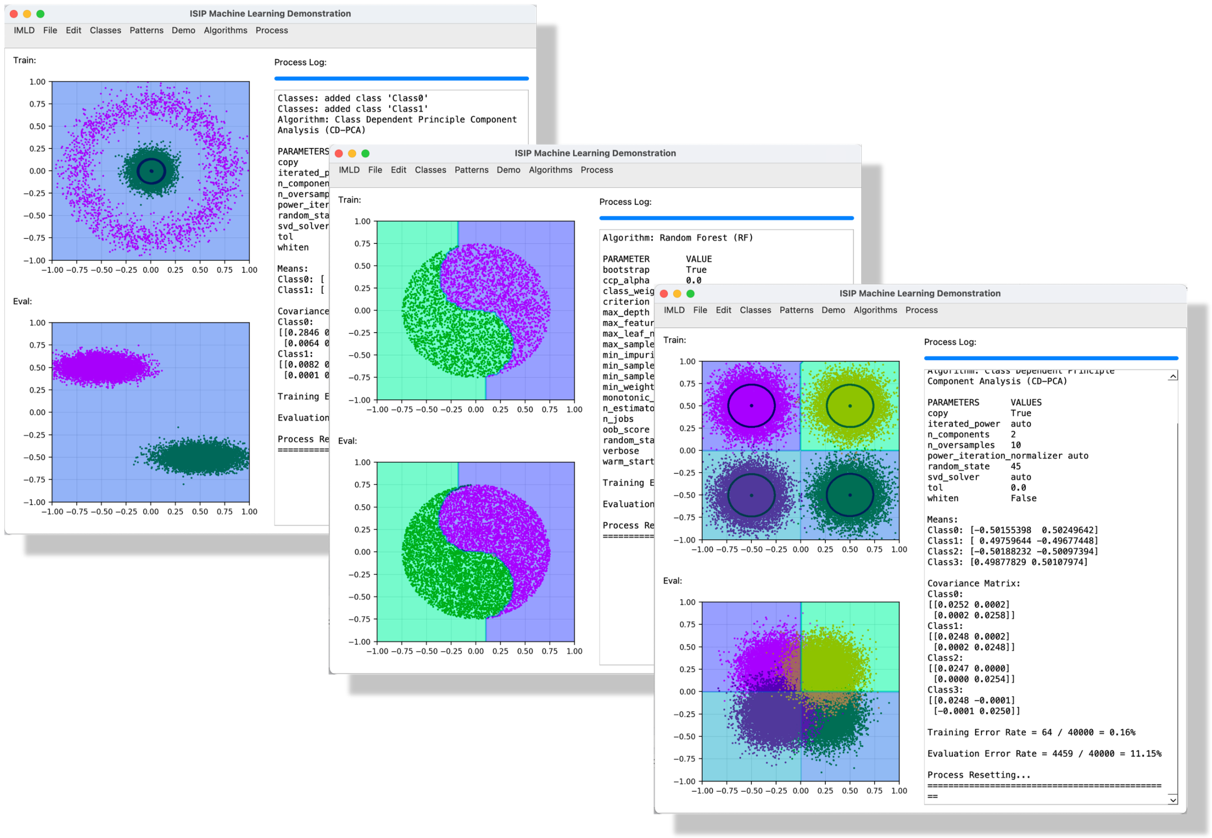
The ISIP Machine Learning Demonstration User Guide

January 1, 2026



ISIP Machine Learning Demonstration
IMLD File Edit Classes Patterns Demo Algorithms Process
Train:
Eval:
Process Log:

```
Classes: added class 'Class0'
Classes: added class 'Class1'
Algorithm: Class Dependent Principle Component
Analysis (CD-PCA)

PARAMETERS
copy
iterated_p
n_componen
n_oversamp
power_iter
random_sta
svd_solver
tol
whiten

Means:
Class0: [
Class1: [

Covariance
Class0:
[[0.2846 0
 [0.0064 0
Class1:
[[0.0082 0
 [0.0001 0

Training E

Evaluation

Process Re
==========
```

```
Algorithm: Random Forest (RF)

PARAMETER       VALUE
bootstrap       True
ccp_alpha       0.0
class_weig
criterion
max_depth
max_featur
max_leaf_n
max_sample
min_impuri
min_sample
min_sample
min_weight
monotonic_
n_estimato
n_jobs
oob_score
random_sta
verbose
warm_start

Training E

Evaluation

Process Re
==========
```

```
Algorithm: Class Dependent Principle
Component Analysis (CD-PCA)

PARAMETERS      VALUES
copy            True
iterated_power  auto
n_components    2
n_oversamples   10
power_iteration_normalizer auto
random_state    45
svd_solver      auto
tol             0.0
whiten          False

Means:
Class0: [-0.50155398  0.50249642]
Class1: [ 0.49759644 -0.49677448]
Class2: [-0.50188232 -0.50097394]
Class3: [0.49877829 0.50107974]

Covariance Matrix:
Class0:
[[0.0252 0.0002]
 [0.0002 0.0258]]
Class1:
[[0.0248 0.0002]
 [0.0002 0.0248]]
Class2:
[[0.0247 0.0000]
 [0.0000 0.0254]]
Class3:
[[0.0248 -0.0001]
 [-0.0001 0.0250]]

Training Error Rate = 64 / 40000 = 0.16%

Evaluation Error Rate = 4459 / 40000 = 11.15%

Process Resetting...
============================================
==
```

Prepared By:

Sohail Aji and Joseph Picone

The Neural Engineering Data Consortium
College of Engineering, Temple University
1947 North 12th Street
Philadelphia, Pennsylvania 19122-6018
Tel: 708-848-2846
Email: {sohail.aji, picone}@temple.edu

NEURAL ENGINEERING
DATA CONSORTIUM

EXECUTIVE SUMMARY

The Institute for Signal and Information Processing (ISIP) Machine Learning Demo (IMLD) is an interactive, visual machine learning (ML) and pattern recognition tool developed entirely with open-source libraries in the Python programming language. It can be easily downloaded and run (*www.isip.piconepress.com/projects/imld*) either through an IDE or a command line/terminal in any operating system. In addition, the platform can be launched directly on most common web browsers, offering a convenient and fully online experience without needing a local installation. This accessibility allows users to experiment with machine learning concepts without a complex setup, making it ideal for both beginners and advanced practitioners. Designed with education in mind, IMLD provides an intuitive and engaging way to explore hands-on classification techniques, enabling users to visualize the concept of how different machine learning algorithm's function.

IMLD's interface is designed to be both functional and educational. The train and eval plotting windows provide a side-by-side view for building and testing models visually, allowing users to see how different algorithms classify data and how changes in parameters affect model performance in real time. The algorithm toolbar simplifies the selection of various ML algorithms and outlines the respective parameters, allowing users to easily experiment with different models and settings. It provides an accessible entry point for understanding how parameter tuning can impact classification outcome and error rate. The process log provides a real-time summary of all users' actions and model decisions, making the workflow transparent and easy to follow.

IMLD provides users with multiple ways to generate and manipulate data for training and evaluation. Users can create their own classes and draw their own two-dimensional (2D) data points in either single point or Gaussian distributions for both the training and evaluation data input windows when generating data. They can also import data in a comma separated value (.csv) format, which is the same format IMLD exports data as well as when users choose to save their own data. Additionally, IMLD includes a selection of pre-built demonstration datasets that are useful for exploring the theoretical properties of algorithms. Other options include toroidal (donut-shaped) distributions and a yin-yang distribution. This unique array of dataset shapes allows users to explore how different classifiers handle data that require complex decision boundaries. This provides deeper insight into the behavior of many common algorithms.

Once the user has their desired data in the train and eval windows, they can choose from a list of algorithms used to classify the data. Algorithms, once selected, will present the user with a list of parameters that they can change to optimize the classification process. The application displays results to a process log window, which also keeps track of the creation and deletion of classes and the algorithm/parameters chosen by the user. Once a model has been created using the training data, it can be applied to blind evaluation data. An error rate will be calculated for that dataset. Users can then decide to delete the results from each window, delete the data itself, or reset the entire program. By resetting, users can use multiple algorithm parameters to compare error rates and optimize their model on the training or evaluation datasets as desired.

IMLD has been in existence in various forms since the mid-1990's. It was originally developed as a Java applet to support graduate-level instruction in machine learning. The current version is written in Python and is implemented in a client/server format so that it can be easily run as a web application. It makes extensive use of standard machine learning libraries such as Python's SKLearn. The model API has been developed in a way that users can easily add new algorithms. This visual structure makes IMLD especially valuable as a teaching tool, allowing users to explore concepts such classification performance and error evaluation in an interactive manor. With its clear design and customizability, IMLD is well suited for classroom demonstrations, labs, or self-guided learning. Additionally, its open-source nature ensures that it remains fully adaptable to new advancements in machine learning research, and educators can use IMLD in their coursework to give students hands-on experience within machine learning.

Table of Contents

1. Introduction

The Interactive Machine Learning Demos (IMLD) platform provides an accessible, hands-on way to explore key concepts in machine learning (ML). Developed and hosted by the Institute for Signal and Information Processing (ISIP: *www.isip.piconepress.com*), the IMLD platform offers a collection of interactive tools designed to illustrate various ML algorithm concepts. Originally developed as a Java applet in the 1990's (Shaffer et al., 1998), It has been used for over 20 years to teach a graduate-level course in machine learning (*https://isip.piconepress.com/courses/temple/ece_8527/*).

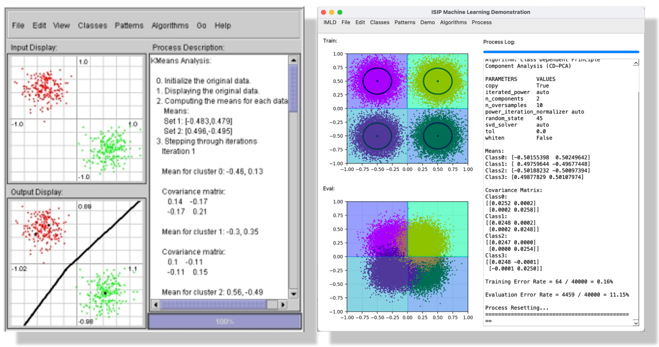
IMLD was created at a time when Java was widely believed to be the future of computing, and Java applets were the primary way to deliver web-based demonstrations. It was part of an extensive library of open-source signal processing tools (Huang & Picone, 2002). Although this app was functional, other languages such as Python began to become the primary environment for the development of machine learning algorithms. Despite running slower than Java in many cases, Python's simpler syntax and flexibility make it highly popular among researchers who develop machine learning technology. These advantages ultimately led to the decision to rebuild IMLD as a Python-based application. This version proved an excellent tool for education and research in machine learning, helping newcomers to grasp abstract concepts while also supporting the research of those more experienced in ML (Thai et al., 2023; Cap et al., 2022).

A major drawback, however, was accessibility, as users could only access the app by downloading and running the Python script locally. This prevented those with minimal technical experience from easily accessing it. The machine learning and graphics libraries use many third-party libraries, and installation of these libraries can often be complex due to versioning issues. This led to a decision to transform IMLD into a web application. This significantly increases its accessibility, allowing it to be easily found and used by anyone with access to a web browser. In addition to running the tool in a web browser, users also have the option to download a standalone version of the application directly from the IMLD website. This allows for offline use and provides greater flexibility for those who prefer working in a local environment.

A key part of IMLD's usability lies in its interface design shown in Figure 1, which shows screenshots of the original Java version and the current Python version. IMLD includes two data analysis windows that visualize the data and the results of an algorithm's attempt to classify the data, a toolbar that lets users select file operations, algorithms, data sets, etc., and a log window that records each stage of the workflow. Together, these features make IMLD a powerful tool for visualizing the application of modern machine learning algorithms to two-dimensional data sets.

This user guide will walk you through navigating the IMLD website, running the interactive demos, adjusting key parameters, and applying the underlying concepts to your own projects. You will learn how to get started quickly, and how to interpret algorithmic outputs. By the end, you should have a clear understanding of the website's offerings and feel comfortable leveraging its demos to deepen your familiarity with ML techniques.

This guide is structured into eight sections that include this introduction (Section 1), an overview of the tool (Section 2), a description of the user interface (Section 3), a description of the algorithms (Section 4), some typical use cases (Section 5), a discussion of how to integrate new algorithms (Section 6), how to download and install the tool (Section 7), and a summary (Section 8).



```
KMeans Analysis:

  0. Initialize the original data.
  1. Displaying the original data.
  2. Computing the means for each data
     Means:
     Set 1: [-0.483,0.479]
     Set 2: [0.496,-0.495]
  3. Stepping through iterations
     Iteration 1

     Mean for cluster 0: -0.46, 0.13

     Covariance matrix:
       0.14   -0.17
      -0.17    0.21

     Mean for cluster 1: -0.3, 0.35

     Covariance matrix:
       0.1    -0.11
      -0.11    0.15

     Mean for cluster 2: 0.56, -0.49
```

```
Algorithm: Class Dependent Principle
Component Analysis (CD-PCA)

PARAMETERS      VALUES
copy            True
iterated_power  auto
n_components    2
n_oversamples   10
power_iteration_normalizer auto
random_state    45
svd_solver      auto
tol             0.0
whiten          False

Means:
Class0: [-0.50155398  0.50249642]
Class1: [ 0.49759644 -0.49677448]
Class2: [-0.50188232 -0.50097394]
Class3: [0.49877829 0.50107974]

Covariance Matrix:
Class0:
[[0.0252 0.0002]
 [0.0002 0.0258]]
Class1:
[[0.0248 0.0002]
 [0.0002 0.0248]]
Class2:
[[0.0247 0.0000]
 [0.0000 0.0254]]
Class3:
[[0.0248 -0.0001]
 [-0.0001 0.0250]]

Training Error Rate = 64 / 40000 = 0.16%

Evaluation Error Rate = 4459 / 40000 = 11.15%

Process Resetting...
============================================
==
```

Figure 1. IMLD began as a Java applet (left) and has evolved into a Python application run via a client/server interface.

2. Overview

The basic IMLD user interface is shown in Figure 2. The application window contains 5 major components: the title bar, the menu bar, the plotting windows, the algorithm toolbar, and the process log. These components were designed to simplify the process of creating data and performing an evaluation. The main components of the user interface are:

(1) **Title Bar:** located at the very top of the window, it displays the title of the application "The ISIP Machine Learning Demo" on the far left, as well as contact and share buttons on the far right. The former is linked to the IMLD web site, where you can find information on the latest software releases. The contact button provides opens a pop-up window that provides users with support resources, including an email address for questions or feedback (help@nedcdata.org). The share button provides a URL to the project web site.

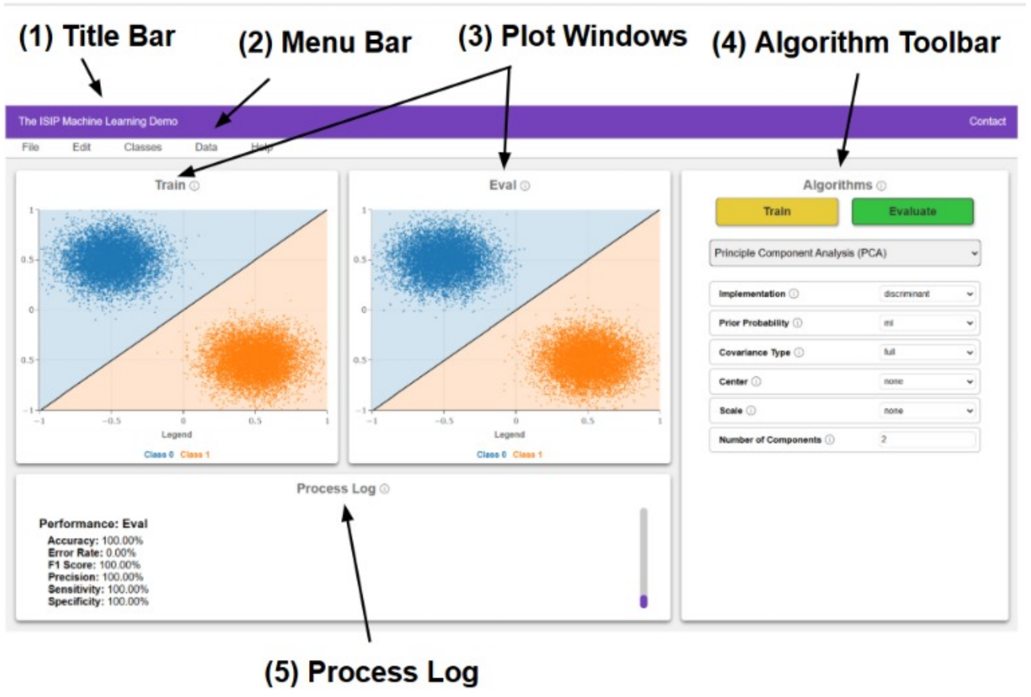
(2) **Menu Bar:** located at the top of the page below the title bar. It contains the options for generating, loading, and saving different datasets. Users can load preset parameter configurations, clear displays, reset data and obtain help (including viewing this manual).

(3) **Plot Windows:** located in the middle of screen, it helps a user visualize each step of the algorithm process. On the left is the train plot, where the user sees the training data distribution and how the model fits that data. On the right is the eval plot, showing the evaluation data distribution and how well their generated model is categorizing the data distribution.

(4) **Algorithm Toolbar:** located on the right side of the window, it allows users to select and configure algorithms. It has two buttons labeled "Train" and "Evaluate" which allow you to apply the selected algorithm to the train or eval data. Under these buttons is a dropdown menu containing each algorithm, where you select which algorithm you would like to use, and set its parameters.

(5) **Process Log:** located at the bottom of the screen, it provides user feedback such as parameter settings, classification error rates and error messages.

The overall application window can be resized as a normal desktop window, and the location of these components will be adjusted accordingly.



Principle Component Analysis (PCA)

Implementation	discriminant
Prior Probability	ml
Covariance Type	full
Center	none
Scale	none
Number of Components	2

```
Performance: Eval
  Accuracy: 100.00%
  Error Rate: 0.00%
  F1 Score: 100.00%
  Precision: 100.00%
  Sensitivity: 100.00%
  Specificity: 100.00%
```

Figure 2. An example of IMLD executing Principal Component Analysis (PCA) with a two-class Gaussian dataset for both training and evaluation data. The decision surface (the black line) visually separates the classes.

3. User Interface

In this section, we describe the features available in the five main components of the user interface. We explain each menu option and button available to the user.

3.1. Title Bar

The title bar contains three key links. The title of the application on the far left, “The ISIP Machine Learning Demo,” is linked to the project web site that contains the latest information on software releases, publications and other relevant resources. The text “Contact” on the right is linked to a page providing contact information to report bugs and receive help. The text “Share” on the far right provides a link to the download site where you can download the source code and a standalone version of the application.

3.2. Menu Bar

The Menu Bar, labeled (2) in Figure *2*, is the primary navigation hub of the IMLD interface, grouping all high-level actions under five intuitive drop downs menus: File, Edit, Classes, Data, and Help. From these menus, users can manipulate data, models and parameter settings.

3.2.1. File

When hovering over “File” in the Menu Bar, shown in Figure 3, a drop-down menu appears displaying various options for managing data, models, and parameters. The File options allow users to load, save, and export datasets and models. The available options include:

- **Load Train Data:** import a dataset to be used for training.
- **Load Eval Data:** import a dataset to be used for evaluation.
- **Load Model Parameters:** import a set of saved model parameters for reusing specific configurations.
- **Load Model:** import a full model with structure and weights.
- **Save Train Data:** save the current training dataset to a file.
- **Save Eval Data:** save the current evaluation dataset to a file.
- **Save Model Parameters:** save the current model parameters for reusability.
- **Save Model:** save a full model with structure and weights.

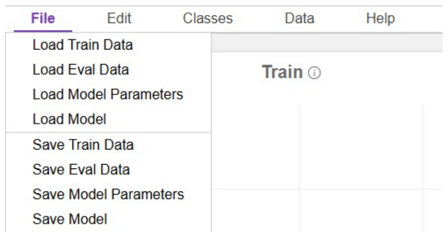


Figure 3. The “File” drop-down menu in the Menu Bar displays options for importing and exporting datasets, models, and parameter configurations.

These options allow users to efficiently manage their datasets and models throughout the machine learning workflow. By enabling the loading of previously saved work, users can resume past sessions or compare results across runs. The ability to save trained models and parameter configurations supports experiment tracking, reproducibility, and iterative testing. This flexibility is especially useful when fine-tuning models, switching between datasets, or applying consistent settings across different experiments.

IMLD uses a structured CSV (Comma-Separated Values) file format to import and export training and evaluation datasets. CSV files are perhaps the most popular format for machine learning research when dealing with small datasets. This format supports both user-defined datasets and datasets automatically generated by IMLD. An example CSV is shown in Figure 4. This format is compatible with several other machine learning tools developed by NEDC.

A CSV file in IMLD consists of two sections: a header and data. The header contains five rows that include the data that allows IMLD to configure itself to the same state it was in when the data was generated. These rows include the filename, the class labels, the colors assigned to each class, and the plot limits as a tuple (x_min, x_max, y_min, y_max).

In the data section, each data point is listed on its own row, and consists of a tuple containing the class label, the x-coordinate, and the y-coordinate. Data points can be interleaved though by default IMLD writes them class by class.

```
# filename: /Downloads/imld_train.csv
# classes: [dog,cat]
# colors: [#1f77b4,#ff7f0e]
# limits: [-2.0,4.0,-1.0,5.0]
#
dog, -0.616837, 0.483275
dog, -0.487216, 0.479231
cat, -0.394414, 0.725841
cat, -0.372843, 0.701493
...
```

Figure 4. A sample of a CSV file generated by IMLD that contains the filename, the names of the two classes, two colors identified using hex color codes, the plot limit boundaries and the data represented as a tuple (class label, x-coordinate and y-coordinate).

In addition to using CSV files for data, IMLD uses a standardized, extensible system to manage algorithm configurations through parameter files written in the Tom's Obvious, Minimal Language (TOML) format. These files define the default settings and input controls for each machine learning algorithm available within the system, enabling users to load, reuse, and modify algorithm presets efficiently. Each algorithm has a unique set of hyperparameters, which are dynamically rendered in the interface based on the information provided in its corresponding TOML file. These files contain a structured, human-readable format for defining default settings and input types for each algorithm, making configuration reusable and easy to manage. An example, which was generated by selecting the algorithm "Support Vector Machine", training a model, and executing the "Save Model" function, is shown in Figure 5.

The "Save Model" function in IMLD writes a model in a Pickle file format (".pkl"). This is a popular Python serialization format that stores model information in a mixture of text and binary data. These are not human readable but follow the same formats used by popular tools such as SKLearn. These files store the internal state of a trained model object, allowing it to be reused or evaluated without needing to retrain it from scratch. Pickled models preserve the structure, learned parameters (e.g., weights, biases), and configuration of an algorithm. This makes .pkl files ideal for saving trained models for future use, sharing models with others, and loading pre-trained models into a new IMLD session for evaluation or demonstration.

Users can save trained models by selecting "Save Model" from the File menu. Models can later be reloaded with the "Load Model" function, allowing for immediate reuse or evaluation without repeating the training process.

```
[Support_Vector_Machines_SVM]
name = "Support Vector Machines (SVM)"

[Support_Vector_Machines_SVM.params.impleme
ntation_name]
type = "select"
default = "sklearn"

[Support_Vector_Machines_SVM.params.c]
type = "float"
default = "1"

[Support_Vector_Machines_SVM.params.gamma]
type = "float"
default = "0.1"

[Support_Vector_Machines_SVM.params.kernel]
type = "select"
default = "linear"
...
```

Figure 5. A sample of a TOML model parameter file generated by the "Save Model" function.

3.2.2. Edit

When hovering over "Edit" in the Menu Bar, shown in Figure 6, a drop-down menu appears providing options to adjust settings and clear various components of the workspace. The Edit options include:

- **Settings:** opens a configuration panel where users can modify parameters related to data processing, visualization, or model training.

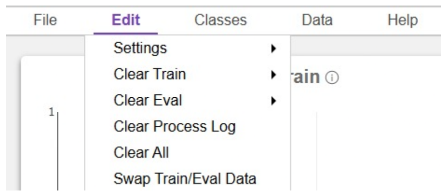


Figure 6. The "Edit" drop-down menu in the Menu Bar, displays options to adjust settings and clear various components of the workspace.

- **Clear Train:** expands to provide options for clearing specific elements of the training dataset.
- **Clear Eval:** expands to provide options for clearing specific elements of the evaluation dataset.
- **Clear Process Log:** removes all entries from the process log, resetting the recorded actions and messages.
- **Clear All:** clears all datasets, logs, and other stored information in the session, resetting the workspace to its initial state.
- **Swap Train/Eval Data:** swaps the datasets used for training and evaluation.

These options allow users to manage their workspace effectively by resetting datasets, removing log entries, or adjusting system settings.

3.2.3. Classes

Figure 7 depicts the process for adding a new class. When hovering over "Classes" in the Menu Bar, an option appears allowing the user to add a new class or select options for an existing class. After hovering over Classes and selecting the Add Class option, the user is prompted to name the new class and select a color for plotting. Once a class is created, it will appear in the dropdown menu when hovering over Classes.

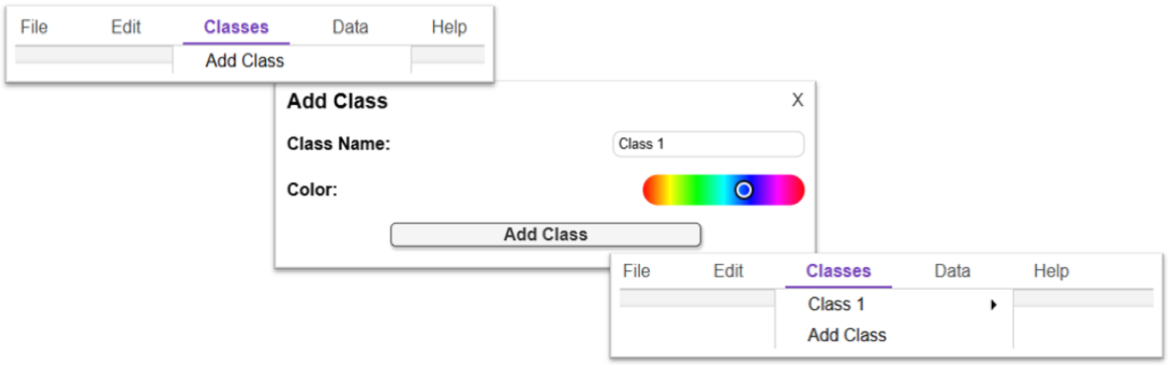


Figure 7. Workflow for adding a new class is shown: the user selects Add Class, enters a class name, chooses a color using the color picker, and confirms by clicking Add Class.

When hovering over the class name in the drop down, a second dropdown appears containing the following:

- **Delete Class:** this function not only erases any drawn data on the train and eval plots but also deletes the entire class. Users can create, delete and modify classes interactively.
- **Draw Points:** when this option is selected the user can manually input data by clicking or dragging their mouse across either the Train or Eval plot windows. As shown in Figure 8 in the right side image, each mouse click or motion will generate individual data points corresponding to the currently selected class. This feature is very useful for creating data sets with specific characteristics that expose strengths and weaknesses of an algorithm.
- **Draw Gaussian:** this option allows users to draw data distributed according to a Gaussian distribution instead of individual points. Once selected, users can click on the plot to define the center (mean) of a Gaussian distribution. As shown in Figure 8 (left), each motion plots more points centered around the cursor.

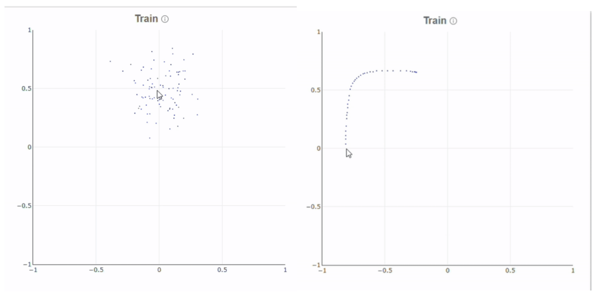


Figure 8. Examples of data input modes: The left plot shows a Gaussian-distributed dataset generated using the Draw Gaussian option, the right plot displays individual points manually drawn with the Draw Points tool.

Note that the drawing features can be used to add new data to existing data sets, including the data sets described below. This makes it very easy to create unique and interesting data sets, and to demonstrate special features of the algorithms.

3.2.4. Data

When hovering over "Data" in the Menu Bar, shown in Figure 9, a dropdown menu appears containing options for several preset data distributions. Once hovering over the desired data distribution, a second dropdown menu will appear where you can select whether you want this distribution to be used for training data (Train) or evaluation data (Eval). The preset data distributions were chosen because they have historical importance to the machine learning community. New data generators can be easily added, as explained in Section 6.

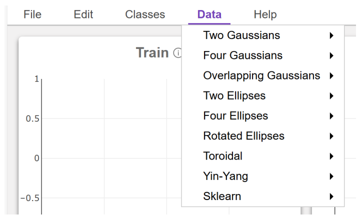
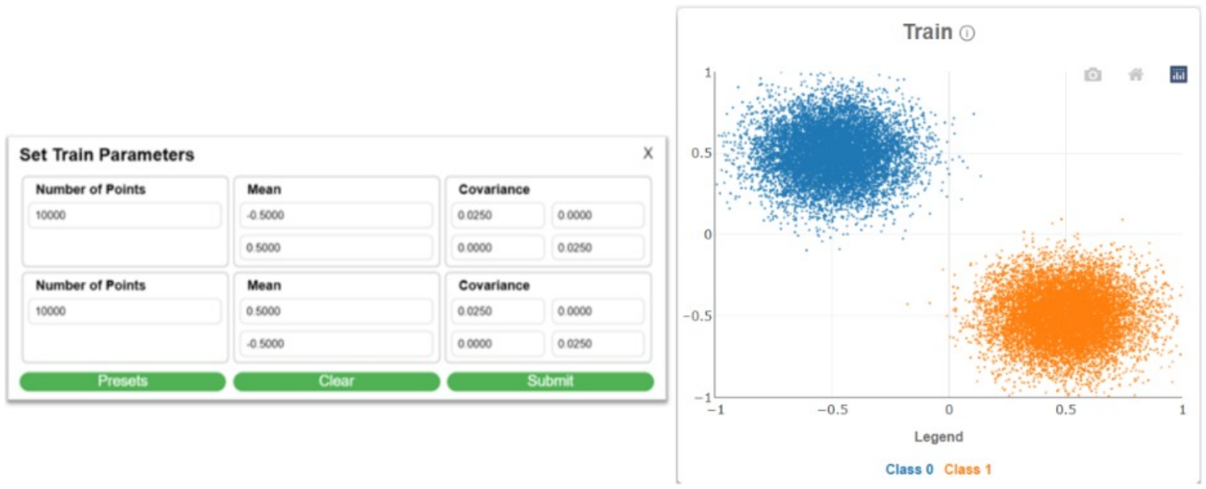


Figure 9. The "Data" drop-down menu in the Menu Bar, displays options to generate some interesting and useful predefined data sets.

The first choice on the menu, show in Figure 10, is referred to as the Two Gaussians data set, and consists of two distinct clusters of points, each sampled from a Gaussian (Normal) distribution. These clusters are



Set Train Parameters

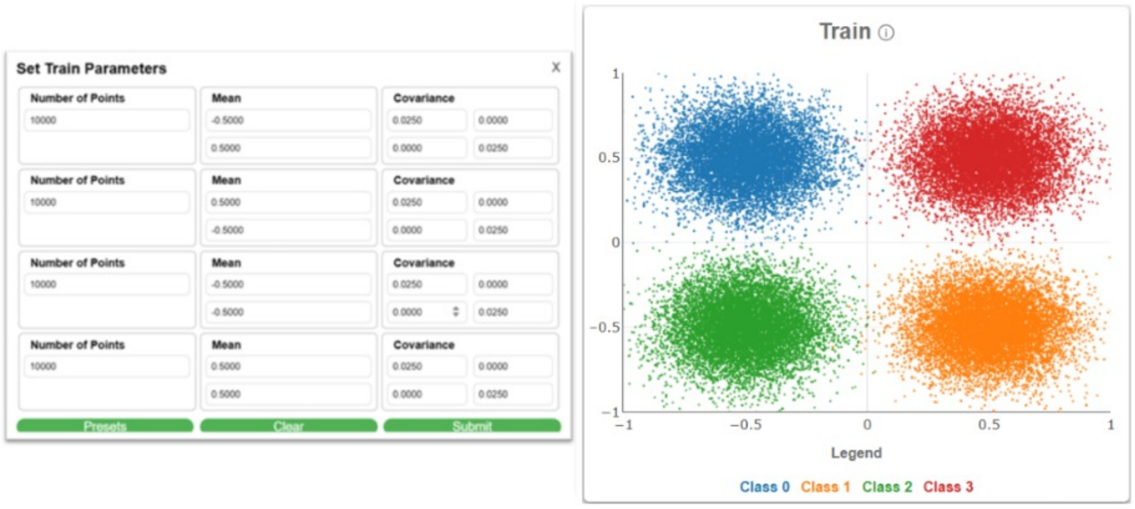
Number of Points	Mean	Covariance	
10000	-0.5000	0.0250	0.0000
	0.5000	0.0000	0.0250
10000	0.5000	0.0250	0.0000
	-0.5000	0.0000	0.0250

Figure 10. An example of the parameter input window for the Two Gaussians dataset

typically placed at separate means with minimal overlap, making them linearly separable and ideal for demonstrating basic classification boundaries. The left image displays the parameters users can control such as the number of points (npts), the mean (location of the center of each class), and the covariance matrix (which controls the spread or shape).

The second choice, Four Gaussians, introduces additional complexity by generating four distinct clusters, each sampled from a Gaussian distribution. As displayed in Figure 11, instead of just two linearly separable groups, this version includes four class labels that are placed at different means with minimal overlap. This allows users to explore how machine learning algorithms handle multi-class classification problems and more intricate decision boundaries. Just like in the Two Gaussians dataset, users can configure the number of points, the mean (center location), and the covariance matrix (which shapes each distribution) for each class, giving them full control over the position, density, and orientation of each class.

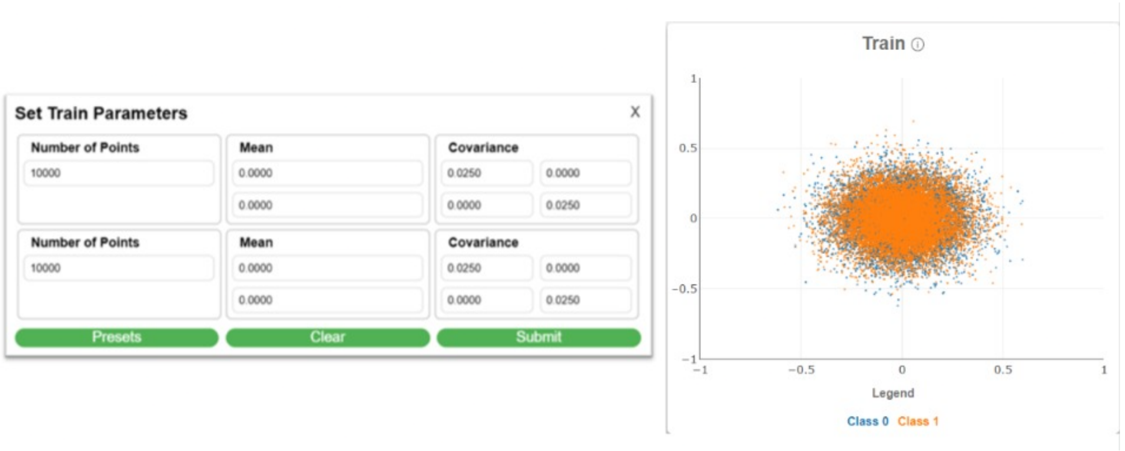
The Overlapping Gaussians dataset, shown in Figure 12, is designed to demonstrate situations where class boundaries are not clean. It consists of two Gaussian clusters where the means and variances are set so that there is significant overlap between the clusters. This creates ambiguity between classes and is ideal for



Set Train Parameters

Number of Points	Mean	Covariance	
10000	-0.5000	0.0250	0.0000
	0.5000	0.0000	0.0250
10000	0.5000	0.0250	0.0000
	-0.5000	0.0000	0.0250
10000	-0.5000	0.0250	0.0000
	-0.5000	0.0000	0.0250
10000	0.5000	0.0250	0.0000
	0.5000	0.0000	0.0250

Figure 11. An example of the parameter input window for the Four Gaussians dataset



Set Train Parameters

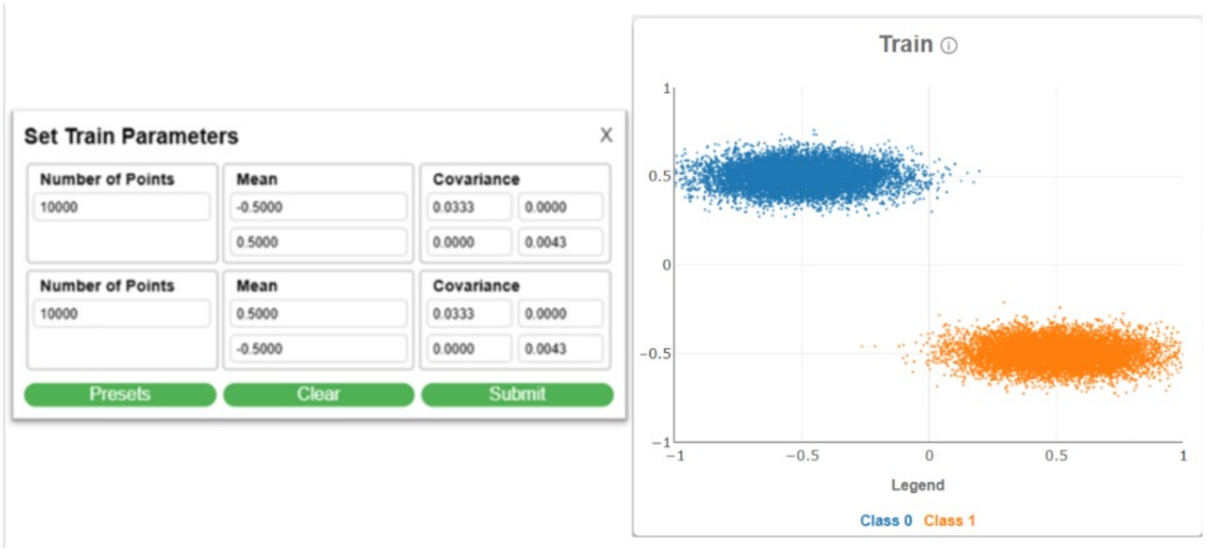
Number of Points	Mean	Covariance	
10000	0.0000	0.0250	0.0000
	0.0000	0.0000	0.0250
10000	0.0000	0.0250	0.0000
	0.0000	0.0000	0.0250

Figure 12. An example of the parameter input window for the Overlapping Gaussians dataset

testing how well a classifier manages uncertainty. Users can adjust the distance between means, the spread (covariance), and the number of samples, allowing them to control how much the distributions intersect.

The Two Ellipses dataset shown in Figure 13 features two class clusters shaped like ellipses rather than circles, indicating anisotropic (non-uniform) Gaussian distributions. These ellipses are defined by their covariance matrices, which stretch the data in specific directions. This dataset is useful for visualizing how algorithms that assume spherical clusters perform when that assumption is violated. Parameters include means, covariances (to control orientation and elongation), and number of points per class.

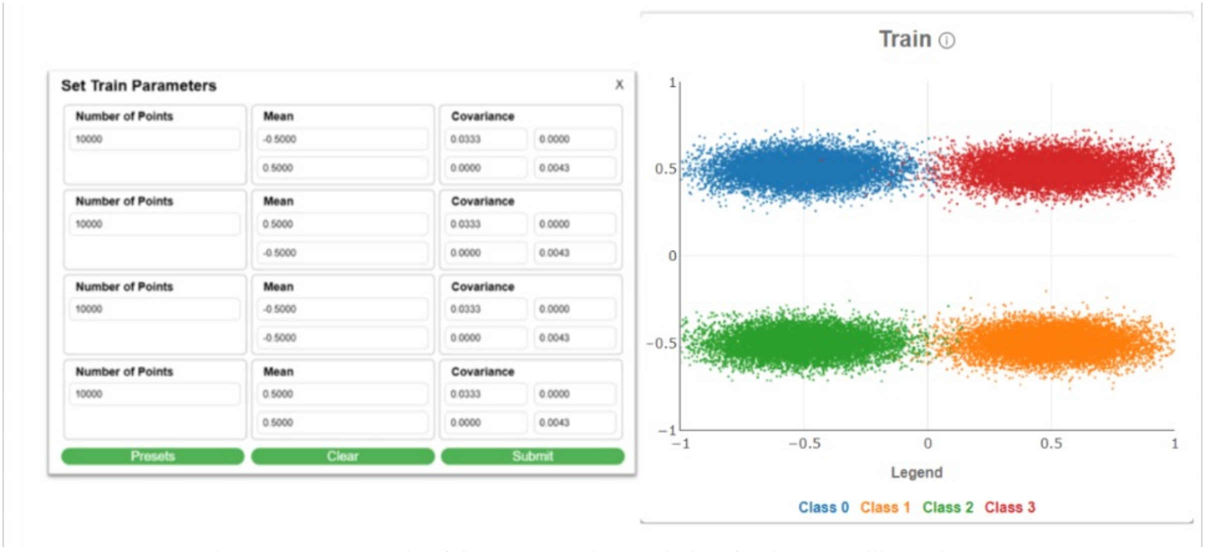
The Four Ellipses dataset, displayed in Figure 14, is an extension of this that introduces four classes, requiring a slightly more sophisticated decision surface. The Rotated Ellipses dataset, displayed in Figure 15, uses Gaussian distributions with non-diagonal covariances matrices. This introduces even more complexity since the direction of variance is not co-aligned with the direction of discrimination. Parameters include the mean, covariance and number of points, allowing users to rotate the ellipses in any direction and analyze classification behavior on skewed data.



Set Train Parameters

Number of Points	Mean	Covariance	
10000	-0.5000	0.0333	0.0000
	0.5000	0.0000	0.0043
10000	0.5000	0.0333	0.0000
	-0.5000	0.0000	0.0043

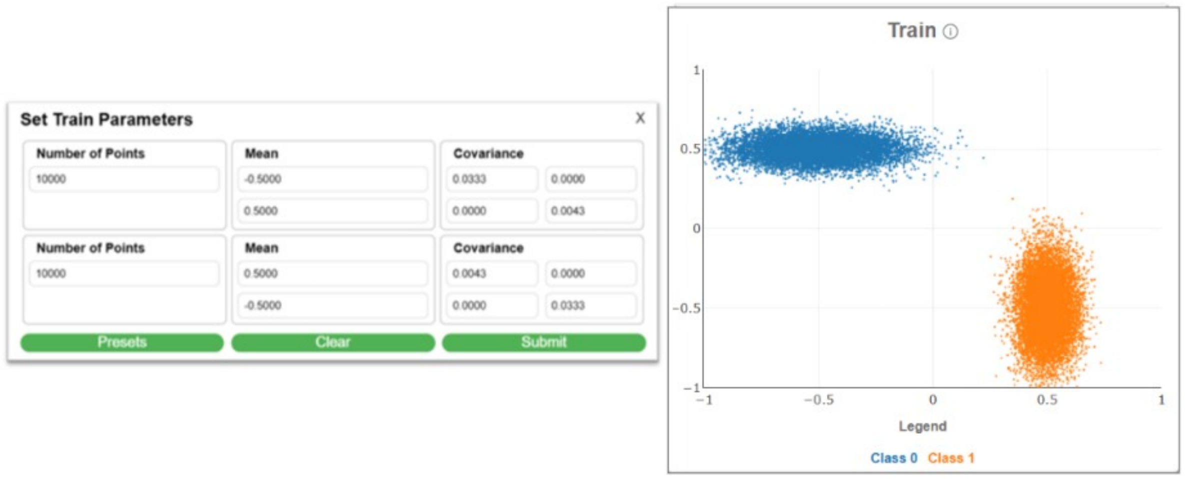
Figure 13. An example of the parameter input window for the Two Ellipses dataset



Set Train Parameters

Number of Points	Mean	Covariance	
10000	-0.5000	0.0333	0.0000
	0.5000	0.0000	0.0043
10000	0.5000	0.0333	0.0000
	-0.5000	0.0000	0.0043
10000	-0.5000	0.0333	0.0000
	-0.5000	0.0000	0.0043
10000	0.5000	0.0333	0.0000
	0.5000	0.0000	0.0043

Figure 14. An example of the parameter input window for the Four Ellipses data set



Set Train Parameters

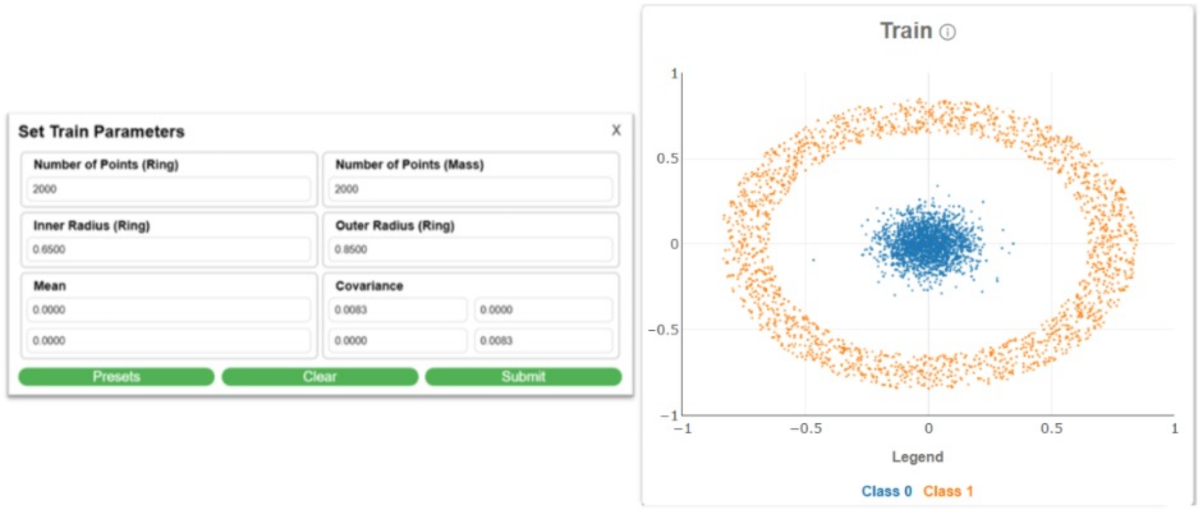
Number of Points	Mean	Covariance	
10000	-0.5000	0.0333	0.0000
	0.5000	0.0000	0.0043
10000	0.5000	0.0043	0.0000
	-0.5000	0.0000	0.0333

Figure 15. An example of the parameter input window for the Rotated Ellipses dataset

The Toroidal dataset displayed in Figure 16, also known as the donut-shaped dataset, arranges data points in a ring or circular pattern. Typically, one class is concentrated in the center while another wraps around it in a circular band. This nonlinearly separable structure is a good test for algorithms that handle complex boundaries. Parameters include the inner and outer radii of the torus, number of samples, and class assignment rules, which determine how inner and outer rings are classified.

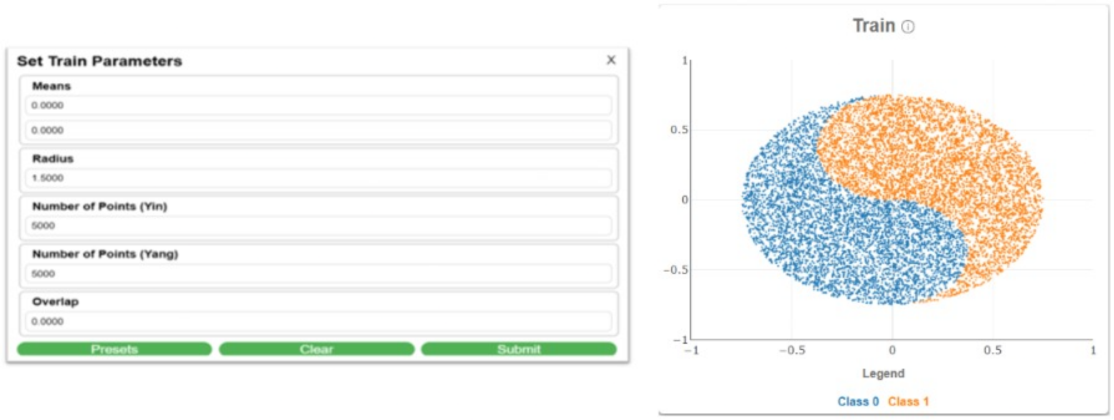
The Yin-Yang dataset introduces a visually complex, spiral-based structure inspired by the iconic Yin-Yang symbol, where two classes intertwine in a curving, symmetrical pattern. This dataset is particularly useful for testing how well algorithms can handle non-linearly separable and non-convex(a shape that a single straight line can't cleanly separate) classification challenges. As shown in Figure 17, users can customize the dataset using several parameters: the means control the center location of the spiral; the radius determines the overall scale of the Yin-Yang shape; and the number of points for each class (Yin and Yang) lets users balance the dataset or simulate class imbalance. Additionally, the overlap parameter allows users to adjust how closely the two spirals intertwine, introducing varying levels of classification difficulty. This dataset is ideal for exploring the strengths and weaknesses of both linear and nonlinear classifiers in a visually intuitive format.

Scikit-Learn's "make_classification" (Sklearn in IMLD) is a data generator that is used to quickly create artificial datasets that mimic the structure of real-world classification problems. Instead of collecting and


Set Train Parameters

Number of Points (Ring): 2000
Number of Points (Mass): 2000
Inner Radius (Ring): 0.6500
Outer Radius (Ring): 0.8500
Mean: 0.0000, 0.0000
Covariance: 0.0083, 0.0000, 0.0000, 0.0083

Figure 16. An example of the parameter input window for the Toroidal dataset


Set Train Parameters

Means: 0.0000, 0.0000
Radius: 1.5000
Number of Points (Yin): 5000
Number of Points (Yang): 5000
Overlap: 0.0000

Figure 17. An example of the parameter input window for the Yin-Yang dataset

cleaning real data (which can be time-consuming), you can use this tool to produce feature matrices (X) and label vectors (y) that follow controllable patterns. With this data generator, you can control the features that are useful for predicting a class, which are redundant and ones that are just uninformative and noisy. You can also control aspects such as the clusters per class, splitting them into multiple Gaussians and even controlling the class balance.

3.2.5. Help

When hovering over "Help" in the Menu Bar, a dropdown menu will appear with options to assist in navigating and using the IMLD website, as displayed in Figure 18. The Help options include the following:

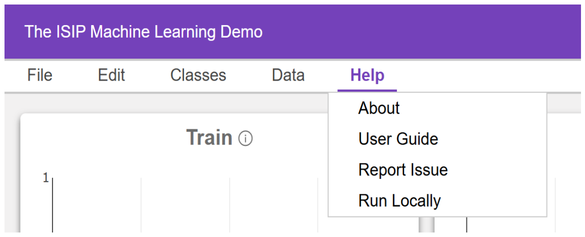


Figure 18. The "Help" drop-down menu in the Menu Bar, displays options for receiving help, including displaying this user guide.

- **About:** selecting this option will open a window that gives a brief explanation of the IMLD app, as well as the current operating version.
- **User Guide:** displays this user manual as a pdf file.
- **Report Issue:** selecting this option opens a dialog box which includes two main input fields. An Issue Title, which is a short summary or heading for the issue being reported, and the Issue Description which is a larger text area where users can describe the issue in more detail.
- **Run Locally:** downloads a copy of the IMLD source code to be ran locally.

Issue reports are sent to the support team at the Neural Engineering Data Consortium (*www.nedcdata.org*). We try to respond to these requests within 24 hours.

3.3. Plotting Windows

The interface contains two side-by-side plotting windows, as shown in Figure 2, that allow users to compare what is called closed-loop performance (training and evaluating on the same data) and open-loop performance (training on the training data and evaluating on the evaluation data). The two windows are labeled as:

- **Train:** displays the training data and the decision surface generated when you press Train. Performance metrics are displayed in the Process Log window at the bottom of the screen.
- **Eval:** displays the evaluation data and how the previously trained model classifies those points when you press Evaluate.

Together, these two visualizations let you see at a glance how well the model generalizes to classify unseen data – data that was not used in the training process.

The small icons in each plotting window, shown in Figure 19, provide some basic interactive controls for the plotting windows. These controls are part of the Python library, Plotly (*https://plotly.com*), that is used to visualize the data and the resulting decision surfaces:

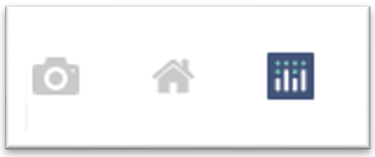


Figure 19. Plot window interactive buttons

- **Camera (Left):** Downloads the current plot as a PNG image.
- **House (Middle):** Resets the axes and zoom to their defaults.
- **Logo (Right):** Opens a link to the Plotly.com web site.

These interactive tools enhance usability, allowing for easier exporting, resetting, and exploration of data.

3.4. Algorithm Toolbar

The Algorithm Toolbar shown in Figure 20 and found on the far-right side of the IMLD interface, is where users select machine learning algorithms, adjust parameters, and initiate training or evaluation of models. This component serves as the main control panel for applying algorithms to the visualized data.

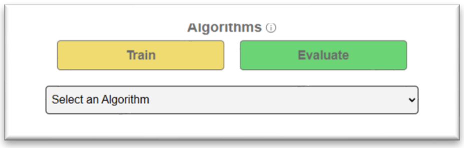


Figure 20. Algorithm tool bar layout

At the top of the toolbar are two prominent buttons: Train and Evaluate. These are used to process data through the selected algorithm. The Train button is used to create a model based on the currently selected algorithm and the training data shown in the Train plot. When clicked, the platform processes the training dataset and builds a model using the specified algorithm and any parameters that have been set.

Once a model is trained, the Evaluate button allows users to apply that trained model to the data displayed in the Eval plot. This step computes the model's performance on new or unseen data, also produces metrics such as accuracy, precision, and error rate, which are logged in the Process Log.

Below the Train and Eval buttons, is the algorithm selection drop down menu shown in Figure 21. IMLD currently supports a broad range of machine learning algorithms, each with its own unique set of parameters. These include popular models such as Principal Component Analysis (PCA), Support Vector Machines (SVM) and K-Nearest Neighbors (KNN), as well as more advanced techniques like Restricted Boltzmann Machines (RBM), Transformers and Quantum Neural Networks (QNN). A total of 17 algorithms are available in the dropdown menu. When an algorithm is selected, the corresponding parameters automatically appear in the Algorithm Toolbar, enabling fine-grained control over behavior, such as adjusting learning rates, selecting solvers, or configuring component counts—depending on the algorithm.

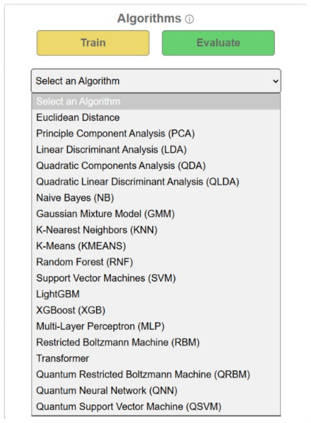

Algorithms
Train
Evaluate
Select an Algorithm
Select an Algorithm
Euclidean Distance
Principle Component Analysis (PCA)
Linear Discriminant Analysis (LDA)
Quadratic Components Analysis (QDA)
Quadratic Linear Discriminant Analysis (QLDA)
Naive Bayes (NB)
Gaussian Mixture Model (GMM)
K-Nearest Neighbors (KNN)
K-Means (KMEANS)
Random Forest (RNF)
Support Vector Machines (SVM)
LightGBM
XGBoost (XGB)
Multi-Layer Perceptron (MLP)
Restricted Boltzmann Machine (RBM)
Transformer
Quantum Restricted Boltzmann Machine (QRBM)
Quantum Neural Network (QNN)
Quantum Support Vector Machine (QSVM)

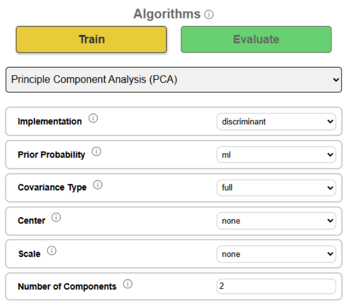
Figure 21. Algorithm selection drop-down menu

For example, Principal Component Analysis (PCA) shown in Figure 22, is a dimensionality reduction technique that transforms potentially correlated variables into a smaller set of uncorrelated variables called principal components. Its goal is to preserve as much variance in the dataset as possible using fewer dimensions. In IMLD, PCA includes several configurable parameters:

- **Implementation:** specifies the backend method used to compute the PCA solution. Options may differ in computational strategy, efficiency, or numerical stability. Selecting the appropriate implementation can impact both runtime and the numerical accuracy of the result.

- **Prior Probability:** determines how prior beliefs about class distributions are incorporated into the analysis. Options like "ml" (Maximum Likelihood) or 'map" (Maximum A Posteriori) allow users to influence model bias based on assumed or known class frequencies.

- **Covariance Type:** controls how the algorithm models variance and correlation across features. A "full"



Principle Component Analysis (PCA)

Implementation	discriminant
Prior Probability	ml
Covariance Type	full
Center	none
Scale	none
Number of Components	2

Figure 22. Principal Component Analysis parameters

covariance type considers all pairwise feature relationships, while simpler alternatives (e.g., “diagonal”) assume features are uncorrelated. This impacts how flexibly the model adapts to data structure.

- **Center:** defines how input data is centered before analysis. Options such as “none”, “tied”, or “untied” determine whether the data is mean-shifted globally, per class, or left as-is. Proper centering is crucial for ensuring meaningful principal component directions.
- **Scale:** specifies the scaling method used to normalize data prior to PCA computation. Options include “none”, “biased”, “unbiased”, and “empirical’, each reflecting a different statistical approach to variance estimation. Scaling ensures features with larger numeric ranges don't dominate the analysis.
- **Number of Components:** sets the number of principal components to retain in the reduced representation. A value of 2, for example, projects the data into two dimensions, preserving only the two directions of maximum variance. This directly controls the dimensionality of the output space.

PCA is a well-known algorithm for doing simple classification tasks and is often one of the first algorithms students will be introduced to in a machine learning course.

3.5. Process Log

The Process Log, located at the bottom of the IMLD interface shown in Figure 2, provides a detailed, step-by-step summary of all actions taken during a session. This includes data generation, class creation, algorithm selection, parameter configuration, model training, and evaluation results. It is updated in real time, offering users a transparent record of their workflow that can be used for debugging, experimentation, and documentation.

When a user creates new classes, the log immediately confirms each addition (e.g., Added class: Class 0). This is followed by detailed summaries whenever a dataset is generated or loaded, including the dataset type (e.g., Two Gaussian, Toroidal, Yin-Yang) and whether it was applied to the Train or Eval window. For each class, the log displays key parameters such as the number of points, the class means, and class covariance matrices.

Once a machine learning algorithm is selected, the log lists the algorithm name and a complete breakdown of its parameter settings, as specified in the algorithm toolbar. This ensures that all user configurations are preserved and visible for later reference. When the user clicks the Train button, the log captures the moment training begins (Process: Train) and may optionally display learned model parameters such as estimated means, priors, or cluster centers, depending on the algorithm.

Following training, the log presents a standardized set of performance metrics: Accuracy, Error Rate, F1 Score, Precision, Sensitivity (Recall), and Specificity.

Clicking the Evaluate button triggers a similar process for the evaluation dataset. The log reports (Process: Eval) and displays the same set of performance metrics, allowing users to compare generalization between training and evaluation phases. In cases where training or evaluation fails (e.g., due to missing data, incompatible configurations, or model errors), the log provides a descriptive error message to help users identify and resolve the issue. By maintaining a comprehensive record of each session, the process log supports reproducibility, comparative testing, and transparent analysis – making it an essential tool for both new learners and advanced users.

4. Algorithms

There are 19 popular machine learning algorithms currently supported in IMLD. These can be roughly clustered into five groups: (1) discriminant, (2) non-parametric, (3) gradient-boosting, (4) neural network and (5) quantum computing. The first two groups are traditionally found in many entry-level machine learning courses. The next two groups are relatively more recent additions and reflect the modern emphasis on neural networks and deep learning. The last group is relatively new and reflects an exciting emerging area of science known as quantum machine learning. These algorithms are experimental in nature and are quite computationally expensive. They are implemented using a quantum computer simulator.

4.1. Discriminant Algorithms

These algorithms are variants of the popular algorithm PCA that attempt to model data using a generative Gaussian distribution-based model.

4.1.1. Euclidean Distance (EUCL)

Euclidean Distance is a simple, intuitive classification method that assigns data points to the class of the closest reference point based on straight-line (L2) distance. While it performs well on linearly separable datasets with evenly spaced data, it is limited in its ability to handle complex, overlapping, or high-dimensional data distributions.

Parameters:

- **Implementation:** Specifies the computational framework used (e.g., "discriminant"), which refers to a basic implementation focusing on class separation based on distance metrics.
- **Weights:** Assigns a weighting factor to each class. This parameter allows for class-specific influence during the classification process. Adjusting these values can simulate class imbalance or prioritize certain classes over others.

4.1.2. Principal Component Analysis (PCA)

Principal Component Analysis is a statistical technique used to reduce the dimensionality of a dataset while preserving as much variance as possible. It works by transforming the original features into a new set of uncorrelated variables called principal components, which are ordered by the amount of variance they capture.

Parameters:

- **Implementation:** specifies the computational method used (e.g., discriminant-based);
- **Prior Probability:** controls how class priors are applied (ml for maximum likelihood, map for maximum a posteriori);
- **Covariance Type:** controls assumptions about the covariance matrix (full: the matrix is fully populated and symmetric, diagonal: assumes the features are decorrelated);
- **Center:** determines how input data is centered (none, tied, untied);
- **Scale:** selects the normalization method applied to input data (none, biased, unbiased, or empirical);
- **Number of Components:** controls how many principal components are retained (e.g., 2 for 2D projection).

4.1.3. Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis is a supervised learning algorithm used for both classification and dimensionality reduction. Unlike PCA, which seeks directions of maximum variance, LDA aims to find the feature space that best separates the classes. It does this by maximizing between-class variance while minimizing within-class variance, resulting in optimal linear decision boundaries for classification tasks. LDA is especially effective when class distributions are approximately Gaussian and share a common covariance structure.

Parameters:

- **Implementation:** specifies the algorithmic backend used for computation;
- **Prior Probability:** controls how class priors are applied (ml or map);
- **Covariance Type:** sets the covariance structure (full or diagonal);
- **Center:** determines how data is centered before analysis (none, tied, untied);
- **Scale:** selects the normalization method applied to input data (none, biased, unbiased, or empirical);
- **Number of Components:** controls how many principal components are retained (e.g., 2 for 2D projection).

4.1.4. Quadratic Components Analysis (QDA)

Quadratic Components Analysis is a supervised classification algorithm that extends Linear Discriminant Analysis (LDA) by allowing each class to have its own unique covariance matrix. This added flexibility enables QDA to model more complex, curved decision boundaries, making it especially effective for problems where class distributions differ in shape, scale, or orientation. QDA calculates class-specific probability distributions and assigns new observations to the class with the highest posterior probability. Its capacity to model class-specific variances and covariances leads to quadratic decision surfaces, giving it a significant advantage over linear models in cases with nonlinearly separable data.

Parameters:

- **Implementation:** specifies the backend method used for computation (e.g., discriminant);
- **Prior Probability:** controls how class priors are applied (ml or map);
- **Covariance Type:** sets the covariance structure (full or diagonal);
- **Center:** determines how the data is centered before analysis (none, tied, or untied);
- **Scale:** selects the normalization method applied to input data (none, biased, unbiased, or empirical);
- **Number of Components:** controls how many principal components are retained (e.g., 2 for 2D projection). This typically matches the number of classes minus one, but QDA can support more due to its nonlinear nature.

4.1.5. Quadratic Linear Discriminant Analysis (QLDA)

Quadratic Linear Discriminant Analysis is a supervised algorithm that extends the principles of LDA by introducing greater flexibility in how it models class distributions. While LDA assumes that all classes share a common covariance structure, QLDA relaxes this assumption – allowing the algorithm to approximate nonlinear (curved) decision boundaries while maintaining the interpretability and computational efficiency of linear methods. This makes QLDA particularly valuable in cases where the class distributions vary modestly but not enough to require a fully nonlinear model like QDA.

Parameters:

- **Implementation:** specifies the discriminant-based solver used to compute the projection;
- **Prior Probability:** controls how class priors are applied (ml or map);
- **Covariance Type:** sets the covariance structure (full or diagonal);
- **Center:** determines how the data is centered before analysis (none, tied, or untied);
- **Scale:** selects the normalization method applied to input data (none, biased, unbiased, or empirical);
- **Number of Components:** controls the number of projection components to retain, usually corresponding to the number of classes minus one.

4.1.6. Naive Bayes (NB)

Naive Bayes is a simple, yet effective supervised learning algorithm based on Bayes' Theorem, which calculates the probability that a data point belongs to a particular class given its features. The "naive" assumption refers to the algorithm treating all features as conditionally independent given the class label, which simplifies computation and often works surprisingly well in practice. Despite its simplicity, Naive Bayes performs competitively on many classification tasks, especially those involving text, document categorization, or other high-dimensional data. It is particularly well suited to problems where the independence assumption is approximately valid, or the dataset is noisy and sparse.

Parameters:

- **Implementation:** specifies the computational backend (e.g., sklearn) used for fitting the model;
- **Prior Probability:** controls how class priors are applied (ml or map).

4.1.7. Gaussian Mixture Model (GMM)

A Gaussian Mixture Model (GMM) is a generative, unsupervised learning algorithm that models the underlying data distribution as a weighted sum of Gaussian distributions. Each component, or mixture, represents a cluster and the overall distribution is a weighted sum of these components. Unlike KMN, which assigns points to clusters based on distance, GMM assigns points based on probability, allowing soft clustering and more flexible cluster shapes. GMMs are particularly useful for identifying subpopulations within complex datasets and can model elliptical clusters with variable sizes and orientations.

Parameters:

- **Implementation:** specifies the method used for fitting the model (e.g., em for Expectation-Maximization);
- **Prior Probability:** determines how class priors are estimated (ml for maximum likelihood);
- **Number of Components:** specifies how many Gaussian components (clusters) the model should fit;
- **Random State:** sets the seed for initialization, ensuring reproducibility across runs.

4.2. Non-Parametric Algorithms

Nonparametric algorithms can be quite powerful but are susceptible to overtraining. IMLD provides a very rich environment in which these issues can be explored via powerful visualizations.

4.2.1. K-Nearest Neighbors (KNN)

K-Nearest Neighbors is a non-parametric, supervised learning algorithm that classifies data points based on the class assignments of the closest training samples. When given a new data point, KNN identifies the K closest labeled training samples - typically using Euclidean distance – and assigns the class that is most common among those neighbors. KNN is intuitive, simple to implement, and does not involve any internal model training, making it ideal for establishing upper bounds on performance.

Parameters:

- **Implementation:** specifies the computational library used to execute the algorithm (e.g., sklearn).
- **Number of Neighbors:** determines the number of nearest neighbors (K) used for majority voting. Common values range from 1 to 10, depending on the size and density of the dataset.

4.2.2. K-Means (KMN)

K-Means is a widely used unsupervised learning algorithm for clustering data into K groups based on similarity. It works by iteratively assigning each data point to the nearest cluster center (centroid), then recalculating the centroids based on the assigned points until convergence. The result is a partitioning of the dataset into K compact, non-overlapping clusters with minimal intra-cluster variance. K-Means is fast, scalable, and particularly effective when the data has distinct spherical clusters of similar size.

Parameters:

- **Implementation:** specifies the underlying framework used to perform clustering (e.g., sklearn);
- **Number of Clusters:** sets the number of clusters (K) that the algorithm will attempt to find;
- **Number of Initializations:** defines how many times the algorithm will be run with different centroid seeds; the best result (lowest within-cluster variance) is selected;
- **Random State:** sets the seed for reproducibility of centroid initialization;
- **Maximum Iterations:** limits the number of iterations for convergence in each run.

4.2.3. Random Forest (RNF)

Random Forest is a powerful ensemble classifier that builds multiple decision trees during training and merges their results to improve accuracy and control overfitting. Each tree is trained on a different bootstrap sample of the data, and features are randomly selected at each split, making the forest robust to noise and variability. It’s particularly well-suited for classification tasks where interactions between features are complex or non-linear. Because the final decision is based on majority voting (classification) or averaging (regression), RNF tends to generalize well without extensive hyperparameter tuning.

Parameters:

- **Implementation:** specifies the framework for execution (e.g., sklearn);
- **Number of Estimators:** sets the number of trees to grow in the forest;
- **Maximum Depth:** limits the depth of each decision tree, helping prevent overfitting;
- **Criterion:** chooses the function used to measure the quality of splits (gini for Gini impurity or entropy for information gain);
- **Random State:** controls randomness in sampling and feature selection for reproducible results.

4.2.4. Support Vector Machines (SVM)

Support Vector Machines are a robust, high-performance classification method that works by finding the optimal hyperplane that maximizes the margin between two classes. SVM is particularly effective for both linearly and nonlinearly separable data through the use of kernel functions, which transform data into higher dimensions where separation is more feasible. SVMs are known for their strong theoretical foundation and are often the go-to method when high accuracy is needed, especially in high-dimensional spaces.

Parameters:

- **Implementation:** specifies the underlying library used to implement the algorithm (e.g., sklearn), which affects how the model is trained and evaluated;
- **Regularization Parameter (c):** balances the trade-off between maximizing the margin and minimizing training error. A smaller C increases regularization, promoting simpler models that generalize better. A larger C reduces regularization, allowing the model to fit the training data more closely.

- **Kernel Coefficient (Gamma):** determines how much influence a single training example has. Lower values imply a wider, more generalized decision boundary, while higher values lead to tighter, more complex boundaries. This parameter is only applicable for non-linear kernels like rbf, poly, or sigmoid.
- **Kernel:** specifies the kernel used to transform the input space. Options include linear (no transformation), and rbf, poly, and sigmoid, nonlinear transformations that allow the model to learn complex boundaries in high-dimensional feature spaces.

4.3. Gradient-Boosting Algorithms

These algorithms use a technique called gradient-boosting for their training method. It gives a prediction model in the form of an ensemble of weak prediction models, i.e., models that make very few assumptions about the data, which are typically simple decision trees.

4.3.1. LightGBM

LightGBM (Light Gradient Boosting Machine) is a high-performance gradient boosting framework developed to handle large datasets with low memory usage and fast training speeds. Unlike traditional boosting methods, it uses histogram-based algorithms and grows decision trees leaf-wise (best-first) rather than level-wise, enabling it to capture complex patterns more efficiently. LightGBM supports classification, regression, and ranking tasks, and includes built-in mechanisms for handling imbalanced data. Its scalability, speed, and accuracy make it particularly effective for large-scale machine learning problems where computational efficiency is critical.

Parameters:

- **Implementation:** specifies the backend used to run the model. *LightGBM* is a fast, efficient gradient boosting framework optimized for large datasets and lower memory usage.
- **Bagging Fraction:** defines the proportion of data used to train each boosting iteration. Using a value less than 1.0 enables bagging, which helps reduce overfitting and improves generalization.
- **Bagging Frequency:** specifies how often bagging is performed (in number of boosting iterations). A value of 0 disables bagging. Higher values perform bagging less frequently**.**
- **Boosting Type:** specifies the boosting method being used.
- **Evaluation Metric:** defines the metric for validation with options such as binary logloss, multi error, and auc.
- **Early Stopping Rounds:** specifies how many rounds with no improvement before training halts. Larger values allow more time but risk overfitting.
- **Validation Fraction:** fraction of training data held out for early stopping. More validation data provides a stronger signal but leaves less for actual training.
- **Is Unbalanced:** if set to 'true', LightGBM automatically adjusts class weights for imbalanced datasets.
- **L1 Regularization:** adds L1 penalty to leaf weights, encouraging sparse solutions by shrinking less important features to zero.
- **L2 Regularization:** adds L2 penalty to leaf weights, smoothing extreme values to reduce overfitting while preserving overall model complexity.
- **Learning Rate**: controls step size shrinkage at each boosting round. Lower rates produce more stable models but require more trees to converge.
- **Max Depth:** limits the depth of each tree. Deeper trees capture complex patterns but can overfit and slow down training.
- **Min Data in Leaf:** specifies the minimum number of samples in a leaf node to avoid creating leaves that overfit on very few observations.

- **Number of Estimators:** total number of trees to build. More trees can improve fit but increase training time and risk overfitting.
- **Number of Leaves:** sets the maximum number of leaves per tree. More leaves allow finer splits but can lead to overly complex models.
- **Random State:** random seed to ensure reproducibility.
- **Solver:** sets the learning objective, automatically setting by default. 'binary' is used for 2-class, 'multiclass' for more, 'regression' for continuous targets.
- **Test Size:** fraction of dataset reserved for testing. Smaller values leave more data for training.
- **Verbosity:** Controls logging detail. Higher values print more training info and diagnostics.

4.3.2. XGBoost (XGB)

XGBoost (Extreme Gradient Boosting) is an optimized gradient boosting framework designed for speed, scalability, and accuracy. It builds an ensemble of decision trees sequentially, where each new tree corrects the errors of the previous ones. It is widely used for both classification and regression tasks and is especially effective for structured/tabular data, ranking problems, and scenarios requiring high predictive performance with efficient computation.

Parameters:

- **Implementation:** defines the backend used to run the model. XGBoost is a scalable and efficient gradient boosting framework commonly used for structured/tabular data.
- **Solver:** the optimization algorithm used to minimize the loss during training. This selection affects convergence speed and final accuracy. Options include popular choices such as "multi: softmax" and "multi: softprob".
- **Evaluation Metric:** defines the validation metric. Options include choices such as "mlogloss" and "error".
- **Early Stopping Rounds:** halts training after no validation improvement for a set number of rounds. Larger values allow more iterations but risk overfitting.
- **Learning Rate:** controls the contribution of each tree to the overall model. Lower rates generalize better but require more boosting rounds.
- **Max Depth:** sets the maximum depth of each tree. Deeper trees capture more complexity but increase overfitting risk.
- **Number of Estimators:** specifies the total number of boosting rounds (trees). More rounds can improve fit but increase training time and risk of overfitting.
- **Number of Threads:** sets the number of CPU threads used for training. Increasing threads speeds up computation on multi-core machines.
- **Random State:** seed used to ensure reproducibility of results.
- **Test Size:** fraction of data held out for testing. Smaller values leave more data available for training.
- **Verbosity:** controls the level of logging output during training. Higher values produce more detailed diagnostics.

4.4. Neural Network-Based Algorithms

We have included some standard neural network algorithms in IMLD that are well-suited to problems involving vector input. Though the theoretical gains achieved by these algorithms on this data are not great, they provide important reference implementations that can be extended to problems where they excel, such as long-term contextual modeling (e.g., ChatGPT).

4.4.1. Multi-Layer Perceptron (MLP)

A Multi-Layer Perceptron is a type of feedforward artificial neural network composed of one or more hidden layers between the input and output. It is a supervised learning algorithm that models complex, nonlinear relationships by learning through backpropagation. MLPs are capable of handling both classification and regression tasks by adjusting their architecture and activation functions. They are particularly effective for datasets where patterns are not easily separable using linear decision boundaries.

Parameters:

- **Implementation:** defines the library or backend used to train the model (e.g., sklearn). Different implementations may offer varying trade-offs in performance, flexibility, or compatibility.
- **Hidden Size:** specifies the number of units or neurons in the hidden layer(s). This determines the capacity of the model to learn complex relationships. A larger hidden size allows the model to capture more intricate patterns but also increases the risk of overfitting.
- **Activation:** determines the nonlinear transformation applied at each neuron. The choice affects how signals propagate through the network and influences learning dynamics. Common choices include:
  - *relu*: efficient and widely used, especially for deep networks.
  - *tanh* or *logistic*: useful in shallower networks or where bounded output is preferred.
- **Solver:** The optimization algorithm used to minimize the loss during training. This selection affects convergence speed and final accuracy. Options include:
  - *adam*: a robust, adaptive method suitable for most tasks.
  - *sgd*: stochastic gradient descent, which may converge more slowly but is highly customizable.
  - *lbfgs*: offers high precision convergence, especially on small to medium-sized datasets.
- **Batch Size:** determines how many samples are processed before updating the model weights. Setting this to auto lets the backend choose an optimal size. Smaller batches lead to faster updates but more variance; larger batches provide more stable updates.
- **Learning Rate:** specifies the policy for adjusting the model's learning rate over time. Common settings include:
  - *constant*: Keeps the learning rate fixed throughout training.
  - *adaptive or invscaling*: Adjusts learning rate as training progresses to fine-tune the model.
- **Initial Learning Rate:** sets the starting step size for the optimization process. This value controls how large each update is during the early stages of training. A value that is too high may cause divergence; a value that is too low may slow learning.
- **Random State:** seeds the random number generator used during initialization and training. This ensures consistent results across repeated runs, improving reproducibility and debugging.
- **Momentum:** used to accelerate training by smoothing updates. It helps the model move consistently in directions that reduce error, avoiding small local minima and speeding up convergence.
- **Validation Fraction:** sets the portion of training data to hold out for validation. This helps monitor overfitting by evaluating performance on unseen data during training.
- **Maximum Iterations:** defines the maximum number of training epochs (complete passes over the data). This prevents excessive runtime and ensures that training halts within a practical timeframe.
- **Shuffle:** if true, the training data is shuffled at each epoch. This reduces bias due to data order and promotes better generalization by exposing the model to different sequences of data.
- **Early Stopping:** if enabled, training halts when the model's performance on the validation set no longer improves. This technique prevents overfitting and saves time by stopping training once optimal learning is reached.

4.4.2. Restricted Boltzmann Machine (RBM)

A Restricted Boltzmann Machine is a generative stochastic neural network used for dimensionality reduction, feature learning, and classification. It consists of a visible layer (input features) and a hidden layer (learned features), with connections only between these two layers, not within them – hence the name "restricted." RBMs learn a probabilistic representation of the input data and can be used as building blocks for deeper models like Deep Belief Networks (DBNs). In IMLD, RBMs are primarily used for unsupervised feature extraction, followed by a classifier (e.g., KNN) for supervised learning. After training, the learned hidden representations are passed to the selected classifier to perform classification tasks.

Parameters:

- **Implementation:** defines the library or backend used to train the model (e.g., sklearn).
- **Classifier:** defines the downstream classifier to apply to the learned features (e.g. KNN).
- **Learning Rate:** controls how fast the model updates weights during training.
- **Batch Size:** sets how many samples are used per training batch. A value of zero corresponds to full-batch training.
- **Verbose:** enables detailed logging output for training diagnostics (0 = off, higher positive values = more info).
- **Random State:** ensures reproducible results by fixing the seed used for random processes.
- **Number of Components:** sets the number of hidden units or features to extract.
- **Maximum Iterations:** limits the number of training passes through the dataset.

4.4.3. Transformer

A Transformer model is a deep learning architecture that relies entirely on self-attention mechanisms to process input data. Originally introduced for natural language processing tasks, Transformers have since been adapted for a variety of domains including classification, sequence modeling, and time-series prediction. Unlike traditional models that rely on recurrence or convolution, Transformers learn relationships between all elements in the input sequence simultaneously, making them especially effective at capturing long-range dependencies. In IMLD, the Transformer is implemented using PyTorch and allows full customization of its architecture, offering flexibility in training and evaluation.

Parameters:

- **Implementation:** selects the backend library used to execute the model (e.g., PyTorch).
- **Epoch:** number of complete passes through the training data.
- **Learning Rate:** determines how quickly the model updates weights during optimization.
- **Batch Size:** number of samples processed before the model's parameters are updated.
- **Embed Size:** dimensionality of the input embeddings that represent each data point.
- **Number of Heads:** defines the number of attention heads in the multi-head attention mechanism, enabling the model to focus on different parts of the data simultaneously.
- **Number of Layers:** controls the depth of the model by setting the number of stacked encoder layers.
- **MLP Dimension:** size of the feed-forward layer following each attention block.
- **Dropout:** introduces regularization to reduce overfitting by randomly setting a fraction of neurons to zero during training.
- **Random State:** sets the seed for reproducibility.
- **Tolerance:** determines the stopping condition for convergence (smaller values require tighter convergence).

- **Validation Fraction:** percentage of data held out for validation during training.
- **Early Stopping:** halts training if performance on validation data stops improving.
- **Max Iterations:** limits the number of training cycles to prevent excessively long training runs.

4.5. Quantum Computing-Based Algorithms

An interesting emerging area of computing is quantum computing. Quantum computers offer the potential to solve certain types of problems very quickly and perhaps find better solutions than conventional machine learning algorithms. In IMLD, we provide three historically important attempts to harness the power of quantum computing for machine learning.

4.5.1. Quantum Restricted Boltzmann Machine (QRBM)

The Quantum Restricted Boltzmann Machine (QRBM) is a hybrid quantum-classical model designed for unsupervised learning and dimensionality reduction using principles from quantum annealing. This model builds upon the structure of a classical Restricted Boltzmann Machine (RBM) but uses a quantum processor to perform optimization tasks. It is particularly well-suited for sampling complex probability distributions and encoding data into lower-dimensional latent spaces. QRBM in IMLD is implemented using the D-Wave quantum platform, which enables annealing-based optimization via binary quadratic models (BQMs). This allows the model to find low-energy representations of input data, mimicking how neural networks compress and interpret features.

Parameters:

- **Implementation Name:** specifies the backend platform, such as “dwave”, which executes the model using D-Wave’s quantum annealer.
- **Provider Name:** identifies the service or interface layer (e.g., “dwave”), which is used to manage access and interaction with the quantum hardware.
- **Encoder Name:** defines the encoding strategy, such as “bqm” (binary quadratic model), which represents the energy function of the RBM in a format solvable by quantum annealing.
- **Number Hidden Units:** controls the number of hidden nodes in the model, which represent latent features extracted from the input data.
- **Number of Shots:** determines how many times the quantum annealer samples from the solution space. More shots provide a better approximation of the probability distribution.
- **Chain Strength:** a hyperparameter that adjusts the penalty for breaking chains of qubits representing logical variables. It affects solution quality and stability.
- **KNN Neighbors:** specifies how many nearest neighbors use in the final classification step after feature extraction, integrating QRBM with a KNN classifier for supervised tasks.

4.5.2. Quantum Neural Network (QNN)

The Quantum Neural Network is a machine learning model that utilizes principles from quantum computing to perform classification and optimization tasks. Unlike classical neural networks that operate on real-valued vectors, QNNs leverage quantum circuits to represent and manipulate data in quantum states. These models are especially well-suited for exploring quantum-enhanced feature spaces and are implemented using frameworks like Qiskit. In IMLD, QNNs are built using parameterized quantum circuits composed of feature maps and variational ansatz layers. These circuits are optimized iteratively using classical optimization algorithms.

Parameters:

- **Implementation Name:** specifies the quantum framework used (e.g., “qiskit”).
- **Provider Name:** defines the service or simulator provider (typically “qiskit”).
- **Hardware:** specifies the execution platform, such as “cpu” for simulation or “qpu” for quantum hardware.
- **Encoder Name:** determines the type of quantum feature map which encodes classical data into quantum states.
- **Number Qubits:** sets the number of qubits used in the quantum circuit, impacting the model's capacity and complexity.
- **Entanglement:** defines the pattern of entanglement among qubits (“full” means all qubits are entangled with one another).
- **Feature Map Repetitions:** controls the number of times the feature map circuit is repeated to enhance representational power.
- **Ansatz Repetitions:** controls how many times the trainable ansatz block is repeated to deepen the circuit.
- **Ansatz Name:** specifies the form of the trainable quantum circuit, such as "real_amplitudes", which applies parameterized rotations.
- **Optimizer Name:** selects the classical optimizer (e.g., “cobyla”) used to tune circuit parameters based on performance.
- **Optimizer Maximum Steps:** limits the number of iterations the optimizer will run, preventing overfitting or long runtimes.
- **Measurement Type:** specifies how measurement results are collected (e.g., “sampler” for probabilistic output).

4.5.3. Quantum Support Vector Machine (QSVM)

The Quantum Support Vector Machine extends classical SVM techniques into the quantum domain by leveraging quantum computing principles such as entanglement and superposition. It maps classical data into a quantum feature space using quantum circuits and then separates data using a kernel-based classification approach. This allows for solving classification problems that may be intractable for classical SVMs on certain datasets. QSVM in IMLD is implemented using Qiskit, a leading quantum computing framework, and can simulate execution on quantum hardware or classical backends.

Parameters:

- **Implementation:** selects the quantum backend library (e.g., Qiskit).
- **Provider Name:** defines the service or simulator provider (typically “qiskit”).
- **Hardware:** specifies the execution platform, such as “cpu” for simulation or “qpu” for quantum hardware.
- **Encoder Name:** determines the type of quantum feature map, such as "zz" for ZZFeatureMap, which encodes classical data into quantum states.
- **Number of Qubits:** sets the number of qubits used in the quantum circuit, impacting the model's capacity and complexity.
- **Entanglement:** defines the pattern of entanglement among qubits (“full” means all qubits are entangled with one another).
- **Feature Map Repetitions:** controls the number of times the feature map circuit is repeated to enhance representational power.
- **Kernel:** specifies the kernel method used to compare quantum states (e.g., “fidelity” measures state overlap).
- **Number of Shots:** indicates how many times the quantum circuit is run to estimate probabilities – higher values yield more stable results but take more computational time.

4.6. Summary

We have attempted to provide a wide range of algorithms that are generally useful to entry-level students interested in understanding how machine learning actually works. As will be discussed in Section 6, it is fairly easy to add new algorithms to IMLD. We expect to continue implementing new algorithms over time as the field evolves. Some of the algorithms described above, such as Transformers, excel when processing long strings of sequential data. However, the implementations provided above are designed to provide fair comparisons of these algorithms on well-known data sets.

5. Typical Use Cases

This section presents a series of typical use cases that walk users through practical workflows. A workflow typically includes dataset creation, algorithm selection, training and evaluation, and interpretation of results. These examples are designed to highlight the strengths and limitations of various algorithms when applied to different data distributions, and to help users develop intuition about how to navigate model selection, parameter tuning, and performance assessment within the IMLD environment.

5.1. PCA vs. QDA for Rotated Ellipses

In this example, we will generate multivariate Gaussian data with an asymmetric covariance structure and explore how well the data can be classified using QDA. To generate Rotated Ellipses in IMLD, navigate to the Data tab in the menu bar and select the Rotated Ellipses option from the dropdown menu, as shown in Figure 23.

After selecting Train or Eval, a parameter configuration window will appear (Figure 24), that allows the user to define the characteristics of the datasets such as class means, covariance matrices, and the number of points.

After selecting "Submit", the resulting data will then be populated in the corresponding Train or Eval plotting windows. The resulting window should resemble Figure 25, displaying two distinct rotated elliptical clusters in both the Train and Eval plotting windows. Their corresponding parameters are displayed in the Process Log.

Next, we will select our desired algorithm. Navigate to the Algorithm Toolbar on the right hand side, and click the drop-down menu titled "Select an Algorithm" and select "Principal Component Analysis (PCA)". After selecting

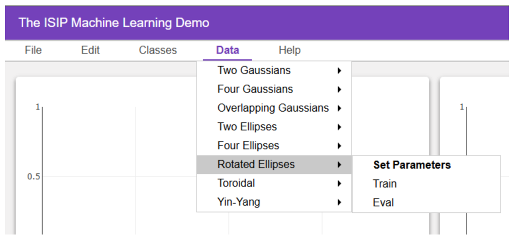
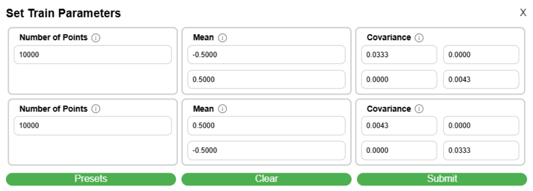
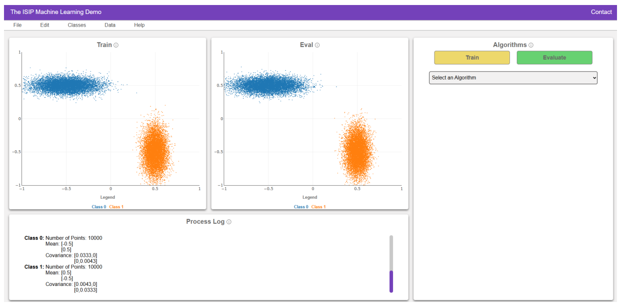


Figure 23. Generation of a Rotated Ellipses dataset


Set Train Parameters

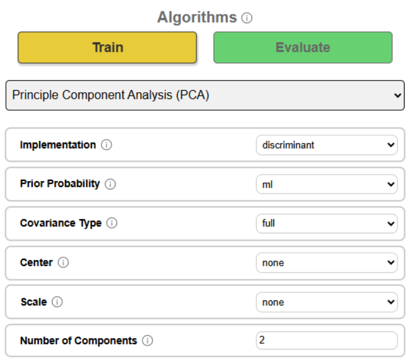
Number of Points	Mean	Covariance	
10000	-0.5000	0.0333	0.0000
	0.5000	0.0000	0.0043
10000	0.5000	0.0043	0.0000
	-0.5000	0.0000	0.0333

Figure 24. Rotated Ellipsis parameter configuration



```
Class 0: Number of Points: 10000
         Mean: [-0.5]
               [0.5]
         Covariance: [0.0333,0]
                     [0,0.0043]
Class 1: Number of Points: 10000
         Mean: [0.5]
               [-0.5]
         Covariance: [0.0043,0]
                     [0,0.0333]
```

Figure 25. Window displaying generated Rotated Ellipses data



Principle Component Analysis (PCA)

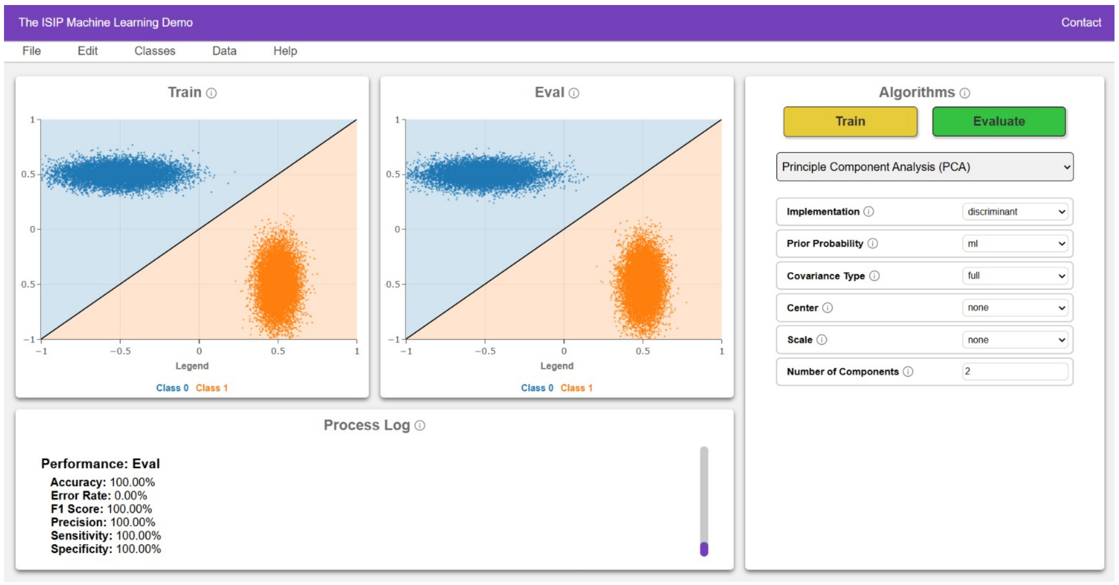
Implementation	discriminant
Prior Probability	ml
Covariance Type	full
Center	none
Scale	none
Number of Components	2

Figure 26. The parameter window for the algorithm 'PCA'

PCA as the algorithm, configure its parameters according to your experimental needs. In this demonstration, we use the default parameters shown in Figure 26. Once configured, click the Train button to apply the PCA transformation to the training data. The Process Log will record details such as class means, prior probabilities, and resulting covariance matrices.

After training, click Evaluate to apply the trained PCA model to the evaluation dataset. The Process Log will now include performance metrics such as accuracy, precision, F1 score, and confusion matrix entries. These results allow the user to assess the model's ability to generalize beyond the training set.

The PCA algorithm creates a linear, or straight, decision boundary by projecting the data onto a new axis. As shown in Figure 27, this works well when the data is already clearly separated. Even though PCA does



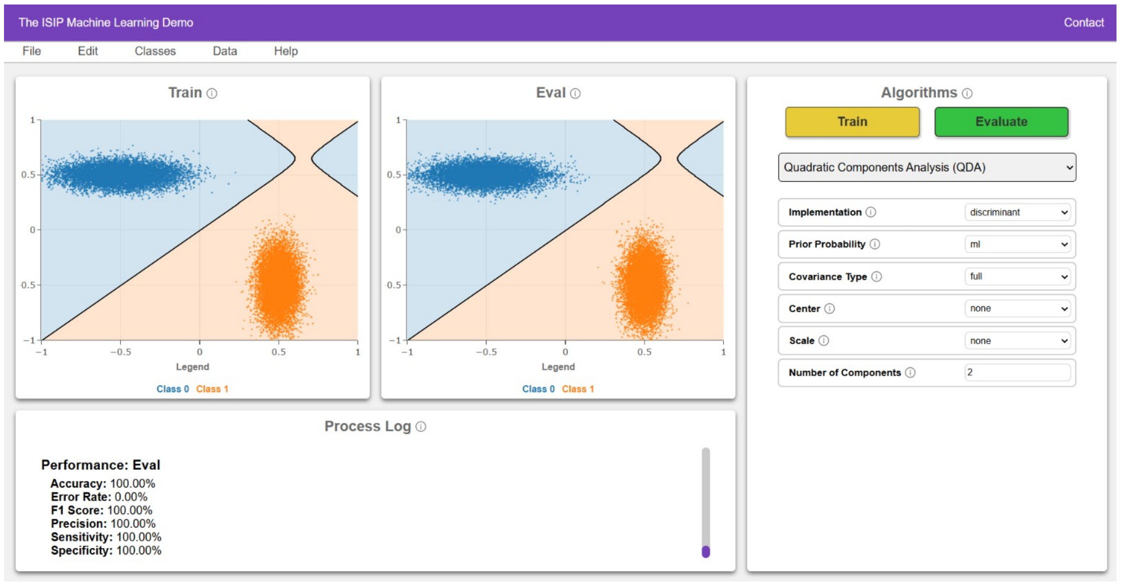
Principle Component Analysis (PCA)

Implementation: discriminant
Prior Probability: ml
Covariance Type: full
Center: none
Scale: none
Number of Components: 2

Process Log

Performance: Eval
Accuracy: 100.00%
Error Rate: 0.00%
F1 Score: 100.00%
Precision: 100.00%
Sensitivity: 100.00%
Specificity: 100.00%

Figure 27. PCA decision boundaries and evaluation metrics



Quadratic Components Analysis (QDA)

Implementation: discriminant
Prior Probability: ml
Covariance Type: full
Center: none
Scale: none
Number of Components: 2

Process Log

Performance: Eval
Accuracy: 100.00%
Error Rate: 0.00%
F1 Score: 100.00%
Precision: 100.00%
Sensitivity: 100.00%
Specificity: 100.00%

Figure 28. QDA Decision boundaries and evaluation metrics

not use class-specific shape or orientation information, it still performs perfectly in this case because the data is clean and well-separated.

Next, repeat the process using Quadratic Discriminant Analysis (QDA) by selecting it from the drop-down menu. As shown in Figure 28, QDA allows each class to have its own shape and spread, which makes it capable of generating more complex decision surfaces. In this case, however, since the data is well separated, both algorithms produce perfect classification.

5.2. QDA vs. RNF for Toroidal Data

To generate Toroidal data in IMLD, navigate to the Data tab in the menu bar and select Toroidal from the dropdown menu (Figure 29). After selecting Train or Eval, a parameter configuration window will appear (Figure 30), allowing the user to define the characteristics of the datasets. After selecting Submit, the resulting data will be populated in the corresponding plotting window. The resulting view should resemble Figure 31, displaying a ring-shaped cluster centered around a circularly shaped inner cluster. All input parameters are displayed in the Process Log.

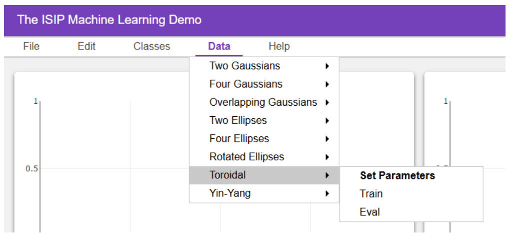
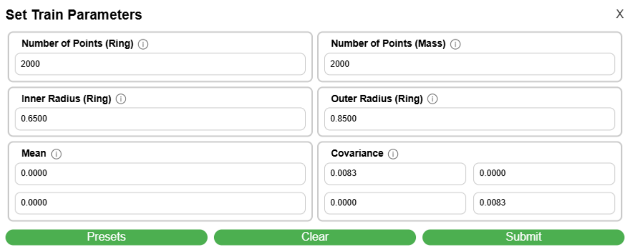


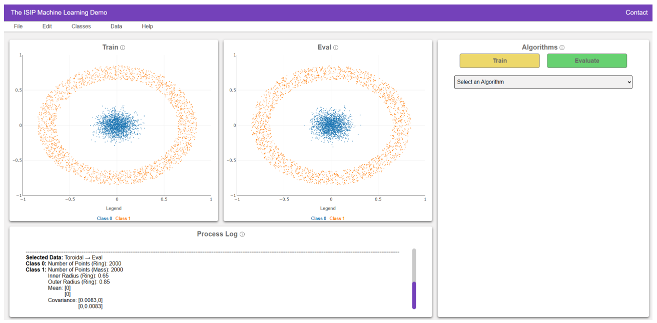
Figure 29. Navigation for the toroidal dataset


Set Train Parameters

Number of Points (Ring): 2000	Number of Points (Mass): 2000
Inner Radius (Ring): 0.6500	Outer Radius (Ring): 0.8500
Mean: 0.0000, 0.0000	Covariance: 0.0083, 0.0000; 0.0000, 0.0083

Figure 30. Toroidal data set parameter configuration

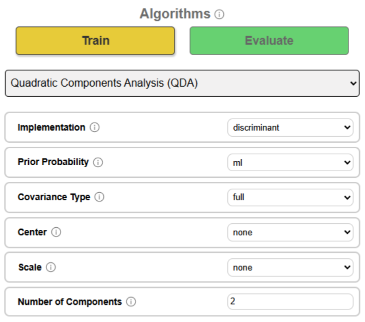
Next, navigate to the Algorithm Toolbar on the right-hand side, open the Select an Algorithm dropdown menu, and choose Quadratic Discriminant Analysis (QDA). Configure the parameters as needed (Figure 32) or use the default settings.



```
Selected Data: Toroidal → Eval
Class 0: Number of Points (Ring): 2000
Class 1: Number of Points (Mass): 2000
         Inner Radius (Ring): 0.65
         Outer Radius (Ring): 0.85
         Mean: [0]
               [0]
         Covariance: [0.0083,0]
                     [0,0.0083]
```

Figure 31. The plotting windows for the toroidal dataset

Click Train to apply QDA to the training dataset. The Process Log will display the resulting class statistics including means, prior probabilities, and the full covariance matrices for each class. Then click Evaluate to apply the trained QDA model to the evaluation set.



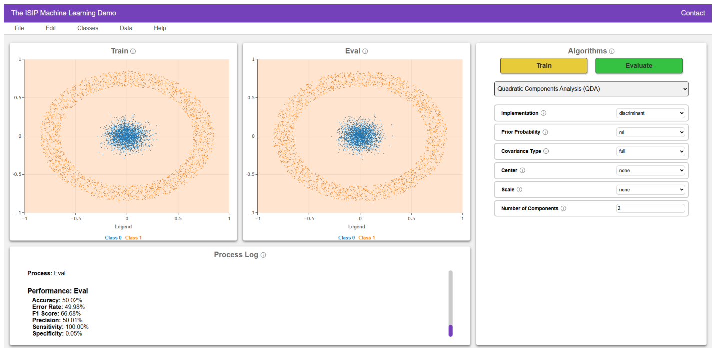
Quadratic Components Analysis (QDA)

Implementation	discriminant
Prior Probability	ml
Covariance Type	full
Center	none
Scale	none
Number of Components	2

Figure 32. The QDA parameter window

The Process Log will now include performance metrics such as accuracy, precision, F1 score, and the confusion matrix. In this demonstration, QDA achieves fails to isolate the two classes, as shown in Figure 33, since it is not well-suited to this type of data.

Now repeat the process using the Random Forest (RNF) algorithm by selecting it from the same dropdown menu. Once selected, click Train to fit the RNF model to the training data, and then Evaluate to apply the model to the evaluation dataset. As shown in Figure 34, RNF forms a very specific decision boundary that closely follows the toroidal contours of each class. The Process Log reports very high accuracy, often exceeding at 99%, along with precise class-specific metrics.



Quadratic Components Analysis (QDA)

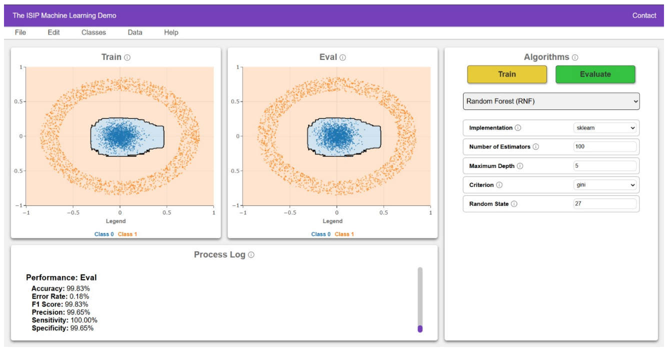
Implementation: discriminant
Prior Probability: ml
Covariance Type: full
Center: none
Scale: none
Number of Components: 2

Process: Eval

Performance: Eval
Accuracy: 50.02%
Error Rate: 49.98%
F1 Score: 66.68%
Precision: 50.01%
Sensitivity: 100.00%
Specificity: 0.05%

Figure 33. QDA decision boundaries for the toroidal dataset

QDA and RNF use fundamentally different approaches to classification, and their performance varies significantly on nonlinear datasets like Toroidal. This use case highlights that while QDA performs well on data with curved but regular shapes (like ellipses), RNF is the better choice when dealing with non-convex or nested topologies such as the toroidal configuration. When interpretability is less critical than performance, RNF's data-driven adaptability offers a clear advantage in both classification accuracy and boundary flexibility.



Random Forest (RNF)

Implementation: sklearn
Number of Estimators: 100
Maximum Depth: 5
Criterion: gini
Random State: 27

Performance: Eval
Accuracy: 99.83%
Error Rate: 0.18%
F1 Score: 99.83%
Precision: 99.65%
Sensitivity: 100.00%
Specificity: 99.65%

Figure 34. RNF decision surfaces

5.3. Creating Custom Data Sets

To create a custom dataset by drawing, first add a class by using the "Add Class" button under the Classes tab (Figure 35). Then, once again hovering over "Classes", select the class you created, and choose Draw Points (for individual clicks) or Draw Gaussian.

In this example, we will be drawing two Gaussian data plots and evaluating using the QDA algorithm. Click or drag inside either the Train or Eval plot to sketch the points you want (Figure 36). When you are satisfied with the sketch, we can save our data.

IMLD supports two drawing modes: points and Gaussian. Gaussian generates data points from a Gaussian window using a paint brush approach. The 'Draw Points' function follows the cursor and lets you insert points in a more granular manner. Existing data sets can be augmented by using these tools to add points. New data sets can also be created starting from a blank canvas.

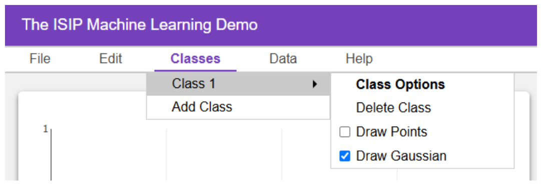
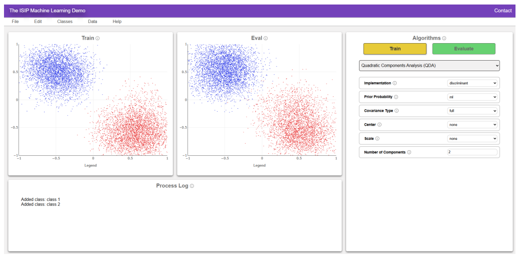


Figure 35. Adding a class to draw Gaussian data

To save the data, go to File → Save Train Data (or Save Eval Data) to export the hand-drawn points as a CSV file (Figure 37). The filename is fixed to the name "imld_train.csv" due to limitations of the client/server interface. Use your desktop tools to rename this file if you want to avoid overwriting it.



Quadratic Components Analysis (QDA)

Implementation: discriminant
Prior Probability: ml
Covariance Type: full
Center: none
Scale: none
Number of Components: 2

Added class: class 1
Added class: class 2

Figure 36. Two manually drawn Gaussian plots

The CSV file structure is shown in Figure 38. The header rows capture class labels, colors, and plot limits so the dataset can be reconstructed later.

To verify the save-and-load workflow, clear the plot (Edit → Clear Train), then load the file you just saved via File → Load Train Data. The points reappear exactly as drawn, confirming that the CSV format preserves all geometry and metadata.

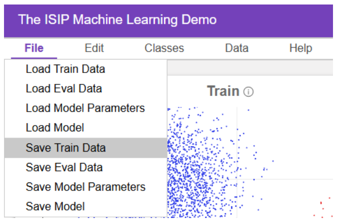
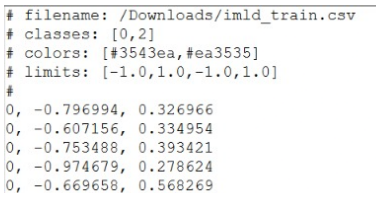


Figure 37. Saving data to a file

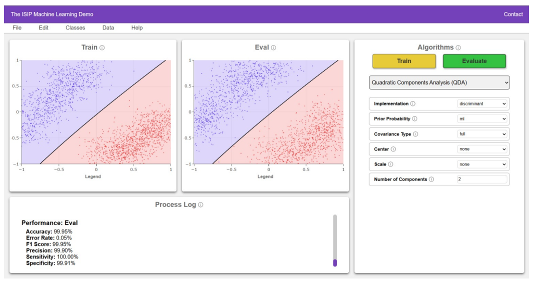


```
# filename: /Downloads/imld_train.csv
# classes: [0,2]
# colors: [#3543ea,#ea3535]
# limits: [-1.0,1.0,-1.0,1.0]
#
0, -0.796994, 0.326966
0, -0.607156, 0.334954
0, -0.753488, 0.393421
0, -0.974679, 0.278624
0, -0.669658, 0.568269
```

Figure 38. The CSV file structure

Next, open the Algorithm Toolbar, select Quadratic Discriminant Analysis (QDA) from the drop-down list. Click Train to fit QDA to your custom drawing; the Process Log now shows class-specific means, priors, and full covariance matrices. Click Evaluate to test on the Eval set (or on the same data if no separate Eval file was loaded). QDA's curved decision surface will bend to match the hand-drawn contours, and the Process Log reports accuracy, precision, recall, and F1 scores for immediate feedback.

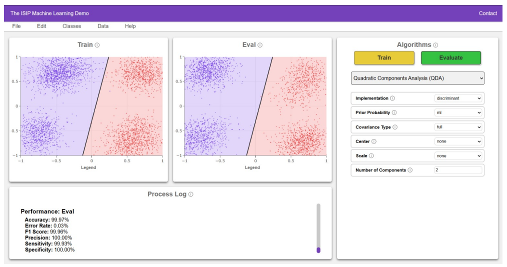
Finally, you may iterate: add extra points, save as a second file, reload, and retrain. With the data loaded, under the "Classes" tab, select the class you'd like to augment, and choose either Draw Points or Draw Gaussian to add new samples to that class. These new points will appear in real time on the plot and be automatically grouped with the selected class. Once the augmentation is complete, save the updated dataset using File → Save Train Data, then retrain the model to observe how the changes affect the classification boundary and performance metrics. This iterative process allows users to explore the effects of class imbalance, added variance, or shifted means, and how QDA adjusts its boundary accordingly through updated class statistics.



Quadratic Components Analysis (QDA)
Implementation: discriminant
Prior Probability: ml
Covariance Type: full
Center: none
Scale: none
Number of Components: 2

Performance: Eval
Accuracy: 99.95%
Error Rate: 0.05%
F1 Score: 99.95%
Precision: 99.90%
Sensitivity: 100.00%
Specificity: 99.91%

Figure 39. QDA classification on the original dataset



Quadratic Components Analysis (QDA)
Implementation: discriminant
Prior Probability: ml
Covariance Type: full
Center: none
Scale: none
Number of Components: 2

Performance: Eval
Accuracy: 99.97%
Error Rate: 0.03%
F1 Score: 99.96%
Precision: 100.00%
Sensitivity: 99.93%
Specificity: 100.00%

Figure 40. QDA Classification on augmented dataset

The first data set shown in Figure 39, demonstrates a dataset with a curved boundary between the classes, which QDA captures well due to its use of class-specific covariance matrices. As shown, QDA achieves nearly perfect performance on this data with 99.95% accuracy and similarly high scores across other metrics – indicating that the model effectively aligns with the data's inherent curvature.

The second, augmented, data set shown in Figure 40 features a revised distribution where the boundary between classes appears more vertical and symmetric. Despite the change in structure, QDA performs remarkably well, achieving 99.97% accuracy. The curved decision boundary continues to model the data effectively, and performance metrics such as precision (100.00%), sensitivity (99.93%), and F1 score (99.96%) remain near perfect.

This comparison demonstrates QDA's versatility. Not only can QDA model complex distributions, but its ability to adapt without overfitting highlights the strength of class-specific covariance modeling. Just as importantly, this example illustrates IMLD's flexibility: users can modify datasets by drawing new points, save and reload data files, and augment specific classes to explore how algorithm performance changes across different scenarios.

6. Extending IMLD

In this section, we describe the architecture of the web application, how the application is organized, and how a user can expand upon the application.

6.1. Software Architecture and Organization

The IMLD application and its source code are packaged in a tar file named *imld.tar.gz*. To download this file, open the IMLD web application, and under the **Help** dropdown menu simply select **Run Locally.** This will prompt your web browser to download *imld.tar.gz*. This is shown in Figure 41.

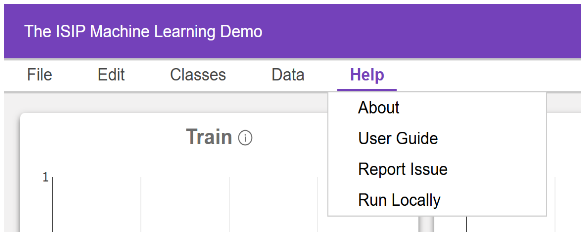


Figure 41. Downloading a local copy of IMLD.

Once the download is completed, simply extract the files to where you'd like IMLD to be stored. In the extracted files, you will find a single folder named v*5.0.1*. This is the home directory. Within the home directory, there are two files and one directory of note:

- **imld.py:** the file that users execute to run IMLD. Calls the IMLD class that is made in the app/ directory.
- **requirements.txt:** contains all the libraries needed to run IMLD.
- **app/:** contains containing the main code of the application.

Within the **app/** directory, IMLD is organized across 4 main directories:

- **backend/:** contains the core logic that drives the application. It includes Python scripts responsible for managing data preprocessing, training, prediction, and evaluation of machine learning models. The backend is where the machine learning algorithms and data generators are executed. The parameter files are also located here.
- **extensions/:** contains the extensions for the application. These extensions are used to add functionality to the Flask server, such as the routes, scheduling, and the base Flask app.
- **static/:** contains all static resources required by the frontend of the application. These include HTML snippets, JavaScript files, CSS style sheets, fonts, and UI assets. Together, these files build the user-facing graphical interface and provide interactive controls for selecting models, tuning parameters, uploading datasets, and visualizing output.
- **templates/:** contains the base HTML template for the IMLD app. This directory is required because the IMLD app is a Flask app, and Flask requires a base HTML template to be used for all the pages in the app. This base HTML template must be in a directory called "templates/" for Flask to find it.
- **__init__.py:** defines the IMLD class that is run within imld.py.

Within the **backend/** directory, the following files exist:

- **algo_params_v00.toml:** contains the parameters for the algorithms that are implemented within IMLD. This file also contains how the user may select parameter values, whether it be through a dropdown menu or integer input.
- **data_params_v00.toml:** contains the parameters for the data generators that are implemented within IMLD. This file also contains how the user may select parameter values, whether it be through a dropdown menu or integer input.
- **nedc_cov_tools.py:** provides functions for computing covariance matrices and related linear algebra operations.
- **nedc_debug_tools.py:** provides classes that facilitate debugging and information display.
- **nedc_file_tools.py:** contains file and path handling utilities. Functions in this module support opening, reading,

and writing data files reliably across environments. This file also contains certain conventions that may be used across other files within the application.

- **nedc_imld_tools.py:** provides IMLD-specific helper functions for formatting responses, transforming UI selections, and interfacing between Flask endpoints and backend processing logic.
- **nedc_ml_tools_data.py:** provides data generators that are to be used for ML Tools.
- **nedc_ml_tools.py:** provides core machine learning functions such as model training, prediction, evaluation, and scoring. This file acts as the main entry point for backend ML processing logic used by IMLD.
- **nedc_qml_base_providers_tools.py:** contains the base classes and hardware configuration class for the quantum machine learning providers.
- **nedc_qml_providers_tools.py:** This file contains all the child classes implementing the QuantumProvider class. Each child class is responsible for implementing the methods required to interact with a specific quantum provider.
- **nedc_qml _tools_constants.py:** contains all the constants that are used in the nedc_qml_tools.py.
- **nedc_qml_tools.py:** contains classes and functions that are used to implement quantum machine learning algorithms.
- **nedc_trans_tools.py:** contains an implementation of the transformer architecture.

Within the **extensions/** directory, the following files are contained:

- **base.py:** This file contains the base Flask app that is used to create the Flask server. It is used to create the Flask app and its configurations. This file is used by /app/__init__.py to load the base Flask app.
- **blueprint.py:** This file contains the blueprint for the Flask app. It is used to create the routes for the Flask app. This file is used by /app/__init__.py to load the blueprint for the Flask app. All of the routes for the Flask app are defined in this file.
- **scheduler.py:** This file contains the scheduler for the Flask app. It is used to create the scheduler for the Flask app. This file is used by /app/__init__.py to load the scheduler for the Flask app. The scheduler is used to schedule the task of removing old cached models from the server.

Within the **static/** directory, the following directories and file are contained:

- **components/:** contains modular JavaScript components that form the IMLD frontend interface. Each file defines a different part of the user interface, such as toolbars, data parameter forms, plot displays, and event handling logic.
- **downloads/:** contains the user's guide and a version of the ISIP Machine Learning Demo.
- **fonts/:** contains the font used across the application, Inter.
- **icons/:** contains the information icon that appears next to parameters, giving the user more background on what each parameter does.
- **index.css:** defines the styling used across the application.

6.2. Customizing IMLD

IMLD is an application designed to be user extensible. For example, a common way a user may choose to extend the application is by implementing a new algorithm or data generator. Thanks to the structure of the application, developers can add these with minimal effort by following existing patterns. Below is an example of how we may go about implementing one of our new native algorithms, LightGBM:

1. **Open nedc_ml_tools.py and define your algorithm:**

    1.1. Be sure that the libraries your algorithm needs to function are defined at the top of the file. In the case of LightGBM, these are “lightgbm”, “sklearn.model_selection.train_test_split”, “sklearn.metrics.accuracy_score”, “sklearn.metrics.confusion_matrix” and “numpy”. Be sure to update requirements.txt when you introduce new libraries to the program.

    1.2. Under the section at line 241 labelled “Algorithm-Specific Parameter Definitions”, define the algorithm name and the available implementation(s):

    ```
    LGBM_NAME = “LGBM”
    LGBM_IMPLS = [IMP_NAME_LGBM]
    ```

    While some algorithms may share the same implementations that are already defined, like “sklearn”, LightGBM has a unique implementation name “lightgbm”. Be sure to define this at line 138:

    ```
    IMP_NAME_LGBM = “lightgbm”
    ```

    You should also investigate algorithm-specific parameters that your algorithm may need to be implemented, as here is where they should be defined. At line 149, there is a list of common names found in many native algorithms in nedc_ml_tools.py. You should use these defined names within your algorithm’s train method, and if necessary, define new names here and use these.

    In LightGBM’s case, there are a few algorithm-specific parameters:

    ```
    LGBM_NAME_BFRAC= “bagging_fraction”
    LGBM_NAME_BFREQ= “bagging_freq”
    LGBM_NAME_BOOST= “boosting”
    LGBM_NAME_ISUNB = “is_unbalance”
    LGBM_NAME_LAMBDAL1= “lambda_l1”
    LGBM_NAME_LAMBDAL2= “lambda_l2”
    LGBM_NAME_MINDATA = “min_data_in_leaf”
    LGBM_NAME_NUML= “num_leaves”
    ```

    1.3. Implement the following methods for your algorithm within a class. In our example of the LightGBM algorithm, we need to implement these methods in the LGBM class:

    - def __init__(self): The user only needs to update these two lines:

      ```
      foo.CLASS_NAME__ = self.class.name__
      self.model_d[ALG_NAME_IMP] = IMP_NAME_doe
      ```

      “foo” is the class name and “doe” is the implementation name.

    - def train(self, data: MLToolsData, write_train_labels: bool, fname_train_labels: str)
    - def predict(self, data: MLToolsData, model = None)
    - def get_info(self)

    It is very helpful to look at how these functions have been implemented in the native algorithms, as methods often either stay the same or are very similar to one another across algorithms, especially those of the same type. For instance, LightGBM and XGBoost, two gradient boosting models, have very similar predict and get_info methods.

    1.4. At line 7490, add your class within the ALGS dictionary. In LightGBM’s case: LGBM_NAME:LGBM.

    1.5. Save the file and close.

2. **Define its configuration in the TOML file algo_params_v00.toml:**

    2.1. In line 47, add your class name in the LIST list. For LightGBM: “LGBM”.

    2.2. Define your parameters. Do note that you should follow the specific format seen throughout the algorithms. In LightGBM’s case:

```
[LGBM]
  name = “LightGBM”
  [LGBM.params]
    [LGBM.params.implementation_name]
```

There are certain rules that need to be adhered to for all the parameters:

- Each param needs a “name”, “type”, “default”, and “description” attribute.
- If the type is a “select”, the param also needs an “options” attribute.
- If the type is an “int” or “float”, the param also needs a “range” attribute.

2.3. Save the file and close.

3. **Run IMLD.**

Your new algorithm should display as an available option within the application. If you’ve added new libraries, be sure to reinstall the libraries in the requirements.txt that now includes the new ones. For troubleshooting, please ensure that both your implementation and parameters line up with the existing formatting.

We strongly recommend reviewing the existing native algorithm implementations and using these as a guide when adding new algorithms.

7. Downloading and Installing IMLD

In this section, we describe how a user can download and install the application locally. We also describe how the application can be installed on a web server using our web server, *www.isip.piconepress.com*, as an example.

7.1. Running IMLD Locally

Running IMLD locally is straightforward with Anaconda. In Section 6.1, we described how to download the tar file containing the application. To successfully run this, however, you need the proper set of Python extensions installed. Follow the steps below to set up an environment specifically for IMLD and launch the application:

1. **Create a Python environment:** Open a terminal window in the root directory where IMLD was installed.
   1.1. conda create --name imld: create a virtual environment specifically for the IMLD application.
   1.2. conda activate imld: activate the newly created environment.
   1.3. conda install pip: ensures that pip is available in the environment for installing Python dependencies.
   1.4. pip install -r requirements.txt: installs all required Python packages listed in requirements.txt, which include Flask, Plotly, and other necessary libraries.
2. **Run the IMLD application:** Once the environment is set up and dependencies are installed, start the application by executing the command ‘*python imld.py*’.
3. **Access the web interface:** After launching, the terminal will display an address such as: *http://localhost:5000*. Open this URL in your web browser to interact with the IMLD web interface.

7.2. Installing IMLD on a Web Server

Currently, the web version of IMLD runs on the NEDC web server (*www.isip.piconepress.com*). A network diagram of the Neuronix computing cloud is shown in Figure 42. An interactive view of the cluster can be found at *https://isip.piconepress.com/projects/neuronix/*. On this server, the IMLD application is deployed using Gunicorn, a Python WSGI HTTP server, in conjunction with Apache acting as a reverse proxy. This architecture allows for a clean separation between application logic and public-facing network services.

IMLD is configured to be used with https. Apache is responsible for terminating SSL/TLS connections, ensuring that all communication between the user's browser and the server is encrypted. Once a secure https request is received, Apache forwards the request to Gunicorn, which runs the IMLD Flask application locally. Gunicorn is configured to launch multiple worker processes, allowing the backend to handle concurrent requests efficiently.

This server setup allows IMLD to be accessed via a stable URL and reliably serve users interacting with the web interface. The use of Gunicorn and Apache together is a best-practice deployment method for Python web applications, and it has proven to be a reliable and robust solution for hosting IMLD in the Neuronix production environment.

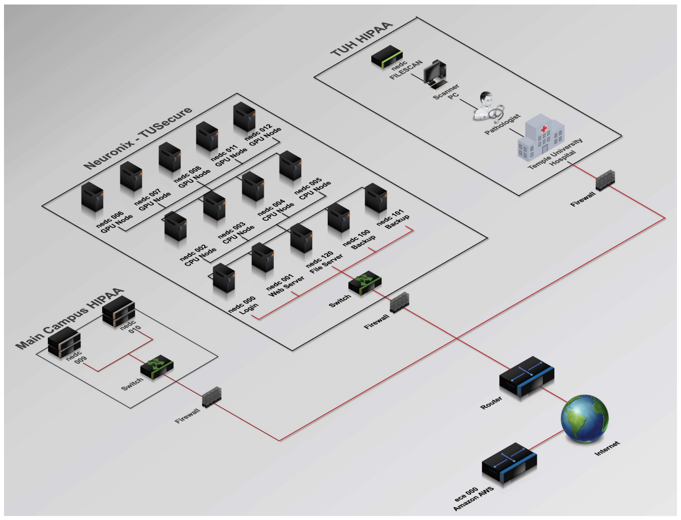


Figure 42. The Neuronix cluster

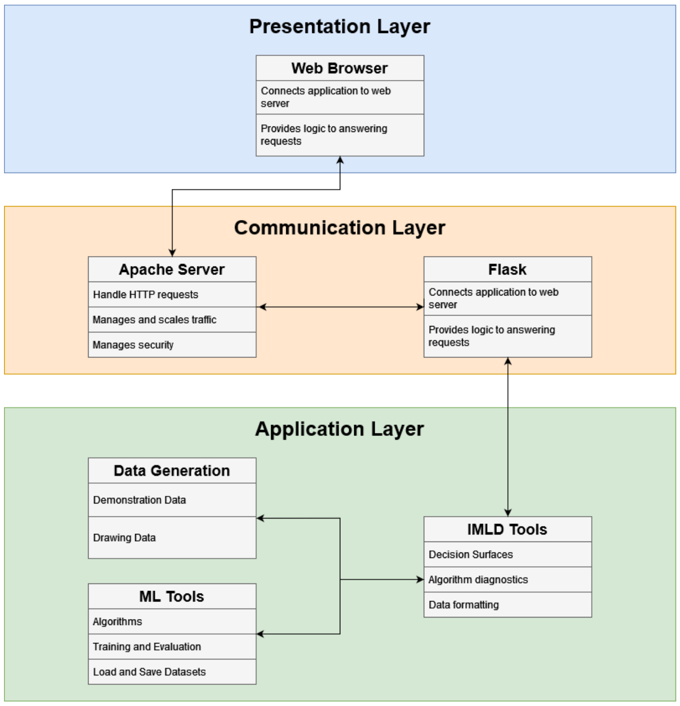


Figure 43. The IMLD architecture

7.3. Deploying IMLD on a Web Server

An overview of the IMLD architecture is shown in Figure 43. Gunicorn is used to deploy IMLD, which is a Python Flask app, because it can be easily used with reverse proxy servers such as Apache and Nginx. Gunicorn acts as an intermediary between a Python application (e.g., Flask) and the client by passing http requests to the application for processing. Gunicorn is very straightforward to set up on an Apache web server, such as that used by the Neuronix cluster. The instructions below describe how to install IMLD on a web server running Rocky Linux.

7.3.1. Create A Python Virtual Environment

Using Anaconda, run these commands:

```
conda create --name imld
conda activate imld
conda install pip
pip install -r requirements.txt
```

Gunicorn is one of many Python applications included in the file *requirements.txt*.

7.3.2. Test Gunicorn

From the IMLD application folder (where imld.py is located), run:

```
gunicorn --workers 1 --bind 127.0.0.1:8000 imld:app
```

This launches Gunicorn with 1 worker, bound to *localhost:8000*. *imld:app* tells Gunicorn to find the app object in imld.py. You can test it at this URL: *http://127.0.0.1:8000*.

7.3.3. Enable Apache Proxy Modules

Open this file:

```
sudo emacs /etc/httpd/conf.modules.d/00-proxy.conf
```

Paste these modules inside:

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
```

These modules together let Apache act as a proxy and forward http requests to other servers. Be sure to save and exit after adding these.

7.3.4. Create Apache Virtual Host Config

Navigate to your Apache sites-available directory and create a file named *imld.conf* or the equivalent:

```
<VirtualHost *:443>
  ServerName imld.local
  ServerAdmin yourname@example.com

  SSLEngine on
  SSLCertificateFile /etc/letsencrypt/live/yourdomain.com/cert.pem
  SSLCertificateKeyFile /etc/letsencrypt/live/yourdomain.com/privkey.pem
  SSLCACertificateFile /etc/letsencrypt/live/yourdomain.com/chain.pem

  ProxyRequests Off
  ProxyPreserveHost On
  <Proxy *>
    Require all granted
  </Proxy>

  ProxyPass "/app" "http://127.0.0.1:8000/"
  ProxyPassReverse "/app" "http://127.0.0.1:8000/"
</VirtualHost>
```

Make sure imld.local is listed in your */etc/hosts* file if you're using it as the domain name. For the Neuronix cluster, this file is */etc/httpd/conf.d/virtualhost.conf*.

7.3.5. Enable the Site and Restart Apache

Test out the Apache server:

```
sudo httpd -t
```

This command checks the Apache config files for syntax errors. If the output of this command is Syntax OK, it means your syntax works and Apache was able to successfully parse it.

Next, restart the server:

```
sudo systemctl restart httpd
```

With Gunicorn running, IMLD should now be accessible through the browser at the specified domain or localhost port.

7.3.6. Set Up ".service" File (Optional)

You can also set up a ".service" file so that you don't have to start Gunicorn every time your server restarts, and you don't need to have a terminal open either. Gunicorn will run in the background as daemon.

First, create a new file titled *imld.service* in this directory: */etc/systemd/system*. Paste this inside, replacing what is applicable for your server:

```
[Unit]
Description=Gunicorn instance to serve IMLD through Apache
After=network.target

[Service]
User=your_user_here
Group=your_group_here
WorkingDirectory=/path/to/your/app
Environment="IMLD_ISSUE_NUM=003"
ExecStart=/bin/bash -c 'echo $IMLD_ISSUE_NUM > /tmp/imld_env.log && exec \
   /path/to/your/conda/env/bin/gunicorn --worker-class eventlet --workers 2 --bind 127.0.0.1:8000 imld:app'

[Install]
WantedBy=multi-user.target
```

After saving and exiting, you then need to then enable and start the service:

```
sudo systemctl daemon-reload
sudo systemctl enable imld.service
sudo systemctl start imld.service
sudo systemctl status imld.service
```

You can check the status of the service by running this command:

```
sudo systemctl status imld.service
```

The service can also get reset by running this command:

```
sudo systemctl restart imld.service
```

Though this adds a bit of complexity to the installation, it is highly recommended you do this to make sure IMLD is available when the server is rebooted.

8. Conclusions and Future Work

IMLD is an application that has been evolving for over 40 years. In the latest version, we're adding the ML algorithms LightGBM and XGBoost. These algorithms represent a major enhancement to IMLD's capabilities, as they introduce gradient boosting—a class of ensemble learning methods not previously native within the application. Unlike traditional algorithms such as Euclidean distance classifiers or linear discriminants, gradient boosting models iteratively build ensembles of decision trees that focus on correcting the mistakes of previous iterations, allowing for highly accurate, nonlinear decision boundaries.

LightGBM and XGBoost are among the most powerful and widely adopted gradient boosting frameworks in modern machine learning. Their addition marks the first time IMLD supports boosted decision tree ensembles, enabling users to explore techniques that are prevalent in both academic research and real-world applications such as finance, healthcare, and competitive data science. Both models function very similarly, using decision trees as their base learners and combining them sequentially to improve the model's performance. Each new tree is then trained to correct the errors made by the previous tree.

With these models, IMLD now supports not just classical, interpretable methods but also high-performance algorithms capable of capturing complex feature interactions. Their inclusion introduces new types of parameters and training behaviors, offering a deeper educational experience and significantly expanding the platform's modeling versatility.

We're also adding Scikit-Learn's make_classification data generator. This tool creates synthetic multiclass datasets and is useful for testing how well classifiers handle noise and complexity. It can introduce redundant and uninformative features, create multiple clusters per class, and apply linear transformations to the feature space. These options allow us to simulate more realistic and challenging classification problems, making it easier to evaluate how robust a model is under different conditions. Note that for make_classification to work in IMLD, which is an application bound to 2D visualizations, datasets that contain more than two features are automatically projected to 2D using PCA.

Finally, we have also added support for quantum computing-based algorithms in v5.0.1. These fairly new algorithms are extremely computationally demanding because they are run via a simulator. As this field matures, we will update IMLD appropriately.

To learn more about IMLD, visit our project page at: *https://isip.piconepress.com/projects/imld/*.

Acknowledgements

IMLD has been a work in progress for many years and has been funded as part of many grants. Most recently, this material is based on work supported by several organizations including the National Science Foundation (grants nos. 2211841 and 1726188 and 1925494), the Temple University Catalytic Collaborative Funding Initiative and most recently by the Pennsylvania Breast Cancer Coalition Breast and Cervical Cancer Research Initiative. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these organizations.

Numerous researchers have contributed to the development of this tool. It would be impossible to list all of them. The original version of the system was derived from work by Janna Shaffer and Daniel May when ISIP was located at Mississippi State University. Most recently, the web version was developed by a senior design team at Temple University that included Raynel Lopez, Shane McNicholas, Brian Thai, and Kayla Toner. Salvatore Tanelli and Sohail Aji were responsible for the current release. Our senior software engineers, Dylan Heathcote, Phuykong Meng and Rick Duncan, also played significant roles in the development of this tool over the years.

References

Cap, T., Kreitzer, A., Miranda, M., & Vadimsky, D. (2022). IMLD: A Python-Based Interactive Machine Learning Demonstration. Senior Design II, College of Engineering, Temple University, 1–12. url: *www. isip.piconepress.com/publications/presentations_misc/2021/senior_design/imld/*.

Cap, T., Kreitzer, A., Miranda, M., Vadimsky, D., & Picone, J. (2021). IMLD: A Python-Based Interactive Machine Learning Demonstration. In I. Obeid, I. Selesnick, & J. Picone (Eds.), Proceedings of the IEEE Signal Processing in Medicine and Biology Symposium (SPMB)(pp. 1–4). Philadelphia, Pennsylvania, USA. (Download). doi: *10.1109/SPMB52430.2021.9672265*.

Huang, K., & Picone, J. (2002). Internet-Accessible Speech Recognition Technology. Proceedings of the IEEE Midwest Symposium on Circuits and Systems, III-73-III–76. doi: *10.1109/MWSCAS.2002. 1186973*.

May, D., & Picone, J.. (2002). The ISIP Pattern Recognition Applet. Institute for Signal and Information Processing, College of Engineering, Mississippi State University. url: *www.isip.piconepress.com/projects/speech/software/demonstrations/applets/util/pattern_recognition/current/*.

McNicholas, S., Thai, B., Toner, K., & Lopez-Morel, R. (2024). ISIP Machine Learning Demo. Senior Design, College of Engineering, Temple University, 1–13. url: *www.isip.piconepress.com/publications/presentations_misc/2024/senior_design/imld/*.

Picone J., Duncan, R., & Hamaker, J.. (2001). Internet-Accessible Speech Recognition Technology. O'Reilly Open Source Convention. url: *www.isip.piconepress.com/publications/conference_presentations/2001/oscon/software/*.

Picone, J. (2000). Internet-Accessible Technology Demonstrations. Institute for Signal and Information Processing, College of Engineering, Mississippi State University. url: *www.isip.piconepress.com/projects/speech/software/demonstrations/*.

Shaffer, J., Hamaker, J., & Picone, J. (1998). Visualization of signal processing concepts. *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, 3, 1853–1856. doi: *10.1109/ICASSP.1998.681824*.

Thai, B., McNicholas, S., Shalamzari, S. S., Meng, P., & Picone, J. (2023). Towards a More Extensible Machine Learning Demonstration Tool. Proceedings of the IEEE Signal Processing in Medicine and Biology Symposium, 1–4. doi: *10.1109/SPMB59478.2023.10372731*.