An End-to-End AutoML Framework for Deploying Classical Machine Learning Models on ESP32 and Arduino Microcontrollers

Huu-Phuoc Nguyen
(Can Tho University, Can Tho, Vietnam
https://orcid.org/0009-0006-5822-4121, huuphuoc.research@gmail.com)

Abstract: This paper presents P-ML, an end-to-end AutoML framework for deploying classical machine learning models on memory-constrained microcontrollers. The framework automates the complete workflow, including data splitting, model selection, hyperparameter optimization, and the generation of optimized Arduino-compatible C++ libraries. P-ML integrates Optuna-based hyperparameter tuning with stratified data splitting using the SPXY method to ensure robust and reliable model selection. The framework has been successfully implemented and validated, enabling automatic library generation for a wide range of classical machine learning models, including SVM variants (linear, RBF, polynomial, sigmoid, NuSVC, LinearSVC), RFF-based SVM, decision tree–based ensembles, multilayer perceptrons, k-nearest neighbors, discriminant analysis, and Naive Bayes classifiers. The generated libraries are compact, efficient, and directly deployable on Arduino Uno, Arduino Nano, and ESP32 platforms using the Arduino IDE.

Keywords: AutoML, Embedded Machine Learning, Resource-Constrained Systems, Code Generation, Software Framework
Categories: B.4.2 (Embedded Systems), C.3.3 (Real-time and Embedded Systems), I.2.6 (Learning)
DOI: 10.5281/zenodo.18130146

1. Introduction

The increasing deployment of Internet of Things IoT systems has driven a strong demand for machine learning inference directly on microcontroller based devices such as Arduino and ESP32. These platforms are widely used in sensor networks wearable systems and edge intelligence applications but they operate under strict memory and computational constraints. In such environments classical machine learning models such as Support Vector Machines decision trees ensemble methods and k nearest neighbors often outperform deep learning approaches for small and medium sized sensor datasets while requiring significantly fewer computational resources.

Despite these advantages deploying classical machine learning models on microcontrollers remains challenging. Existing embedded machine learning solutions either rely on high level runtimes that introduce considerable memory and performance overhead or focus primarily on deep learning models which leads to complex deployment pipelines and limited suitability for resource constrained devices. Furthermore current tools lack an automated and systematic workflow that bridges conventional machine learning development in Python with efficient native C C plus plus deployment on Arduino compatible platforms. To address these limitations this paper introduces P ML an end to end AutoML framework specifically designed for deploying classical machine learning models on memory constrained embedded systems. P ML automates the complete workflow including data splitting model selection hyperparameter optimization and the generation of compact Arduino compatible C plus plus libraries.

The framework integrates Optuna based hyperparameter optimization with stratified data splitting using the SPXY method to ensure reliable and deployment oriented model evaluation. A broad range of classical machine learning algorithms is supported and trained models are automatically converted into optimized C plus plus code suitable for Arduino Uno Arduino Nano and ESP32 platforms. The entire pipeline is unified within an intuitive user interface which eliminates the need for manual configuration or specialized embedded systems expertise.

2. Architecture of the P-ML Framework

2.1 Data Splitting Algorithms

The P-ML framework adopts a SPXY-based hold-out data splitting strategy for model training and evaluation. This method is designed to create independent training and testing sets while preserving class distribution and maximizing data diversity in the training set.

Using SPXY, the dataset is divided into training and testing subsets according to a predefined ratio (e.g., 70/30). The algorithm is applied independently within each class to maintain label balance. Training samples are selected such that they are maximally distant from each other in the feature space, which increases coverage of the input domain and improves the model’s ability to generalize.

For example, consider a dataset containing 150 samples from 3 classes (50 samples per class) with a 70% training ratio. SPXY selects 35 samples per class for training and 15 samples per class for testing, resulting in 105 training samples and 45 independent test samples. The model is trained once using the training set and evaluated on the test set, closely reflecting real-world deployment conditions where unseen data is encountered.

To prevent information leakage, feature normalization is performed exclusively on the training set. The resulting statistics (mean and scale) are then applied to the test set and later embedded into the generated microcontroller library. This ensures consistency between training, evaluation, and on-device inference.

Compared to random hold-out splitting, SPXY provides a more representative and diverse training set, particularly for small to medium-sized datasets. While the method does not reuse test samples for training, it offers a clear and deployment-oriented evaluation protocol that aligns well with embedded and IoT applications, where models are typically trained once and deployed without further retraining.

2.2 Hyperparameter Optimization

The P-ML framework employs Optuna to automatically search for and select optimal hyperparameters for each machine learning model. Instead of manually testing a limited set of predefined values, Optuna evaluates dozens of parameter combinations in a guided manner and identifies the configuration that yields the best performance.

For example, in the case of an SVM model, rather than fixing parameters such as C = 1.0 and kernel = 'rbf', the system explores multiple values, e.g., $C \in \{0.1, 1, 10, 50\}$, different kernel types, and corresponding gamma settings, then selects the combination that maximizes accuracy. In practice, this approach often leads to substantial performance improvements compared to manual tuning (e.g., increasing accuracy from around 80% to 85–90%).

Optuna operates through trials, where each trial corresponds to a specific hyperparameter configuration. All trials are managed within a study, which records and compares results to determine the best-performing parameters. The optimization process is driven by the Tree-structured Parzen Estimator (TPE) sampler, which learns from previous trials to prioritize promising regions of the search space, rather than relying on random or exhaustive grid exploration. Compared to random search (slow and unguided) and grid search (computationally expensive and coarse), TPE achieves faster convergence and higher efficiency.

To reduce training time, the framework incorporates Median Pruning, which enables early termination of poorly performing trials. During cross-validation, if the performance of a trial after the initial folds is significantly worse than the median performance of previous trials, the trial is pruned and training is stopped. For example, if most configurations achieve around 75–80% accuracy, a trial that only reaches 45% after the first fold will be discarded immediately, avoiding unnecessary computation.

Each trial is evaluated using k-fold cross-validation (e.g., $k = 5$). For a proposed parameter set, the model is trained and evaluated across five different splits, and the final trial score is computed as the mean accuracy across folds. If a trial is pruned early, its score is based only on the completed folds. For instance, one trial may achieve accuracies of 80%, 82%, 85%, 83%, and 84%, yielding a mean accuracy of 82.8%, while another may be terminated after the first fold due to poor performance.

Compared to manual parameter selection, this approach is objective, automated, and more effective, while reducing the risk of missing promising parameter combinations. Compared to grid search, Optuna significantly reduces the number of required training runs through its TPE strategy and pruning mechanism. The main drawback is the higher initial computational cost compared to fixed-parameter training, as each model must be trained multiple times.

In the current configuration, each model is optimized using 50 trials with 5-fold cross-validation (configurable by the user), corresponding to a maximum of 250 training runs. However, the actual runtime is substantially reduced through pruning and timeout constraints. The final outcome of this process is a set of best hyperparameters and best accuracy, which are then used to retrain the model and generate an optimized C++ library for microcontroller deployment.

2.3 Microcontroller Library Generation

The conversion from scikit-learn models (Python) to C/C++ libraries for ESP32 and Arduino Mega is performed by directly extracting the internal parameters of trained models and mapping them to static data structures in C/C++. The objective is to ensure that inference on the microcontroller reproduces exactly the same mathematical computations as the original Python model.

For all models, the first step is extracting the parameters of the StandardScaler fitted on the training set. Specifically, the arrays scaler.mean_ and scaler.scale_ are exported as constant float arrays in C/C++. During inference, each input feature vector from sensors is normalized using the same formula as in Python:

$$x' = (x - \mu) / \sigma$$

where μ and σ are the stored mean and scale values. This guarantees that the input distribution during deployment matches the training phase. Additionally, the list of classes_, class names, and feature order are hard-coded into the library to ensure correct label mapping and prevent input misalignment.

For kernel-based SVMs (SVC, NuSVC), the framework extracts key attributes from the scikit-learn model, including support_vectors_, dual_coef_, intercept_, and kernel parameters (kernel, gamma, degree, coef0). Support vectors are stored as two-dimensional arrays (or flattened arrays) in C/C++, while dual coefficients and intercepts are stored as corresponding weights and biases for each one-vs-one classifier. During inference, the embedded prediction function replicates the Python computation: for each support vector, the kernel function is evaluated (e.g., RBF kernel:

$$\exp(-\gamma * ||x - x\_i||\hat{}2)),$$

multiplied by the dual coefficient, summed with the intercept, and combined through voting to select the class with the highest score.

For LinearSVC, the process is simpler: only coef_ and intercept_ are extracted. The generated C/C++ code computes the dot product between the input vector and the linear weights, resulting in a compact and memory-efficient model suitable for constrained devices.

In Decision Tree models, attributes such as tree_.feature, tree_.threshold, tree_.children_left, tree_.children_right, and tree_.value are extracted directly from scikit-learn. These arrays are converted into constant C/C++ data structures and used to reproduce the same tree traversal logic as in Python. During inference, the firmware starts at the root node, compares the relevant feature with the threshold, follows the left or right child accordingly, and stops at a leaf node, where the class with the highest frequency or probability in tree_.value is selected.

For MLP (neural networks), the framework extracts the full lists of coefs_ and intercepts_, each corresponding to the weights and biases of a network layer. These matrices are stored as C/C++ arrays in the correct layer order. The activation function type (relu, tanh, or logistic) is also hard-coded. During inference, each layer performs a matrix–vector multiplication, adds the bias, and applies the specified activation function. Preserving the exact layer order and activation functions ensures that the microcontroller output matches the Python model.

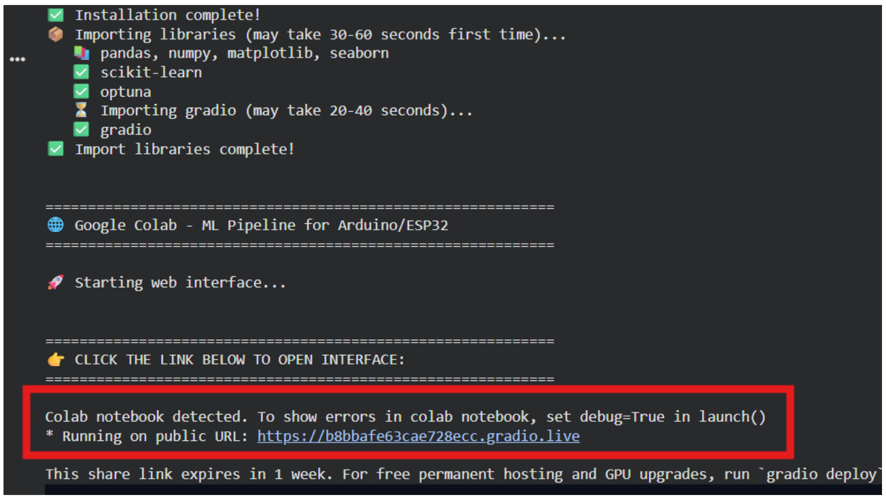
For KNN, since the model does not learn parameters, the framework directly extracts X_train and y_train (or a reduced subset) from Python and stores them in the C/C++ library. During inference, the firmware computes distances (typically Euclidean) between the input sample and stored samples, selects the k nearest neighbors, and applies majority voting. On Arduino Mega, the number of stored samples may be limited to fit memory constraints, while ESP32 can typically retain the full dataset.

For Naive Bayes, all statistical parameters are extracted from Python: theta_ (mean), var_ (variance), and class_prior_ for GaussianNB, or log-probability tables for BernoulliNB. These values are mapped to constant arrays in C/C++, and inference consists of computing log-likelihoods using the standard statistical formulas and selecting the class with the highest posterior probability. Similarly, for LDA/QDA, class means, priors, and covariance matrices (shared for LDA, class-specific for QDA) are extracted and used to compute linear or quadratic discriminant functions in the firmware.

In summary, the entire conversion process does not rely on approximation or retraining. Instead, it performs a controlled transfer of mathematical parameters from Python to C/C++ and faithfully reproduces the original inference computations. As a result, the generated libraries for ESP32 and Arduino Mega exhibit prediction behavior consistent with the original scikit-learn models, while still allowing memory and performance optimizations tailored to each hardware platform.

3. User Guide
3.1 Running the Code on Google Colab

To begin, access the P-ML source code via the provided Google Colab link [https://colab.research.google.com/drive/1G3PQG_6mmQgMDnm88vhXh-duST0GOgO2].
Once opened, save a personal copy of the notebook to your own Google Drive to enable editing and execution.
After saving, simply run the existing code cells in order. The notebook will automatically install all required dependencies. Once the setup process is complete, a URL pointing to the graphical user interface (GUI) will be printed in the output. Click this link to open the P-ML web interface in a new browser tab and start working with the system.



```
✅ Installation complete!
📦 Importing libraries (may take 30-60 seconds first time)...
   📊 pandas, numpy, matplotlib, seaborn
   ✅ scikit-learn
   ✅ optuna
   ⏳ Importing gradio (may take 20-40 seconds)...
   ✅ gradio
✅ Import libraries complete!


============================================================
🌐 Google Colab - ML Pipeline for Arduino/ESP32
============================================================

🚀 Starting web interface...


============================================================
👉 CLICK THE LINK BELOW TO OPEN INTERFACE:
============================================================
Colab notebook detected. To show errors in colab notebook, set debug=True in launch()
* Running on public URL: https://b8bbafe63cae728ecc.gradio.live

This share link expires in 1 week. For free permanent hosting and GPU upgrades, run `gradio deploy`
```

Figure 1: Link to the P-ML graphical interface

3.2 Uploading the Dataset
Data File Requirements
The uploaded dataset must satisfy the following conditions:
The file format must be Excel (.xlsx).
The first row must contain column names.
Each feature must be stored in a separate column.
A column named Class is mandatory, as it contains the ground-truth labels.
If this column is missing, the system will raise an error.
Feature column names can be arbitrary and user-defined.
A sample dataset file is available at the provided reference link [https://raw.githubusercontent.com/HuuPhuoc2411/P-ML/main/DATA_SAMPLE.xlsx].
Data Upload Options
P-ML supports two different data upload scenarios:

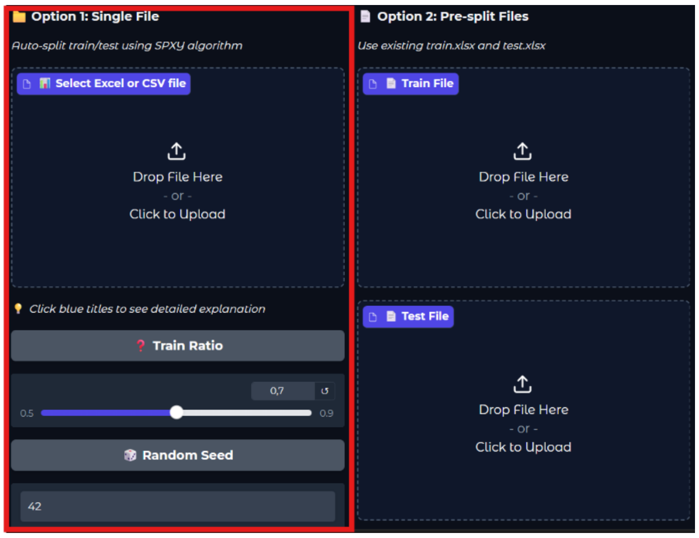


Figure 2: Uploading a single dataset for automatic SPXY splitting

If your data has not been pre-split, upload a single Excel file containing the full dataset. The system will automatically apply the SPXY algorithm to split the data into training and testing sets using a default ratio of 70/30 (which can be adjusted by the user).
The generated training and testing datasets are also saved as separate Excel files, allowing them to be reused in future experiments.

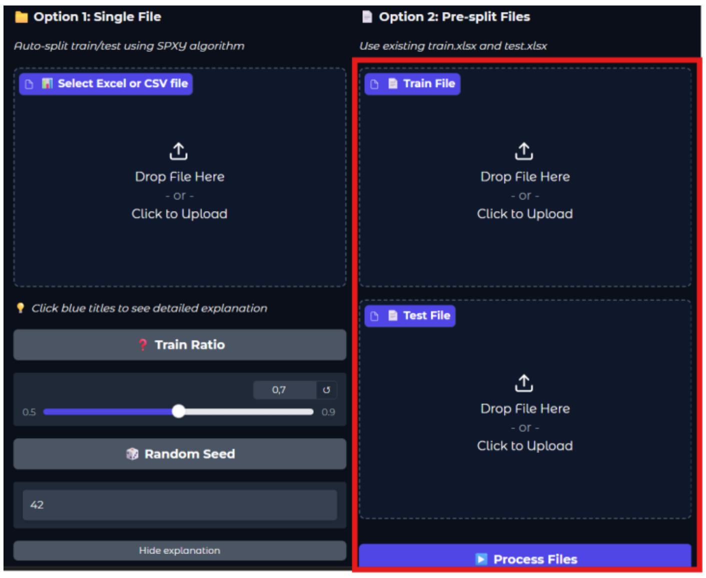


Figure 3: Uploading pre-split training and testing files
If you have already prepared separate training and testing files, upload them directly using the corresponding fields under Option 2: Pre-split Files.

After uploading, the system displays class distribution plots for both training and testing sets. These plots allow users to visually verify whether the class proportions are balanced across the two splits. Ideally, similar distributions indicate a successful and reliable split.

Once this step is complete, the user must manually proceed to the next stage to select and train models.

3.3 Model Selection and Training

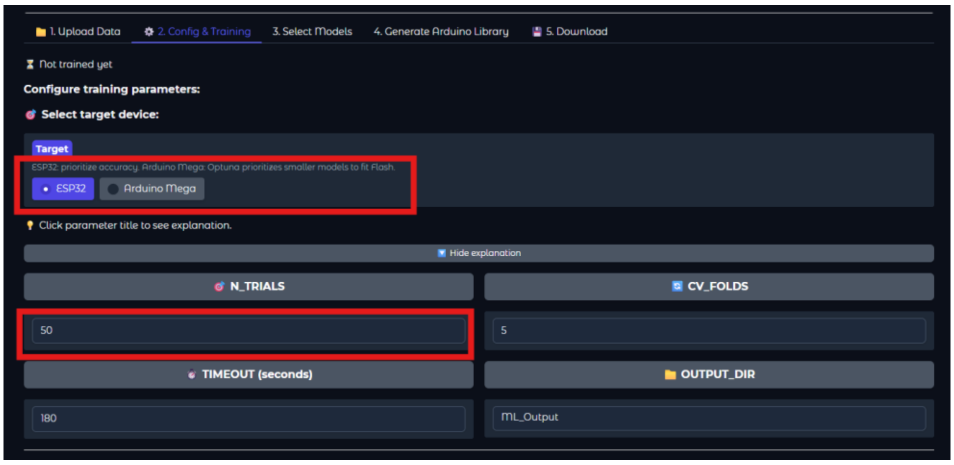
The first configuration choice is selecting the target platform for library generation: ESP32 or Arduino. This selection determines memory constraints and optimization strategies used later in the pipeline.

The user then configures the training parameters:

- Number of trials per model: how many times each model is trained with different hyperparameter configurations.
- Timeout: the maximum allowed training time for a single model. If the timeout is reached before completing all trials, the process stops and the best parameters found so far are retained.
- CV folds: the number of data folds used internally during training for performance estimation.
- Output folder name: the directory name under which all results and generated files will be saved.

Next, the user selects the models to be trained under "Select model groups to train". It is possible to select all available model groups or only specific algorithms of interest.

Finally, the training process is initiated by clicking the Train button. Depending on the dataset size, number of selected models, and training trials, this process may take a considerable amount of time.



Not trained yet

Configure training parameters:

Select target device:

Target

ESP32: prioritize accuracy. Arduino Mega: Optuna prioritizes smaller models to fit Flash.

[✓ ESP32] Arduino Mega

Click parameter title to see explanation.

N_TRIALS	CV_FOLDS
50	5

TIMEOUT (seconds)	OUTPUT_DIR
180	ML_Output

Figure 4: Training configuration interface

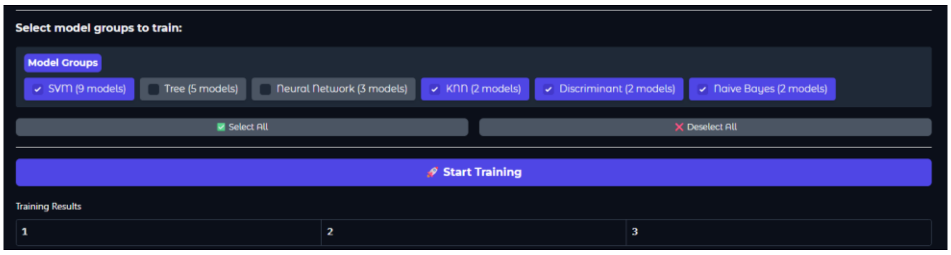


Figure 5: Model group selection

3.4 Selecting the Best Models for Library Generation

After training is complete, the system presents a list of trained models along with their performance metrics. At this stage, users can select which models will be exported as embedded libraries.

Seven different filtering criteria are available to help identify the best-performing models. Additionally, users can specify the number of models to select, with a default value of ten.

A special Manual Selection option is also provided. When enabled, users can explicitly choose specific models by entering their corresponding indices from the displayed list, bypassing automatic performance-based ranking. This is particularly useful when certain models are preferred for practical or experimental reasons.

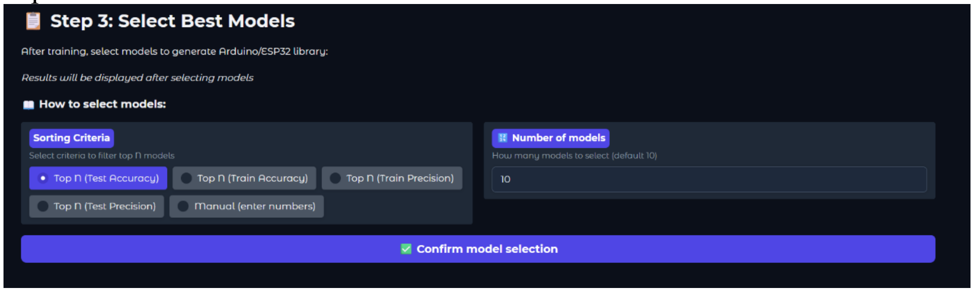
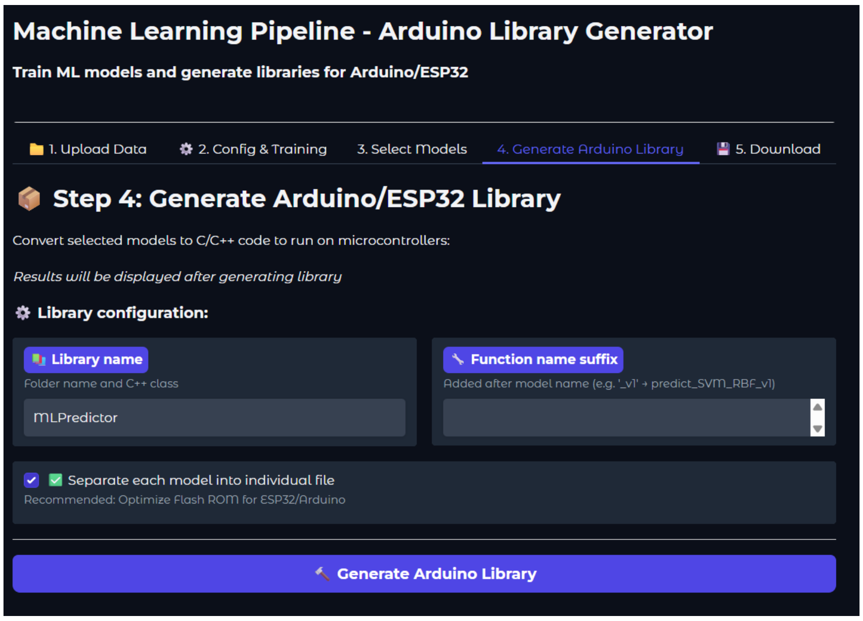

Step 3: Select Best Models
After training, select models to generate Arduino/ESP32 library:
Results will be displayed after selecting models
How to select models:
Sorting Criteria
Select criteria to filter top N models
● Top N (Test Accuracy)
Top N (Train Accuracy)
Top N (Train Precision)
Top N (Test Precision)
Manual (enter numbers)
Number of models
How many models to select (default 10)
10
Confirm model selection

Figure 6: Model filtering and manual selection interface

3.5 Generating the Embedded Library

Once the desired models are selected, the user proceeds to the Generate Arduino Library page.
Here, the user specifies:
The library name
A custom suffix for the prediction functions
Each generated prediction function is automatically named based on the original model name combined with the user-defined suffix, ensuring clarity and avoiding naming conflicts.
After configuration, clicking the Generate button initiates the library generation process, which converts the selected models into optimized C/C++ code suitable for the chosen microcontroller platform.


Machine Learning Pipeline - Arduino Library Generator

Train ML models and generate libraries for Arduino/ESP32

Step 4: Generate Arduino/ESP32 Library

Convert selected models to C/C++ code to run on microcontrollers:

Results will be displayed after generating library

Library configuration:

Library name: MLPredictor
Folder name and C++ class

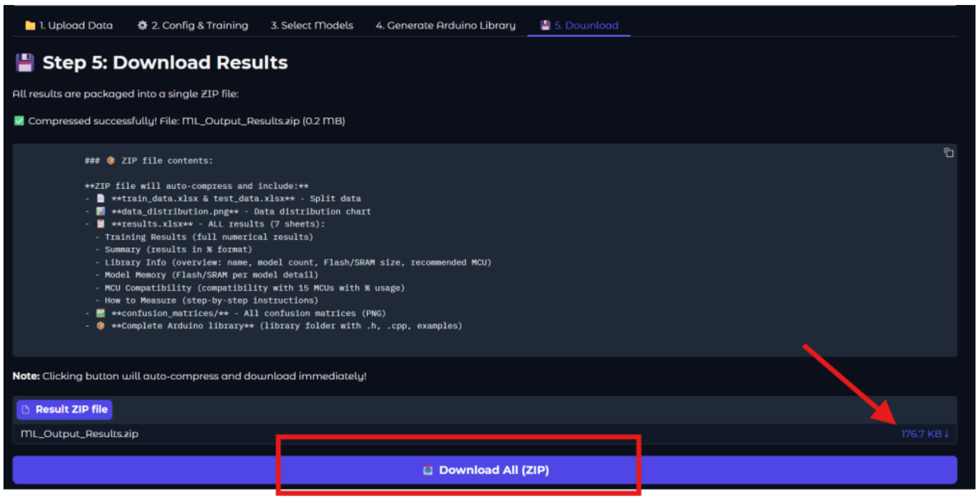
Function name suffix:
Added after model name (e.g. '_v1' → predict_SVM_RBF_v1)

[✓] Separate each model into individual file
Recommended: Optimize Flash ROM for ESP32/Arduino

Figure 7: Library naming and function configuration

3.6 Downloading the Results

In the final step, users can download all generated outputs. Clicking the Download button compresses the entire result directory into a single archive file (RAR format).
Due to Google Colab's interface behavior, users may need to click the download location again, as indicated in the interface, to complete the file transfer to their local machine.



Step 5: Download Results

All results are packaged into a single ZIP file:

✅ Compressed successfully! File: ML_Output_Results.zip (0.2 MB)

```
### 📦 ZIP file contents:

**ZIP file will auto-compress and include:**
- 📄 **train_data.xlsx & test_data.xlsx** - Split data
- 📊 **data_distribution.png** - Data distribution chart
- 📋 **results.xlsx** - ALL results (7 sheets):
  - Training Results (full numerical results)
  - Summary (results in % format)
  - Library Info (overview: name, model count, Flash/SRAM size, recommended MCU)
  - Model Memory (Flash/SRAM per model detail)
  - MCU Compatibility (compatibility with 15 MCUs with % usage)
  - How to Measure (step-by-step instructions)
- 🖼️ **confusion_matrices/** - All confusion matrices (PNG)
- 📦 **Complete Arduino library** (library folder with .h, .cpp, examples)
```

Note: Clicking button will auto-compress and download immediately!

Result ZIP file: ML_Output_Results.zip 176.7 KB ↓

Figure 8: Downloading the generated archive

3.7 Output File Structure

The downloaded archive contains the following components:

- One Excel file summarizing the training results and model performance

- One PNG image showing the dataset distribution
- One directory containing confusion matrix images for all trained models
- One directory containing the generated embedded libraries and Arduino IDE example code
- If SPXY splitting was used, two additional Excel files containing the training and testing datasets

This structured output enables easy inspection, reuse, and deployment of the trained models across different embedded platforms.

4. Discussion and Limitations

The experimental results indicate that the libraries generated for the ESP32 platform consistently achieve higher prediction accuracy than those deployed on Arduino-based microcontrollers. This performance gap primarily stems from hardware constraints. ESP32 devices provide substantially larger memory resources, enabling models to be deployed with their full parameter sets and without aggressive simplification. In contrast, Arduino platforms—particularly Arduino Mega—are subject to strict limitations on both SRAM and Flash memory. As a result, several models must be simplified through parameter reduction, pruning, or structural optimization in order to fit within the available resources, which can lead to a measurable degradation in predictive performance.

In addition to memory constraints, computational performance is also a critical limitation on Arduino platforms. In particular, linear SVM variants (SVC_linear or SVM_linear), despite their relatively compact memory footprint, often exhibit very slow inference speed on Arduino microcontrollers. This slowdown is caused by the need to compute dot products between the input vector and high-dimensional weight vectors for each class, which becomes computationally expensive on low-frequency CPUs without hardware acceleration. Consequently, even models that fit within memory limits may fail to meet real-time inference requirements on Arduino devices.

To mitigate these limitations on memory- and compute-constrained Arduino platforms, we recommend avoiding reliance on the output of a single model. Instead, an ensemble-style voting strategy can be employed, where multiple lightweight models are deployed in parallel and the final decision is determined by majority voting across their predictions. For example, deploying a set of ten simplified models and selecting the class predicted most frequently can significantly improve robustness and overall accuracy, effectively compensating for the performance loss introduced by individual model simplifications. While this approach is primarily motivated by Arduino's constraints, it is equally applicable to ESP32 platforms and can further enhance stability and reliability in real-world IoT deployments.

Finally, it is important to note that the size of the generated embedded library is not fixed. The memory footprint grows with both the number of training samples (e.g., support vectors in SVM or stored instances in KNN) and the number of output classes, especially for multi-class classifiers that rely on one-vs-one or one-vs-rest strategies. Users must therefore carefully consider dataset size, class count, and model complexity when targeting highly constrained platforms such as Arduino Mega, as exceeding these limits may result in compilation failures, degraded runtime performance, or the need for further model simplification.

5. License and Copyright Statement