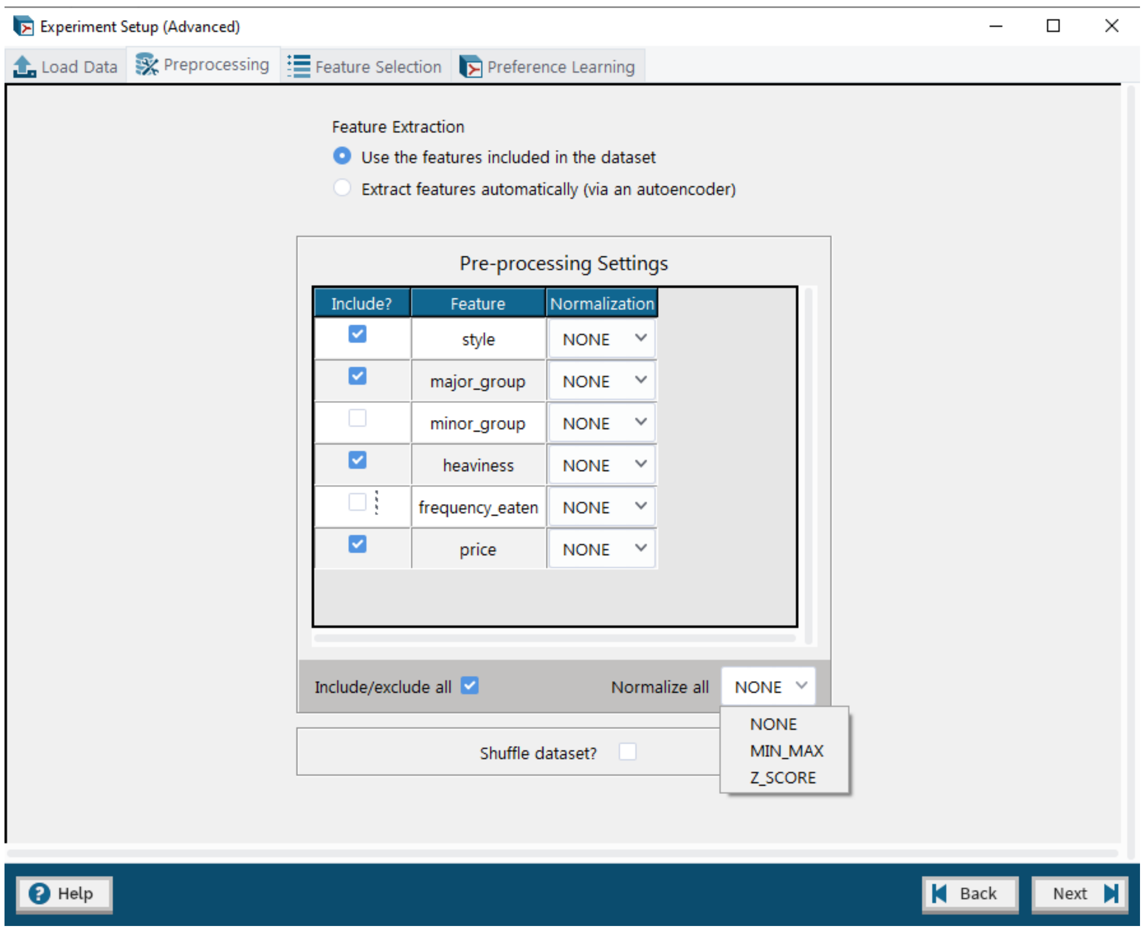
Epochs: 10

Figure 12: Feature Extraction | Advanced

Apart from feature extraction, in advanced mode the dataset preprocessing offers feature normalization and dataset shuffling prior to running the experiment.

3.1.2.2 Feature Normalization

By default, all features in the dataset are normalized using the Min-Max method which transforms the values of the given features to fit the range of 0 to 1. In advanced mode Z-score method can also be used, which transforms the values in order to have an average value of the feature equal to 0 and its standard deviation equal to 1. Feature normalization prior to model fitting is important, because variables that are measured at different scales do not contribute equally to the model fitting and might end up creating a bias.



Feature Extraction

● Use the features included in the dataset

○ Extract features automatically (via an autoencoder)

Pre-processing Settings

Include?	Feature	Normalization
✓	style	NONE
✓	major_group	NONE
☐	minor_group	NONE
✓	heaviness	NONE
☐	frequency_eaten	NONE
✓	price	NONE

Include/exclude all ✓ Normalize all NONE

NONE
MIN_MAX
Z_SCORE

Shuffle dataset? ☐

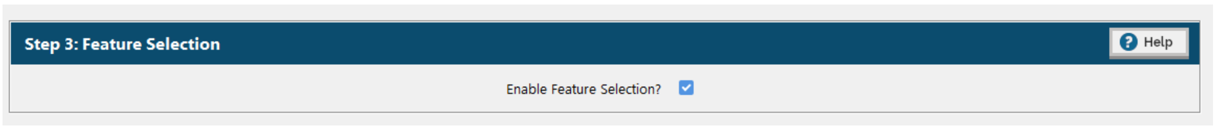
Figure 13: Feature Normalization | Advanced User

3.1.2.3 **Feature Selection**

Feature Selection is the process of selecting out the most significant features from our dataset. In Machine Learning, removing the less relevant to the predictive model features is considered a good practice. In advanced mode the user can manually deactivate any irrelevant features. However, manually identifying these features is not always possible or optimal. For example, when the dataset contains a huge amount of features, the user has to spend a lot of time and effort in order to choose the correct features. PLT offers an automatic feature selection algorithm in such cases: the Sequential Forward Selection method.

3.1.2.4 **Sequential Forward Selection**

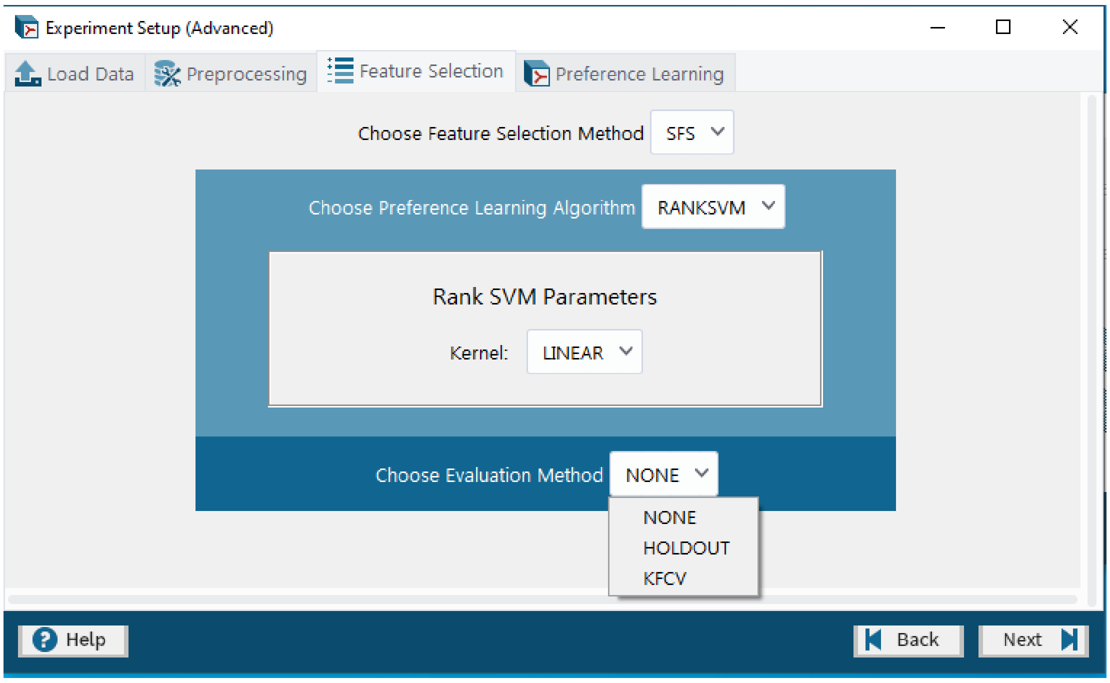
In Sequential Forward Selection the procedure begins with an empty feature set and the feature that is added in every step results in the maximization of set performance over all the remaining features. When an added feature yields equal or lower performance to the performance obtained without it, the selection part is over. The performance of each subset of features is computed as the prediction accuracy of a model trained with those features as an input. Also, the trained model is tested using Holdout validation in order to ensure that it performs well on new data.



Enable Feature Selection? ✓

Figure 14: Feature Selection | Beginner

In advanced mode the user can choose which preference learning algorithm implemented in the tool will be used to create the model for sequential forward selection. In addition, the user can choose whether to train the model using the complete dataset and assessing the performance as the percentage of correctly classified training pairs or can test the generality of the results by using Holdout Validation or K-Fold Cross Validation.



Choose Feature Selection Method SFS

Choose Preference Learning Algorithm RANKSVM

Rank SVM Parameters

Kernel: LINEAR

Choose Evaluation Method NONE

NONE
HOLDOUT
KFCV

Figure 15: Feature Selection | Advanced

3.1.2.5 **Holdout Validation**

A part of the dataset is used to train the model while the rest of the dataset is used to validate performance of the model. The split ratio is 70:30.

3.1.2.6 **K-Fold Cross Validation**

In k-fold cross-validation, the data is divided into k folds. k-1 folds are used for the model training and the last fold is for testing. This method ensures that each fold of the dataset has the opportunity to be used in the test set 1 time and used to train the model k-1 times. Once the process is completed, it returns the percentage of correctly classified pairs not used for training.

3.1.3 Preference Learning Algorithms

PyPLT offers three options of preference learning algorithms for both user modes: RankSVM, Backpropagation and RankNET.

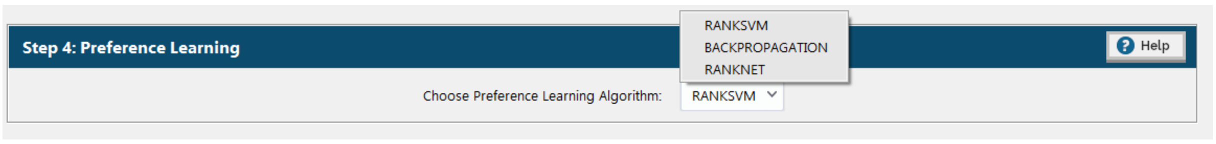
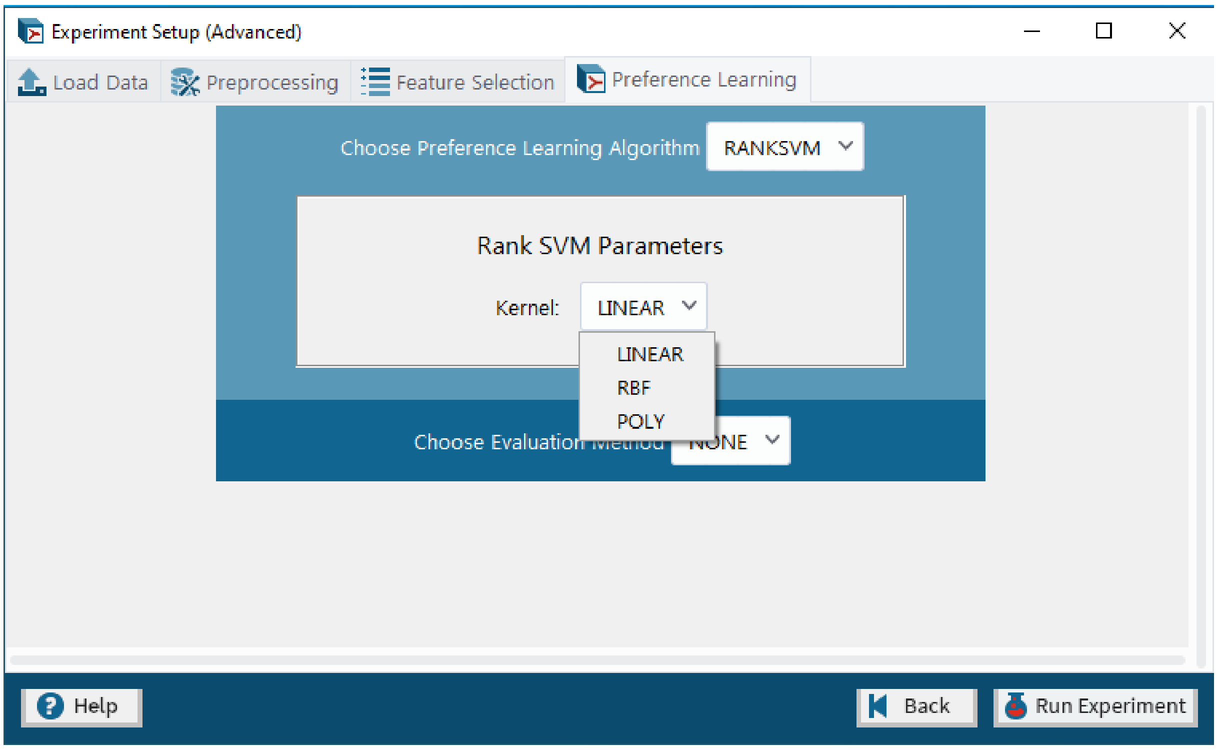

Step 4: Preference Learning
RANKSVM
BACKPROPAGATION
RANKNET
Help
Choose Preference Learning Algorithm:
RANKSVM

Figure 16: Preference Learning Algorithms | Beginner

3.1.3.1 RankSVM

The RankSVM algorithm is a rank-based variation of the traditional Support Vector Machine algorithm. SVM maps the data instances onto geometric points in a high-dimensional space according to their features via a pre-defined kernel function. The kernel transforms a low-dimensional input space into a higher dimensional. Then, the algorithm attempts to split the instances according to their annotated category via a hyperplane. The hyperplane divides a set of objects that have different class memberships. The goal is the optimal segregation of the given dataset. Unseen data instances may be mapped to the space according to their features and a category is produced (output) based on which sub-space they correspond to [1].



Choose Preference Learning Algorithm: RANKSVM

Rank SVM Parameters

Kernel: LINEAR

LINEAR
RBF
POLY

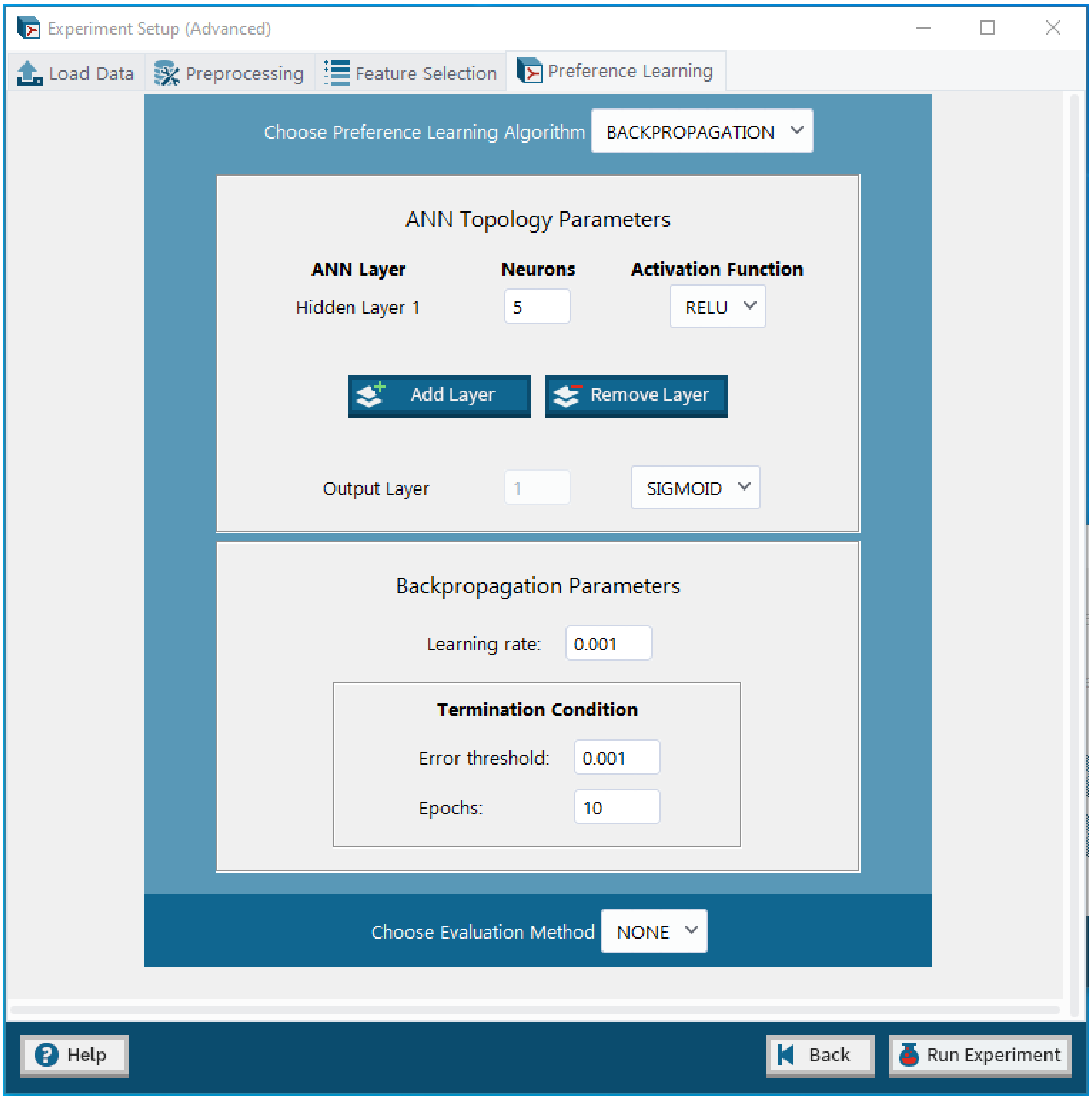
Choose Evaluation Method: NONE

Figure 17: RankSVM Parameter Selection in advanced mode

3.1.3.2 **Backpropagation**

The principle of the backpropagation approach is to model a given function by altering internal weightings of input signals to produce an expected output signal. Training is performed with a supervised learning method. An input is forward propagated in order to calculate an output and the error is back- propagated in order to train the network.

Backpropagation is a gradient-descent algorithm that iteratively optimizes an error function by adjusting the weights of an artificial neural network model. The error function used is the Rank Margin function and the total error is averaged over the complete set of pairs in the training dataset. If the error is below a given threshold, training stops before reaching the specified number of epochs and the current weight values are returned as the final model. In the beginner mode, the network topology contains one hidden layer of 5 neurons and uses the ReLu activation function for each neuron in that layer, while the output neuron uses the Logistic Sigmoid activation function. In advanced mode, the user specifies the desired parameters [1].



Choose Preference Learning Algorithm: BACKPROPAGATION

ANN Topology Parameters

ANN Layer	Neurons	Activation Function
Hidden Layer 1	5	RELU
Output Layer	1	SIGMOID

Backpropagation Parameters

Learning rate: 0.001

Termination Condition

Error threshold: 0.001

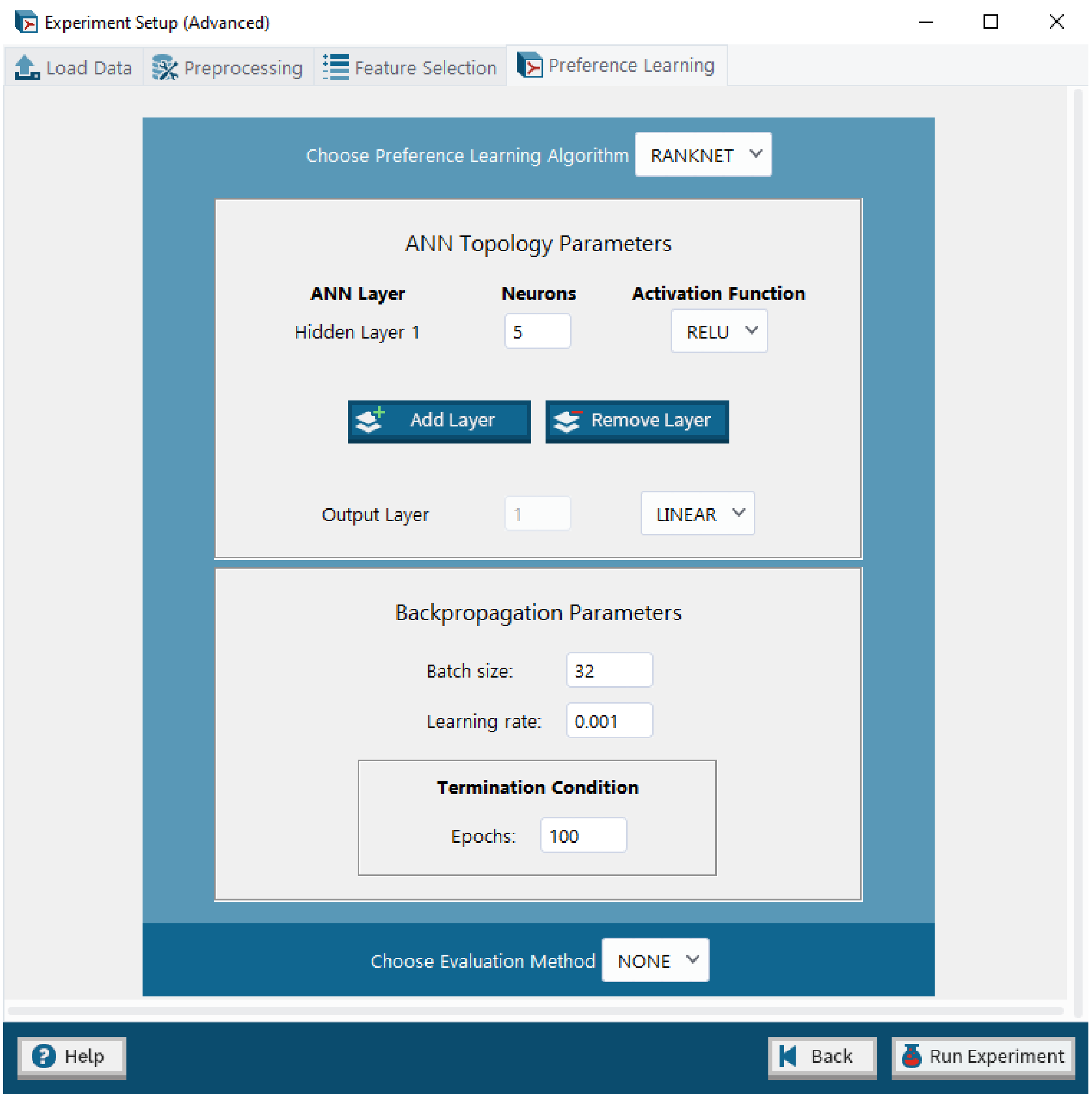
Epochs: 10

Choose Evaluation Method: NONE

Figure 18: Backpropagation Parameter Selection in advanced mode

3.1.3.3 **RankNET**

RankNET algorithm is a variant of the Backpropagation algorithm that handles ordered pairs of data using a probabilistic cost function. The error function used is the binary cross-entropy function and the total error is averaged over the complete set of pairs in the training dataset as in the previous algorithm.



Choose Preference Learning Algorithm: RANKNET

ANN Topology Parameters

ANN Layer	Neurons	Activation Function
Hidden Layer 1	5	RELU
Output Layer	1	LINEAR

Backpropagation Parameters

Batch size: 32

Learning rate: 0.001

Termination Condition

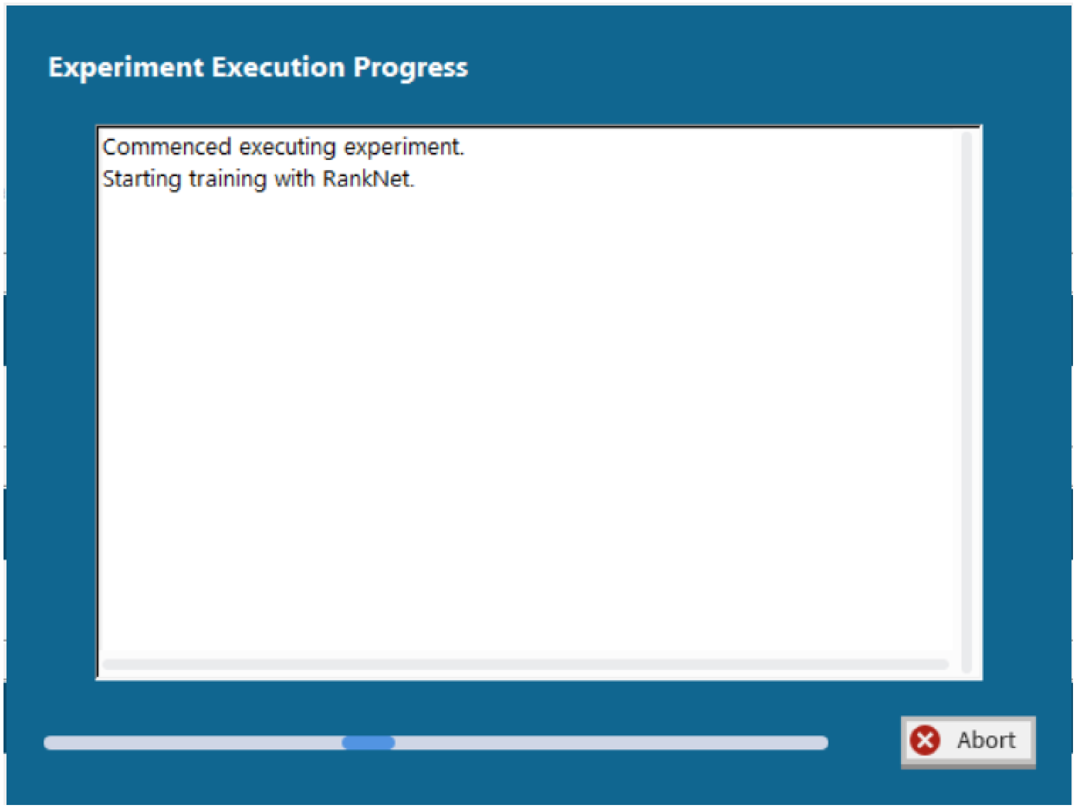
Epochs: 100

Choose Evaluation Method: NONE

Figure 19: RankNET Parameter Selection in advanced mode

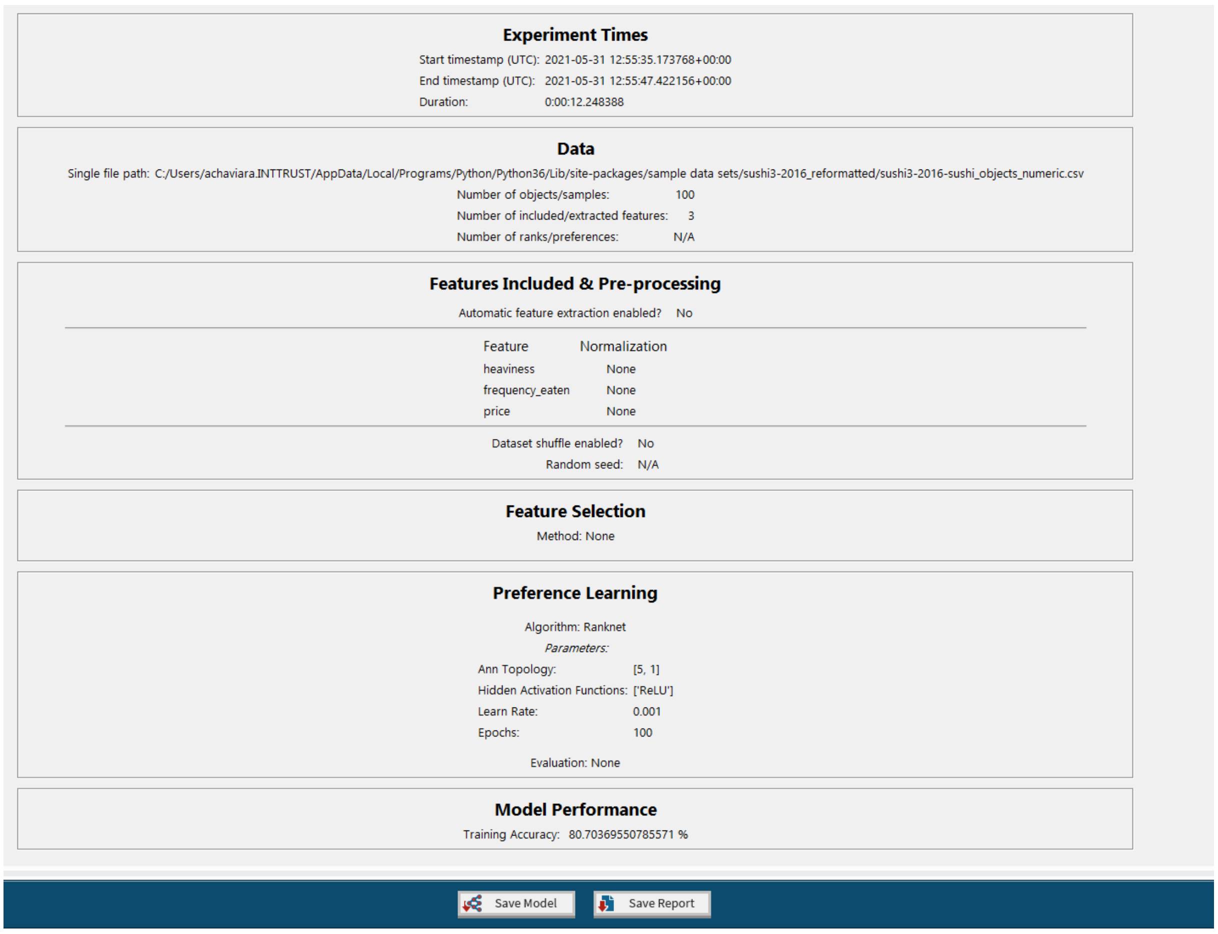
3.1.4 Experiment Reporting and Model Storage

Once the experiment is set up and the execution begins, a progress report appears (Figure 20). When the whole process is completed, a summary screen follows displaying the configuration of each step and the training/validation accuracies of the final model (Figure 21). The user can save the experiment report as well as the final model to file.



```
Commenced executing experiment.
Starting training with RankNet.
```

Figure 20: Execution Progress Window



```
Experiment Times
Start timestamp (UTC):  2021-05-31 12:55:35.173768+00:00
End timestamp (UTC):    2021-05-31 12:55:47.422156+00:00
Duration:               0:00:12.248388

Data
Single file path:  C:/Users/achaviara.INTTRUST/AppData/Local/Programs/Python/Python36/Lib/site-packages/sample data sets/sushi3-2016_reformatted/sushi3-2016-sushi_objects_numeric.csv
Number of objects/samples:               100
Number of included/extracted features:     3
Number of ranks/preferences:             N/A

Features Included & Pre-processing
Automatic feature extraction enabled?   No

Feature          Normalization
heaviness           None
frequency_eaten     None
price               None

Dataset shuffle enabled?   No
            Random seed:   N/A

Feature Selection
Method: None

Preference Learning
Algorithm: Ranknet
Parameters:
Ann Topology:                  [5, 1]
Hidden Activation Functions:   ['ReLU']
Learn Rate:                    0.001
Epochs:                        100

Evaluation: None

Model Performance
Training Accuracy:  80.70369550785571 %
```

Figure 21: Experiment Report

4

Fixing a broken pyPLT

4.1.1 Graphical Users Interface problem

When the dataset contains a large amount of data, the graphical users interface freezes and stops being responsive. As a result, the user is not aware of the progress and whether the loading process is aborted or slowed down. The problem appears because the GUI is running on the same thread as the main logic.

A thread is a separate flow of execution. Threading is the ability to have different, individual units of a process (running program) running concurrently. Each process has at least one thread: the main thread (or the process itself). Multiple threads work together to achieve a common goal. The resources and memory available to the main thread, are also shared among the other threads within the same process.

An important advantage of multi-threading is the speed up of the execution, provided that the program has multiple CPUs. Also, other tasks can be performed simultaneously with the I/O operations. For example in a program where a thread involves an algorithm and another thread the input reading. The algorithm doesn't have to wait for the input to be fully read and respectively the input reading doesn't wait for the algorithm to complete its calculations.

However, for python cases, there is a limitation stating that only a thread can be run at a time due to Global Interpreter Lock. Even in cases of processors with multiple cores, the global interpreter allows only one thread to be executed at a time. That happens in order to prevent overwriting of data in memory in case multiple threads might try to acquire the memory. However, in Python 3, each thread is given a defined amount of time to execute, such as 5 milliseconds. If the thread doesn't release the lock after 5ms, it will be forced to release it, in order to prevent starvation of CPU time. Also, CPU-bound jobs are prioritized compared to I/O operations.

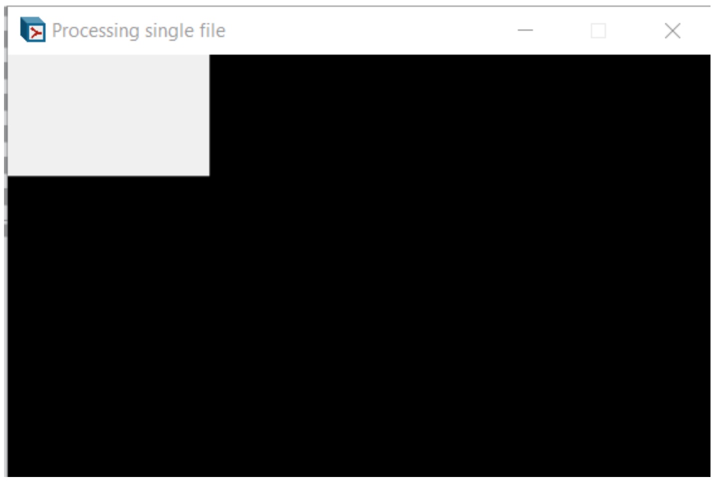
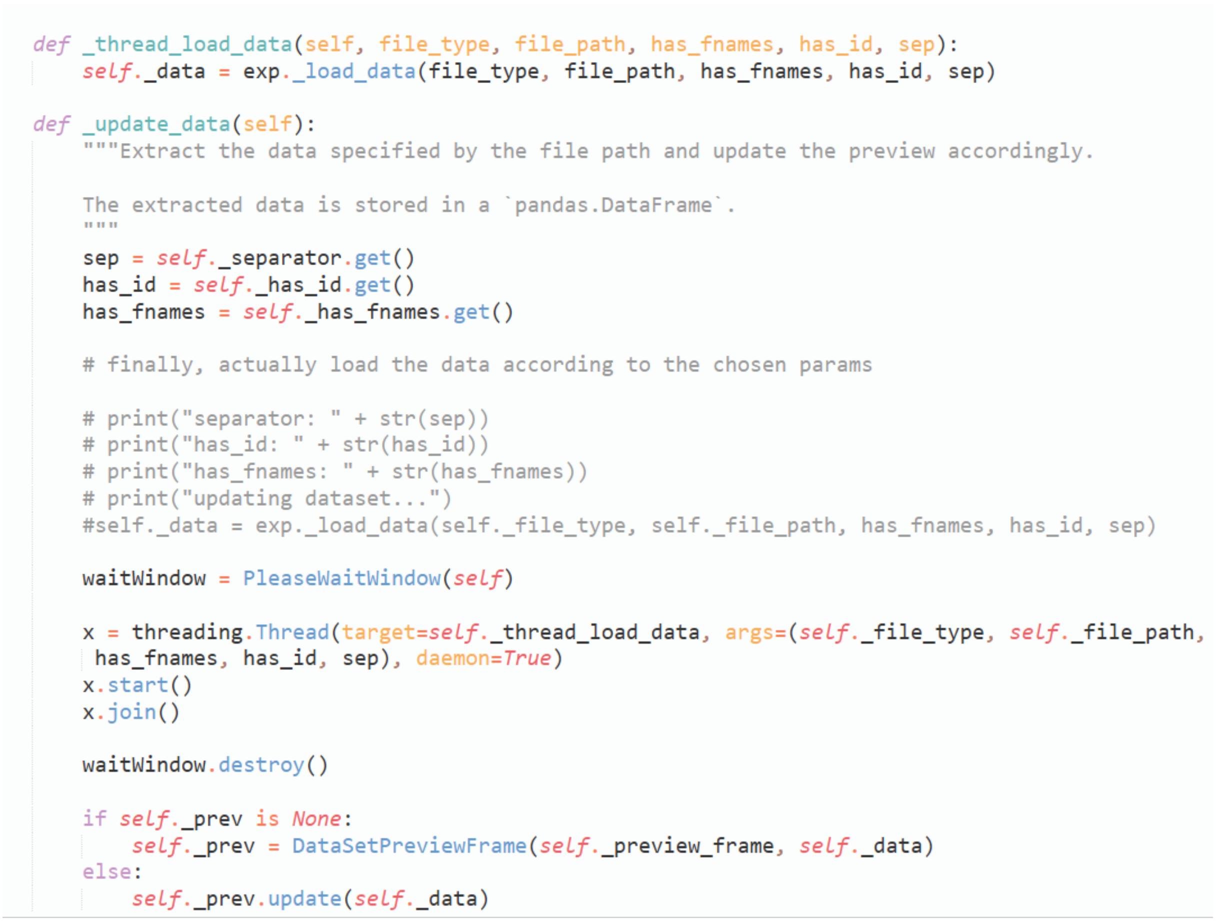


Figure 22: Freezing window while loading large dataset

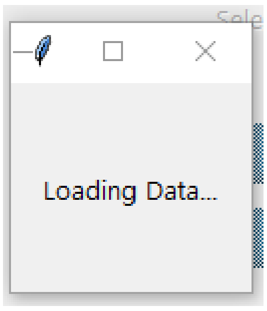
In order to solve the freezing problem, a thread was created which calls a function that loads the data. The new function is **_thread_load_data** and the function **update_data** that originally loaded the data, now creates the thread and passes the parameters to the new function (Figure 23).



```
def _thread_load_data(self, file_type, file_path, has_fnames, has_id, sep):
    self._data = exp._load_data(file_type, file_path, has_fnames, has_id, sep)

def _update_data(self):
    """Extract the data specified by the file path and update the preview accordingly.

    The extracted data is stored in a `pandas.DataFrame`.
    """
    sep = self._separator.get()
    has_id = self._has_id.get()
    has_fnames = self._has_fnames.get()

    # finally, actually load the data according to the chosen params

    # print("separator: " + str(sep))
    # print("has_id: " + str(has_id))
    # print("has_fnames: " + str(has_fnames))
    # print("updating dataset...")
    #self._data = exp._load_data(self._file_type, self._file_path, has_fnames, has_id, sep)

    waitWindow = PleaseWaitWindow(self)

    x = threading.Thread(target=self._thread_load_data, args=(self._file_type, self._file_path,
     has_fnames, has_id, sep), daemon=True)
    x.start()
    x.join()

    waitWindow.destroy()

    if self._prev is None:
        self._prev = DataSetPreviewFrame(self._preview_frame, self._data)
    else:
        self._prev.update(self._data)
```

Figure 23: Threading implementation

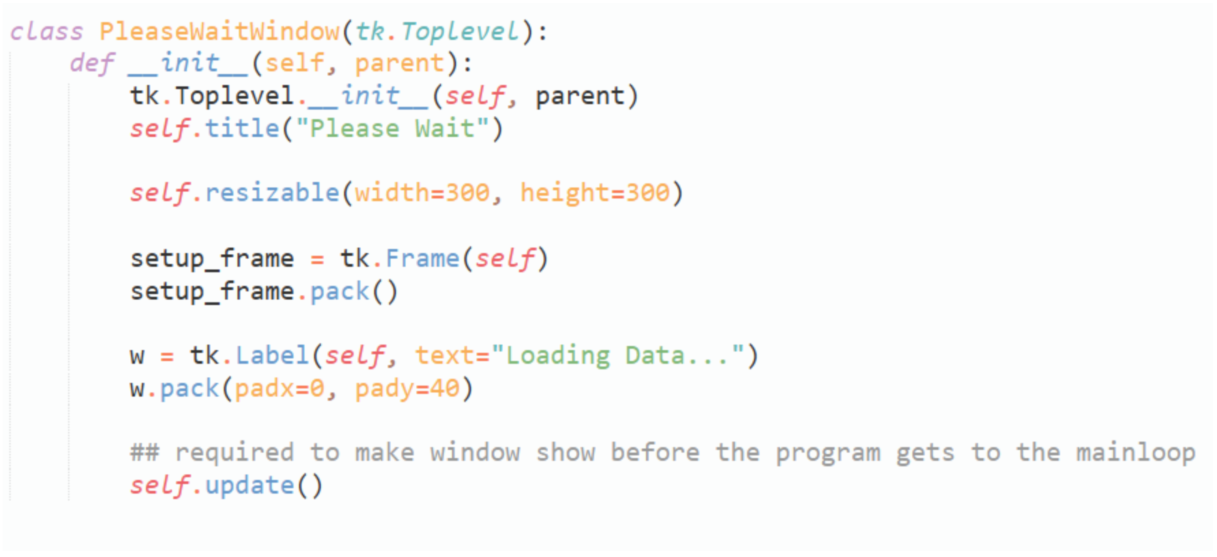
However, the problem seems to remain. A wait window was added, so the user knows that the system is working while waiting for a large dataset to be fully loaded (Figure 24).



Loading Data...

Figure 24: Wait window

The wait window creation is shown in figure 25:



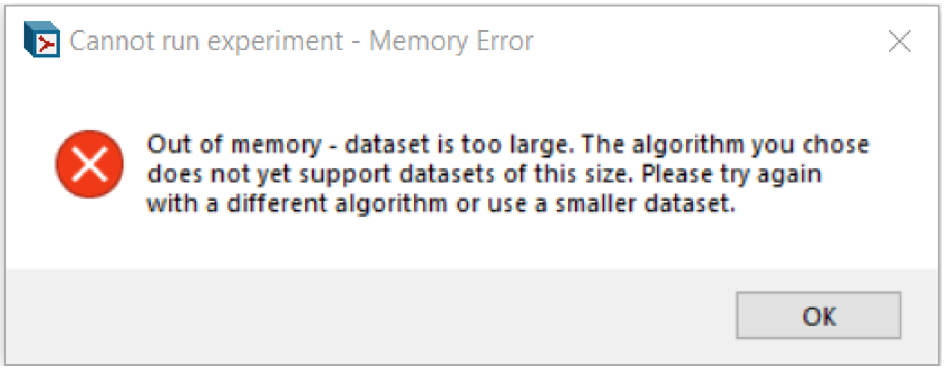
```
class PleaseWaitWindow(tk.Toplevel):
    def __init__(self, parent):
        tk.Toplevel.__init__(self, parent)
        self.title("Please Wait")

        self.resizable(width=300, height=300)

        setup_frame = tk.Frame(self)
        setup_frame.pack()

        w = tk.Label(self, text="Loading Data...")
        w.pack(padx=0, pady=40)

        ## required to make window show before the program gets to the mainloop
        self.update()
```

Figure 25: Wait window class

When the LoadingParamsWindow (Figure 19) is created for the first time it gets hidden and the wait window is created (Figures 24, 25). Join function waits until the thread execution is completed before it proceeds, as the data must be loaded before proceeding. Once the data is fully loaded, wait window gets destroyed (Figure 23) and the main window appears again.

4.1.2 Memory error

The tool was tested with artificially created datasets of various sizes, in order to generate the memory error (Figure 26).



Cannot run experiment - Memory Error

Out of memory - dataset is too large. The algorithm you chose does not yet support datasets of this size. Please try again with a different algorithm or use a smaller dataset.

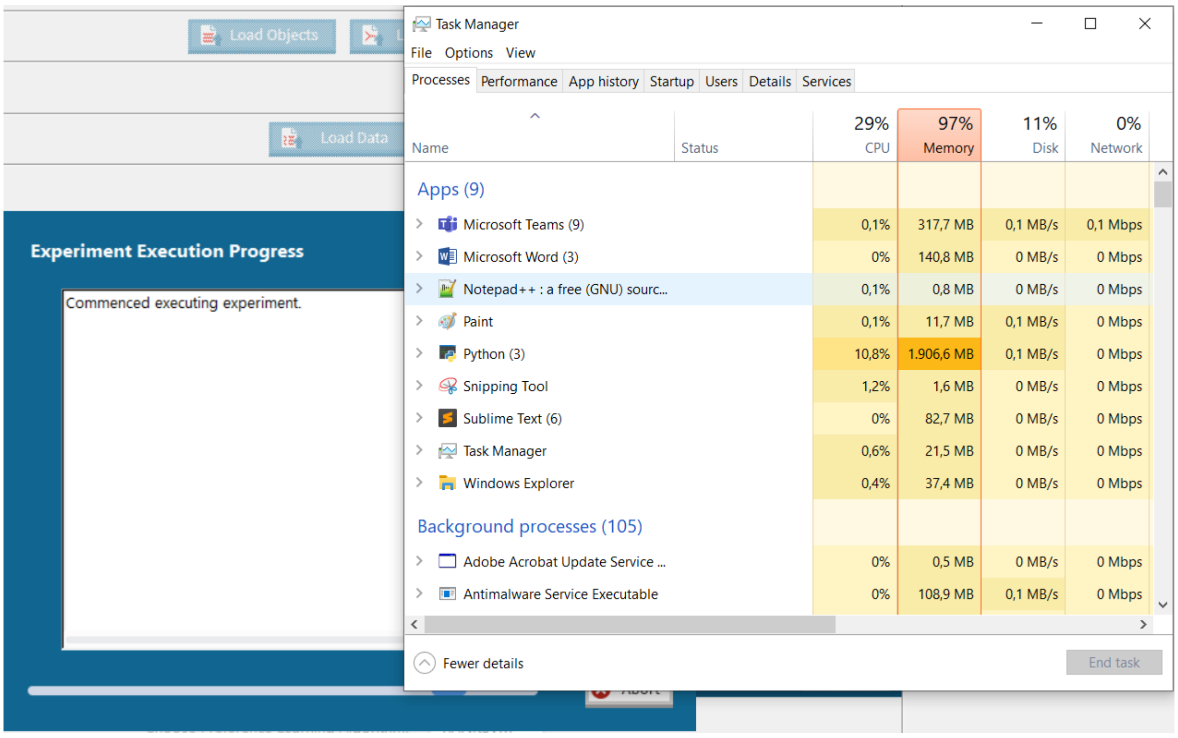
OK

Figure 26: Memory error

Three large files were created:

a. Dataset with 10.000 rows:

The file is 324KB and the program runs without errors, using a significant part of memory. After an hour it was not completed so the execution was stopped manually.



Commenced executing experiment.

Task Manager

File Options View

Processes Performance App history Startup Users Details Services

Name	Status	29% CPU	97% Memory	11% Disk	0% Network
Apps (9)					
Microsoft Teams (9)		0,1%	317,7 MB	0,1 MB/s	0,1 Mbps
Microsoft Word (3)		0%	140,8 MB	0 MB/s	0 Mbps
Notepad++ : a free (GNU) sourc...		0,1%	0,8 MB	0 MB/s	0 Mbps
Paint		0,1%	11,7 MB	0,1 MB/s	0 Mbps
Python (3)		10,8%	1.906,6 MB	0,1 MB/s	0 Mbps
Snipping Tool		1,2%	1,6 MB	0 MB/s	0 Mbps
Sublime Text (6)		0%	82,7 MB	0 MB/s	0 Mbps
Task Manager		0,6%	21,5 MB	0 MB/s	0 Mbps
Windows Explorer		0,4%	37,4 MB	0 MB/s	0 Mbps
Background processes (105)					
Adobe Acrobat Update Service ...		0%	0,5 MB	0 MB/s	0 Mbps
Antimalware Service Executable		0%	108,9 MB	0,1 MB/s	0 Mbps

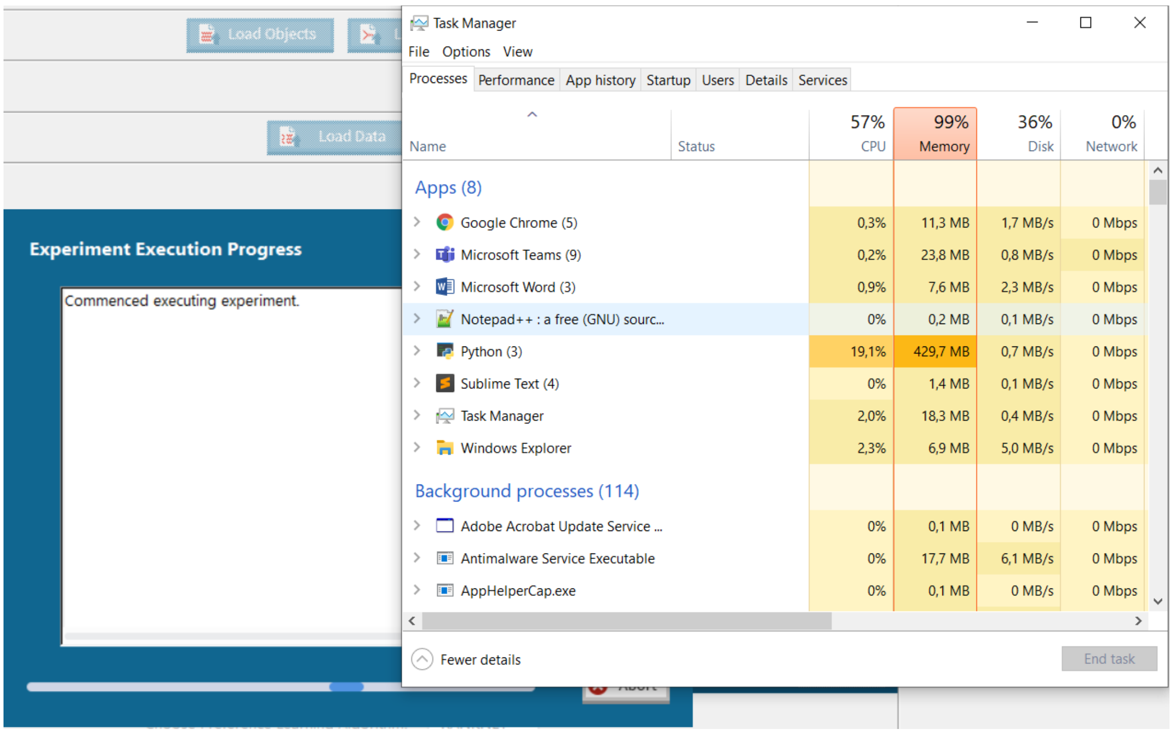
Fewer details

End task

Figure 27: Experiment with 10k dataset

b. Dataset with 50.000 rows:

The program could run for a few seconds without error, but the operating system had slowed down significantly and was practically frozen. After a few seconds the memory error appeared and the execution was aborted. A large use of memory was noticed.


Load Objects
Load Data
Experiment Execution Progress
Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Commenced executing experiment.

Name	Status	57% CPU	99% Memory	36% Disk	0% Network
Apps (8)					
Google Chrome (5)		0,3%	11,3 MB	1,7 MB/s	0 Mbps
Microsoft Teams (9)		0,2%	23,8 MB	0,8 MB/s	0 Mbps
Microsoft Word (3)		0,9%	7,6 MB	2,3 MB/s	0 Mbps
Notepad++ : a free (GNU) sourc...		0%	0,2 MB	0,1 MB/s	0 Mbps
Python (3)		19,1%	429,7 MB	0,7 MB/s	0 Mbps
Sublime Text (4)		0%	1,4 MB	0,1 MB/s	0 Mbps
Task Manager		2,0%	18,3 MB	0,4 MB/s	0 Mbps
Windows Explorer		2,3%	6,9 MB	5,0 MB/s	0 Mbps
Background processes (114)					
Adobe Acrobat Update Service ...		0%	0,1 MB	0 MB/s	0 Mbps
Antimalware Service Executable		0%	17,7 MB	6,1 MB/s	0 Mbps
AppHelperCap.exe		0%	0,1 MB	0 MB/s	0 Mbps

Figure 28: Experiment with 50k dataset

c. Dataset with 1.000.000 rows: The error appears immediately and the execution is aborted.

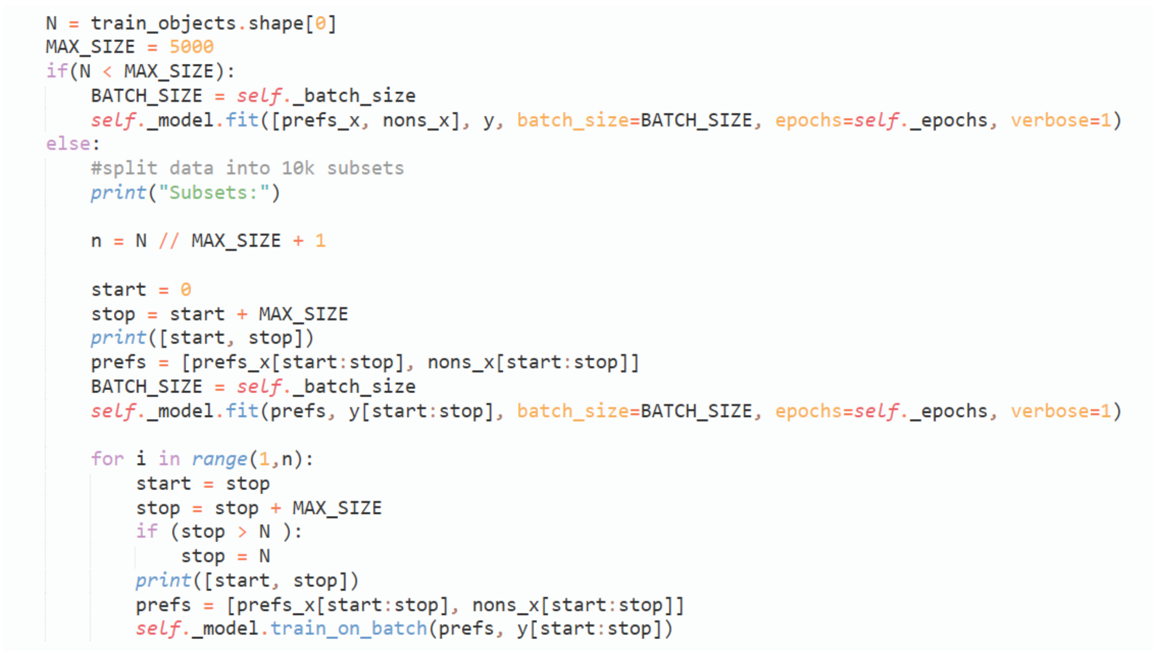
When the amount of memory consumed by all running processes exceeds the amount of RAM available, the operating system places pages from one or more virtual address spaces to the hard disk. These "paged out" pages are stored in one or more files (Pagefile.sys files) in the root of a partition on Windows computers. Each disk partition can have one such file. The page file's location and size are set in System Properties.

From the previous we understand that the page file is the virtual memory on the hard disk. Our personal computer has 8GB of RAM, which seems insufficient, so free space in the virtual memory is allocated. There is a need for continuous data transfer from disk to memory and vice versa and as a result the system almost stops to respond.

The reason seems to be the need to reserve space N^2 (e.g. for a table of similarities or distances between each pair of points). For instance:

For N = 50000, the table has 2,500,000,000 elements. The float type is usually 8 bytes, so the above table needs 20GB of memory. For big data, different types of algorithms or computers with large capabilities are required, e.g. cloud clusters. So, depending on the user's computer memory, there is a limit.

To resolve the memory error in our toolbox, we use the ability of keras to update an existing model using the train_on_batch function. So, if we see that the data is greater than a certain limit, e.g. 5000 samples, we break them into "packages" of 5000, we train the model in the first package and then for each other package we use train_on_batch. A similar logic was already used in the Backpropagation algorithm, so it was added to Ranknet.



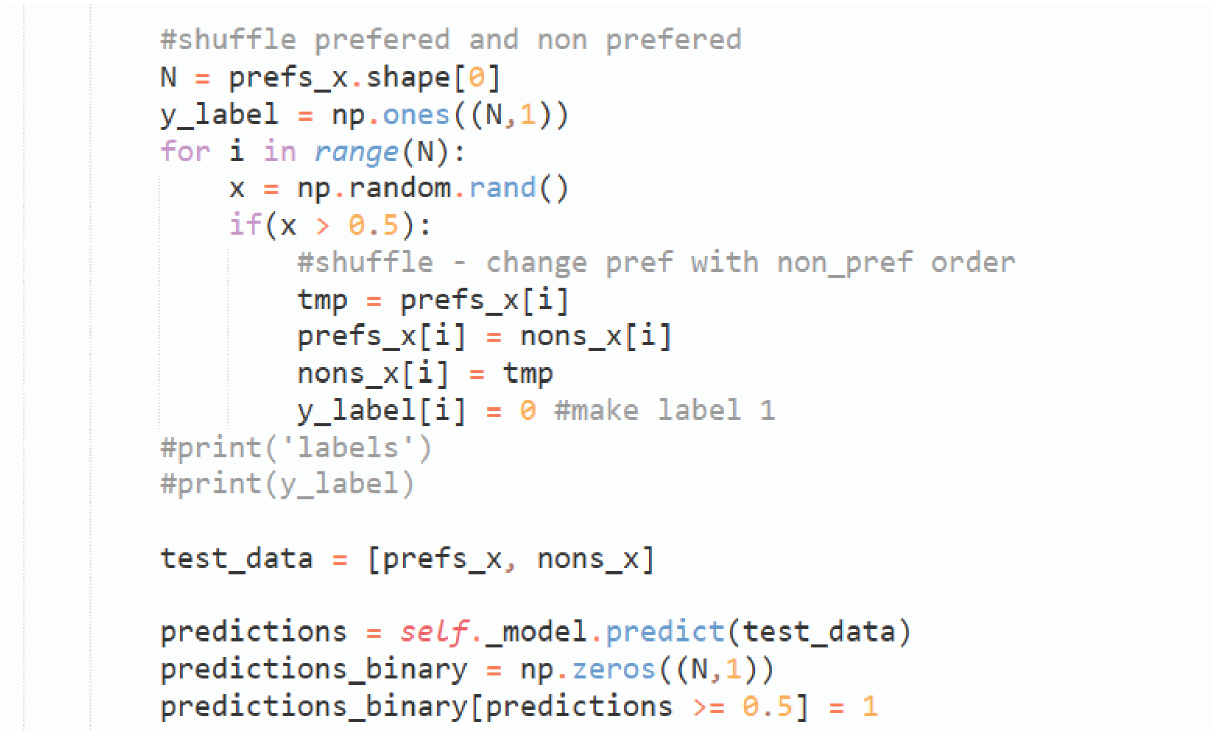
```
N = train_objects.shape[0]
MAX_SIZE = 5000
if(N < MAX_SIZE):
    BATCH_SIZE = self._batch_size
    self._model.fit([prefs_x, nons_x], y, batch_size=BATCH_SIZE, epochs=self._epochs, verbose=1)
else:
    #split data into 10k subsets
    print("Subsets:")

    n = N // MAX_SIZE + 1

    start = 0
    stop = start + MAX_SIZE
    print([start, stop])
    prefs = [prefs_x[start:stop], nons_x[start:stop]]
    BATCH_SIZE = self._batch_size
    self._model.fit(prefs, y[start:stop], batch_size=BATCH_SIZE, epochs=self._epochs, verbose=1)

    for i in range(1,n):
        start = stop
        stop = stop + MAX_SIZE
        if (stop > N ):
            stop = N
        print([start, stop])
        prefs = [prefs_x[start:stop], nons_x[start:stop]]
        self._model.train_on_batch(prefs, y[start:stop])
```

Figure 29: RankNET batch implementation

The tool was tested with a 10k sample file and RankNET algorithm. The execution was successfully completed after 3.5 hours without a memory error.

4.1.3 RankNet problem (sorting)

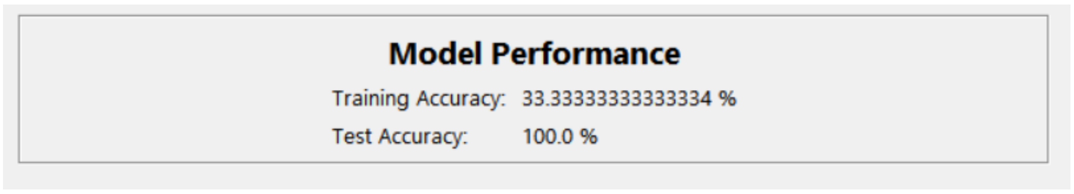
In RankNet, training pairs of data points are sorted into preferred and non-preferred subsets. During the test, the points are also sorted into two lists prefs_x and nons_x, so the following issue occurs. As the network is trained to predict the preferred set, this sorting results in a naive algorithm achieving unreasonably high accuracies. The algorithm achieves great accuracy, simply always predicting "1". In order to solve any problems, we take the two lists and randomly (with a probability of 50%) we exchange the corresponding points, so that in the end the two lists have points from each category. In the test function of ranknet.py the following code is added, so that the two test_data lists are "mixed".



```
#shuffle prefered and non prefered
N = prefs_x.shape[0]
y_label = np.ones((N,1))
for i in range(N):
    x = np.random.rand()
    if(x > 0.5):
        #shuffle - change pref with non_pref order
        tmp = prefs_x[i]
        prefs_x[i] = nons_x[i]
        nons_x[i] = tmp
        y_label[i] = 0 #make label 1
#print('labels')
#print(y_label)

test_data = [prefs_x, nons_x]

predictions = self._model.predict(test_data)
predictions_binary = np.zeros((N,1))
predictions_binary[predictions >= 0.5] = 1
```

Figure 30: Shuffle dataset RankNET

The y_label list has the actual categories: it contains 1 if the first point is preferred, otherwise we set 0 if the second point is preferred. This list will be compared with the prediction to evaluate the performance of the algorithm.

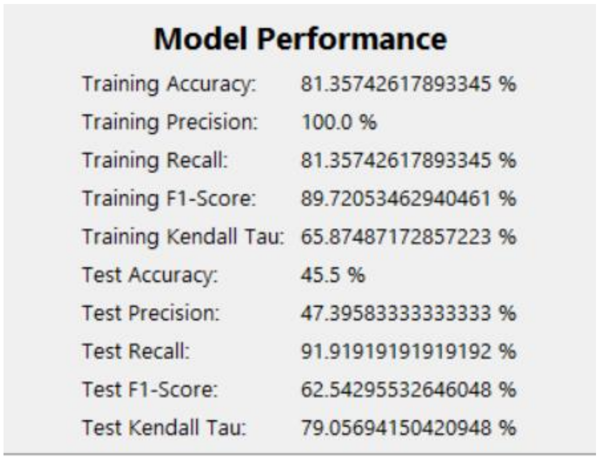


Model Performance

```
Training Accuracy:  33.33333333333334 %
Test Accuracy:      100.0 %
```

Figure 31: Training and test accuracy before exchange

After exchanging the points in order to have mixed points from each category, the test accuracy scores achieved are more reasonable.



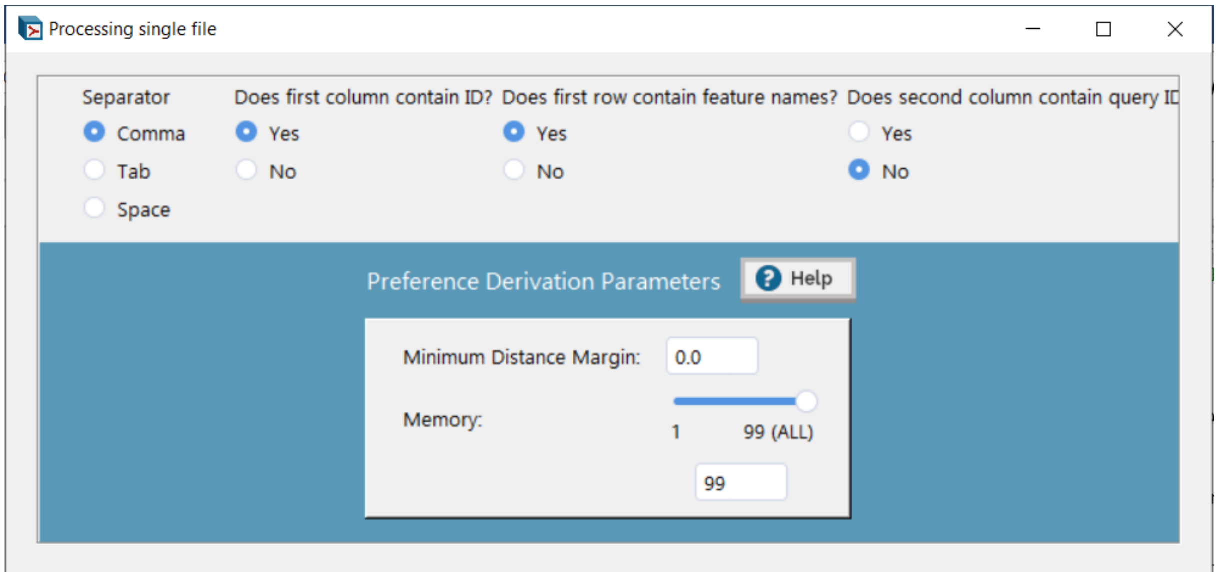
Model Performance

```
Training Accuracy:     81.35742617893345 %
Training Precision:    100.0 %
Training Recall:       81.35742617893345 %
Training F1-Score:     89.72053462940461 %
Training Kendall Tau:  65.87487172857223 %
Test Accuracy:         45.5 %
Test Precision:        47.39583333333333 %
Test Recall:           91.91919191919192 %
Test F1-Score:         62.54295532646048 %
Test Kendall Tau:      79.05694150420948 %
```

Figure 32: Training and test accuracy after exchange

4.1.4 Query ID

In the current implementation, if we want to perform several preprocessing in our data, for example normalization, the whole dataset will be affected. In order to be able to transform a part of our data separately, a query id column has been added. The user is asked whether the data has a Query ID in the second column of the csv input file (Figure 31).


Processing single file

Separator: [✓ Comma] Tab Space

Does first column contain ID? [✓ Yes] No

Does first row contain feature names? [✓ Yes] No

Does second column contain query ID: Yes [✓ No]

Minimum Distance Margin: 0.0

Memory: 1 99 (ALL) 99

Figure 33: Query ID

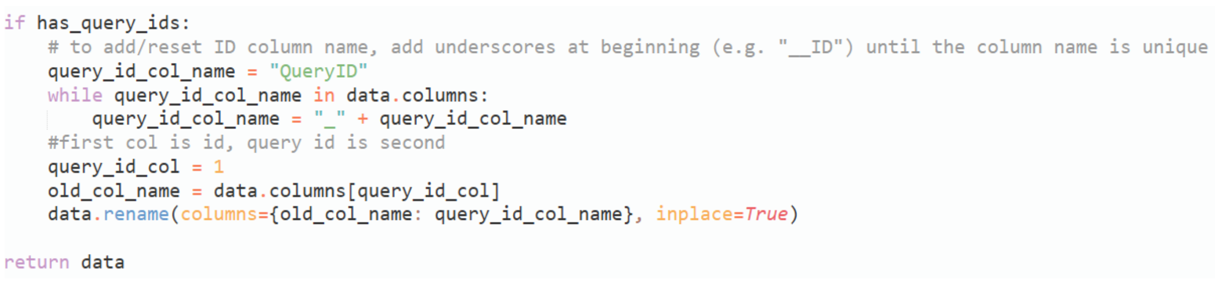
The previous was implemented as following (Figure 32). In params.py file, the variable self._has_query_id was added, with initial value False.

```
self._has_query_id = tk.BooleanVar()
self._has_query_id.set(False)  # false by default
```

```
# Check Query ID
has_qid_label = tk.Label(other_frame3, text="Does second column contain query ID?")
has_qid_label.grid(row=0, column=0, sticky='nw')
has_qid_opt1 = ttk.Radiobutton(other_frame3, variable=self._has_query_id, text="Yes", value=True,
                               command=self._update_data, style='PLT.TRadiobutton')
has_qid_opt1.grid(row=1, column=0, sticky='nw')
has_qid_opt2 = ttk.Radiobutton(other_frame3, variable=self._has_query_id, text="No", value=False,
                               command=self._update_data, style='PLT.TRadiobutton')
has_qid_opt2.grid(row=2, column=0, sticky='nw')
```

Figure 34: Query ID implementation

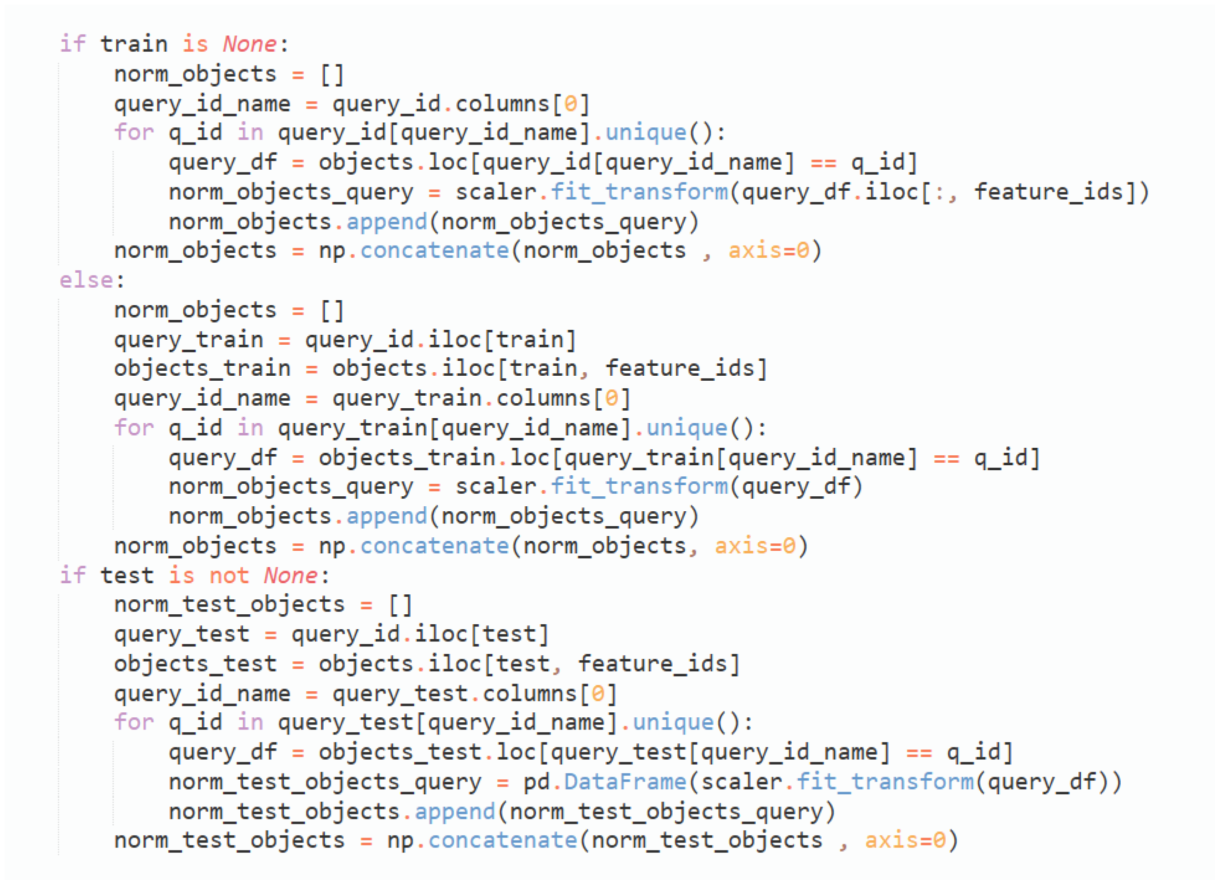
Then, in the experiment.py file, in the _load_data function the following lines were added to the code (Figure 33), which rename the second column to QueryID if the user has previously selected Yes in the corresponding query. Note that the first column is always the ID, which either comes from the data or if the file has no IDs, is automatically created by _load_data.



```
if has_query_ids:
    # to add/reset ID column name, add underscores at beginning (e.g. "__ID") until the column name is unique
    query_id_col_name = "QueryID"
    while query_id_col_name in data.columns:
        query_id_col_name = "_" + query_id_col_name
    #first col is id, query id is second
    query_id_col = 1
    old_col_name = data.columns[query_id_col]
    data.rename(columns={old_col_name: query_id_col_name}, inplace=True)

return data
```

Figure 35: Rename query ID column in load_data function

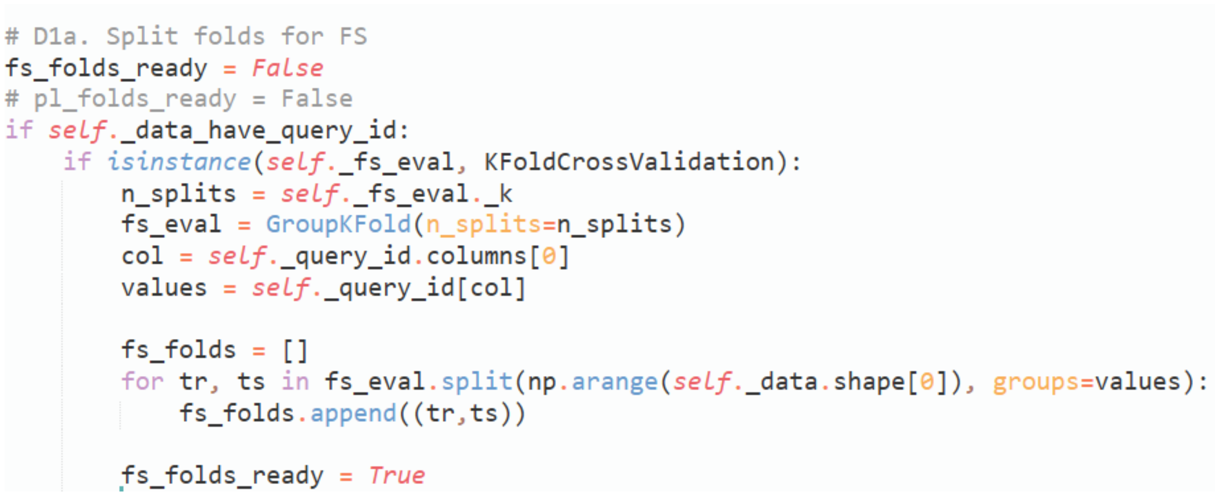
The queryID column is then used to normalize and split the data into train and test subsets. For normalization (_normalize function in experiment.py file), this is applied per query ID, for example as follows (Figure 34):



```
if train is None:
    norm_objects = []
    query_id_name = query_id.columns[0]
    for q_id in query_id[query_id_name].unique():
        query_df = objects.loc[query_id[query_id_name] == q_id]
        norm_objects_query = scaler.fit_transform(query_df.iloc[:, feature_ids])
        norm_objects.append(norm_objects_query)
    norm_objects = np.concatenate(norm_objects , axis=0)
else:
    norm_objects = []
    query_train = query_id.iloc[train]
    objects_train = objects.iloc[train, feature_ids]
    query_id_name = query_train.columns[0]
    for q_id in query_train[query_id_name].unique():
        query_df = objects_train.loc[query_train[query_id_name] == q_id]
        norm_objects_query = scaler.fit_transform(query_df)
        norm_objects.append(norm_objects_query)
    norm_objects = np.concatenate(norm_objects, axis=0)
if test is not None:
    norm_test_objects = []
    query_test = query_id.iloc[test]
    objects_test = objects.iloc[test, feature_ids]
    query_id_name = query_test.columns[0]
    for q_id in query_test[query_id_name].unique():
        query_df = objects_test.loc[query_test[query_id_name] == q_id]
        norm_test_objects_query = pd.DataFrame(scaler.fit_transform(query_df))
        norm_test_objects.append(norm_test_objects_query)
    norm_test_objects = np.concatenate(norm_test_objects , axis=0)
```

Figure 36: Feature normalization with query ids

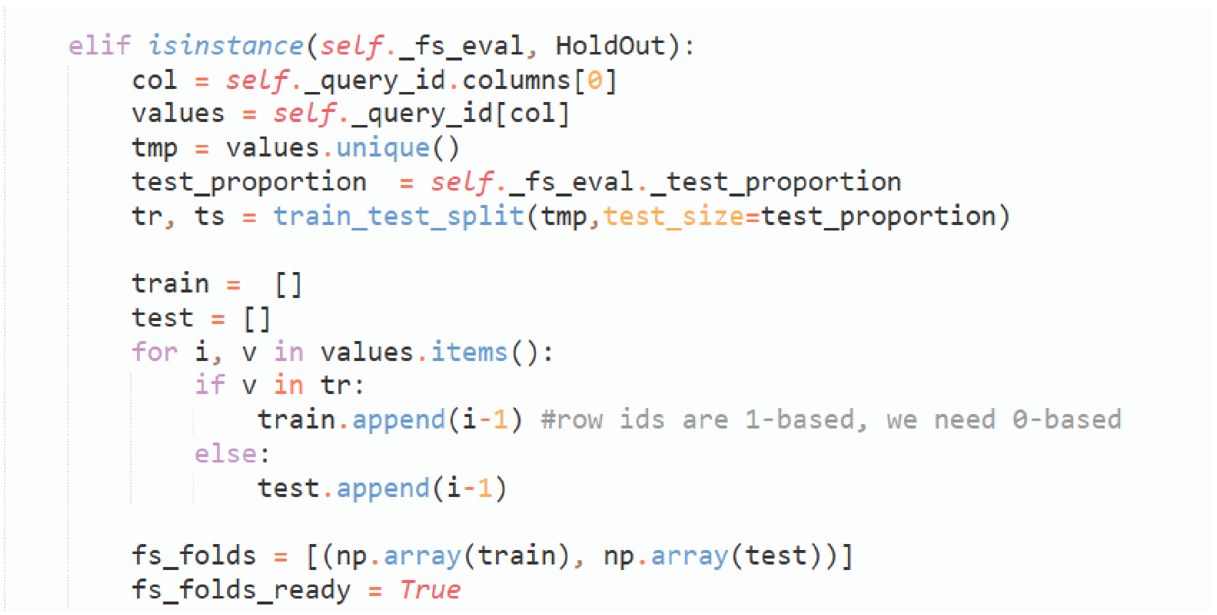
To split the data into train-test, for the K-fold method, we take the variable _k which indicates the number of folds from the KFoldCrossValidation class and we use the GroupKFold function provided in the scikit-learn package which performs the desired function (Figure 35):



```
# D1a. Split folds for FS
fs_folds_ready = False
# pl_folds_ready = False
if self._data_have_query_id:
    if isinstance(self._fs_eval, KFoldCrossValidation):
        n_splits = self._fs_eval._k
        fs_eval = GroupKFold(n_splits=n_splits)
        col = self._query_id.columns[0]
        values = self._query_id[col]

        fs_folds = []
        for tr, ts in fs_eval.split(np.arange(self._data.shape[0]), groups=values):
            fs_folds.append((tr,ts))

        fs_folds_ready = True
```

Figure 37: KFoldCrossValidation with query ids

In the case of HoldOut, we take the variable _test_proportion declared by the user and use the train_test_split function of the scikit-learn package. Specifically, we apply it to the unique query ids, so that the whole id is either in the train or in the test subset. Based on the unique ids, we distribute each line of data in the corresponding subset:

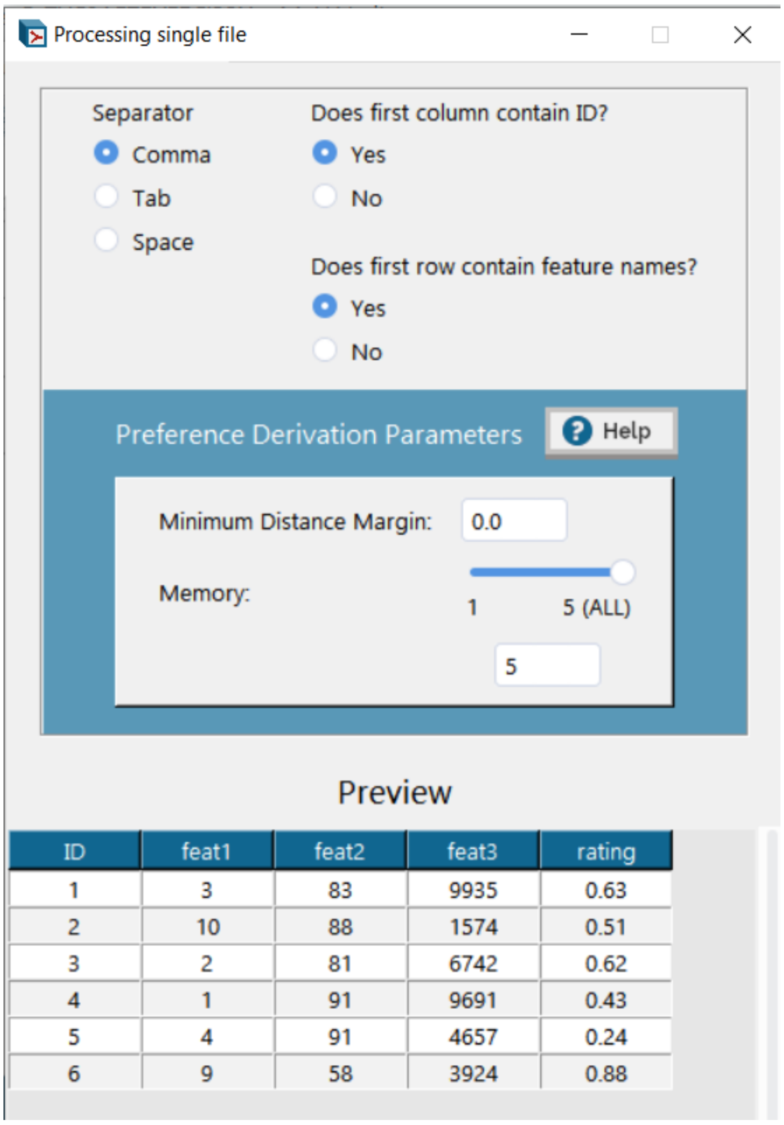


```
    elif isinstance(self._fs_eval, HoldOut):
        col = self._query_id.columns[0]
        values = self._query_id[col]
        tmp = values.unique()
        test_proportion  = self._fs_eval._test_proportion
        tr, ts = train_test_split(tmp,test_size=test_proportion)

        train =  []
        test = []
        for i, v in values.items():
            if v in tr:
                train.append(i-1) #row ids are 1-based, we need 0-based
            else:
                test.append(i-1)

        fs_folds = [(np.array(train), np.array(test))]
        fs_folds_ready = True
```

Figure 38: HoldOut Validation with query ids

The above was implemented both for the separation during the feature selection as well as during the creation of the machine learning models.

As mentioned before, prior to the addition of the query ID column in the dataset, the user could not perform a pre-processing on a proportion of data with some common characteristic. For example, for a dataset containing data from multiple participants, it wasn't possible to normalize the values of a feature per participant. Below is a preview of the fully loaded dataset (Figure 39) and the options for pre-processing (Figure 40). The user can choose whether to perform min_max or z_score normalization only per feature.


Processing single file

Separator: [✓ Comma] Tab Space

Does first column contain ID? [✓ Yes] No

Does first row contain feature names? [✓ Yes] No

Preference Derivation Parameters — Help

Minimum Distance Margin: 0.0

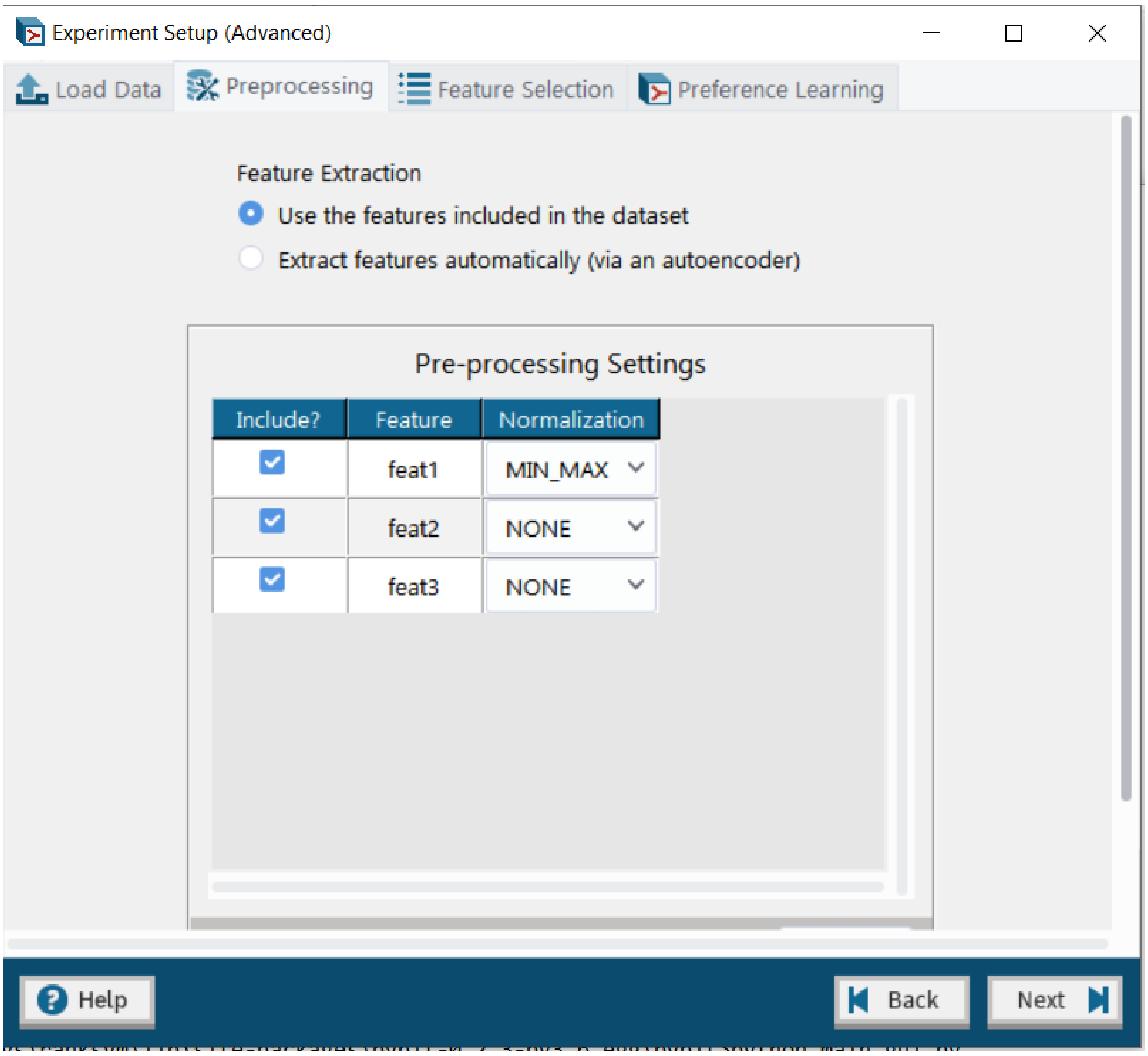
Memory: 1 — 5 (ALL): 5

Preview

ID	feat1	feat2	feat3	rating
1	3	83	9935	0.63
2	10	88	1574	0.51
3	2	81	6742	0.62
4	1	91	9691	0.43
5	4	91	4657	0.24
6	9	58	3924	0.88

Figure 39: Preview of the fully loaded dataset

Let's perform min_max normalization for feature 1 (Figure 40):



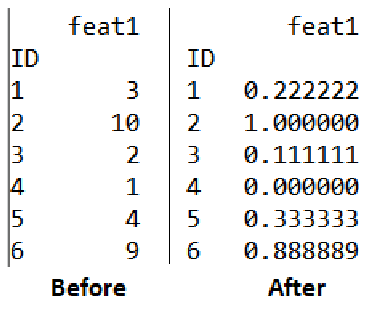
Feature Extraction
- [✓ Use the features included in the dataset]
- Extract features automatically (via an autoencoder)

Pre-processing Settings

Include?	Feature	Normalization
✓	feat1	MIN_MAX
✓	feat2	NONE
✓	feat3	NONE

Figure 40: Pre-processing settings

As we can see below (Figure 41) all the values of feature 1 are normalized regardless of the participant:

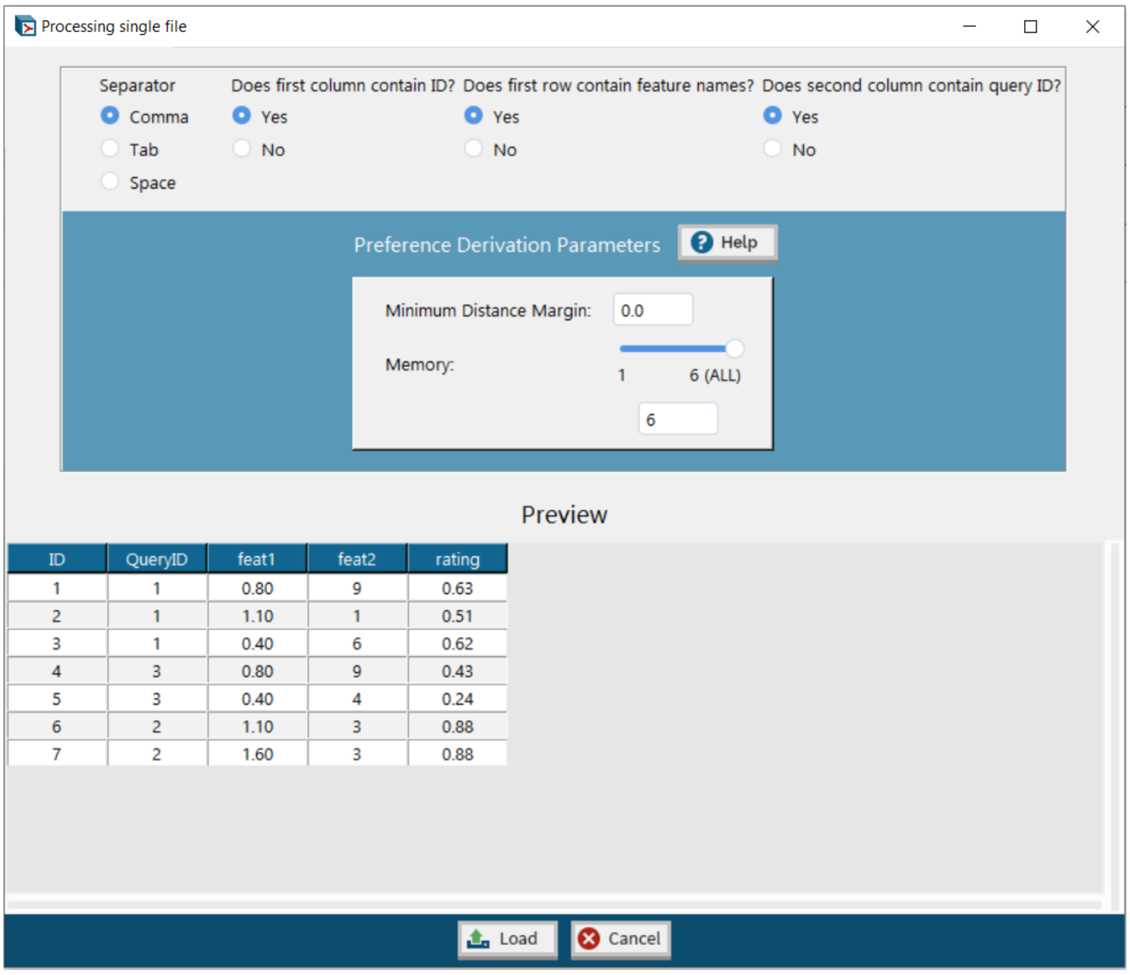


```
     feat1 | ID     feat1
ID         | 1   0.222222
1        3 | 2   1.000000
2       10 | 3   0.111111
3        2 | 4   0.000000
4        1 | 5   0.333333
5        4 | 6   0.888889
6        9 |
```

Before **After**

Figure 41: Feature 1 normalization

By adding a query_id column in the dataset (Figure 42), which contains a unique number for every participant's data the user can perform normalization per query_id which means per participant (Figures 43 and 44).



Processing single file

Separator: [✓ Comma] Tab Space

Does first column contain ID? [✓ Yes] No

Does first row contain feature names? [✓ Yes] No

Does second column contain query ID? [✓ Yes] No

Preference Derivation Parameters

Minimum Distance Margin: 0.0

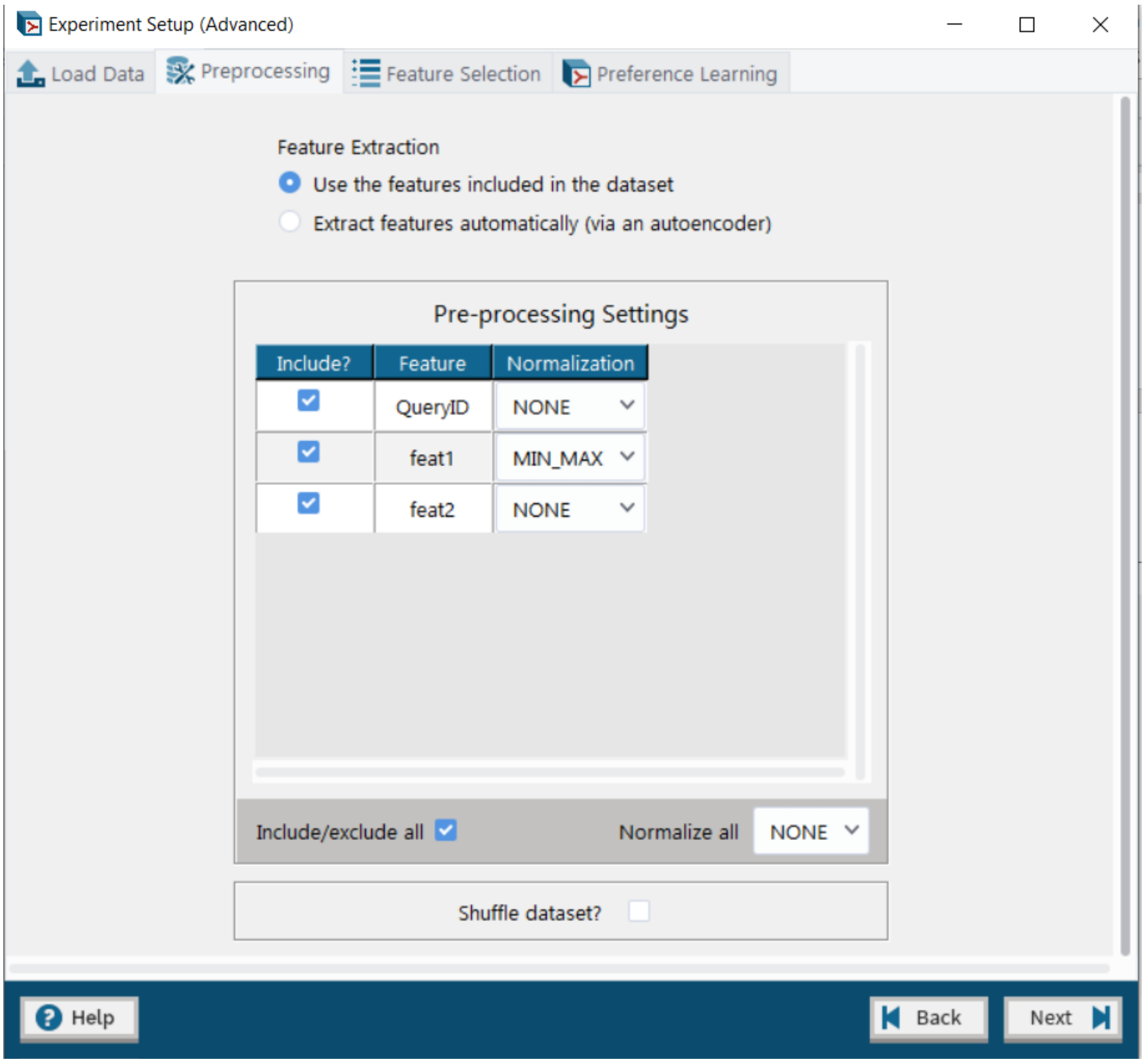
Memory: 1 — 6 (ALL): 6

Preview

ID	QueryID	feat1	feat2	rating
1	1	0.80	9	0.63
2	1	1.10	1	0.51
3	1	0.40	6	0.62
4	3	0.80	9	0.43
5	3	0.40	4	0.24
6	2	1.10	3	0.88
7	2	1.60	3	0.88

Figure 42: Dataset including a query_id column

For example if we select min_max normalization for feature 1 (Figure 43), we observe that it is performed per query_id and not for all the values of the feature (Figure 44).


Experiment Setup (Advanced)

Load Data | Preprocessing | Feature Selection | Preference Learning

Feature Extraction

● Use the features included in the dataset

○ Extract features automatically (via an autoencoder)

Pre-processing Settings

Include?	Feature	Normalization
✓	QueryID	NONE
✓	feat1	MIN_MAX
✓	feat2	NONE

Include/exclude all ✓ Normalize all NONE

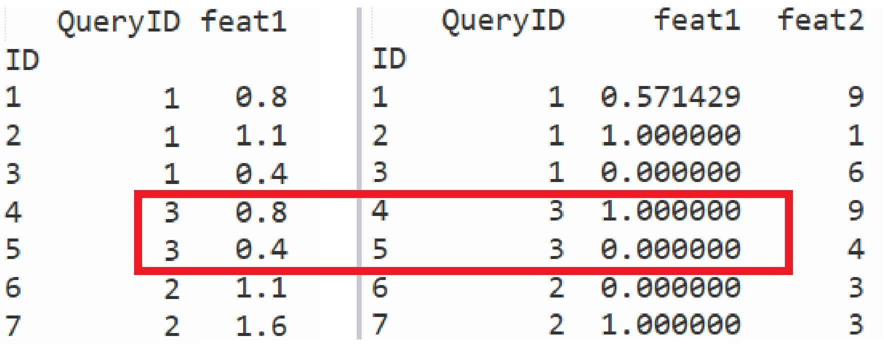
Shuffle dataset? ☐

Help Back Next

Figure 43: Min_max normalization for feature 1 per query_id

More specifically, let’s observe the procedure of min_max normalization for the feature 1, for every participant. As mentioned before, the Min-Max method transforms the values of the given feature to fit the range of 0 to 1. For the participant with the query_id equal to 3, the values of 0.8 and 0.4 were normalized to 1 and 0 respectively (Figure 44).

For the participant with the query_id equal to 1, the values of 0.8, 1.1 and 0.4 were transformed to 0.57 , 1 and 0 respectively. As we can notice, the procedure is performed per query_id without considering the values of the others.



```
    QueryID feat1
ID
1         1   0.8
2         1   1.1
3         1   0.4
4         3   0.8
5         3   0.4
6         2   1.1
7         2   1.6
```

```
     QueryID     feat1  feat2
ID
1          1  0.571429      9
2          1  1.000000      1
3          1  0.000000      6
4          3  1.000000      9
5          3  0.000000      4
6          2  0.000000      3
7          2  1.000000      3
```

Figure 44: Normalization of feature 1 per query_id

5

Augmenting the functionality of pyPLT

5.1 *Estimator’s performance metrics*

Cross-validation is a way to evaluate the performance of an estimator. A model must be capable of making useful predictions on yet-unseen data, not just repeating the labels of the samples it has already seen. Thus, we split our dataset in two parts, in order to train and evaluate our classification predictive model.

A very useful tool for predictive modeling is a Python package called scikit-learn. Scikit-learn comes with a number of built-in functions for assessing model performance. For all the algorithms, in the function calc_train_accuracy and test, apart from the metric accuracy which was the only performance measure in pyplt, we added the calculation of the metric precision, recall and F1 score, based on the according functions from the package.

We load a dataset with actual labels and the prediction probabilities for the model. So the function inputs are two 1xdimensional arrays:

y_label: which contains the ground truth (correct) target values and

y_pred: which contains the target values returned by the classifier

The (default) return value is a floating type score depending on the metric.

Typically, a threshold is established to determine whether prediction probabilities are designated as predicted positive or negative. Let's say the cutoff point is 0.5. In its most basic form, we can divide our samples into four groups based on a real label and a predicted: False Positive, True Negative, True Positive, False Negative. For example a value of 1 for which the model predicts 1 is a True Positive.

	Actual	Probability	Predicted
True Positive	1	0.68	1
False Positive	0	0.52	1
True Negative	0	0.23	0
False Negative	1	0.48	0

Figure 45: Categories of predicted labels

Below is another form of display for representing the four categories:

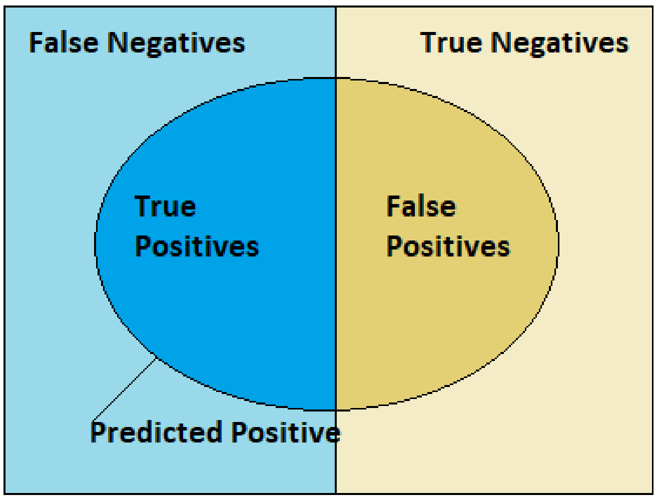


Figure 46: Display of predicted label categories

This display is really useful for understanding the calculation of the below measures: Accuracy, Precision, Recall, F1 score which are further described below.

5.1.1 Accuracy score

The function computes subset accuracy in multilabel classification: the set of labels predicted for a sample (y_pred) must exactly match the corresponding set of labels in y_label. The return value is the fraction of correctly classified samples which is an outcome of the below:

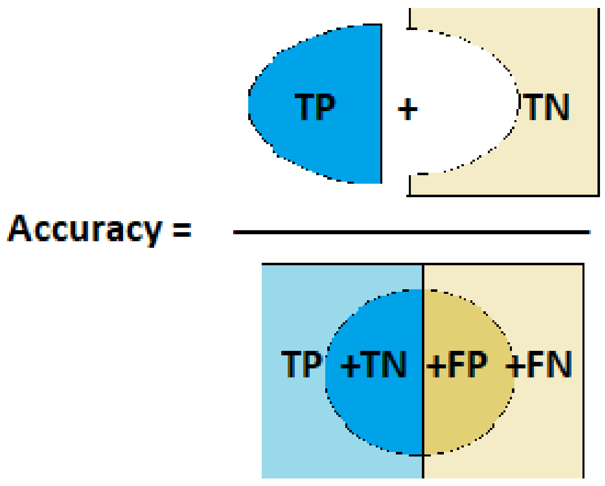


Figure 47: Accuracy score formula

where:

TP: True Positive

TN: True Negative

FP: False Positive

FN: False Negative

The best accuracy score is 1 equal to 100%, where every prediction is correct. Though, we cannot consider accuracy as the absolute metric. If the model tries to forecast something that only happens once in a hundred times, the accuracy is 99 percent, implying that the event never occurred. So we need an additional performance metric in order to capture all the events we care about, known as recall or sensitivity.

5.1.2 Recall

The classifier's recall refers to its capacity to locate all positive samples. The return value is the fraction of correctly classified positive samples which is an outcome of the below:

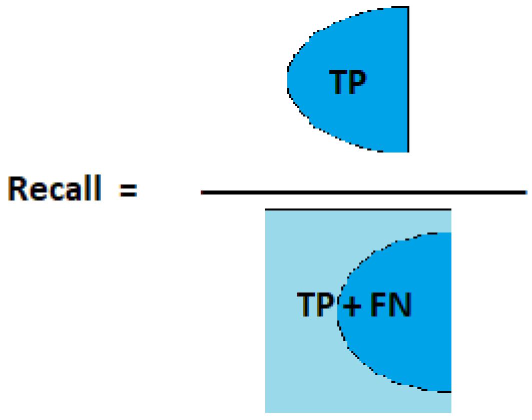


Figure 48: Recall score formula

An easy way to increase the recall score is to lower the threshold for positive predicted. This tactic results in an increase of the number of false positives, so another performance metric appears in order to solve this.

5.1.3 Precision

Precision refers to the classifier's ability to avoid classifying a negative sample as positive. As demonstrated below, the return value is the fraction of predicted positive events that really happen to be positive: In binary classification the precision of the positive class is calculated, while in multiclass the weighted average of each class is measured.

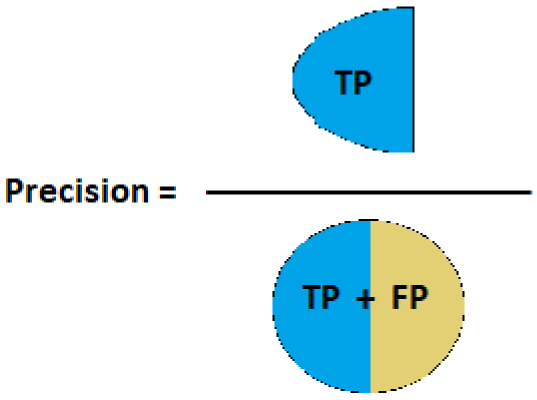


Figure 49: Precision score formula

Comparing two models, if the first is better at both recall and precision we are more confident to choose it. However, if the first model scores better at recall and the second is better at precision it gets complicated. In order to ease our decision we use another metric called F1 score.

5.1.4 F1 score

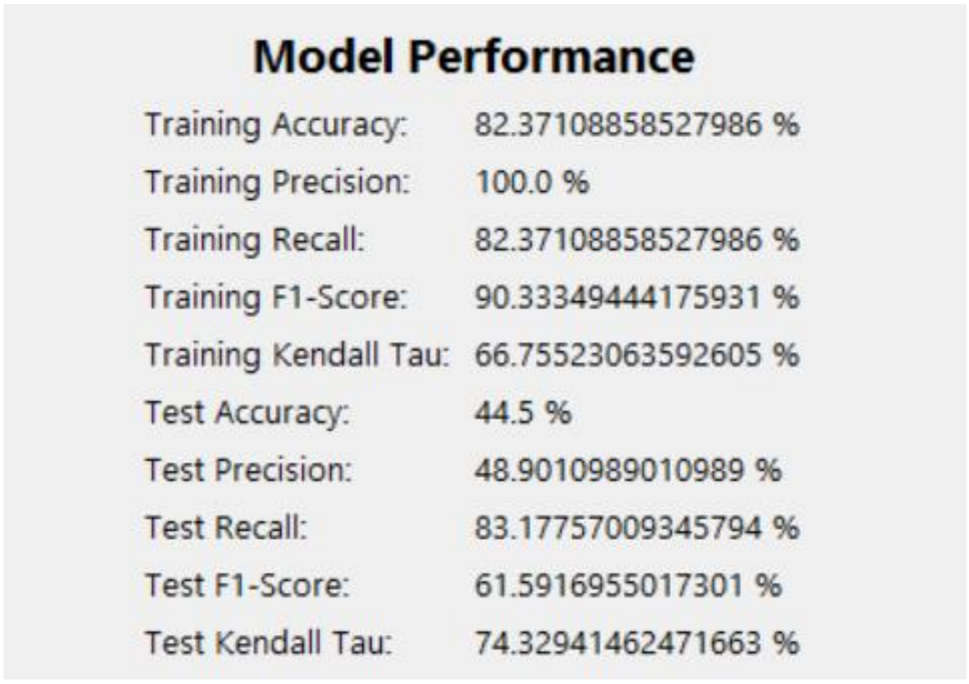
F1 score is also known as balanced F-score or F-measure. This metric combines the precision, recall measures and it can be described as a weighted average of them. In the multi-class and multi-label case, it is the average of the F1 score of each class with weighting depending on the average parameter.

The following formula is used to determine the f1 score:

F1 = 2 * (precision * recall) / (precision + recall)

We observe that each performance metric has its own set of advantages and drawbacks. A variety of metrics exist which can be used to help us decide which is the most appropriate for our specific case. Furthermore, it is possible to observe all of the available measures in order to choose the best model.

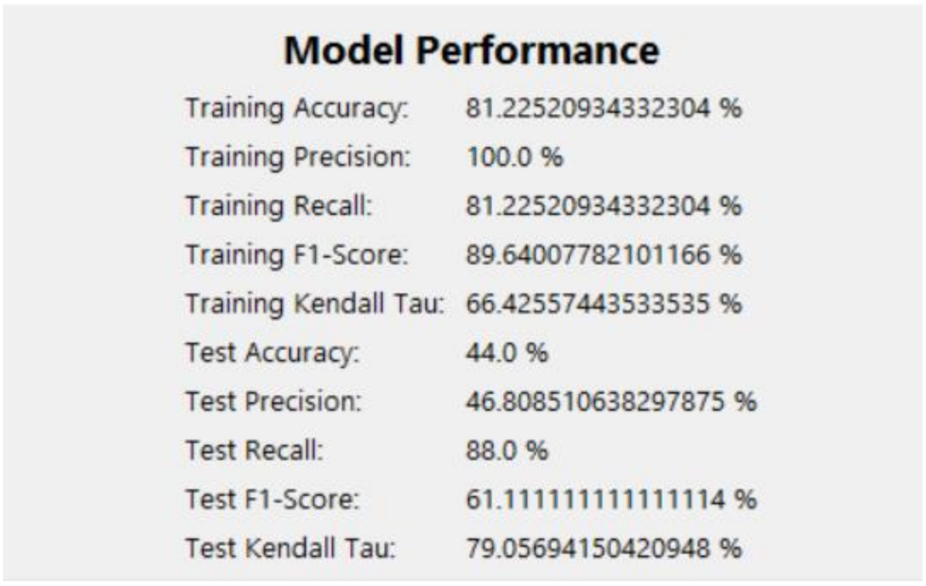
In figures 50 and 51 we can see an example of the calculated performance metrics for two models of ranknet algorithm:



Model Performance

Training Accuracy:	82.37108858527986 %
Training Precision:	100.0 %
Training Recall:	82.37108858527986 %
Training F1-Score:	90.33349444175931 %
Training Kendall Tau:	66.75523063592605 %
Test Accuracy:	44.5 %
Test Precision:	48.9010989010989 %
Test Recall:	83.17757009345794 %
Test F1-Score:	61.5916955017301 %
Test Kendall Tau:	74.32941462471663 %

Figure 50: Performance metrics model1



Model Performance

Training Accuracy:	81.22520934332304 %
Training Precision:	100.0 %
Training Recall:	81.22520934332304 %
Training F1-Score:	89.64007782101166 %
Training Kendall Tau:	66.42557443533535 %
Test Accuracy:	44.0 %
Test Precision:	46.808510638297875 %
Test Recall:	88.0 %
Test F1-Score:	61.111111111111114 %
Test Kendall Tau:	79.05694150420948 %

Figure 51: Performance metrics model2

5.1.5 Kendall’s tau to a reconstructed ground truth score

Kendall’s tau is a measure of the correspondence between two rankings. The scipy.stats kendalltau function has been utilized, which includes the implementation of two variants of Kendall’s tau (default Tau-b and Stuart’s Tau-c). They mainly differ in how they are normalized within the range -1 to 1.

- Tau-c: preferable for the analysis of data based on non-square (i.e. rectangular) contingency tables
- Tau-b: appropriate if both variables’ scales have the same number of possible values before ranking
- Tau-a: does not account for ties in any way. In the absence of ties, tau-b and tau-c both descend to tau-a.

As ties, we describe certain objects that we are incapable of distinguishing a clear preference between them, thus we consider them as tying and regard them as equal.

The function takes two arguments, x, y, which are vectors of size N and show the ranking of the N elements. It returns a correlation float which is the tau statistic and a pvalue float which is the value for a test whose null hypothesis is an absence of association. The result is a factor between -1 and 1, depending on the level of association. 1 implies 100% negative association, while 0 indicates no relationship and a value of +1 shows full agreement.

The following is the definition of Kendall's tau as used in scipy.stats:

tau_b = (P - Q) / sqrt((P + Q + T) * (P + Q + U))

tau_c = 2 (P - Q) / (n**2 * (m - 1) / m)

where

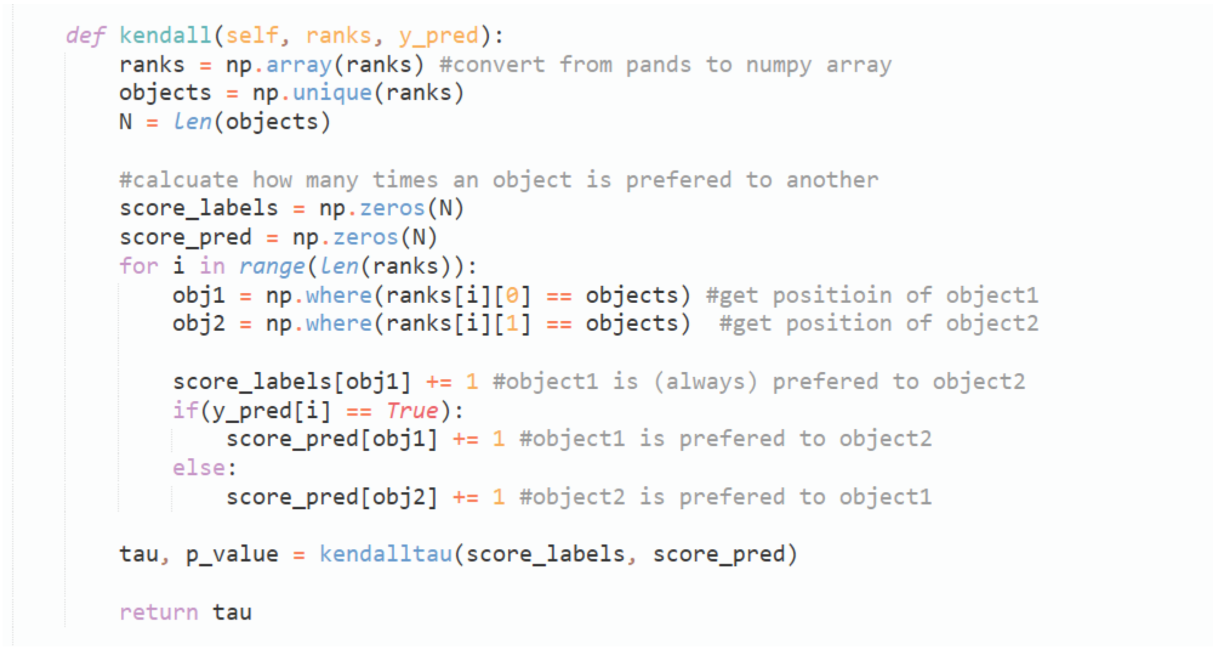
P,Q: number of concordant/discordant pairs,

T,U: number of ties only in x/y. If a tie occurs for the same pair in both x and y, it is excluded from both T and U.

n: total number of samples

m is the number of distinct values in x or y, whichever is smaller.

In train, test functions we have the actual rankings as a list of point pairs showing that the former is preferred to the latter. We also have the result as a logical variable, if the first point is preferred to the second. Based on these, we define the following function (Figure 52)



```python
def kendall(self, ranks, y_pred):
    ranks = np.array(ranks) #convert from pands to numpy array
    objects = np.unique(ranks)
    N = len(objects)

    #calcuate how many times an object is prefered to another
    score_labels = np.zeros(N)
    score_pred = np.zeros(N)
    for i in range(len(ranks)):
        obj1 = np.where(ranks[i][0] == objects) #get positioin of object1
        obj2 = np.where(ranks[i][1] == objects)  #get position of object2

        score_labels[obj1] += 1 #object1 is (always) prefered to object2
        if(y_pred[i] == True):
            score_pred[obj1] += 1 #object1 is prefered to object2
        else:
            score_pred[obj2] += 1 #object2 is prefered to object1

    tau, p_value = kendalltau(score_labels, score_pred)

    return tau
```

Figure 52: Kendall_tau implementation

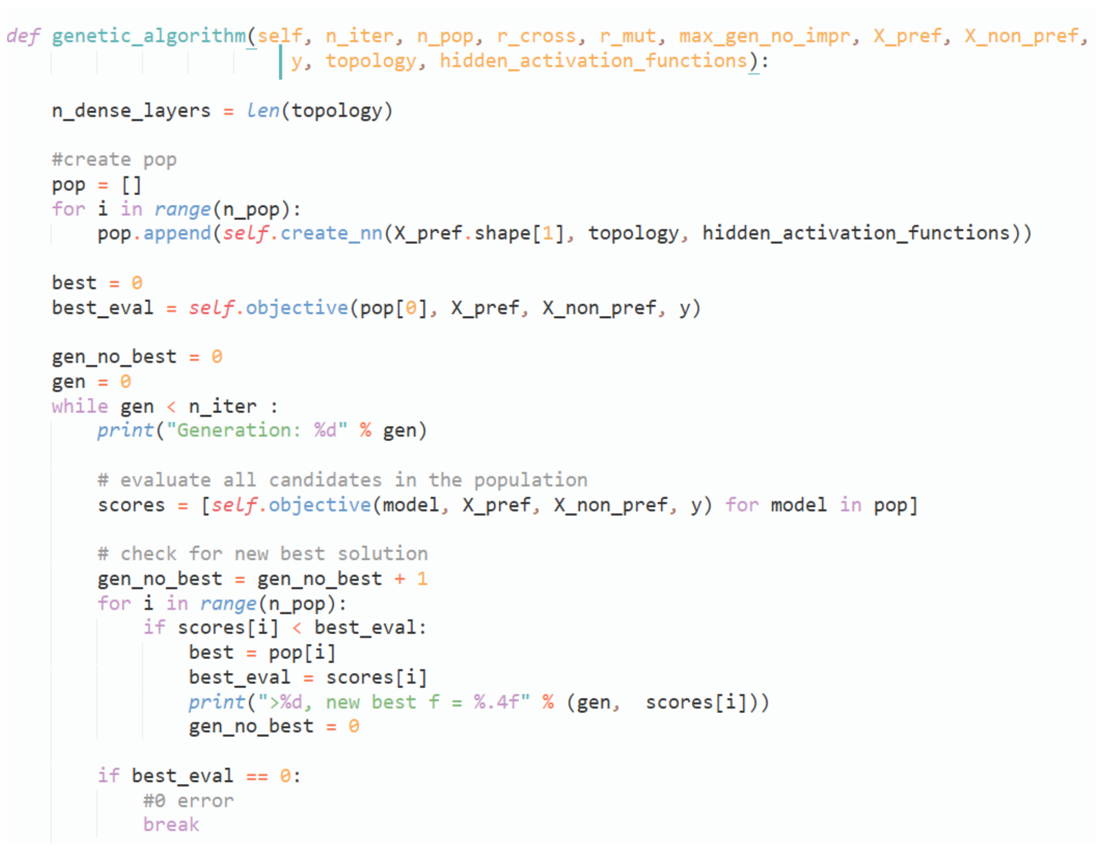
In fact, we create two score variables, one for the actual ranking and one for the prediction, which show for each point how many times it is preferred over the other points. Kendall function returns the tau statistic, which is calculated by kendalltau function. All the evaluation metrics are presented on a scale of 100% (Figures 50, 51).

5.2 *NeuroEvolution Algorithm*

The main idea is that instead of applying an algorithm for neural network training, that is, for calculating parameters such as backpropagation, a genetic algorithm will be used. As a fitness function to be minimized, we define the error of the model during the prediction.

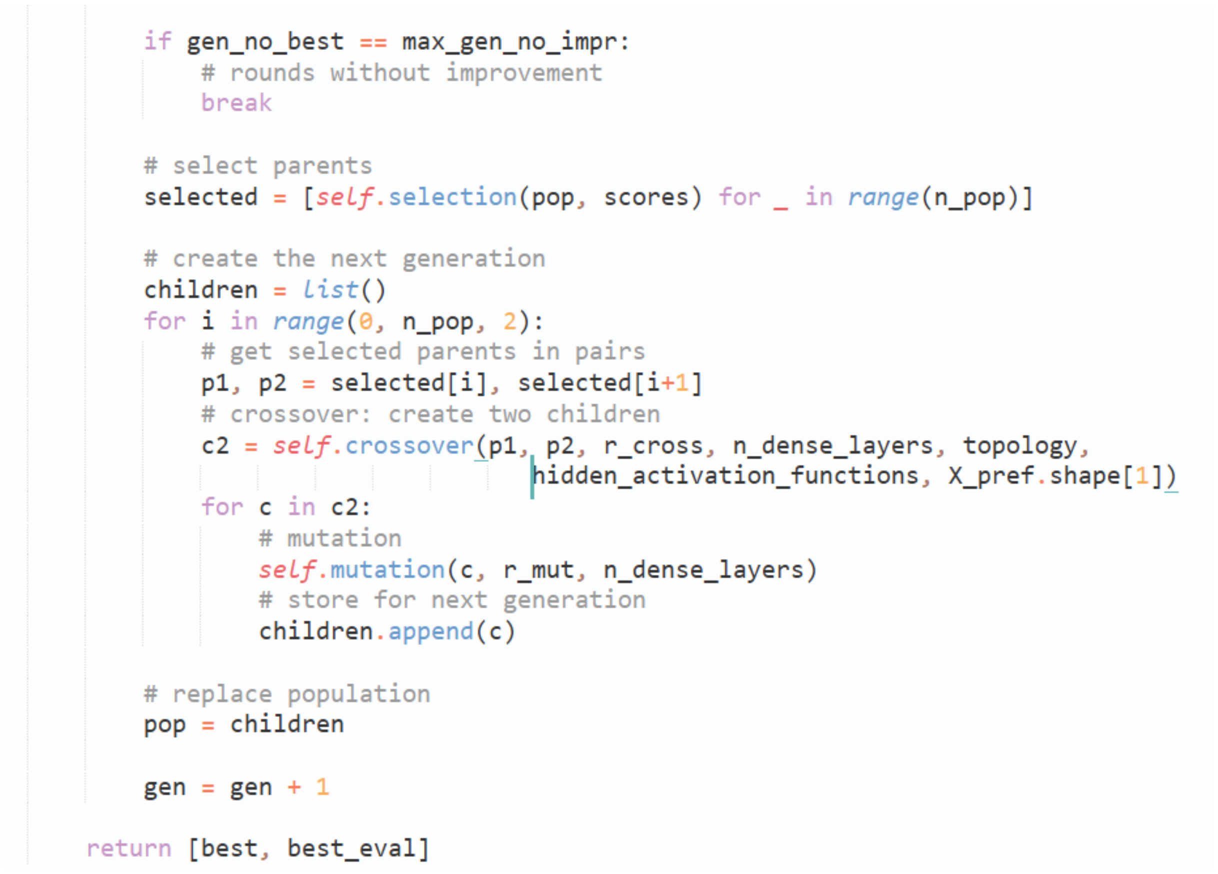
At first we create a population of neural networks with the same topology, with the only difference being the random weights of the hidden levels. Then, we apply the operators of the genetic algorithm (selection, crossover, mutation) and create new generations of neural networks. The algorithm terminates when we have an error of 0, or when the maximum number of generations is reached or when the objective function for a number of generations is not improved (defined by the user). In order to apply the operators, we combine all the parameters (weights and biases) into one vector. After the operators are applied and possibly changes are made, we assign the values from the vector to the corresponding parameters of the network.

The genetic algorithm's basic framework is illustrated in Figure 53. At first, a generation of neural networks is created. Then, in an iteration loop, we evaluate the candidates in the population and select the best individuals for reproduction with the tournament method (we compare each individual with two random and select the best) (Figure 56).



```python
def genetic_algorithm(self, n_iter, n_pop, r_cross, r_mut, max_gen_no_impr, X_pref, X_non_pref,
                      y, topology, hidden_activation_functions):

    n_dense_layers = len(topology)

    #create pop
    pop = []
    for i in range(n_pop):
        pop.append(self.create_nn(X_pref.shape[1], topology, hidden_activation_functions))

    best = 0
    best_eval = self.objective(pop[0], X_pref, X_non_pref, y)

    gen_no_best = 0
    gen = 0
    while gen < n_iter :
        print("Generation: %d" % gen)

        # evaluate all candidates in the population
        scores = [self.objective(model, X_pref, X_non_pref, y) for model in pop]

        # check for new best solution
        gen_no_best = gen_no_best + 1
        for i in range(n_pop):
            if scores[i] < best_eval:
                best = pop[i]
                best_eval = scores[i]
                print(">%d, new best f = %.4f" % (gen,  scores[i]))
                gen_no_best = 0

        if best_eval == 0:
            #0 error
            break
```

Figure 53: Generation creation & population evaluation



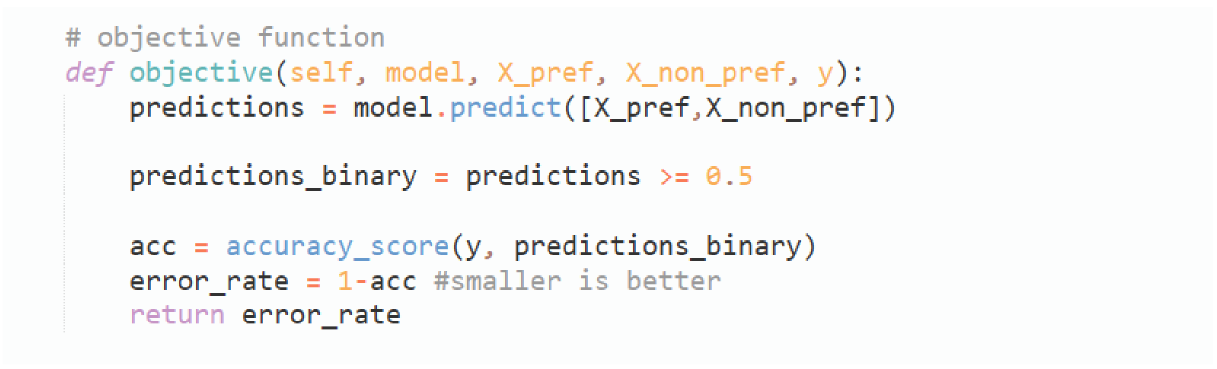
```
        if gen_no_best == max_gen_no_impr:
            # rounds without improvement
            break

        # select parents
        selected = [self.selection(pop, scores) for _ in range(n_pop)]

        # create the next generation
        children = list()
        for i in range(0, n_pop, 2):
            # get selected parents in pairs
            p1, p2 = selected[i], selected[i+1]
            # crossover: create two children
            c2 = self.crossover(p1, p2, r_cross, n_dense_layers, topology,
                                hidden_activation_functions, X_pref.shape[1])
            for c in c2:
                # mutation
                self.mutation(c, r_mut, n_dense_layers)
                # store for next generation
                children.append(c)

        # replace population
        pop = children

        gen = gen + 1

    return [best, best_eval]
```

Figure 54: Creation of the next generation

For every two people as parents we create two children, who are copies of them. With an r_cross probability we will have a parent crossover (ie we will take half the vector from one and half from the other, where the intersection point is chosen randomly). Also, with a probability of r_mut, a weight can mutate, ie be replaced by a random number. Children are the new population for the next generation (Figure 54). If there is no improvement for a number of generations, the iteration is stopped.

The objective function calculates the error rate of a model of the population (Figure 55).

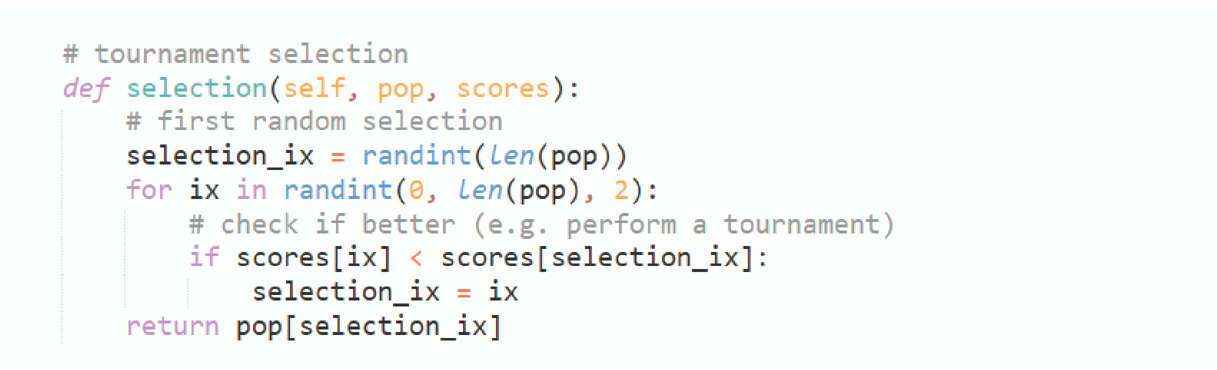
A prediction is performed for every pair of objects and is transformed to a binary value using a threshold of 0,5 . The error rate is equal to (1-accuracy score), thus a model with a smaller error is better.



```python
# objective function
def objective(self, model, X_pref, X_non_pref, y):
    predictions = model.predict([X_pref,X_non_pref])

    predictions_binary = predictions >= 0.5

    acc = accuracy_score(y, predictions_binary)
    error_rate = 1-acc #smaller is better
    return error_rate
```

Figure 55: Objective function

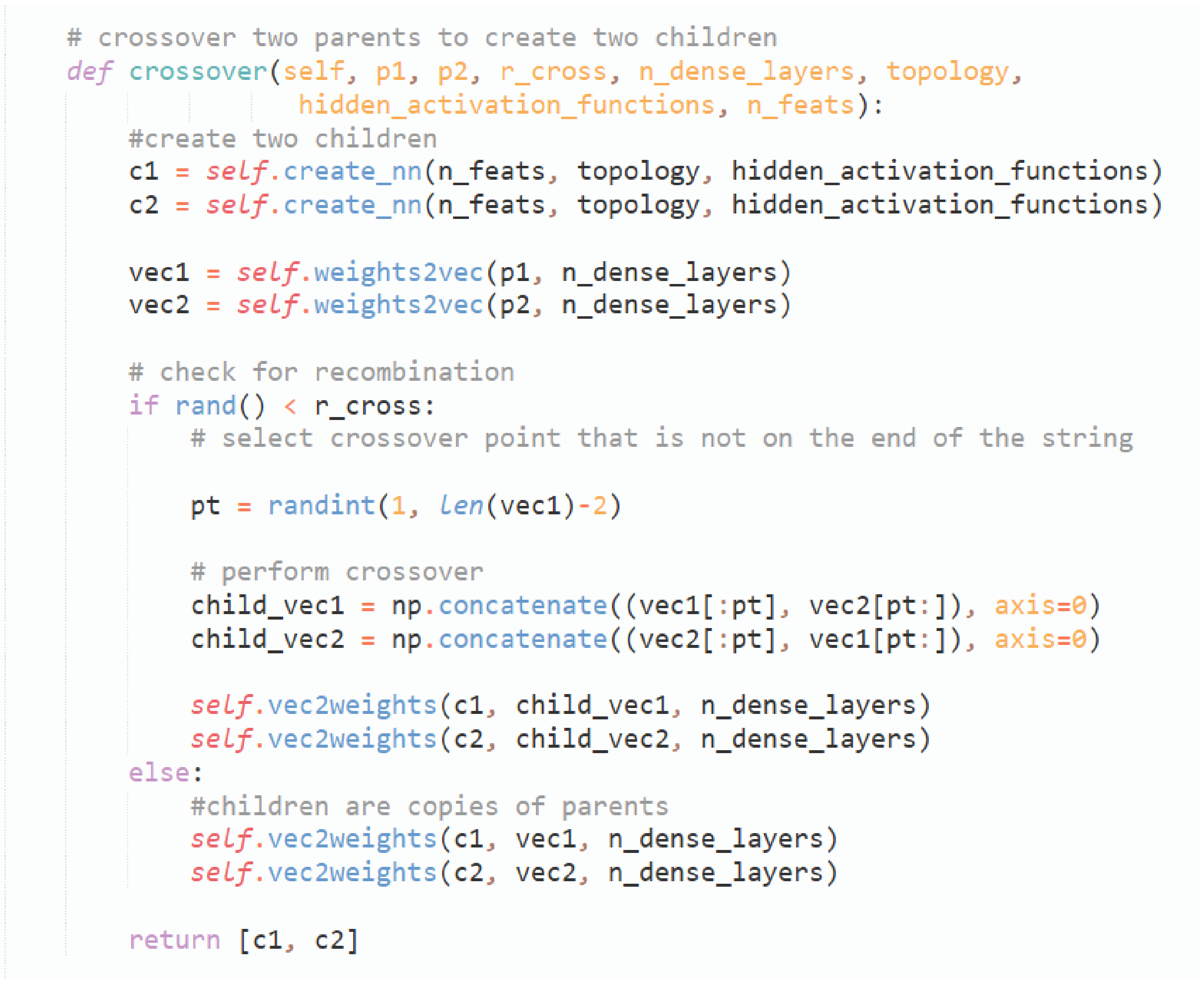
In tournament selection, an individual from the population is randomly selected along with two others. The one with the best score (smaller error rate) is returned (Figure 56).



```python
# tournament selection
def selection(self, pop, scores):
    # first random selection
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), 2):
        # check if better (e.g. perform a tournament)
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    return pop[selection_ix]
```

Figure 56: Tournament selection

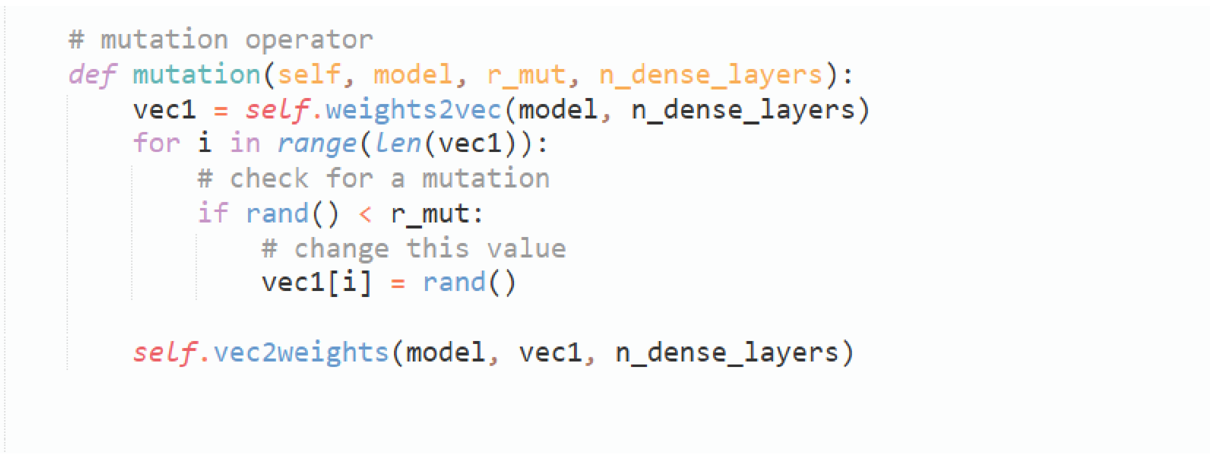
In crossover, two children are created. The weights of their parents are assigned to vectors using weights2vec auxiliary function. A random value is selected and compared to the r_cross. If the random value is smaller than the probability set by the user, then we select a random crossover point. The crossover point cannot be at the end of the vector. The first child contains the vector values of the first half of parent1 (until the crossover point) and the values of the second half of the parent2 (after the crossover point). The other child vice versa. If the initial selected value is larger than the r_cross probability, the children are same as their parents. Finally the weights of the new children are transformed from the vector using vec2weights auxiliary function and are returned (Figure 57).



```python
# crossover two parents to create two children
def crossover(self, p1, p2, r_cross, n_dense_layers, topology,
                    hidden_activation_functions, n_feats):
    #create two children
    c1 = self.create_nn(n_feats, topology, hidden_activation_functions)
    c2 = self.create_nn(n_feats, topology, hidden_activation_functions)

    vec1 = self.weights2vec(p1, n_dense_layers)
    vec2 = self.weights2vec(p2, n_dense_layers)

    # check for recombination
    if rand() < r_cross:
        # select crossover point that is not on the end of the string

        pt = randint(1, len(vec1)-2)

        # perform crossover
        child_vec1 = np.concatenate((vec1[:pt], vec2[pt:]), axis=0)
        child_vec2 = np.concatenate((vec2[:pt], vec1[pt:]), axis=0)

        self.vec2weights(c1, child_vec1, n_dense_layers)
        self.vec2weights(c2, child_vec2, n_dense_layers)
    else:
        #children are copies of parents
        self.vec2weights(c1, vec1, n_dense_layers)
        self.vec2weights(c2, vec2, n_dense_layers)

    return [c1, c2]
```

Figure 57: Crossover operator

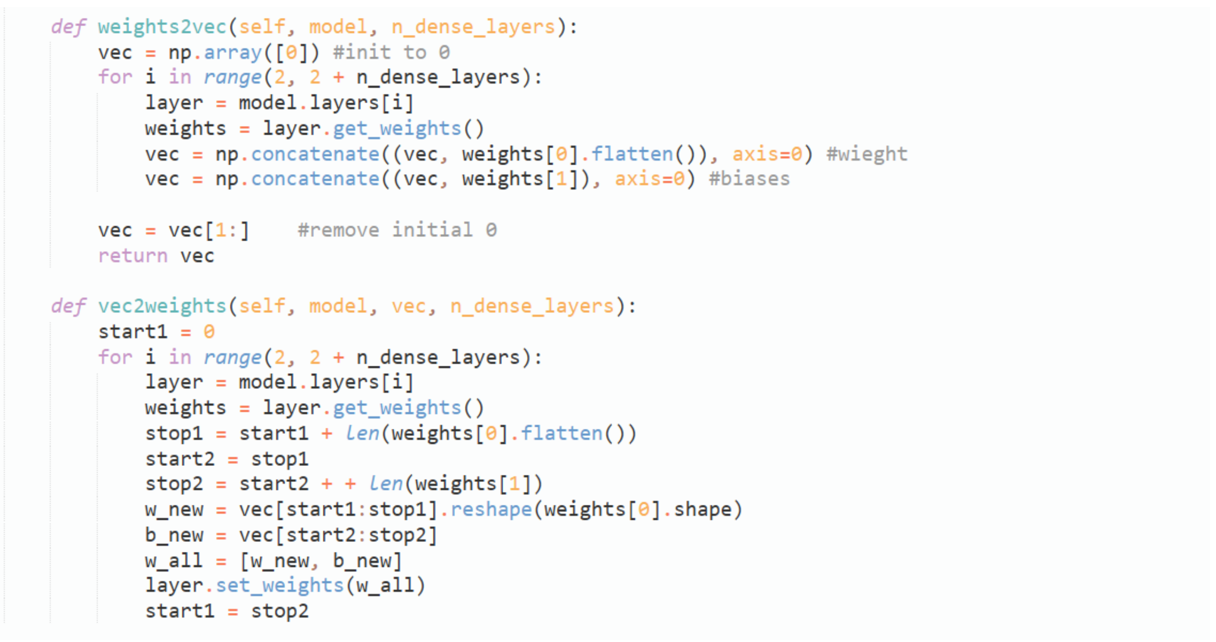
In mutation, for all the weights of the model, a random value is selected and compared to the r_mut. If the random value is smaller than the probability set by the user, then the weight is mutated i.e. changed to a random value. The weights are returned to the model (Figure 58).



```python
# mutation operator
def mutation(self, model, r_mut, n_dense_layers):
    vec1 = self.weights2vec(model, n_dense_layers)
    for i in range(len(vec1)):
        # check for a mutation
        if rand() < r_mut:
            # change this value
            vec1[i] = rand()

    self.vec2weights(model, vec1, n_dense_layers)
```

Figure 58: Mutation operator

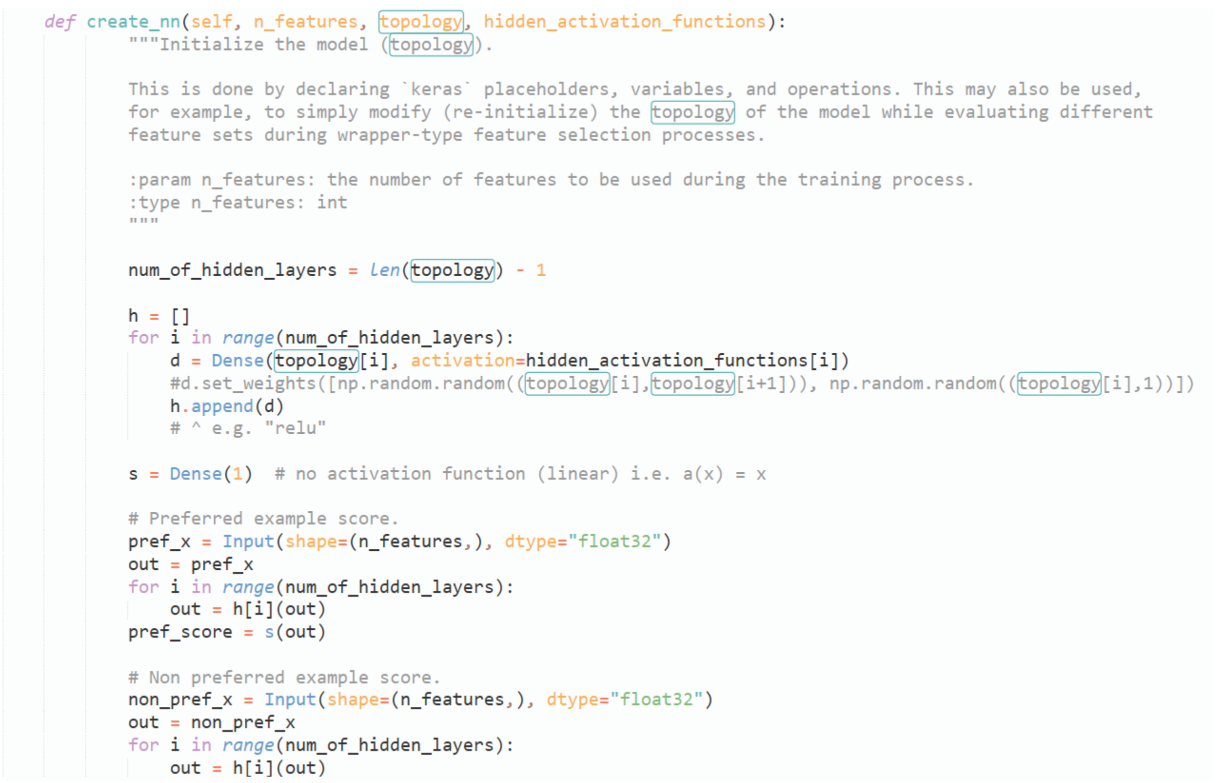
As shown above, because the genetic algorithm requires the data to be in a vector, we have to place all the weights and biases in a vector and vice versa, set the weights and biases from the vector. This is performed with the following auxiliary functions (Figure 59). Weights[0] includes the weights and weights[1] includes the biases.



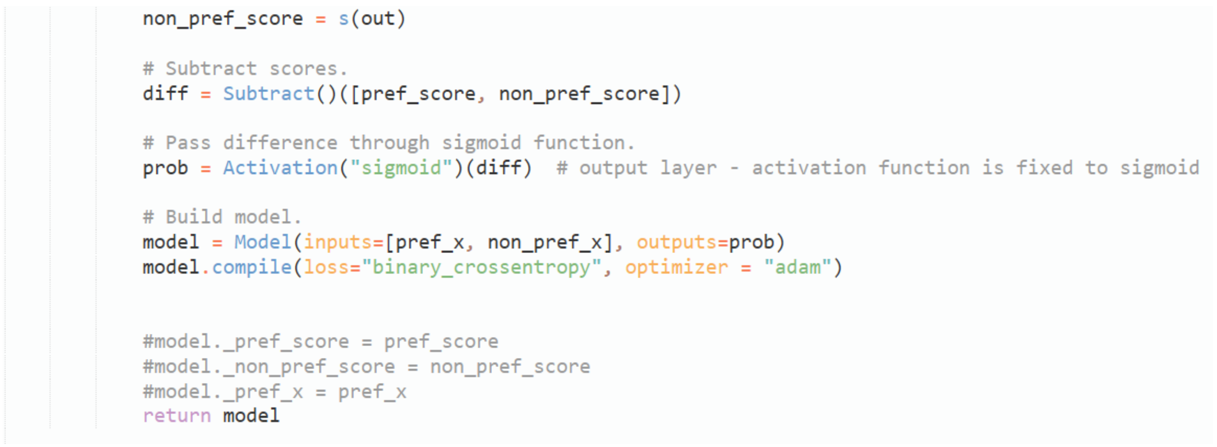
```python
def weights2vec(self, model, n_dense_layers):
    vec = np.array([0]) #init to 0
    for i in range(2, 2 + n_dense_layers):
        layer = model.layers[i]
        weights = layer.get_weights()
        vec = np.concatenate((vec, weights[0].flatten()), axis=0) #wieght
        vec = np.concatenate((vec, weights[1]), axis=0) #biases

    vec = vec[1:]    #remove initial 0
    return vec

def vec2weights(self, model, vec, n_dense_layers):
    start1 = 0
    for i in range(2, 2 + n_dense_layers):
        layer = model.layers[i]
        weights = layer.get_weights()
        stop1 = start1 + len(weights[0].flatten())
        start2 = stop1
        stop2 = start2 + + len(weights[1])
        w_new = vec[start1:stop1].reshape(weights[0].shape)
        b_new = vec[start2:stop2]
        w_all = [w_new, b_new]
        layer.set_weights(w_all)
        start1 = stop2
```

Figure 59: Auxiliary functions

Finally, the neural network creation function (Figure 60) is similar to the case of backpropagation: we take the two inputs (preferred and non-preferred data), pass them through one or more hidden layers (it is the user's choice) and then calculate their difference and provide the output.



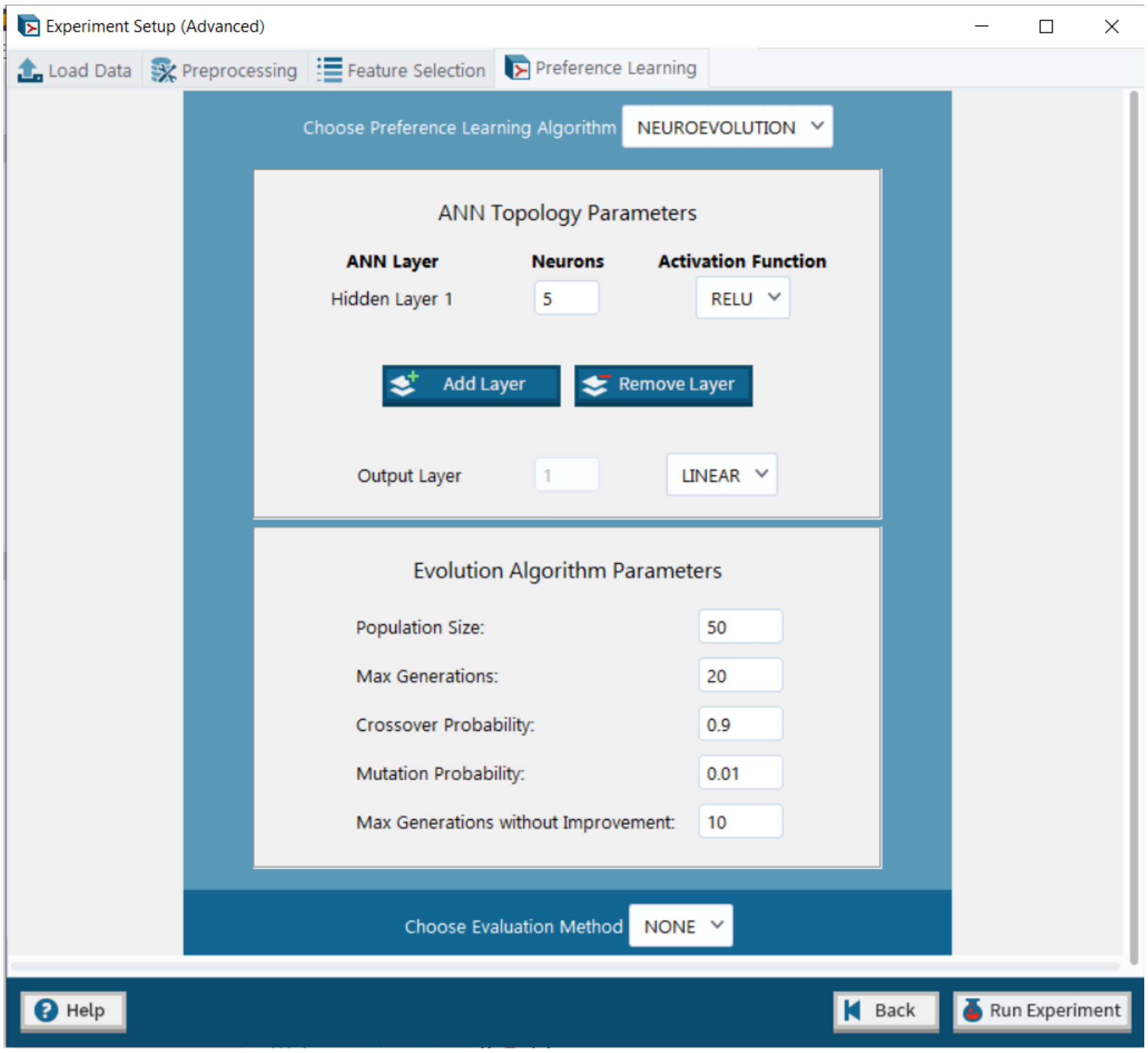
```
def create_nn(self, n_features, topology, hidden_activation_functions):
        """Initialize the model (topology).

        This is done by declaring `keras` placeholders, variables, and operations. This may also be used,
        for example, to simply modify (re-initialize) the topology of the model while evaluating different
        feature sets during wrapper-type feature selection processes.

        :param n_features: the number of features to be used during the training process.
        :type n_features: int
        """

        num_of_hidden_layers = len(topology) - 1

        h = []
        for i in range(num_of_hidden_layers):
            d = Dense(topology[i], activation=hidden_activation_functions[i])
            #d.set_weights([np.random.random((topology[i],topology[i+1])), np.random.random((topology[i],1))])
            h.append(d)
            # ^ e.g. "relu"

        s = Dense(1)  # no activation function (linear) i.e. a(x) = x

        # Preferred example score.
        pref_x = Input(shape=(n_features,), dtype="float32")
        out = pref_x
        for i in range(num_of_hidden_layers):
            out = h[i](out)
        pref_score = s(out)

        # Non preferred example score.
        non_pref_x = Input(shape=(n_features,), dtype="float32")
        out = non_pref_x
        for i in range(num_of_hidden_layers):
            out = h[i](out)
```



```
        non_pref_score = s(out)

        # Subtract scores.
        diff = Subtract()([pref_score, non_pref_score])

        # Pass difference through sigmoid function.
        prob = Activation("sigmoid")(diff)  # output layer - activation function is fixed to sigmoid

        # Build model.
        model = Model(inputs=[pref_x, non_pref_x], outputs=prob)
        model.compile(loss="binary_crossentropy", optimizer = "adam")


        #model._pref_score = pref_score
        #model._non_pref_score = non_pref_score
        #model._pref_x = pref_x
        return model
```

Figure 60: Creation of Network

Modifications have been performed in several files of pyplt package in order to adjust the neuroevolution algorithm to the gui in both beginners and advanced mode. Advanced users have the ability to set the desired values of the parameters of the algorithm (Figure 61), while beginners can perform an experiment with the default values (Figure 62).



Choose Preference Learning Algorithm: NEUROEVOLUTION

ANN Topology Parameters

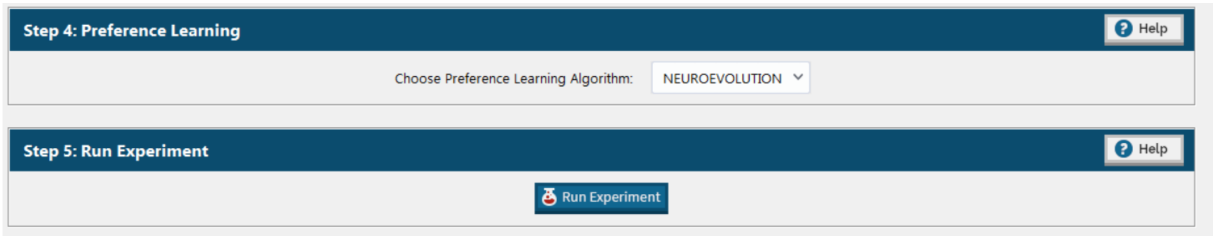
ANN Layer	Neurons	Activation Function
Hidden Layer 1	5	RELU
Output Layer	1	LINEAR

Evolution Algorithm Parameters

Population Size:	50
Max Generations:	20
Crossover Probability:	0.9
Mutation Probability:	0.01
Max Generations without Improvement:	10

Choose Evaluation Method: NONE

Figure 61: Advanced user's menu Neuroevolution algorithm


Step 4: Preference Learning

Choose Preference Learning Algorithm: NEUROEVOLUTION

Step 5: Run Experiment

Figure 62: Beginner's menu Neuroevolution algorithm

Due to the large number of neural networks that are created in each generation, for which we need to evaluate their performance, the algorithm is quite slow. Further improvements will be discussed in the last chapters. Regarding the performance of NEAT, as it contains randomness multiple experiments have been performed. A few of the models that were produced from NEAT are included in the Appendix section, along with the inferred models from Backpropagation, RankNET and RankSVM. The same dataset was used for all the experiments.

6

Usability testing of pyPLT

6.1 Method of testing

A usability testing survey has been conducted in order to summarize the experience of the participants while using the Preference Learning Toolbox. Several users have participated to the survey, from multiple backgrounds. The data was collected from **ten participants** in total (4 females and 6 males) aged between 24 and 36.

The participants received a list of four tasks along with a small questionnaire and an executable file of the PyPLT. The first task involved the loading of a simple dataset via beginner mode in order to familiarize with the tool. The users loaded three datasets of five, ten and fifty thousand registries and wrote down their observations about the process. In the second task, they loaded a dataset of 100 registries and performed an experiment via beginner mode using RankNET algorithm. They saved the experiment report as well as the derived model. Also, they kept a screenshot of the evaluation metrics of the model during training and testing.

The following tasks were a bit more advanced. In the third task, the participants had to load a dataset containing an extra column with query id values. A selection of the proper pre-processing option was important in order to load the dataset properly, without missing columns. They wrote down the minimum & maximum values per query id and feature. Then, they performed min-max normalization per query-id to the first feature. They ran the experiment using RankNet preference learning algorithm and wrote down the normalized values of the selected feature. Task four was the most demanding, as the users had to perform an experiment with Neuroevolution algorithm, by adjusting its parameters and selecting the Holdout Evaluation method.

After completing the assigned tasks, the participants filled a brief usability questionnaire. PLT questionnaire contained a few questions answered on a five-point scale (e.g., strongly agree, agree, neutral, disagree, and strongly disagree) along with some open ended questions.

6.2 Analysis

The first questions were helpful in order to summarize the users' background. The majority of them answered "Neutral" "Agree" and "Strongly Agree" to the question whether they are familiar with computer science. The next question was "I am familiar with Machine Learning". Users who answered "Strongly disagree", "Disagree" or "Neutral" could be considered novice and those who answered "Agree" or "Strongly Agree" could be considered experienced, as they are familiar with machine learning concepts. Regarding the use of preference learning algorithms most of them answered "Strongly disagree" and "Disagree", thus we cannot select between relevant and irrelevant participants to preference learning. Furthermore, only one of our participants had previous experience with PyPLT and thus we cannot divide our participants between experienced and novices. The background of the users is displayed in figure 63.

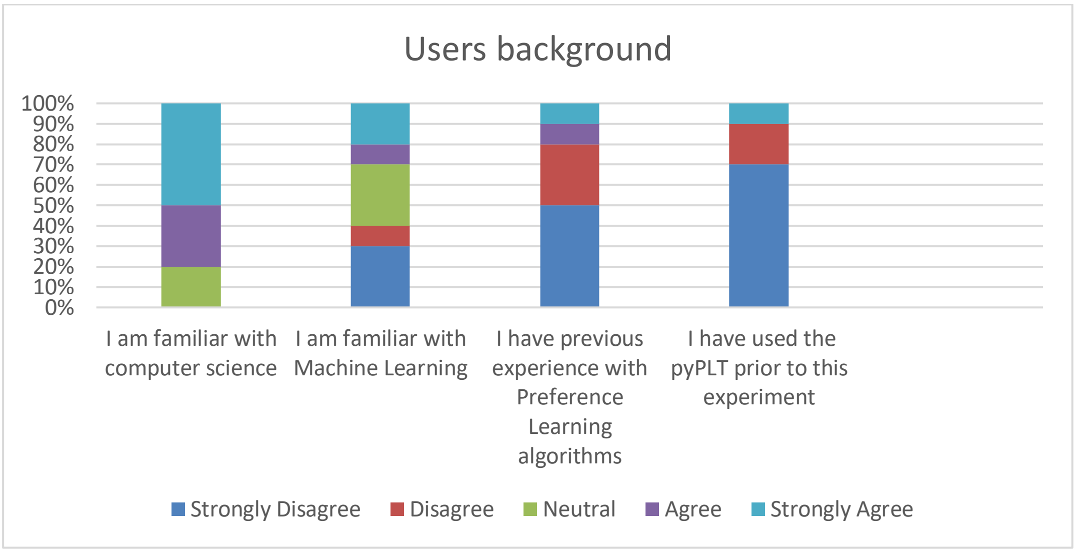


Figure 63: Users background

As mentioned previously, most of the participants consider themselves familiar with computer science. Regarding the open ended question about their field of studies, six of them study in fields concerning computer science, such as computer engineering, informatics, game research and affective computing. Three of the users are closely related to computer science, in the field of engineering, such as electronic, environmental and civil engineering. One of them is studying applied mathematics. Figure 64 shows the field of studies of the participants.

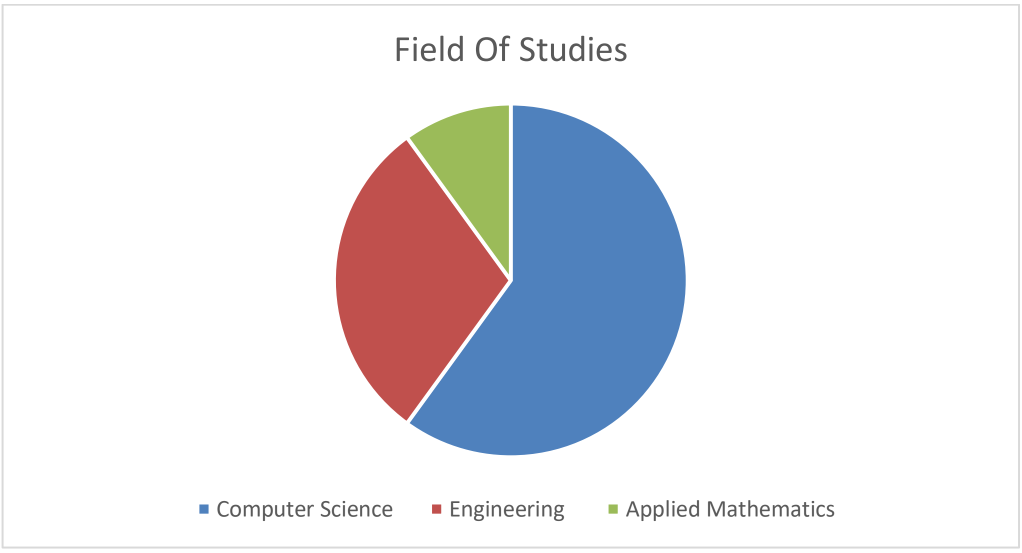


Figure 64: Field of studies

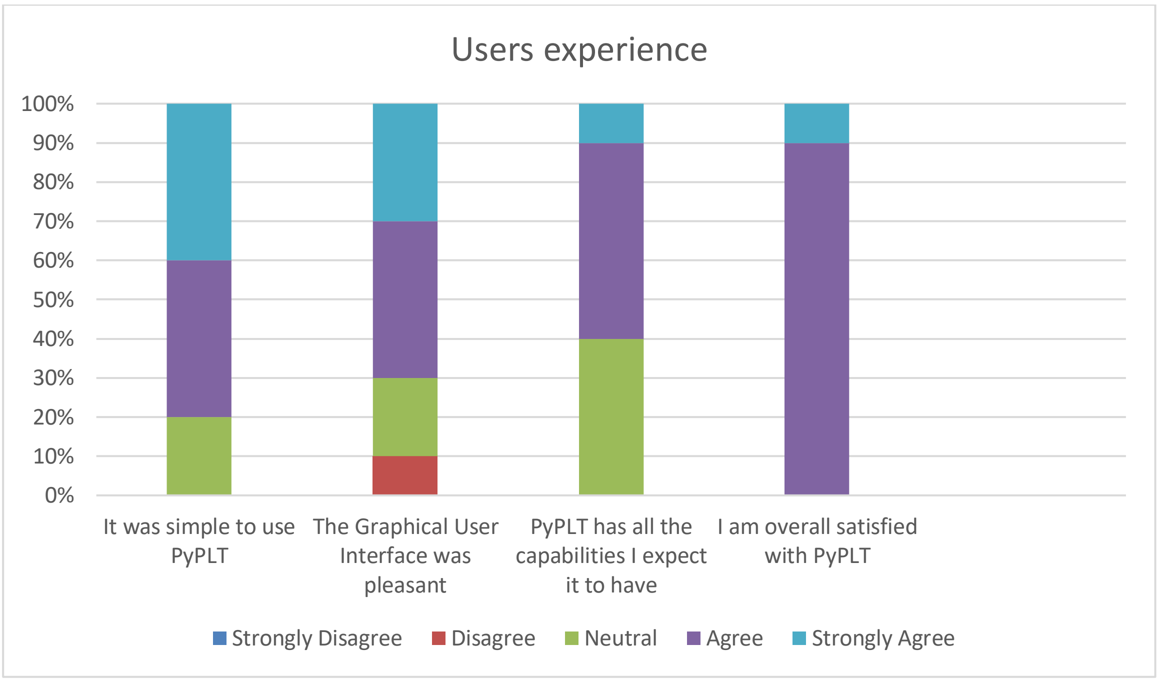


Figure 65: Users experience chart

Figure 65 shows the percentage of positive, negative and neutral responses to the questions related to the participants experience after using PyPLT. The vast majority of responses are positive as the highest percentages are achieved from answers “Agree” and “Strongly Agree”. Specifically, eight users responded that they were satisfied with how easy it is to use PyPLT and six agreed that PyPLT has all the capabilities they expect it to have. In addition, all of them were generally satisfied with the toolbox.

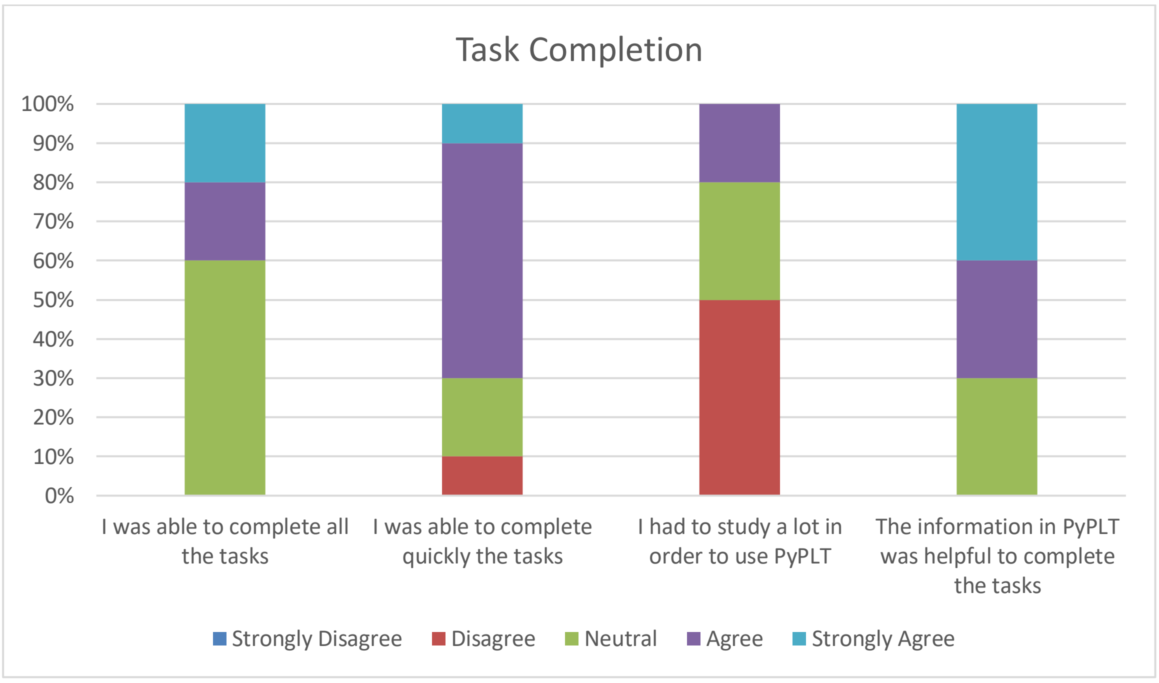


Figure 66: Task completion chart

Figure 66 shows the percentage of responses to the questions related to the task completion. Seven users agreed that the information in PyPLT was helpful to complete the tasks and that they were able to complete them quickly. Five of them answered that they needn't study a lot prior to using the toolbox, along with two participants staying neutral. At the question "I was able to complete all the given tasks" there are six neutral answers which is explained with the open-ended question "Is there a task you couldn't complete? Please specify". Most of the participants found difficulty in locating the normalized values during task 3. Only two of them managed to locate the normalized values. Furthermore, when comparing the users' screenshots, it was noticed that a participant had performed normalization to query_id instead of feature 1.

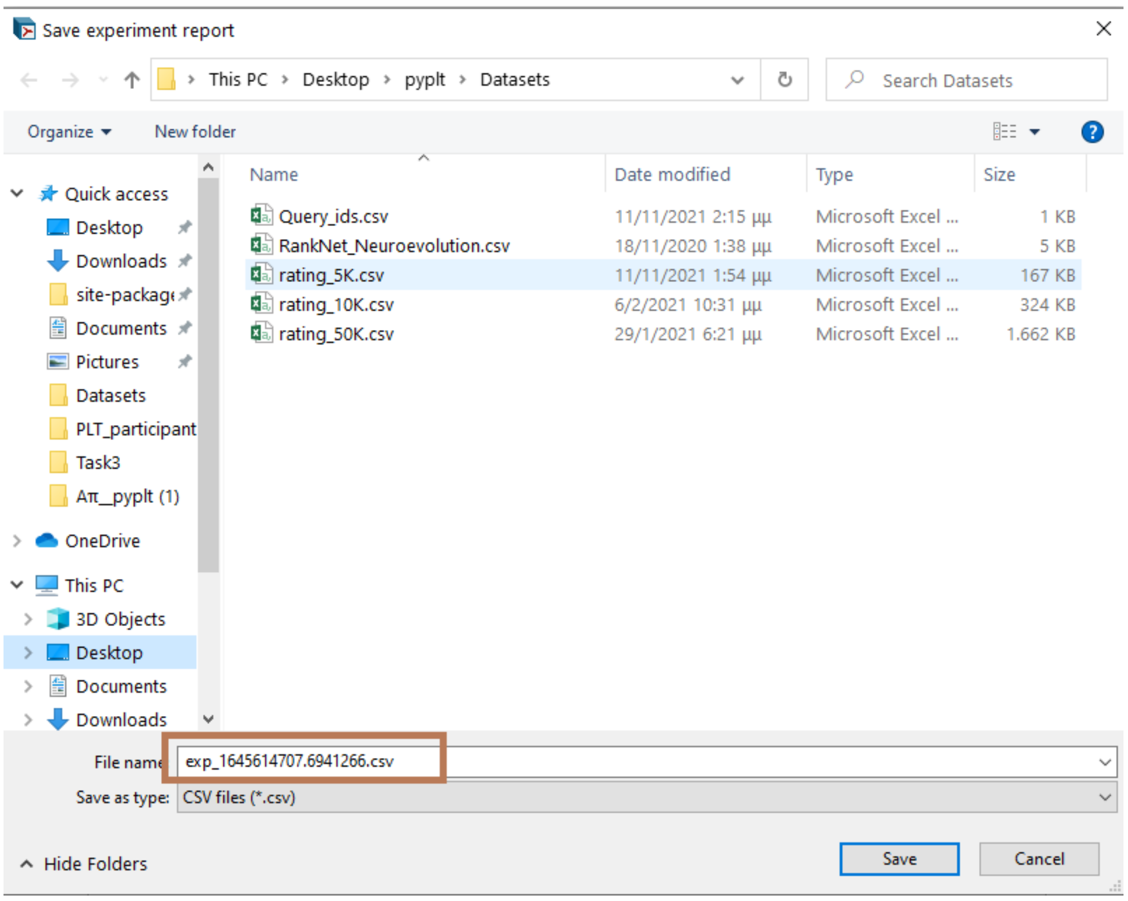
6.3 Efficiency and Users feedback

In the first task, participants loaded the datasets of five, ten and fifty thousand registries. This facilitates the use of large datasets, as they can be loaded and used efficiently for processing with PyPLT. None of the users commented on the dataset loading, as they didn't face any problem. Overall, the users were satisfied of PyPLT and made interesting comments regarding the addition of functionalities to the toolbox and the improvement of its features.

The most common response to the open-ended question "What did you most like about PyPLT" was the Graphical User Interface. Nine out of ten users answered that the GUI was very informative and guided them through the experiment. Also, a participant commented positively on the speed of the process, as he noticed that there is no delay between the steps in advanced mode. Regarding the question "What did you least like about PyPLT", half of the users answered the normalized values, as they didn't show up in the final report. One of them stated that the query_id should not appear among the features in the normalization step, as it can be confusing for the user and there is no need of normalizing the query_id values.

Furthermore, in the question "Which feature of the PyPLT would you improve" most of the improvements concerned task 3 and feature normalization. Some of the participants suggested the display of the normalized values in a separate window right after the process in order to reassure that it has been performed correctly. Others commented that the values could show up in the final report, in order to be able to save them along with the report. A participant suggested that the query_ id values should be hidden as an option among the features for normalization in the optimization step. Regarding the graphical interface a user proposed another design, in order to view all the functionality without having to maximize the window. Also, another suggestion was in advanced mode, when completing the experiment, by closing the last window all the previous windows should close and return to the first screen. Regarding the evaluation metrics, a user noticed that Kendall's tau doesn't correspond to any ratio and should be written as a fraction instead of a percentage.

Finally, interesting thoughts appeared regarding the question “Which functionality would you add to pyPLT”. Many users noticed that the experiment report and the produced model can only be saved in comma separated value files. They proposed a nicer display of the report, as it could be more comprehensible if the file could be saved in pdf or as an image. Furthermore, they suggested a graphical representation of the model. A user observed that when saving a report or a model, the default title of the file contains numbering and suggested more appropriate default names such as the simplest form report and model (Figure 67). At last, a participant proposed the clustering for the automatic recognition of categories within the sample.



Name	Date modified	Type	Size
Query_ids.csv	11/11/2021 2:15 μμ	Microsoft Excel ...	1 KB
RankNet_Neuroevolution.csv	18/11/2020 1:38 μμ	Microsoft Excel ...	5 KB
rating_5K.csv	11/11/2021 1:54 μμ	Microsoft Excel ...	167 KB
rating_10K.csv	6/2/2021 10:31 μμ	Microsoft Excel ...	324 KB
rating_50K.csv	29/1/2021 6:21 μμ	Microsoft Excel ...	1.662 KB

File name: exp_1645614707.6941266.csv

Save as type: CSV files (*.csv)

Figure 67: Default name when saving a report/model

7

Discussion & Conclusions

7.1 Limitations of PyPLT

PyPLT offers a variety of preprocessing options, feature selection and popular algorithms for each stage of the modeling process. In order to produce accurate and efficient models, a combination of proper dataset processing, selection and careful parameterization of the preference algorithm is necessary. The procedure of choosing the appropriate dataset processing and algorithm is important and depends on the dataset and the aim of the experiment. Lack of dataset preparation or the selection of an unsuitable method can lead to the construction of models with reduced accuracy.

The input of the toolbox is a single format file for problems where a total order of objects exists, or a dual format where a partial order of objects is known. In both cases, the files must be of comma-separated-values, containing only numeric values. In addition, the output of PyPLT, which is an experiment report and the derived model is also in csv format. This fact limits the capabilities of the tool, as it restrains the use of image or video as a dataset and thus the save of the final output.

Another possible limitation could be the constraint of time during Neuroevolution algorithm. Due to the nature of the preference learning algorithm, it is not possible to predict the duration of the execution. As a large number of neural networks are created in each generation, for which we need to evaluate their performance, the algorithm is quite slow. A model could be derived from the first generations if the minimum error is reached, or the exploration may continue for dozens of generations, if the algorithm continues finding better descendants.

7.2 Future Work

PyPLT is an open source software, designed to facilitate further development and improvement. Thus any researcher or user can add or improve the functionality of the toolbox. During the pre-processing stage, additional methods could be added regarding feature selection as well as more feedback on each step. As for the modelling stage, more preference learning or deep learning algorithms could be added. By improving the tool's ability to prepare the dataset and augmenting the range of available algorithms, the construction of more accurate models is reassured.

Along with the addition of new algorithms, methods and other options, an interesting addition would be to handle 2D data such as images and videos as input. This would require enhancements to the deep learning capabilities of PyPLT by integrating convolutional layers with ANN models. Furthermore, relating to the output of the toolbox, the derived model could be exported in a graphical representation as an image along with the experiment's report.

Future work on PyPLT could include extending the tool to allow saved (pre-trained) models to be loaded into the toolbox to predict new instances and continue training. The feedback of pre-trained models, along with their exposure to new data would result to even more accurate and effective models.

Based on the analysis of the user study, future development should include improvement regarding the display of feature normalization during pre-processing. In addition, the methods of saving the experiment report or the model should be further enriched as marked by the users.

7.3 Summary of Contributions - Conclusions

PyPLT is an accessible software to researchers of affective computing and human computer interaction at large. Through its graphical user interface, users without prior experience can set up and run an experiment quickly. Furthermore, developers have the ability to modify or improve the toolbox. Given the importance of ordinal labelling and the lack of tools for handling ordinal datasets, PyPLT is an important addition as it is designed specifically for ordinal data modelling.

The current thesis updated and extended the toolbox framework by investigating the errors noted by the users. Large datasets can be processed, a variety of evaluation metrics is offered along with a new preference learning algorithm. PyPLT has taken one step forward in ordinal data processing and the production of more reliable and valid predictive models. Depending on the nature of the data and the use of the inferred model, researchers have the ability to select the proper preprocessing and modelling. Also, the ground for isolating groups of data has been prepared, as the toolbox receives and processes parts of data, without affecting the whole dataset.

The usability testing confirmed the ease of use of PyPLT. Most of the participants were overall satisfied of the graphical user interface and the toolbox, as they managed to complete quickly most of the assigned tasks.

References

[1]. Farrugia, V. E., Martínez, H. P., & Yannakakis, G. N. (2015). The preference learning toolbox. arXiv preprint arXiv:1506.01709.

[2]. Camilleri, E., Yannakakis, G. N., Melhart, D., & Liapis, A. (2019, September). PyPLT: Python Preference Learning Toolbox. In 2019 8th International Conference on Affective Computing and Intelligent Interaction (ACII) (pp. 102-108). IEEE.

[3]. Yannakakis, G. N., Cowie, R., & Busso, C. (2018). The ordinal nature of emotions: An emerging approach. IEEE Transactions on Affective Computing, 12(1), 16-35.

[4]. Domshlak, C., Hüllermeier, E., Kaci, S., & Prade, H. (2011). Preferences in AI: An overview. Artificial Intelligence, 175(7-8), 1037-1052.

[5]. Pantic, M., & Rothkrantz, L. J. M. (2000). Automatic analysis of facial expressions: The state of the art. IEEE Transactions on pattern analysis and machine intelligence, 22(12), 1424-1445.

[6]. Scherer, K. R., & Ceschi, G. (1997). Lost luggage: a field study of emotion–antecedent appraisal. Motivation and emotion, 21(3), 211-235.

[7]. Cowie, R., Cox, C., Martin, J. C., Batliner, A., Heylen, D., & Karpouzis, K. (2011). Issues in data labelling. In Emotion-oriented systems (pp. 213-241). Springer, Berlin, Heidelberg.

[8]. Brooks, R. A. (2018). Intelligence without reason (pp. 25-81). Routledge.

[9]. Rockwell, A. (2017). The History of Artificial Intelligence. Harvard University.

[10]. Alzubi, J., Nayyar, A., & Kumar, A. (2018, November). Machine learning from theory to algorithms: an overview. In Journal of physics: conference series (Vol. 1142, No. 1, p. 012012). IOP Publishing.

[11]. Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. Science, 349(6245), 255-260.

[12]. Mitchell, T. M. (1999). Machine learning and data mining. Communications of the ACM, 42(11), 30-36.

[13]. Hastie, T., Tibshirani, R., & Friedman, J. (2009). Overview of supervised learning. In The elements of statistical learning (pp. 9-41). Springer, New York, NY.

[14]. Brownlee, J. (2016). Supervised and unsupervised machine learning algorithms. Machine Learning Mastery, 16(03).

[15]. Shobha, G., & Rangaswamy, S. (2018). Machine learning. In Handbook of statistics (Vol. 38, pp. 197-228). Elsevier.

[16]. Liu, Y., Zhang, H., Zeng, L., Wu, W., & Zhang, C. (2017). MLBench: How Good Are Machine Learning Clouds for Binary Classification Tasks on Structured Data?. arXiv preprint arXiv:1707.09562.

[17]. Nokeri, T. C. (2021). An Introduction to Simple Linear Regression. In Data Science Revealed (pp. 1-43). Apress, Berkeley, CA.

[18]. Maulud, D., & Abdulazeez, A. M. (2020). A Review on Linear Regression Comprehensive in Machine Learning. Journal of Applied Science and Technology Trends, 1(4), 140-147.

[19]. Kleinbaum, D. G., & Klein, M. (2010). Introduction to logistic regression. In Logistic regression (pp. 1-39). Springer, New York, NY.

[20]. Fürnkranz, J., & Hüllermeier, E. (2010). Preference learning and ranking by pairwise comparison. In Preference learning (pp. 65-82). Springer, Berlin, Heidelberg.

[21]. Yannakakis, G. N., & Hallam, J. (2011, October). Ranking vs. preference: a comparative study of self-reporting. In International conference on affective computing and intelligent interaction (pp. 437-446). Springer, Berlin, Heidelberg.

[22]. Yannakakis, G. N., & Martínez, H. P. (2015). Ratings are overrated! Frontiers in ICT, 2, 13.

[23]. Agatonovic-Kustrin, S., & Beresford, R. (2000). Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research. Journal of pharmaceutical and biomedical analysis, 22(5), 717-727.

[24]. Dupond, S. (2019). A thorough review on the current advance of neural network structures. Annual Reviews in Control, 14, 200-230.

[25]. Feng, J., & Lu, S. (2019, June). Performance analysis of various activation functions in artificial neural networks. In Journal of physics: conference series (Vol. 1237, No. 2, p. 022030). IOP Publishing.

[26]. Vani, S., & Rao, T. M. (2019, April). An experimental approach towards the performance assessment of various optimizers on convolutional neural network. In 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI) (pp. 331-336). IEEE.

[27]. Floreano, D., Dürr, P., & Mattiussi, C. Neuroevolution: from architectures to learning. Evolutionary Intelligence. 1 (1), 47–62 (2008).

[28]. Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., & Clune, J. (2017). Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. arXiv preprint arXiv:1712.06567.

[29]. Kearney, William. T. (2016). Using genetic algorithms to evolve artificial neural networks. Colby College. Honors Thesis. Paper 818.

[30]. Peng, Y., Chen, G., Singh, H., & Zhang, M. (2018, July). NEAT for large-scale reinforcement learning through evolutionary feature learning and policy gradient search. In Proceedings of the Genetic and Evolutionary Computation Conference (pp. 490-497).

[31]. Lopes, P., Liapis, A., & Yannakakis, G. N. (2017). Modelling affect for horror soundscapes. IEEE Transactions on Affective Computing, 10(2), 209-222.

Appendix

In the following tables there is an overview of the evaluation metrics during testing of the computational models created using pyplt. The Sonancia Audio Clipdataset [31] was used in the experiment with the algorithms Backpropagation, RankNET, RankSVM and NEAT.

	Backpropagation	RankNET	RankSVM
Accuracy	51.68%	82.7%	49.5%
Precision	100%	50.26%	42.70%
Recall	51.68%	87.8%	84.04%
F1 score	68.14%	63.94%	56.63%
Kendall Tau	33.60%	75.7%	74.63%

Figure 68: Performance of models derived from Backpropagation, RankNET, RankSVM

	NEAT 1st model	NEAT 2st model	NEAT 3st model
Accuracy	52%	55.8%	81.57%
Precision	40.32%	38.09%	100%
Recall	84.26%	86.74%	81.57%
F1 score	54.54%	52.94%	89.85%
Kendall Tau	76.24%	78.44%	66.54%

Figure 69: Performance of models derived from NEAT algorithm