

perClass Mira 2.4 User's Guide

Table of contents

Introduction	5
Release Notes	6
Installation	11
System requirements	11
Installer	11
First start	13
Uninstall	15
Checking for updates	15
Walkthrough example	16
Selecting project type	16
Adding images	16
RGB preview from selected bands	17
Selecting spectral band	18
Adjusting display range	19
Defining classes	20
Labeling examples	21
Training a model	21
Sample validation	22
Improving the model	24
Labeling more examples	24
Active learning	26
Tuning model performance	28
Limiting spectral range	30
Executing on new images	31
Exporting solutions for deployment	31
Performance optimization	32
Performance constraints	34
Marking images for testing	36
Error visualization	37
Interactive error visualization in image confusion matrix	38
Visualization	39
Scaling	41
Constraining visualization to foreground	41
Mapping wavelengths to cube bands	43
Object segmentation	43
Object labels and object decisions	44
Displaying object list	45
Object modes	46
Object classification	47
Regression	48
Step 1: Pixel classifier	48
Step 2: Object segmentation	49
Step 3: Point annotation	50
Step 3a: Removing point annotations	51
Step 4: Build regression model	51

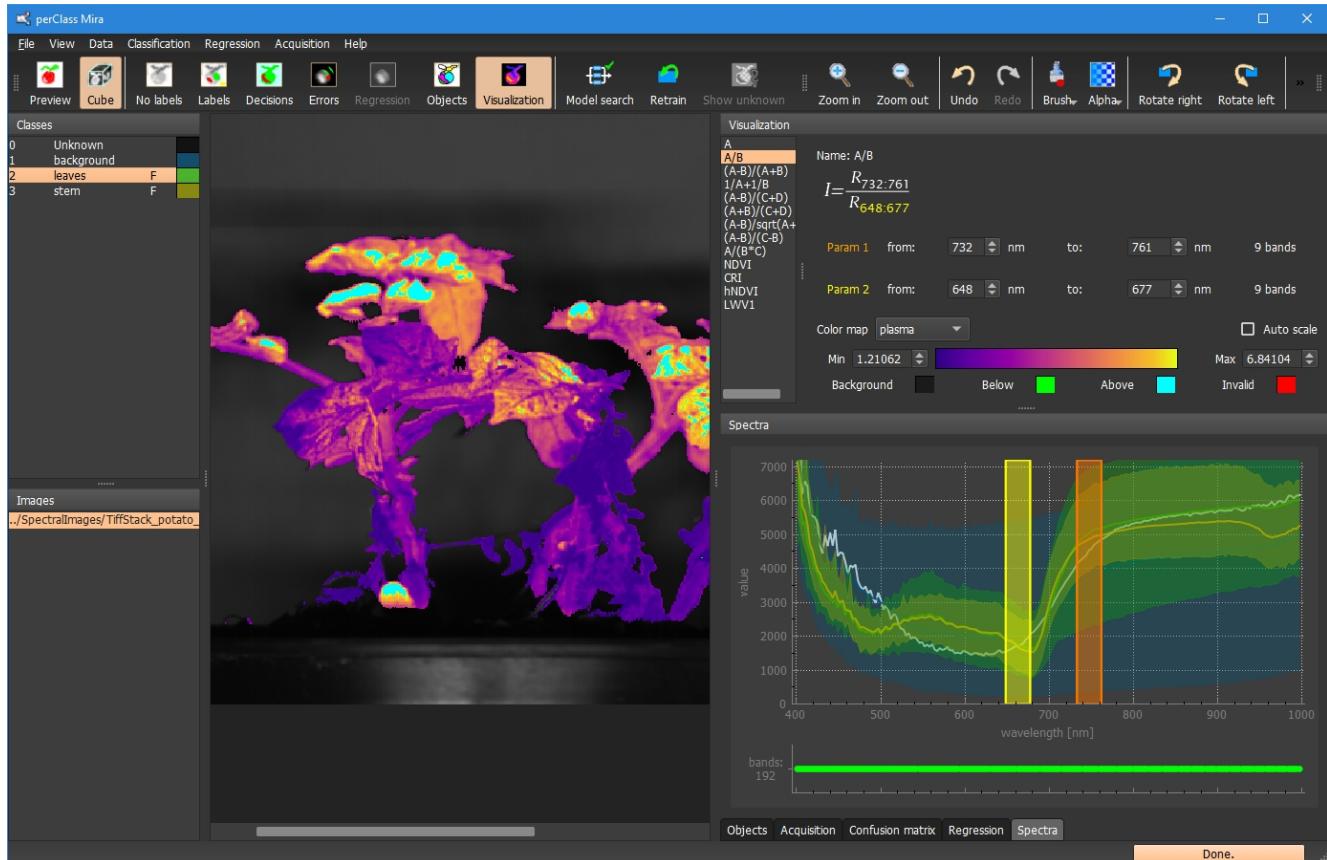
Step 5: Adding test examples	52
Improving regression models	53
Exporting regression results to Excel	55
Importing annotations from Excel	55
Per-pixel regression output	58
Live data acquisition	59
Installation of Specsensor SDK	60
Initializing acquisition	61
live data processing	62
Segmenting out objects	63
Closing live acquisition mode	64
Reference information	64
Project types	65
Supported image formats	66
Image zoom	66
Keyboard shortcuts	66
perClass Mira Runtime API	67
mira_Init	68
Error codes	69
mira_GetVersion	70
mira_GetErrorCode	70
mira_GetErrorMsg	70
mira_RefreshDeviceList	70
mira_GetDeviceCount	71
mira_GetDeviceName	71
mira_SetDevice	71
mira_LoadModel	72
mira_LoadCorrection	72
mira_SetMinObjSize	73
mira_SetSegmentation	73
mira_GetInputWidth	73
mira_GetInputHeight	74
mira_GetInputBands	74
mira_GetInputDataType	74
mira_GetInputDataLayout	74
mira_GetMaskType	75
mira_StartAcquisition	75
mira_ProcessFrame	75
mira_ProcessCube	76
mira_StopAcquisition	76
mira_GetFrameDecisions	76
mira_GetDecCount	77
mira_GetDecName	77
mira_GetDecColor	77
mira_GetObjCount	78
mira_GetObjDataInt	78
mira_GetObjDataClassSize	79

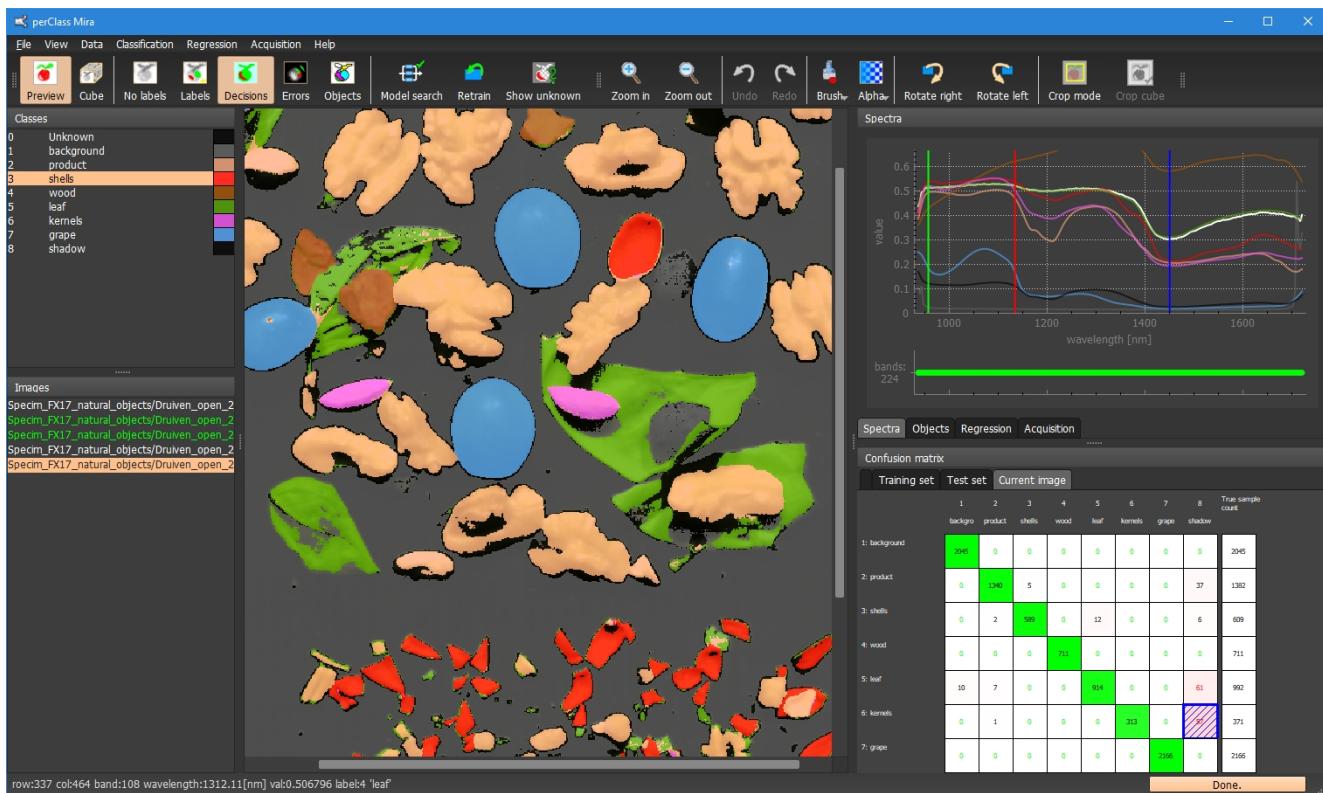
mira_GetObjDataClassFrac	79
mira_SaveImage	80
mira_Release	80

Introduction

perClass Mira

perClass Mira is a user interface for interpretation of spectral images. It allows users to define classification and regression solutions and deploy them in custom applications. It automatically selects machine learning models. It enables users to understand problems at hand and interactively improve the solutions.





Release Notes

2.4 28-sep-2020

- added reflectance correction for Headwall project type (correction by whiteReference and darkReference ENVI cubes in the same directory)
 - allows loading of externally corrected cubes in the same project
 - enables multiple scans per directory sharing the same correction
 - default cube extension is .bin, arbitrary extensions are supported
 - to apply correction at runtime, pass directory containing whiteReference and darkReference scans to `mira_LoadCorrection` (example:
`mira_LoadCorrection(pmr, "path_to_dir_with_correction_files", NULL)`)
- added general ENVI project type supporting arbitrary cube file extension
- added Corning project type
 - added perClass Mira Runtime support for native BIP data stream corrected with dark reference inside the camera
- improved selection of multiple images (click and drag supported, no image reload in multiple selection)
- improved drag&drop of directories (adding all files within each dropped dir)
- added support for NVIDIA CUDA11 (Ampere)
- when using floating licenses, specific licensing product can be requested based on `floatingLicenseProduct` setting in `mira.ini` (`mira` for perClass Mira Dev and `mira.gui` for perClass Mira)
- when importing regression annotation from Excel, existing points are removed to avoid duplicates
- fixed a problem when adding regression annotation to all objects in each scan
- fixed problem when label painting with large brushes
- fixed memory leak in loading large number of specim FX scans
- fix for dropped frames at the start of live acquisition session
- at runtime, all projects (including line-scans) must explicitly enable object segmentation with `mira_SetSegmentation(pmr, 1)`

2.3 26-jun-2020

- support for foreign object detection with truly unknown objects
 - label materials you know. Enable *Show unknown* to highlight all materials unseen in training.
 - user-adjustable sensitivity on per-class basis provides extra control (slider via the right-click in the class-list)
 - objects unseen in training can be segmented out (flag *Unknown* decision as foreground)
 - the new foreign object optimizer is on by default, can be disabled in Classification menu.
- color wells display transparency (change alpha for a specific class in the color dialog or by via alpha toolbar button by holding Ctrl)
- crop improvements
 - crop rectangle line thickness auto-adjusted for very large cubes
 - adjust crop rectangle by dragging lines
- segmentation improvements
 - support for up to 20 foreground classes including access to their content information
 - per-object results can be batch-exported to Excel including per-class content in each object
 - fix for a crash due to changing object size in live acquisition mode
- confusion matrix improvements
 - added light mode (to allow copy/paste directly to documents)
 - added option to copy as text for direct copy/paste to Excel
- fixed live acquisition issue when Specim calibration file (.scp) was not found
- fix for min/max visualization setting in presence of NaNs and infinite values
- support for case insensitive fields ENVI in header files (for Python integration)
- runtime improvements
 - support for region of interest (ROI) for snapshots. Applying classifier only to specific ROI.
 - support for object segmentation for snapshot use-cases (Imec project type, float data type, BIP layout)

2.2 29-apr-2020

- new [Visualization mode](#) showing computed indices using different common equations
 - define using individual wavelengths or wavelength ranges
 - auto-scaling and manual scaling
 - indication of below, above and invalid values
 - define wavelength ranges interactively in spectral plot
 - render using different colormaps
- improved [regression](#)
 - visualize [per-pixel regression output](#) (e.g. distribution of moisture)
 - [import point annotations from Excel](#) (matching scan names exactly or with regular expressions)
 - move and edit point annotations
 - use only specific subset of spectral bands
 - show cross-validated regression error (RMSECV) which has the same units as the regressed value
 - when hovering over the results in the regression plot, display specific annotation points with their true and estimated values
 - visual indication that some point annotations are not linked to objects (e.g. point not on foreground class)
 - [export regression results in Excel](#) together with per-object size, bounding boxes, true and estimated regression outputs
- perClass Mira Runtime improvements
 - added model export for perClass Mira Runtime (new "Mira Pipeline" .mpl format using base64 encoding)
 - added API to query expected data type, data layout and geometry of data from spectral camera
 - added support for all object segmentation configurations created in the GUI including per-object content retrieval and object classification by rules
 - added snapshot processing mode (`mira_ProcessCube`). Currently only pixel decisions are provided, not yet the object segmentation or content.
- added support for OceanInsight Spectrocam and Pixelcam data formats

- added support for ENVI cubes with uint32 data type and little-endian float
- added classifier preprocessing (smoothing, 1st and 2nd derivative)
- export and import labels as PNG images
- export per-image results to Excel allowing quick summary of fraction of decisions within foreground (e.g. disease within plant leaves)
- update of live acquisition using Specim SpecSensor SDK
 - Applying regression both per-object and per-pixel in live acquisition
 - Calibration pack information stored in settings, reused for further sessions
- fixes in object panel: When retraining the classifier, object classification rules are preserved
- adding default class color map
- repeatable object label colors (can be change using random seed dialog)
- added per-class transparency (alpha setting in the color dialog and using the toolbar transparency slider - hold Ctrl to change only the current class alpha)

2.1 18-feb-2020

- Specim FX project type allows scan directories with different name than raw cube in capture sub-folder
- Unicode support in image file names for ENVI-based formats
- providing informative error messages when image cannot be loaded
- adding Cubert Tiff project type with native support for Cubert Ultris camera
- adding Headwall project type
- license file can be drag & dropped from Explorer to the license dialog
- RGB bands are set based on ENVI header file
- mira.log file is now written to AppData/Roaming, not to the installation directory (now by default in Program Files (x86))
- fix of calibration pack loading in SpecSensor
- labels can be exported into .png files
- ENVI import supports int16 data type
- when the number of samples is too low, the output window shows a red message that can provide details on click
- when alpha is too low (high label transparency), the toolbar alpha button blinks to remind the user that labels may be badly visible

2.0 18-oct-2019

- new Cubert ENVI project enabling data from Cubert Ultris and upsampled UH185 images
- perClass Mira Runtime binaries adding dongle support

2.0 10-oct-2019

- adding support for double-precision ENVI data cubes
- supporting model deployment for execution on live data from Cubert Ultris light-field hyperspectral camera
- enabling Cubert plugin export for ENVI-based projects.
- fixes in live acquisition using Specim FX cameras when device loading fails or opening FileReader gets cancelled
- fixing a crash due to very large training set
- fixing a bug in error visualization mode where switching to images without labels did not show proper image

2.0 20-sep-2019

- Fix: Installation directories with non-ASCII characters are now supported
- Live acquisition executables for Specim cameras included (perClass_Mira_live.exe and perClass_Mira_gpu_live.exe)
- Senop project: Images are automatically processed with per-band gain

2.0 6-sep-2019

- Estimate [object quality using regression](#) (examples: sugar content estimation per tomato)
 - annotate quality per object
 - automatic model selection reporting performance (R^2 and Q^2 statistics)
 - user-defined pre-processing (smoothing and derivatives)
 - apply regression to new images (show a bounding box + regression output per object)
 - allow localized information extraction by a radius around annotation points
- Images can be [flagged for testing only](#) (not used for building the model)
 - **Test confusion matrix** provides a detailed view of the performance on test images
- [Error visualization mode](#) brings insight in model performance.
 - visualize where the current model fails
 - this helps to identify incorrect labels or (together with test image flagging) whether the data is well represented in the training set
 - **Image confusion matrix** shows only labeled examples on the current image
 - [interactive error visualization](#) by moving mouse over the image confusion matrix
- [Object segmentation mode](#) with multiple options
 - one object / one class mode for object detection (e.g. detect plastic pieces in a food product stream for automatic removal)
 - one object / multiple classes for object classification (e.g. detect potato pieces, classify entire piece as defective if it contains more than 5% of greening or rot inside)
 - visualizing object labels or object decisions
 - object decisions by majority vote or rules (size of or fraction of a specific class)
- Usability improvements
 - **assign label stroke to the current class.** This allows one to exclude a specific label stroke from training and see the impact on model performance (define an additional class and exclude it, assign strokes to it and retrain)
 - the data validation mechanism excluding invalid spectra is now off by default. It can be enabled using context menu in the spectral plot.
 - all modes (labels, decisions, errors, objects) accessible by direct keystrokes
 - confusion matrix size can be decreases/increased (useful for large number of classes)
 - auto-check for software updates + direct link to download latest version from the GUI (Help / Check for updates)
- experimental [Live data acquisition](#) from **Specim FX cameras** using Specsensor SDK (needs to be installed separately)
 - apply a classifier and object segmentation to a live data stream
 - live visualization of processing speed and drop frame indication to assess production performance
 - user-control of exposure and camera frame-rate
 - supports practical situations where production light conditions are different from the training situation
 - the white and dark references used for live data processing can be specified without model retraining
 - automatic handling of spectral and spatial binning based on specific scan meta-data
 - support for **outdoor operation**: Define white reference by specifying an image region where a reference tile was placed
 - **recording data** from a live acquisition in the standard LUMO format

1.4 22-may-2019

- **perClass Mira Runtime** is now included in the distribution
 - high throughput (1.5ms/frame on NVIDIA GPU in an example foreign object detection project, Specim FX17, 640 spatial pixels, 224 bands, 6 materials)
 - the runtime directly reports object positions, sizes and classes
 - support for **NVIDIA Jetson** platform (both ARM CPU and NVIDIA GPU backend)
 - support for line-scan use-case on Specim projects (specific white/dark correction format)
- **Linux build** for both perClass Mira GUI and perClass Mira Runtime
 - accelerated CPU and GPU support on Linux

- new high-throughput segmentation engine
 - automatically discarding objects smaller than user-defined minimal size
 - supporting multiple foreground classes
 - high-speed line-scan segmentation with constant per-frame speed
- export visualization as PNG images (band or RGB, with labels, pixel decisions or segmented objects)\
- for Cubert projects, proper wavelength ranges are shown

1.3 8-feb-2019

- zoom using mouse wheel now follows cursor
- image rotation using toolbar buttons (and > < keyboard shortcuts)
- adding images using drag and drop from Windows explorer
- support for ENVI files with high-endian byte order uint16 (byte order=1)
- saved projects now preserve settings of the current band, R,G,B lines and allow direct execution of the trained model when project is loaded
- exported decision images (PNGs) contain meta-data such as class count and class names accessible by standard tools such as [tweakpng](#) or Matlab imfinfo command
- multiple directory selection for Specim FX and Tiff stack project types can be enabled in mira.ini file (using useNativeDirSelection=false). It is not enabled by default because it uses a non-native file dialog.
- new project type for Senop cameras (formerly Rikola)

1.2 5-dec-2018

- Added [band-selection widget](#). It is now possible to manually select the wavelengths used for building models
 - Band brushing allows quick selection or clearing of wavelength ranges
 - Exported models start from the full set of wavelengths but use only the selected subset for the model. This allows quick deployment of different models to custom applications assuming full spectrum (single binding with perClass Runtime is needed)
- Added export of labeled data to perClass Toolbox sddata format
- Added export of entire data cube in Matlab format as 3D matrix
- Models results are now repeatable with a new random seed dialog controlling the internal data partitioning process.
- Separate CPU-only and CPU+GPU builds are available. The CPU-only build is always available by default to avoid issue related to GPU drivers or CUDA versions installed. The CPU+GPU executable is called perClass_Mira_gpu.exe
- Band index and the wavelength number are now updated on the status bar when dragging the band line in spectral plot
- Added support for logging of status messages when starting up the application. This is useful to understand some issues with GPU installations and CUDA versions. Logging is off by default, can be switched on in the mira.ini file.
- Licensing improvements:
 - For activated licenses, there is now an auto-update mechanism that pulls updated license from the activation server when the application starts. The application may be used without on-line connection - it is needed only once in two weeks.
 - Adding support for floating licenses obtained over network from a license server. Floating licenses are now checked out one per session.
- Fixed wrong file name of previous project used for saving new project with File/Save command
- Fixed a crash when preview image could not be loaded

1.1 10-sep-2018

- [confusion matrix view](#) showing detailed error information
 - interactive performance optimization in a confusion matrix (slider in right-click context menu or a mouse wheel on confmat entries)
 - confusion matrix shows normalized errors and precisions, absolute sample counts available as well
 - quickly switch to confmat with 'c' key and to spectral plot with 's' key

- define **performance constraints** via double click on a confusion matrix field (create/remove constraint)
- constraints may be adjusted live by Ctrl+mouse wheel
- constraints may be enabled/disabled to understand available performance options
- move between available solutions fulfilling all constraints with [and] shortcuts
- **[preview image from user-adjustable R,G and B bands](#)** when spectral cube is loaded
 - this view improves labeling experience for many material types that look similar in a single band but their differences may be highlighted in R,G,B view
- **undo/redo** for label painting speeds up labeling
- **image crop** providing significant memory use reduction and processing speedups
 - when a project with a cropped image is loaded, the original cube is loaded and cropped
 - original cube may be loaded as a new image and multiple crops from the same cube are supported
- including perClass Runtime DLL and example of spectral cube processing in C
 - support for both **single precision** and double precision pipelines (with a new perClass 5.4 Runtime)
 - significant speedup of exported classifiers
 - legacy export option supporting older deployed runtimes <= 5.2
- a **preview rotation** command allows one to fix the rotation between preview and spectral cube (e.g. on Specim IQ projects)
- adding an option to **exclude a class from training** (right-click in class list or press 'x')
 - this allows one to quickly check the impact of specific classes on the overall solution
- option to purchase a license online and directly turn the demo into a commercial product
- dialog to request Skype/Teamviewer session on start up
- fix for a wrong class index after removing a class
- fix for clear labels of an image

1.0 13-jul-2018

- fix for a dock shift bug (when resizing a docked window and clicking on the image, the docked pannel resized back)
- adding band line dragging by mouse
- adding max valid line which is automatically set on image load
- when user is on preview and tries painting, a dialog is shown to load the entire cube (allows quick image changing without load)

1.0 29-may-2018

- first public release

Installation

System requirements

perClass Mira system requirements:

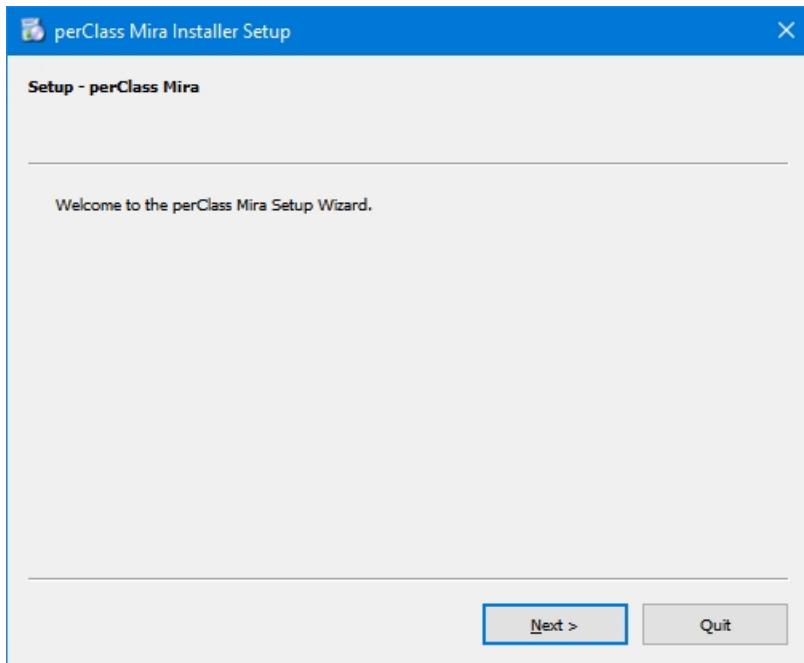
- PC with MS Windows (version 7 or higher) 64bit
 - memory of at least 8GB
- (optional) NVIDIA GPU supporting CUDA 9 or higher library (needs to be installed separately)
 - GPU memory at least 4GB
- (optional) mouse with a scroll-wheel is useful for many operations
- (optional) for live data acquisition using Specim FX cameras, Specsensor SDK needs to be installed

Installer

perClass Mira installer will guide you through the installation process.

Welcome screen

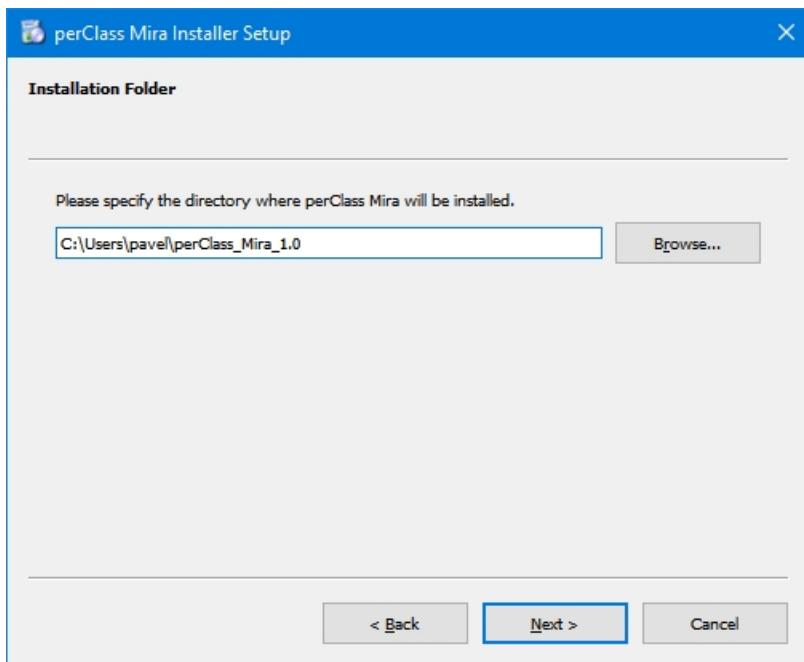
When the installer is launched, the following welcome screen appears:



TIP: In some situations, buttons at the bottom may not be directly visible. It is sufficient to resize the installer window to make buttons appear.

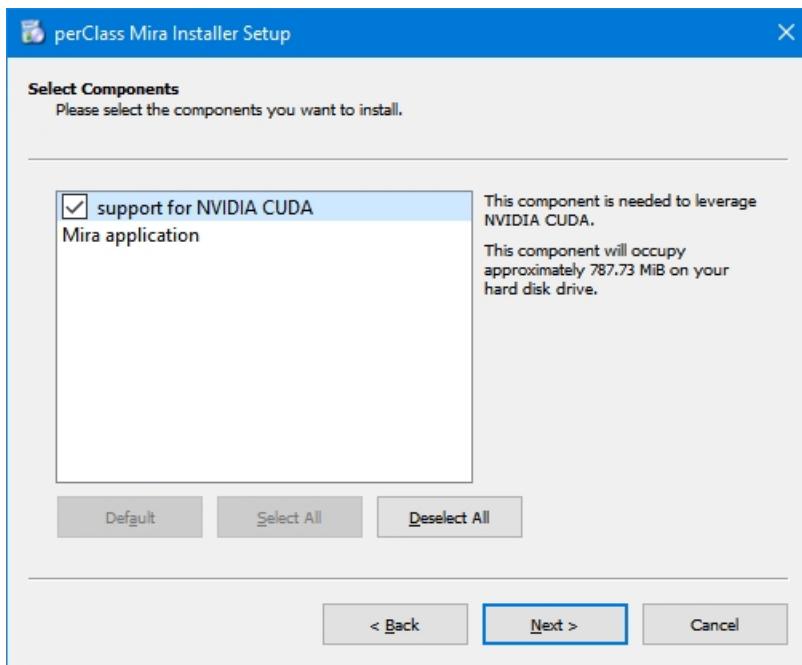
Installation directory

Next, installation directory can be defined. By default, perClass Mira is installed in a user directory



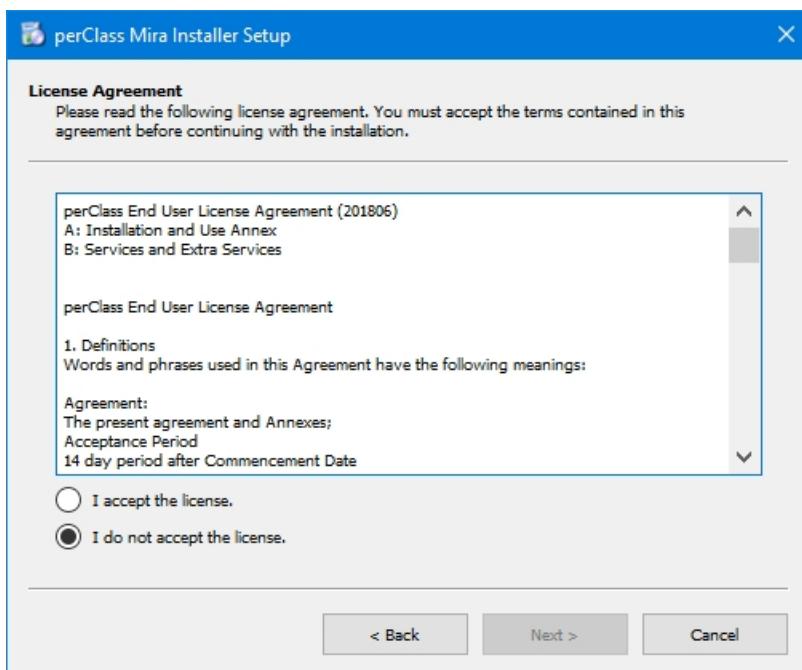
Component selection

In the next window, software components may be selected. Although it is possible to deselect NVIDIA GPU support, we recommend keeping the default setup if possible.



License agreement

In the next step, license agreement is provided and needs to be agreed in order to continue the installation.

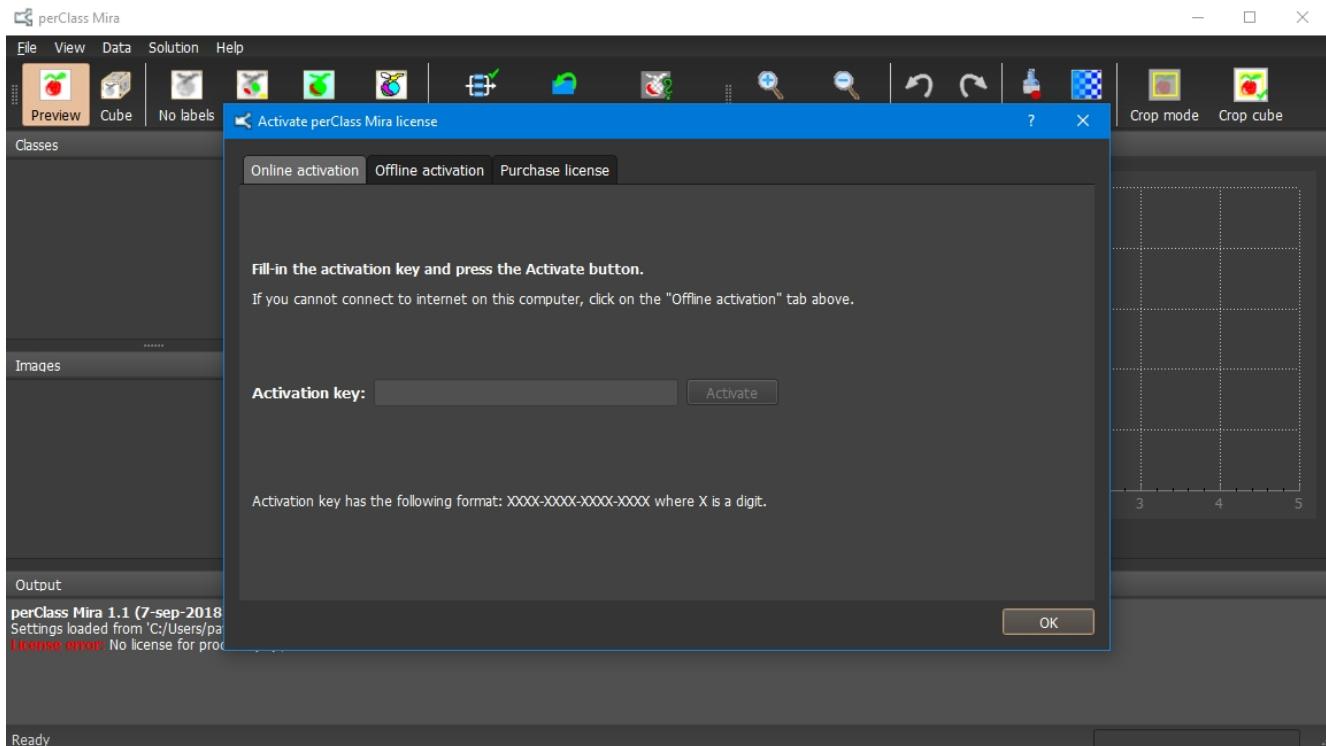


Start Menu Shortcut

Finally, start menu shortcut may be defined.

First start

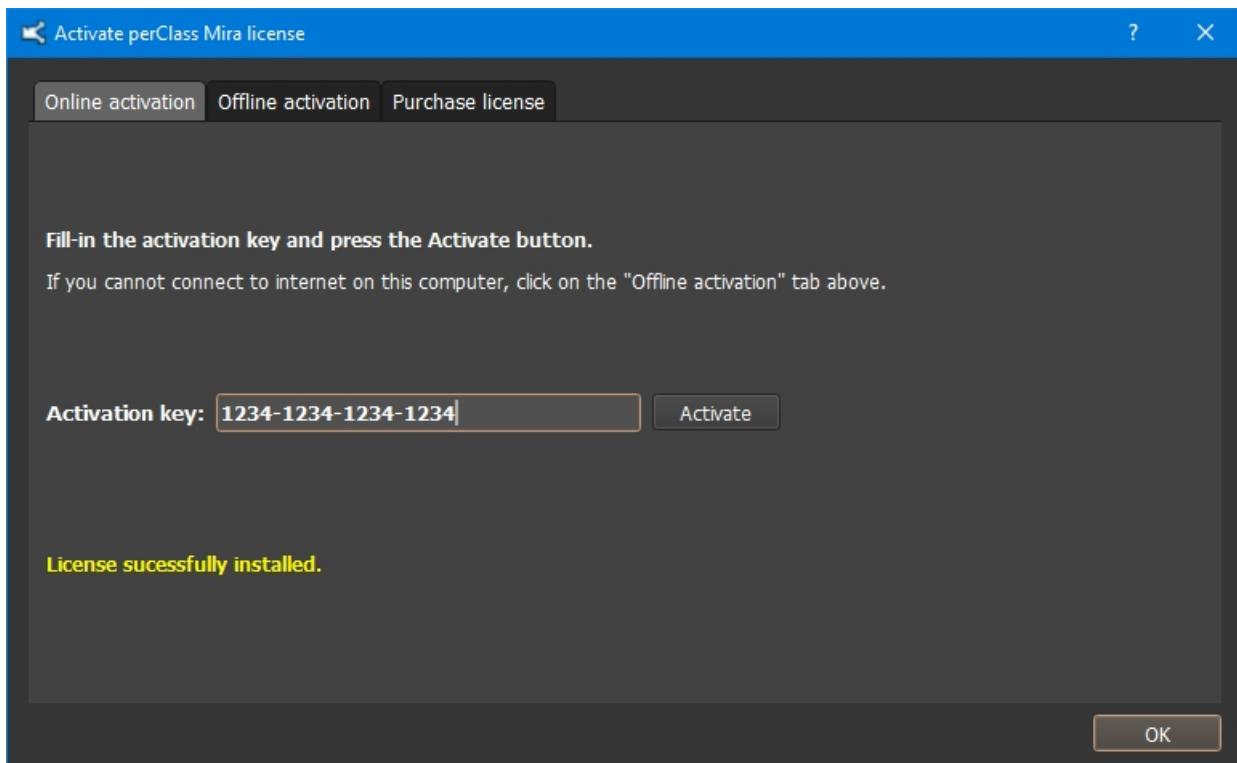
When starting perClass Mira for the first time, there is no license file present. Therefore, an activation dialog appears:



You need to provide an activation key that will pull a license, specific to your machine.

The key has twelve digits in a format such as: 1234-1234-1234-1234. Naturally, you need to use the key you obtained for a demo or for the full version, not this example.

By clicking *Activate* button, the license is installed:



TIP: The license file is stored in `c:\users\USERNAME\AppData\Roaming\perClassBV` directory together with perClass Mira configuration file.

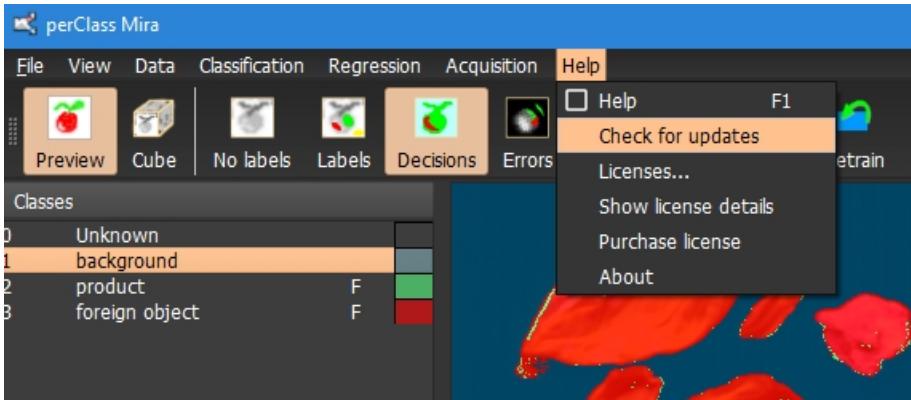
Uninstall

In order to uninstall perClass Mira, run the **maintenancetool.exe** application in its installation directory (by default **c:\users\USERNAME\perClass_Mira_2.4**)

Checking for updates

perClass Mira automatically checks for software updates. The default behaviour is that once in 10 days perClass BV server is contacted requesting the latest published software release. If a newer version exists a *Check for updates* dialog is shown with the details.

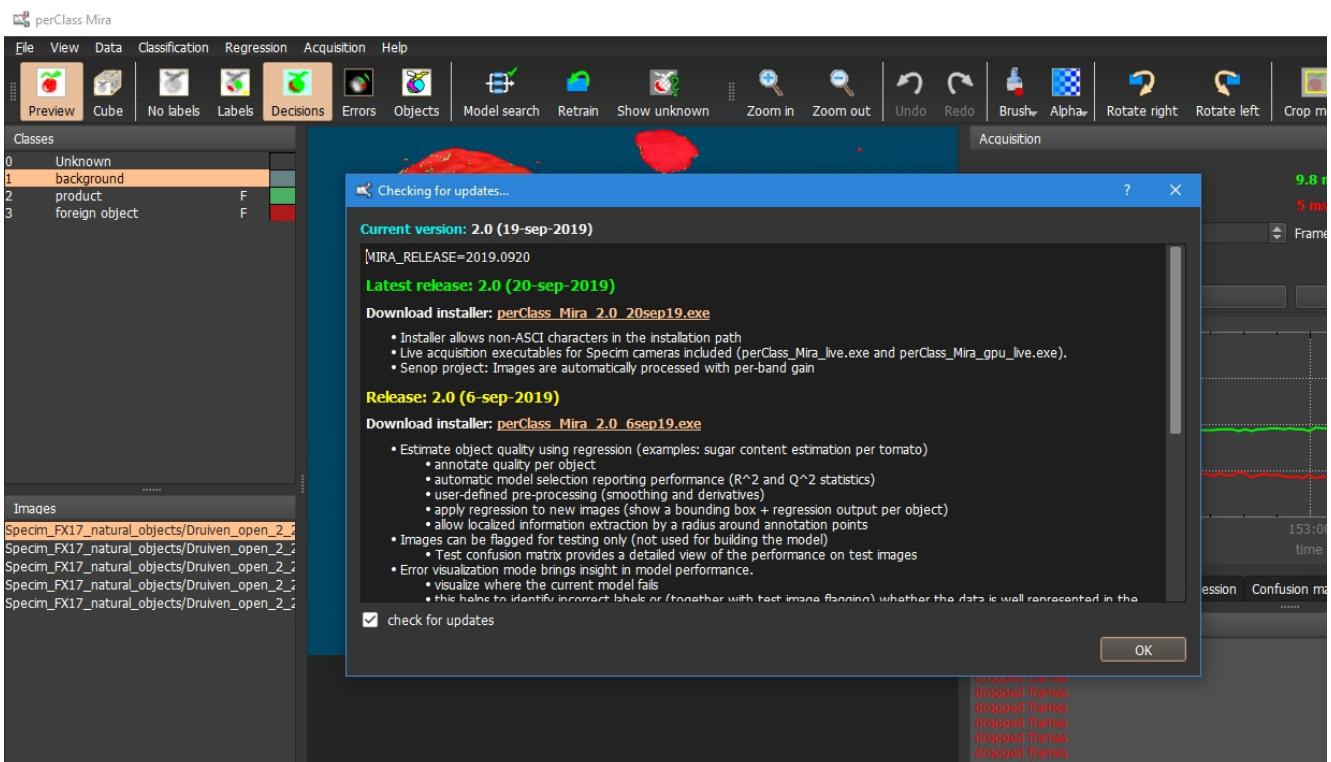
The check for updates can be also initiated manually from the *Help* menu.



The Update dialog shows latest release notes and download links.

The user needs to manually click on the download link and install the software.

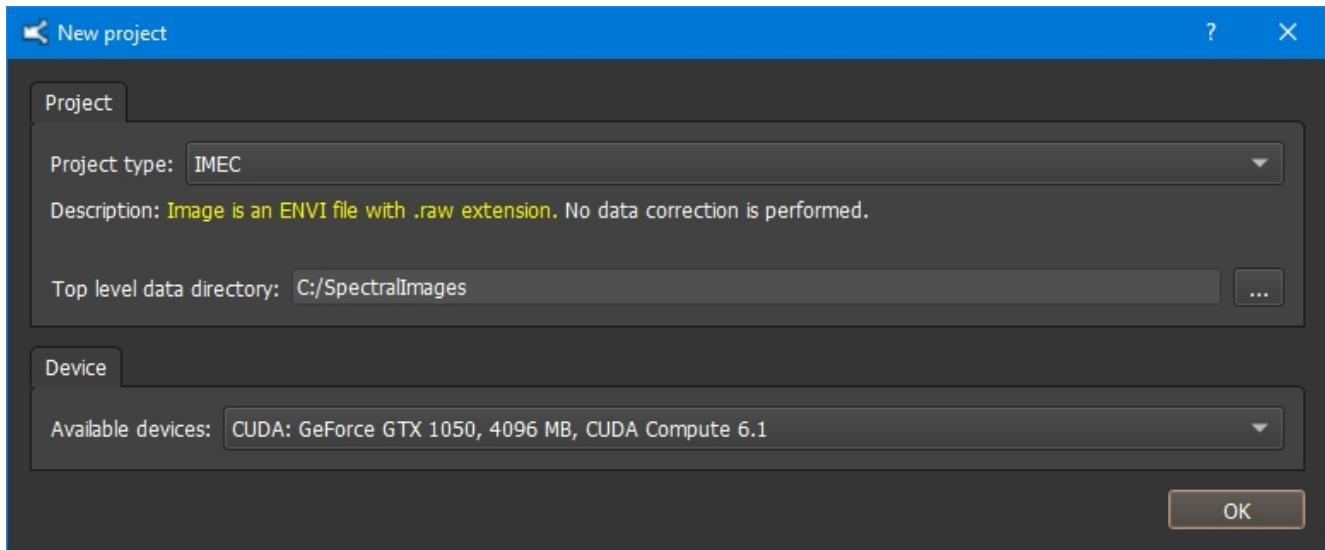
The default update checking behaviour can be switched off by the checkbox in the updates dialog or in the **mira.ini** file.



Walkthrough example

Selecting project type

Select *New Project* from *File* menu. A dialog will appear:



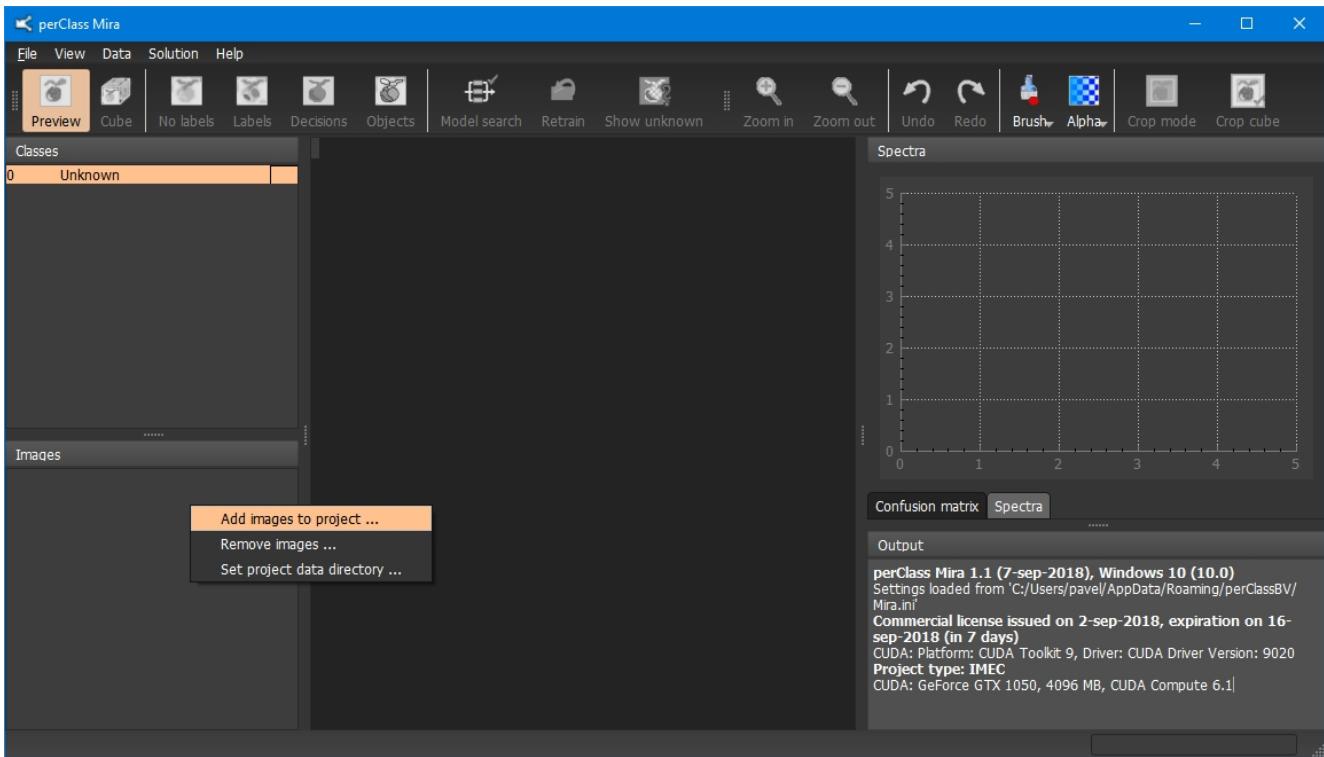
In the New project dialog, we need to [specify project type](#), top level data directory and computational device

- *Project type* specifies the type of images. For example, IMEC project type expects ENVI files with .raw extension.
- *Top level data directory* specifies a directory under which all files are located (possibly in sub-directories). The idea behind is that data is stored in a directory hierarchy e.g. data drive or a network drive. perClass Mira does not write into data directly (the only exception being Specim FX project when white correction is included in a scan using *Create white correction* command)
- *Device* combo box allows us to select computational device such as a graphic card.
 - **TIP:** To use NVIDIA graphics card, CUDA 9.0 or later needs to be installed. When installing CUDA, graphical card drivers are often bundled in the same installer. Make sure you are not selecting by default older graphics card drivers than already present on your system. We recommend always performed custom installation making sure that newest possible NVIDIA drivers are used.
 - Device selection is not saved in the project and may be changed anytime based on the current situation of the used computer system. To do that simply select *New Project*, select graphical card of interest and then *Open project* you saved earlier.

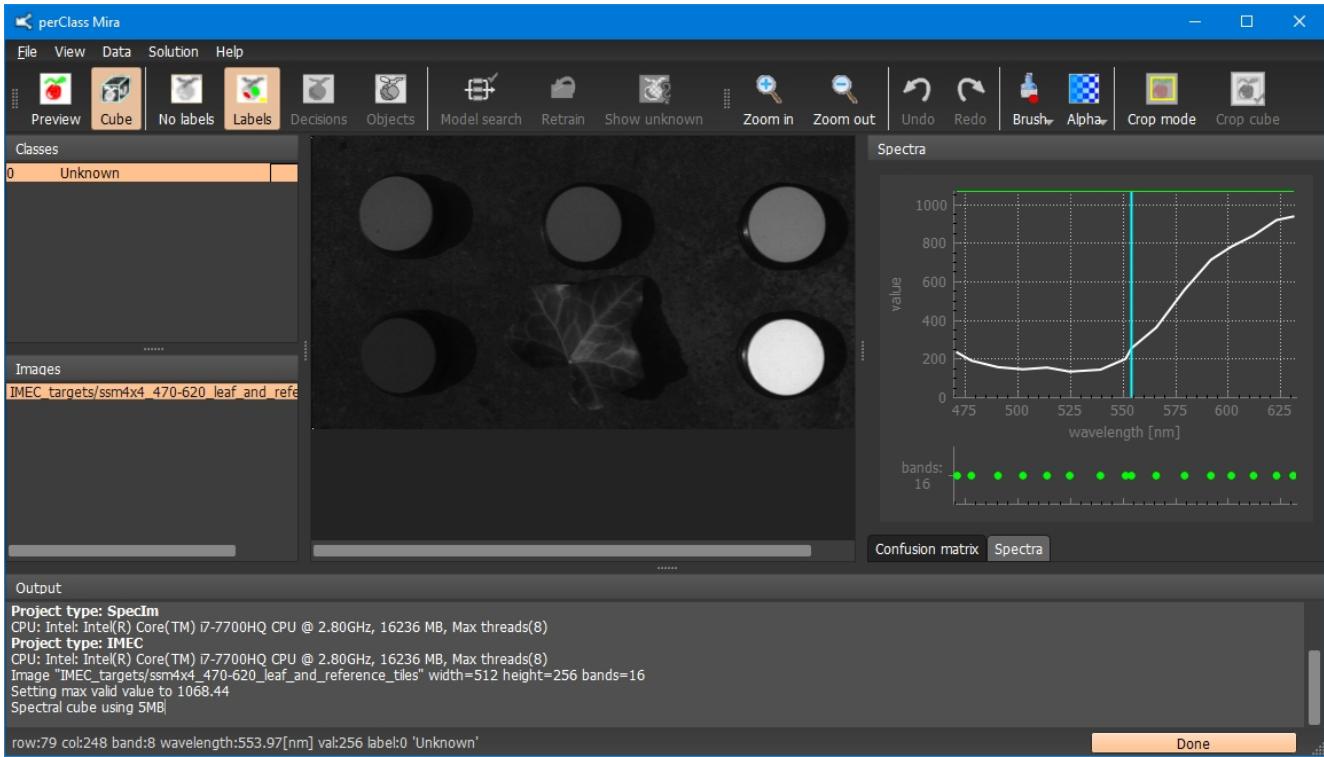
Adding images

Add image selecting *Add images to project...* command in:

- Context menu in the image list pannel (right-click) to open
- File menu



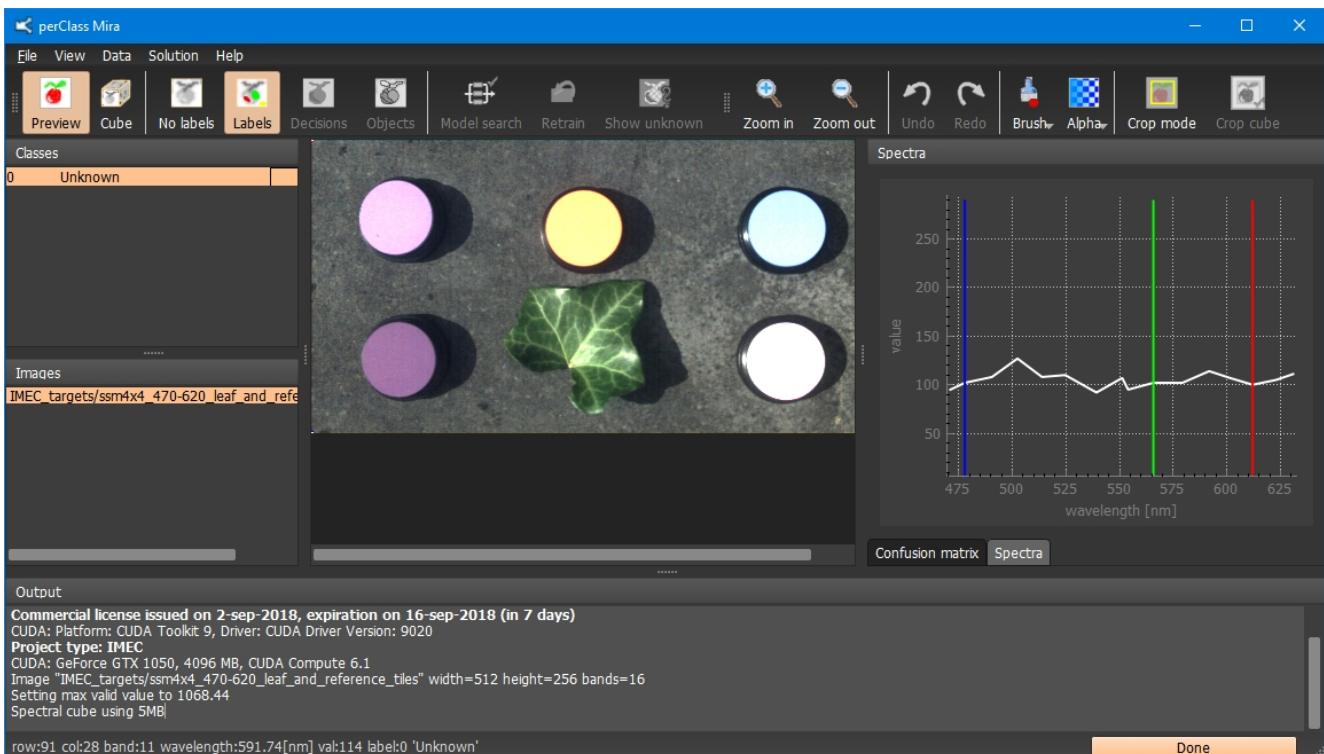
A file dialog will open where you can select one or more images.



RGB preview from selected bands

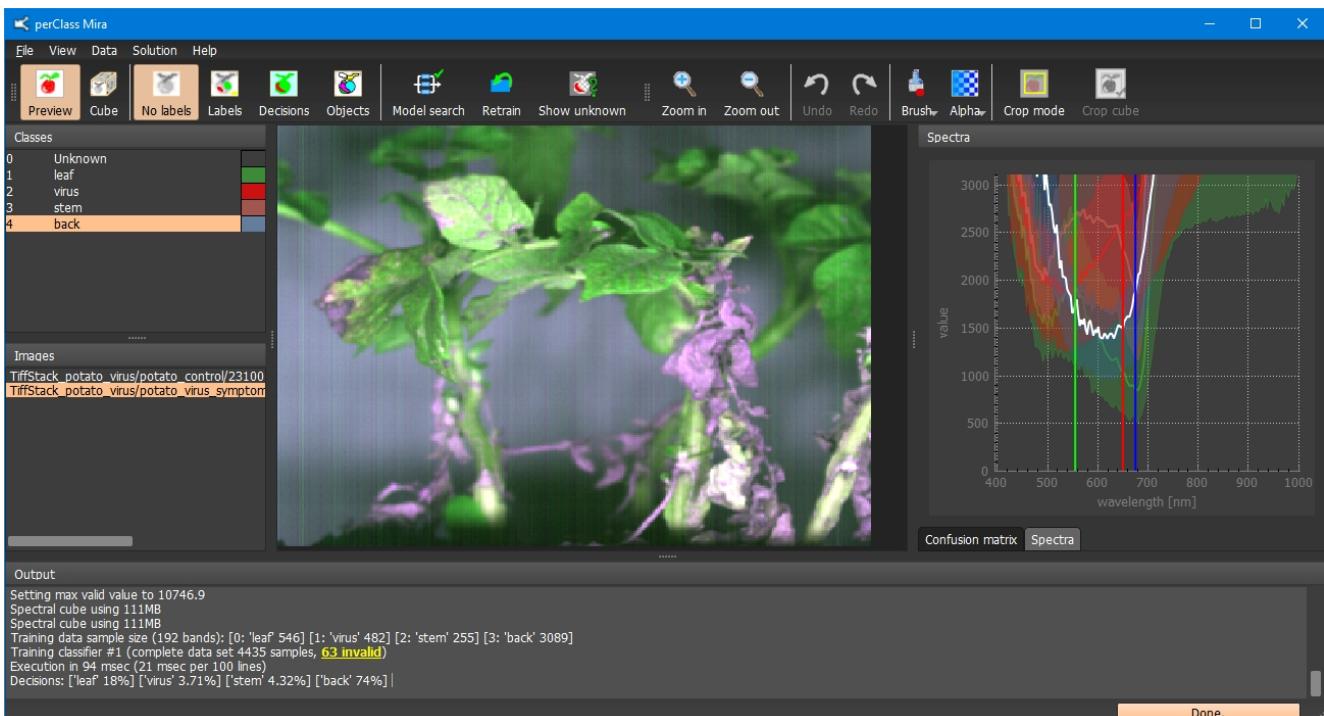
If a spectral cube is loaded, we may visualize RGB preview image from specific hand selected bands. When pressing the *Preview* button, the spectral plot will show R,G, and B band lines.

We may drag them to change the way preview image is generated:



TIP: To brighten or darken the image, use the mouse scroll wheel in the spectral plot

RGB preview is especially useful with natural objects and remote sensing scenes where certain types of materials may be visually distinguished simplifying the labeling process. For example, the image below shows a potato plant with some leaves infected with a virus (courtesy WUR University). Note that the virus-infected leaves exhibit distinct color in RGB preview. We would not be able to distinguish them on any single band image of the same cube without prior knowledge or without considering specific macroscopic folding of the leaves.



Selecting spectral band

You may select a spectral band in several ways:

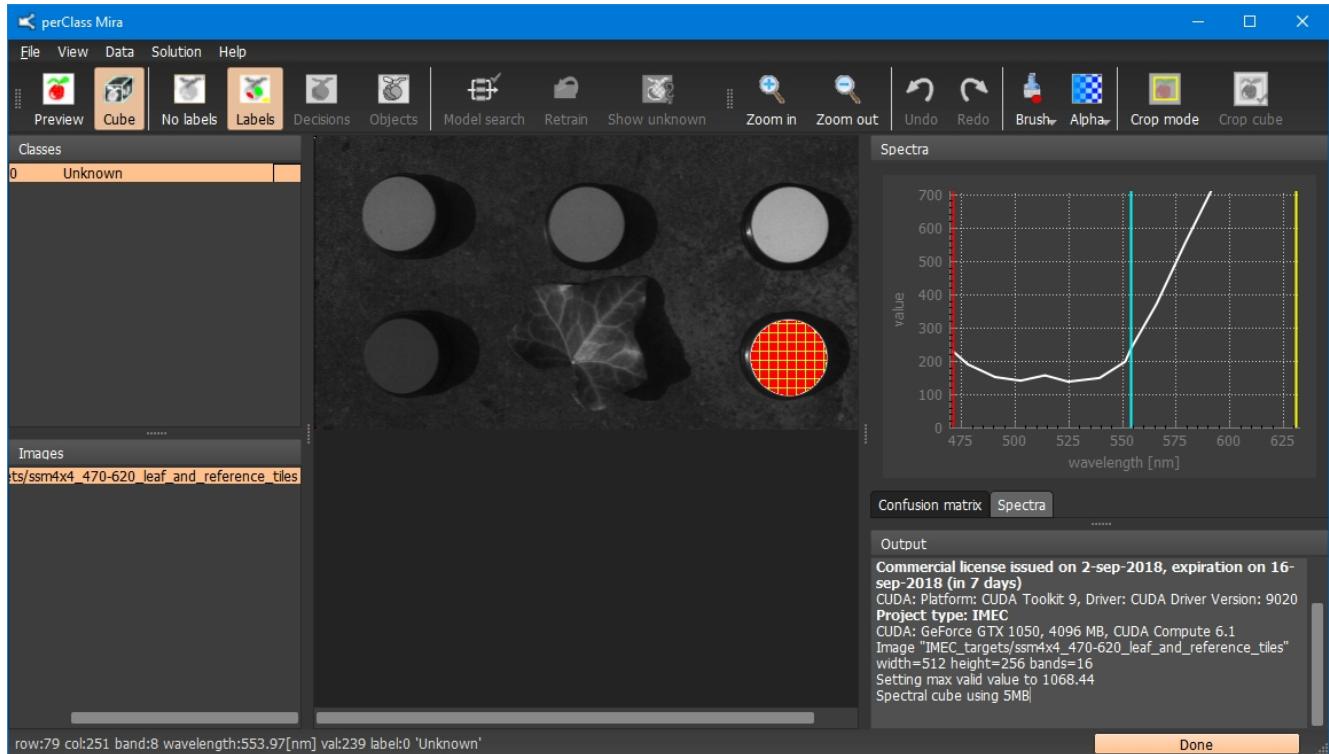
- By clicking in the spectral plot panel on a desired band
- Via Page Up/Page Down keys
- By a mouse scroll-wheel when holding Shift key in the image view

Adjusting display range

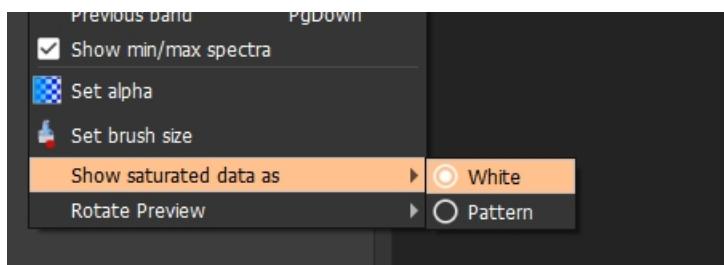
We may wish to adjust visualization range to see more details in dark image areas. This is possible using mouse scroll-wheel in the spectral pannel

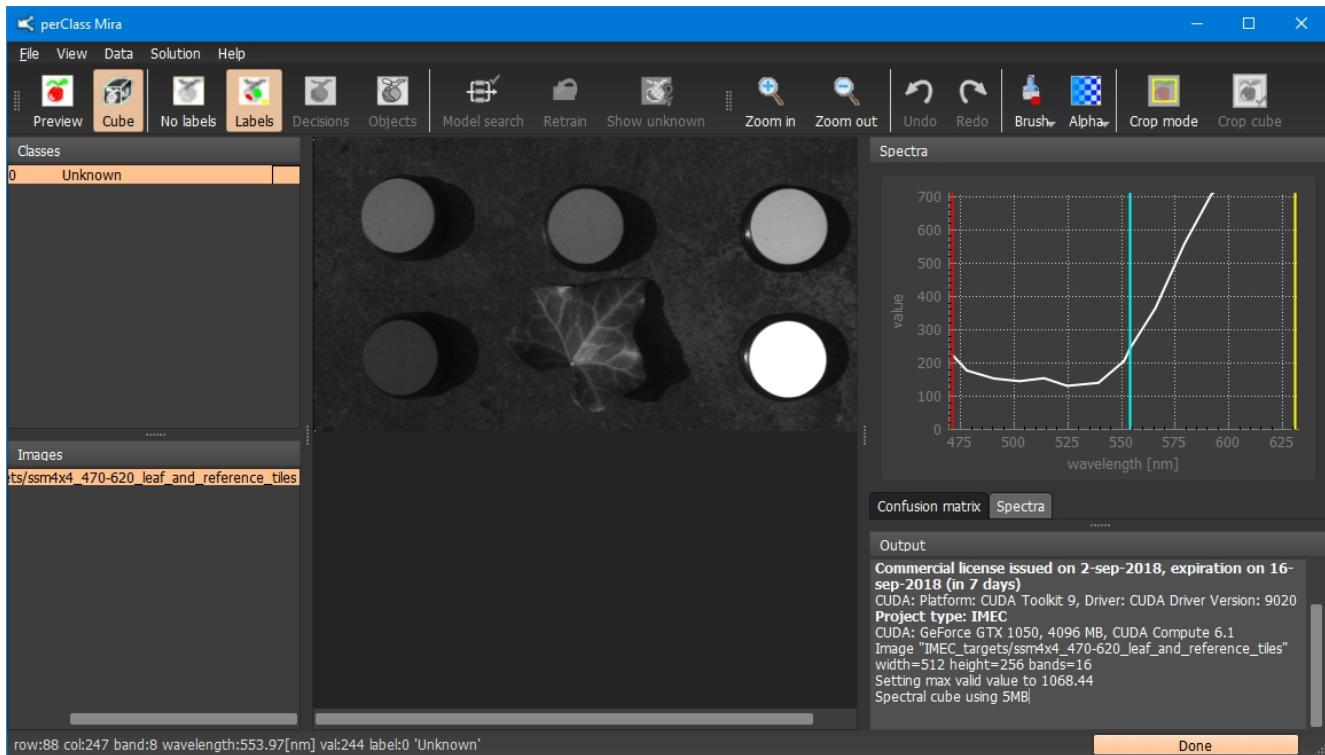
Alternatively, if spectral pannel is selected (e.g. by clicking) you may use cursor Up and Down keys.

By default, image pixels with a value out of the current display range is rendered in red/yellow cross pattern. This is to highlight out-of-range areas.



It is possible to change the out-of-range rendering style in by *Show saturated data as* command in the *View* menu. Alternative style is to paint out-of-range values as white:



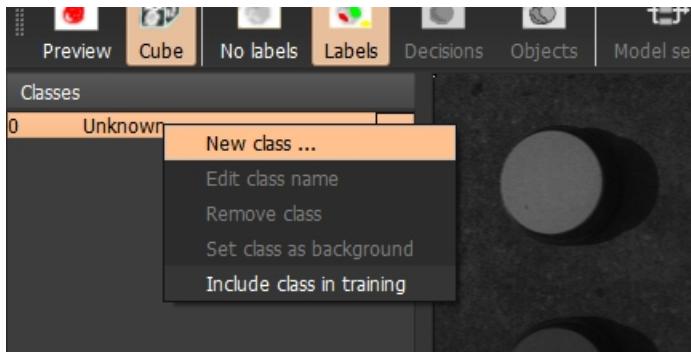


Defining classes

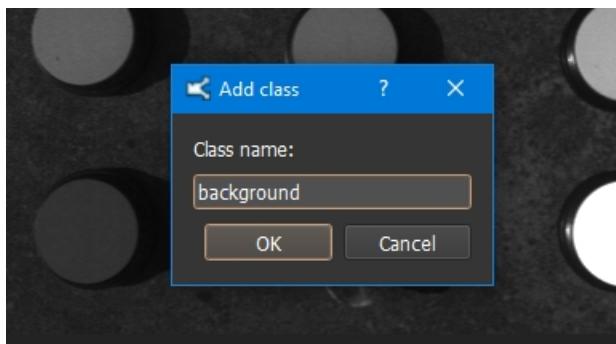
To build classification models, we first need to define classes of interest. We use the term "class" as a synonym of category - a group defining a material or function of a set of pixels. For example, in a potato segmentation task, we may want to define *leaf*, *stem*, and *background* as separate classes. Each pixel belongs only to one class.

perClass Mira interface presents a list of classes as a docked window, by default on the left side.

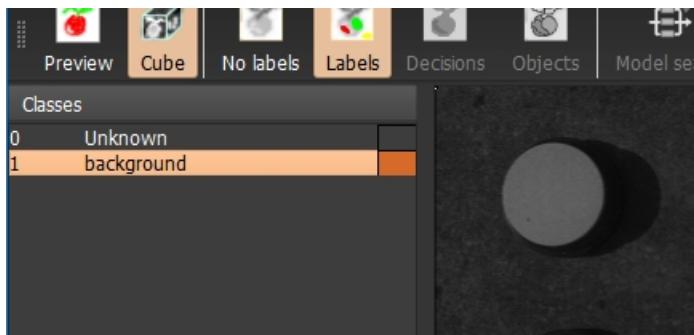
We may create a new class by right-clicking in the class list and selecting *New class* command:



A dialog window will open asking for the name of the new class:



The class is then added to the class list:

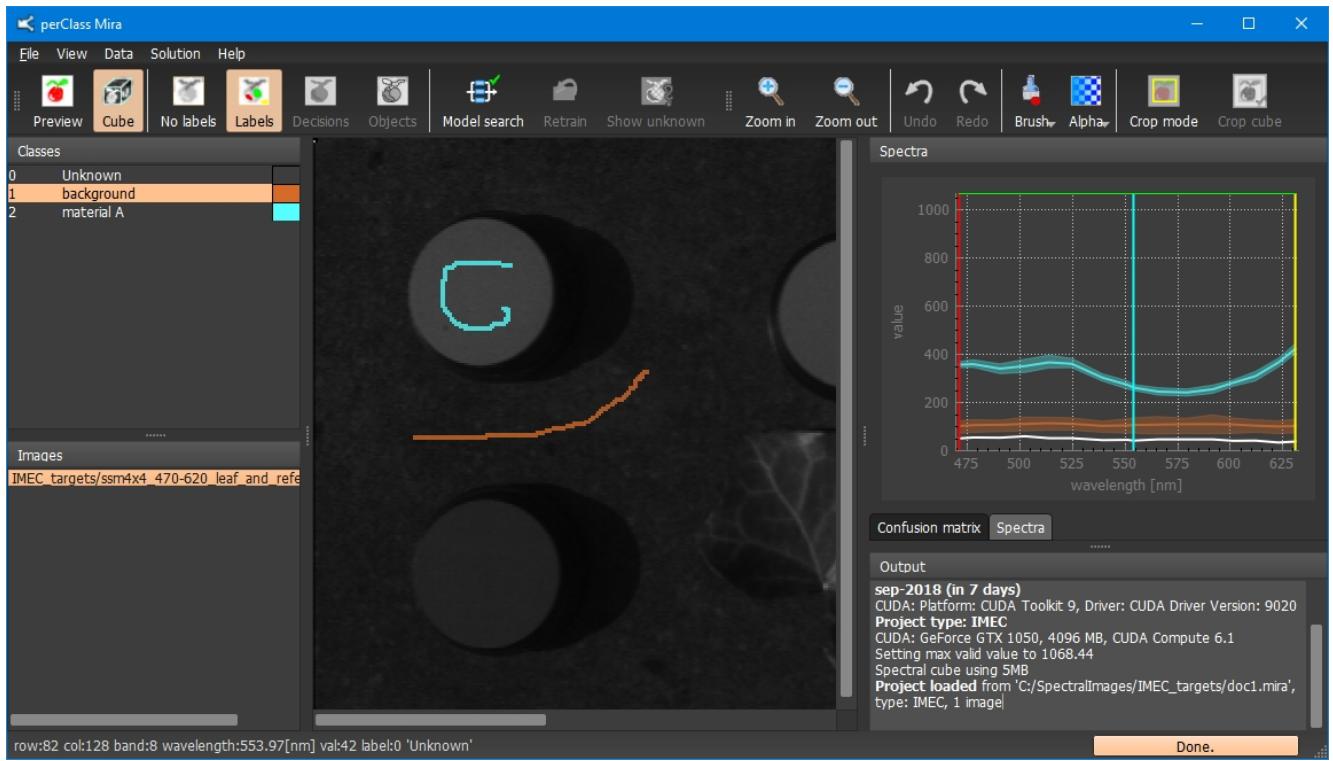


We may change class color by clicking on the color swatch widget in the right part of the list item.

TIP: To add a class quickly with a generated name and a random color, press *n* key

Labeling examples

Paint labels by selecting a class in the class list and moving mouse over image while holding the left mouse button.



You can see that spectral plot is updated after each stroke with information on class distribution. The thick central line represents class mean and the colored area min/max range for each wavelength.

TIP: To remove labels paint while holding *Shift* key or select the "Unknown" class.

TIP: To quickly switch between classes, press the corresponding digit key (e.g. press '1' to paint background in our example)

We may switch off label layer using *No labels* command on the toolbar.

TIP: We may switch to the last image layer by pressing *Space* key. In our example, this would flip between class labels and no labels.

Training a model

Once at least two classes with sufficient number of examples are labeled, we may build a statistical classification

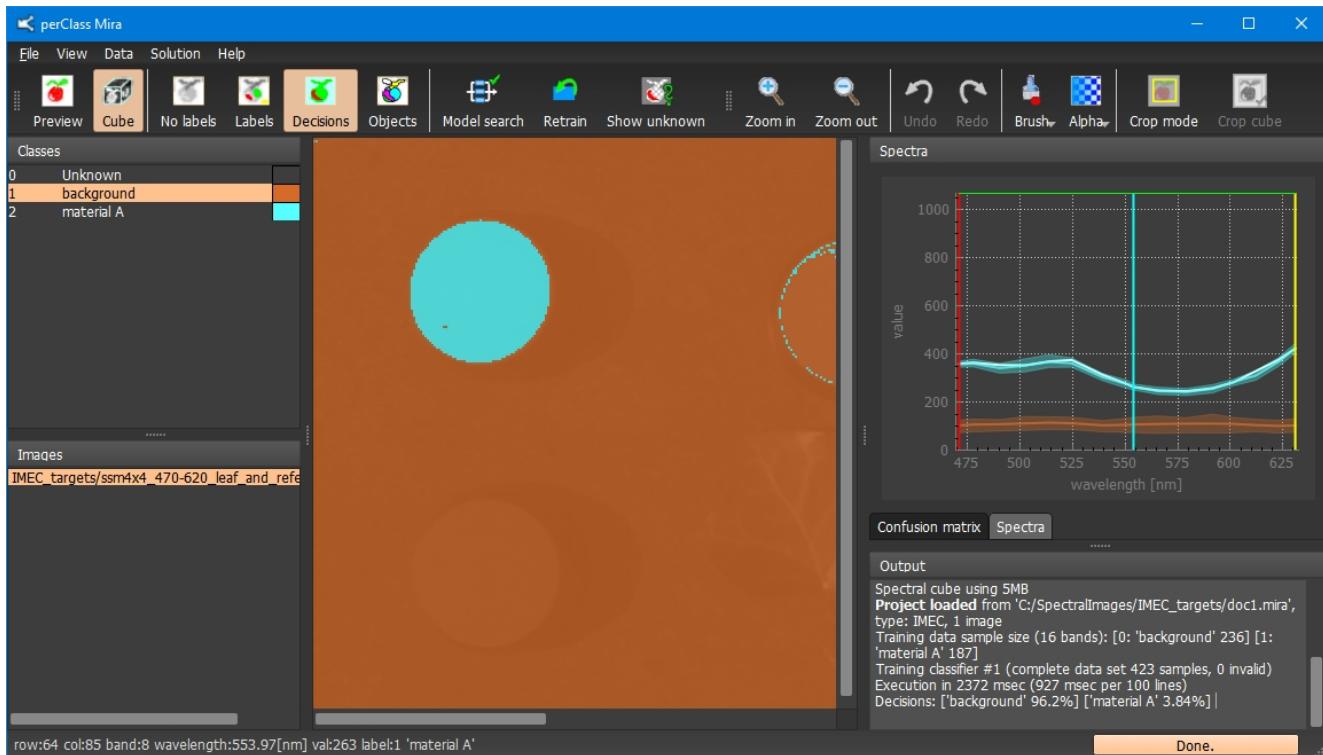
model.

perClass Mira uses a simple scheme for data validation that allows us to easily discard outliers. It is specified by the green line at the top of the plot representing the maximum valid value. The spectra with values higher than the green line will be discarded from training.

To train a model, we need to make sure that our training data is under this green line. We will discuss this in detail in [the next section](#). To adjust the max validation value, simply move the green line by mouse.

The first time, we need to select a good model using *Model search* command (or 'm' key shortcut). The stage of the search is indicated by the progress bar in the right-bottom corner.

After the search, the best data representation and model is used to label the entire image. Note that this automatically the interface to the *Classifier decisions* layer.



Each pixel in our image is labeled to one of the defined classes.

TIP: We may switch back to the label layer by pressing *Space bar* key

When we paint more examples, we do not need to re-run the search. Instead, may only retrain the best selected model (*Retrain model* command or 'r' key shortcut)

TIP: If we include many more examples adding new sub-types of our classes (e.g. dark bacground), it may be beneficial to re-run the model search. The reason is that different model may be relevant in this new situation.

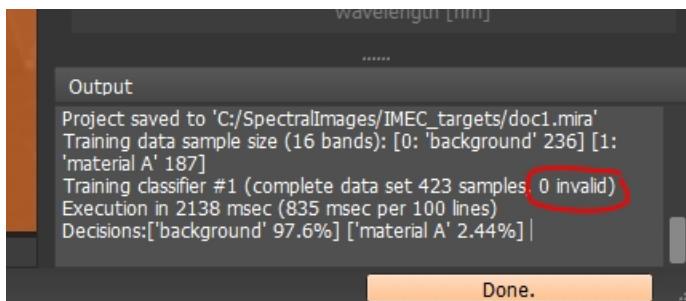
Sample validation

By default, perClass Mira 2.0 and higher leverages all labeled examples from training images to build a classification model. There are situations where some labeled examples should be excluded from training. For example, there may be artifacts and spikes on spectra, uninformative areas due to sensor saturation or defective pixels. In such cases a simple data validation mechanism can be applied excluding bad spectra from training.

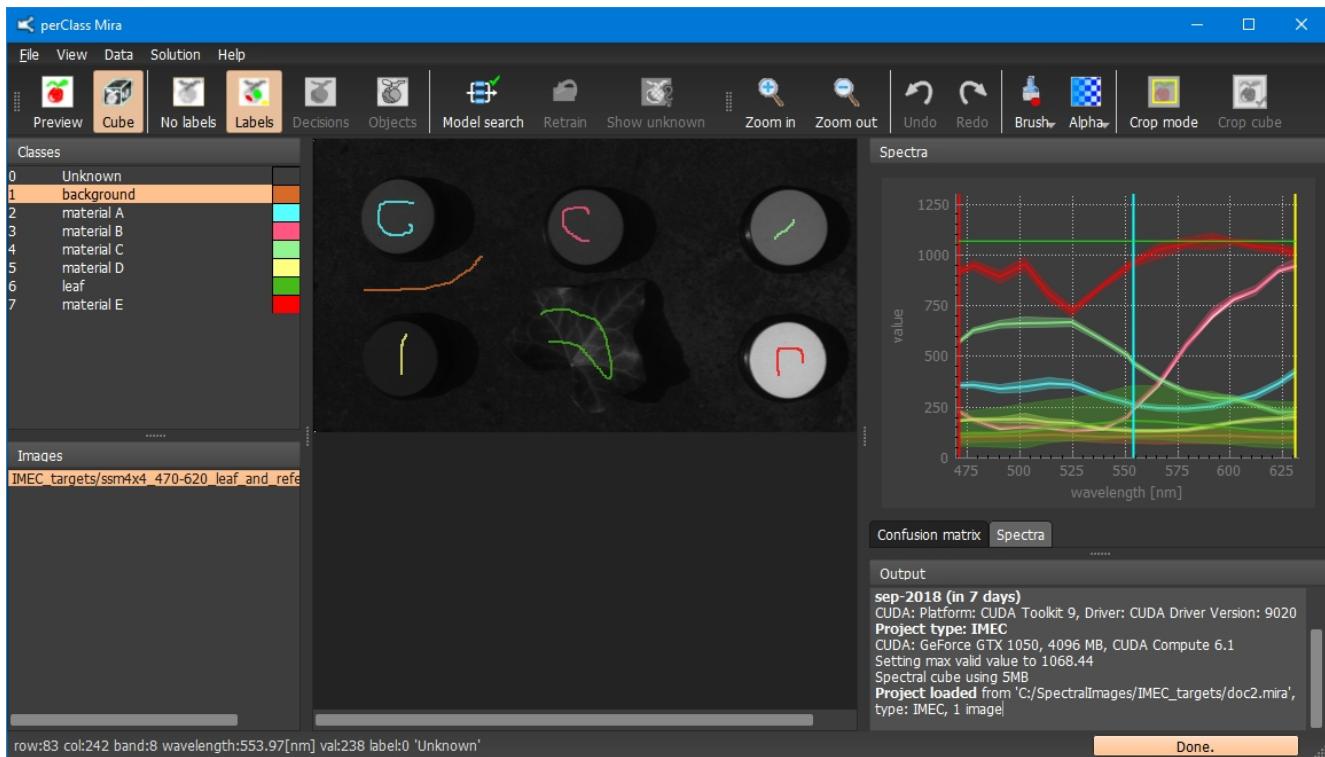
In the spectral plot, one may set a "maximum validation value" using the context menu. A green horizontal line will appear. By adjusting the green line in the spectral plot, we set a bound on spectra included in the training set. This allows us to remove badly scanned areas, outliers with spectral artifacts and other pixels if out-of-range values. Note, that this removal only concerns the training set, during model execution all image pixels are processed. For a method to highlight outliers in execution, see the section [Active learning](#)

You can remove the validation line by double-clicking on it.

Upon training, the number of invalid examples removed from the training set, is displayed in the output window:



If we create more classes in our example we can see situation as follows:

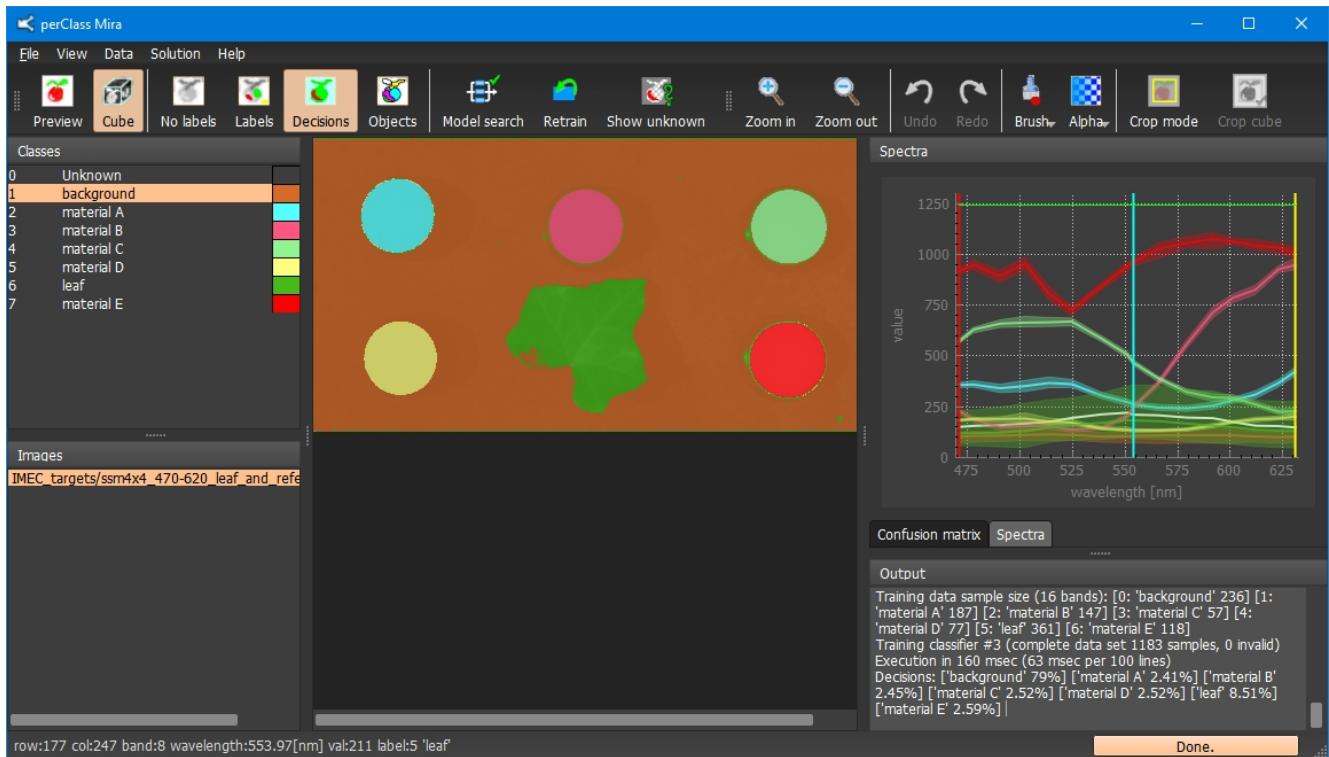


Some of the spectra of the red class "material E" are above the green validation line between 575 and 605 nm.

If we try to run to train a model, we will see the number of samples for "material E" highlighted in red due to insufficient number of examples (less than 50):



The solution is simple: We will move the green line in the spectral plot above all relevant data. Now, the classifier can be built just fine using all labeled examples:

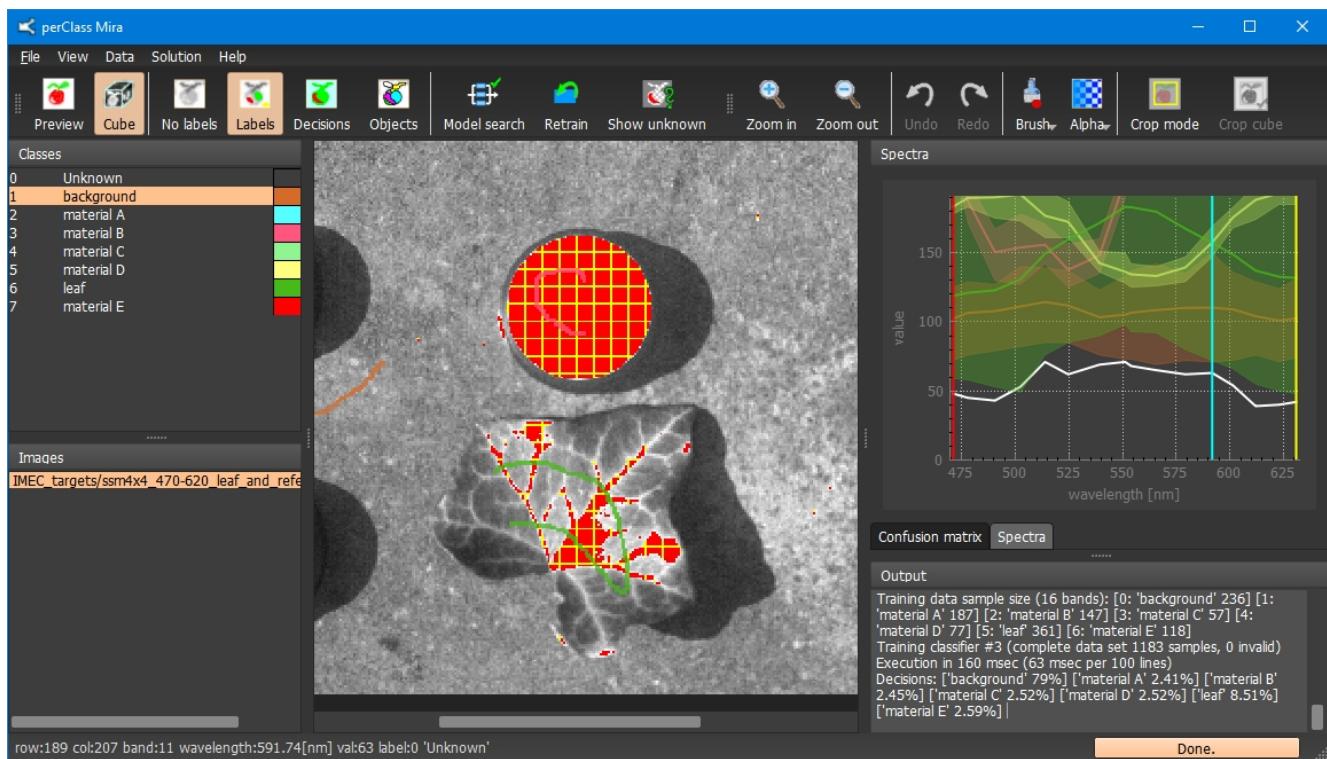


Improving the model

Labeling more examples

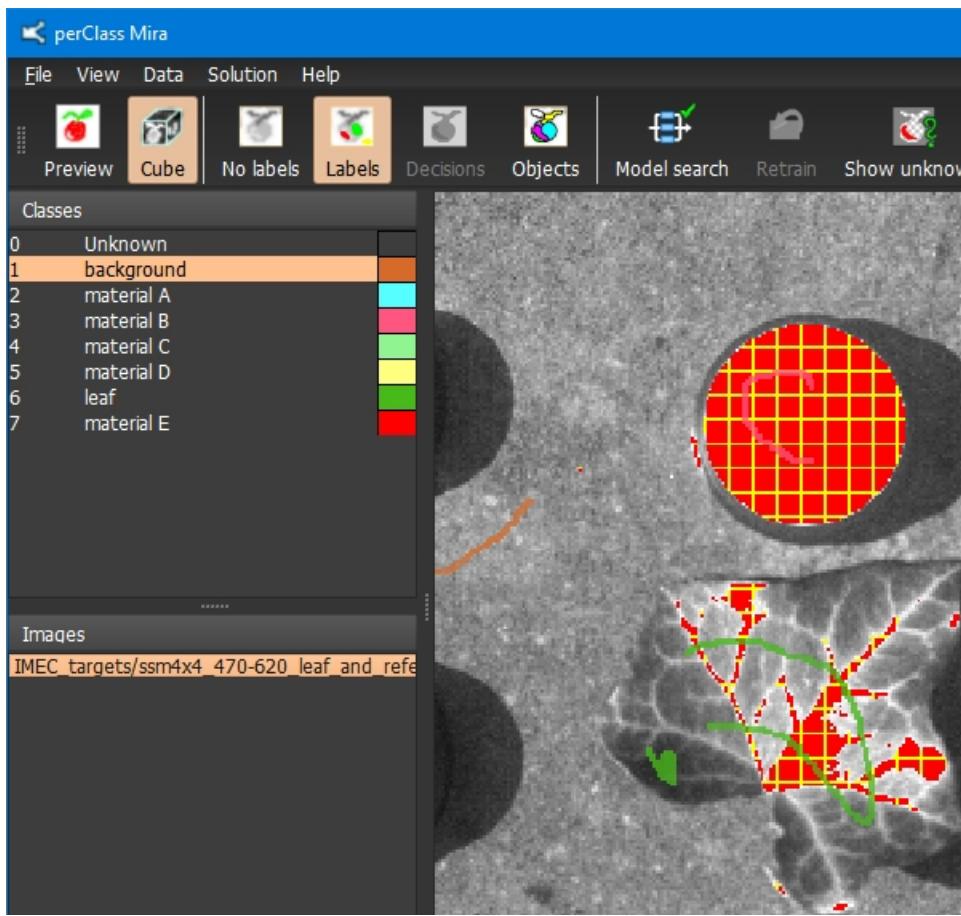
We may improve the model by labeling more examples in areas where classification does not work as expected.

For example, the leaf in the center of the image has a hole labeled as background. We may first inspect, how the area looks when zooming. We also adjust the spectral display range to see more details in dark areas:

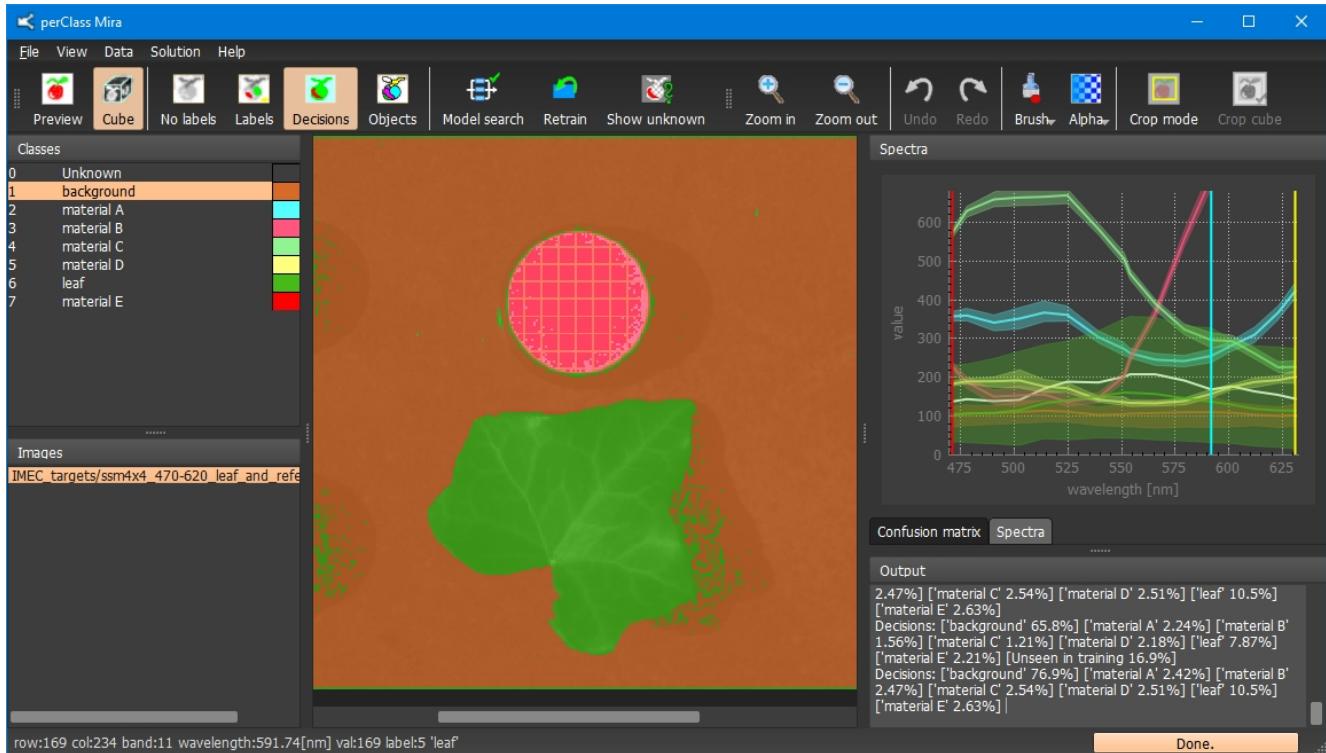


We can see that indeed the leaf is without any hole and provide more examples of the leaf class in this area.

TIP: To select a class based on image we may right-click on a desired pixel (labeled as 'leaf') and select Set 'leaf' as *current* in the context menu. Alternative is to press 't' keystroke while hovering mouse over any pixel of the desired class.



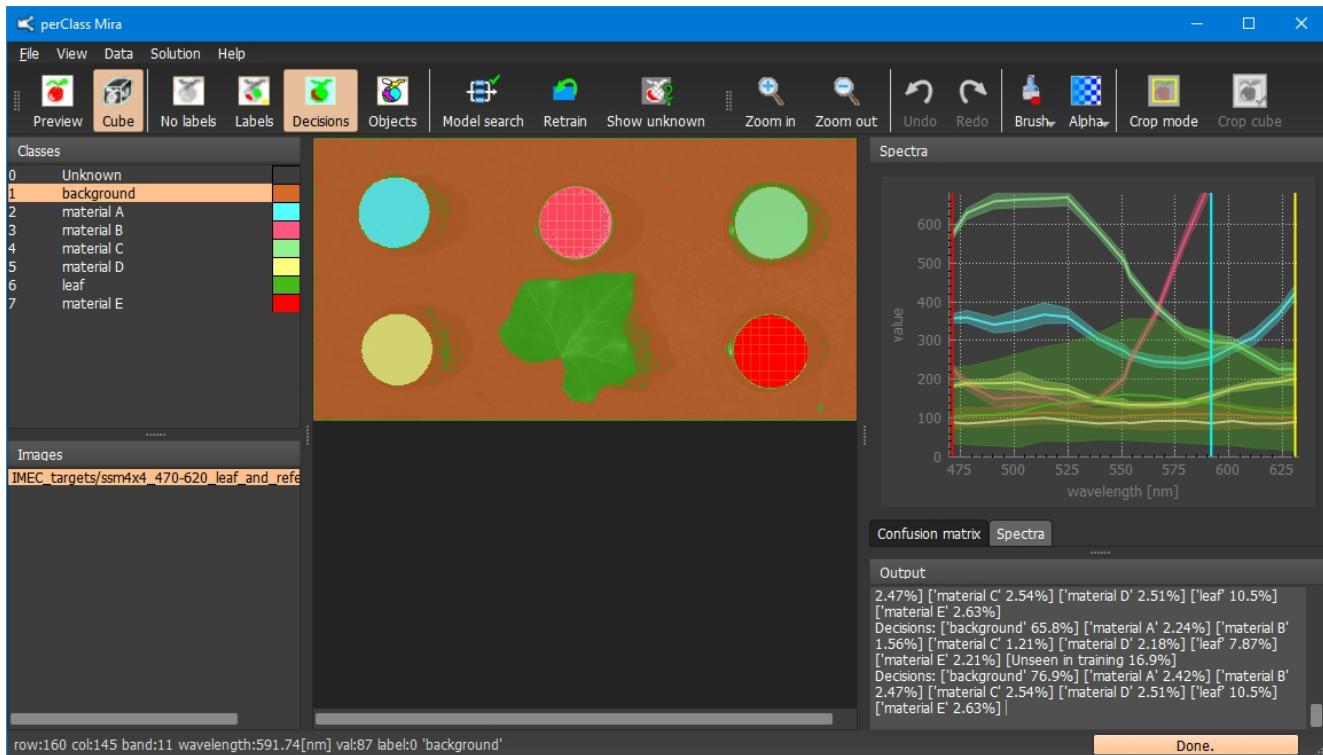
Now we may retrain the classifier. The hole is closed. Note the problematic classification in the right part of the leaf. We will discuss it in the next section.



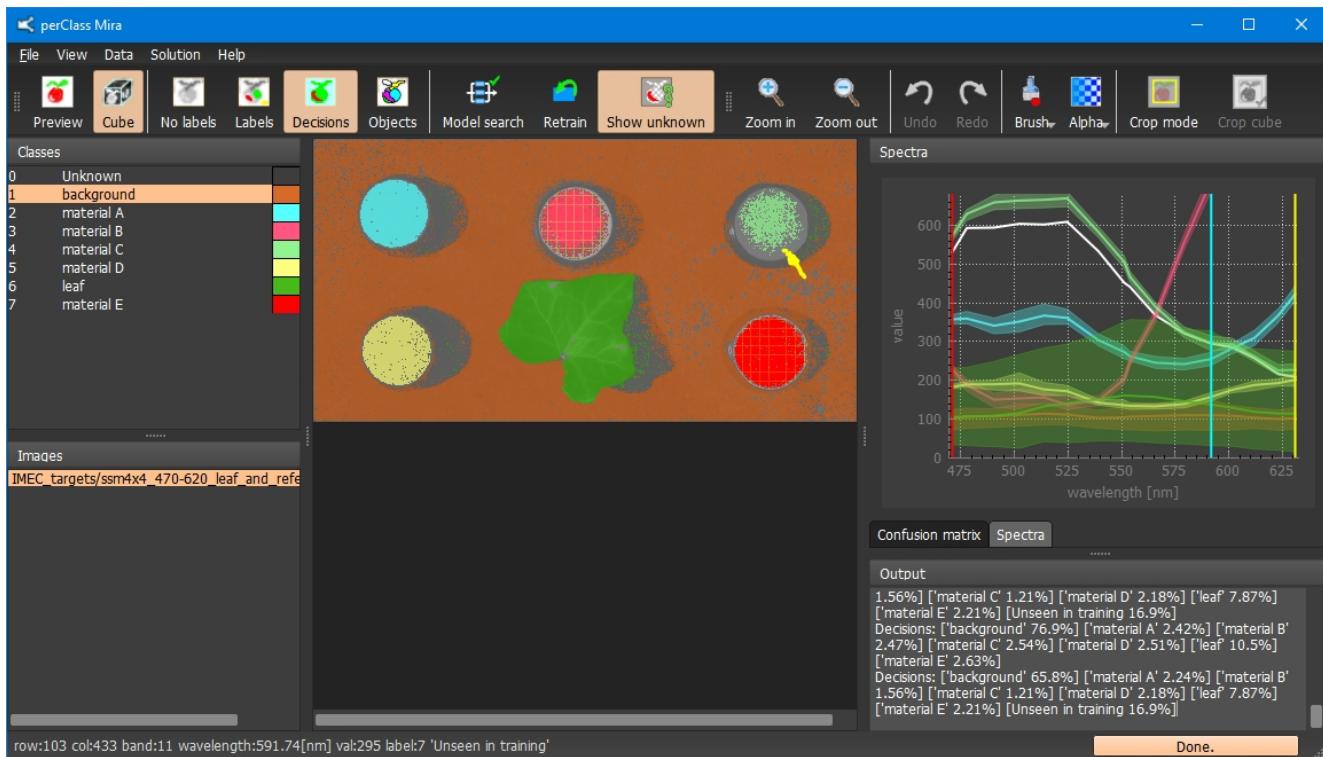
Active learning

perClass Mira provides easy to use active learning functionality offering easy-to-use guidance in labeling the most relevant examples.

Let us consider the classifier we've built so far:



To enable active learning tool, click *Show unknown data* button on the toolbar (or 'u' keystroke). Portions of the image will be now highlighted in transparency:

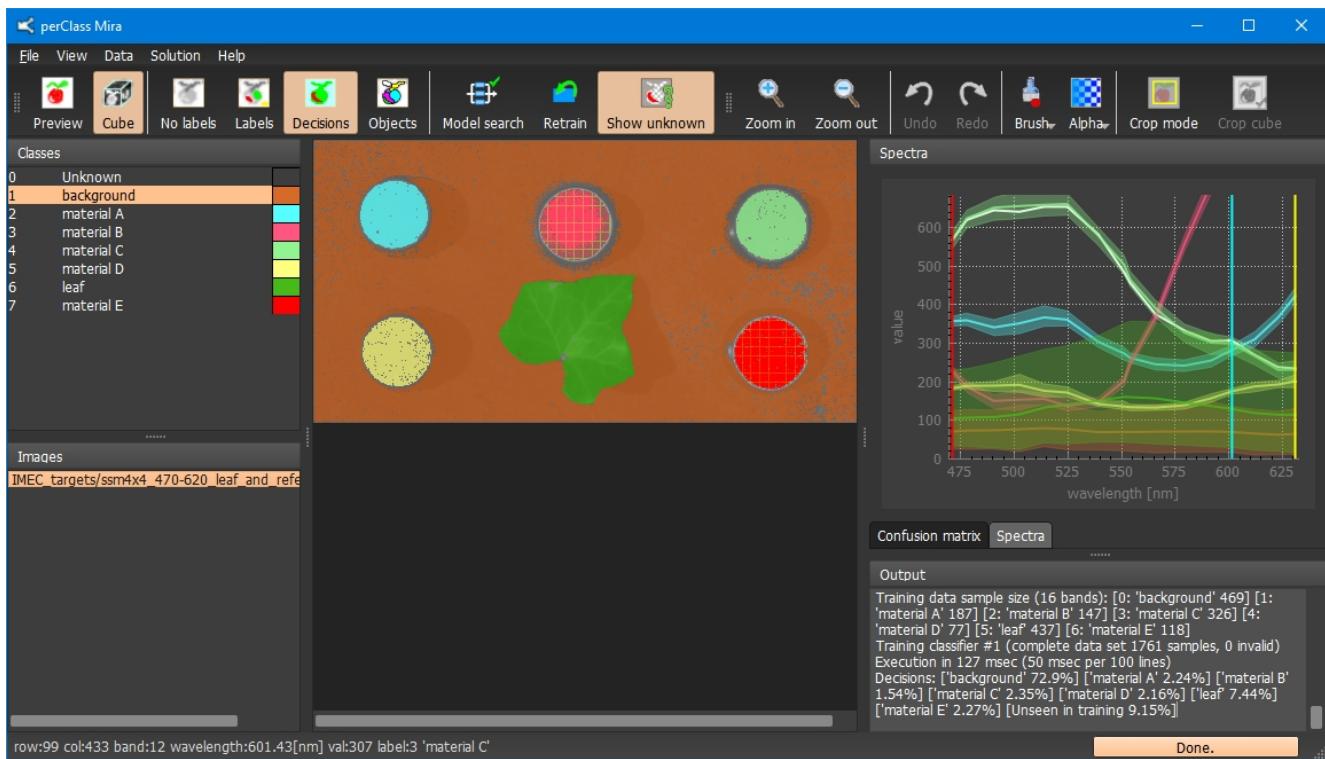
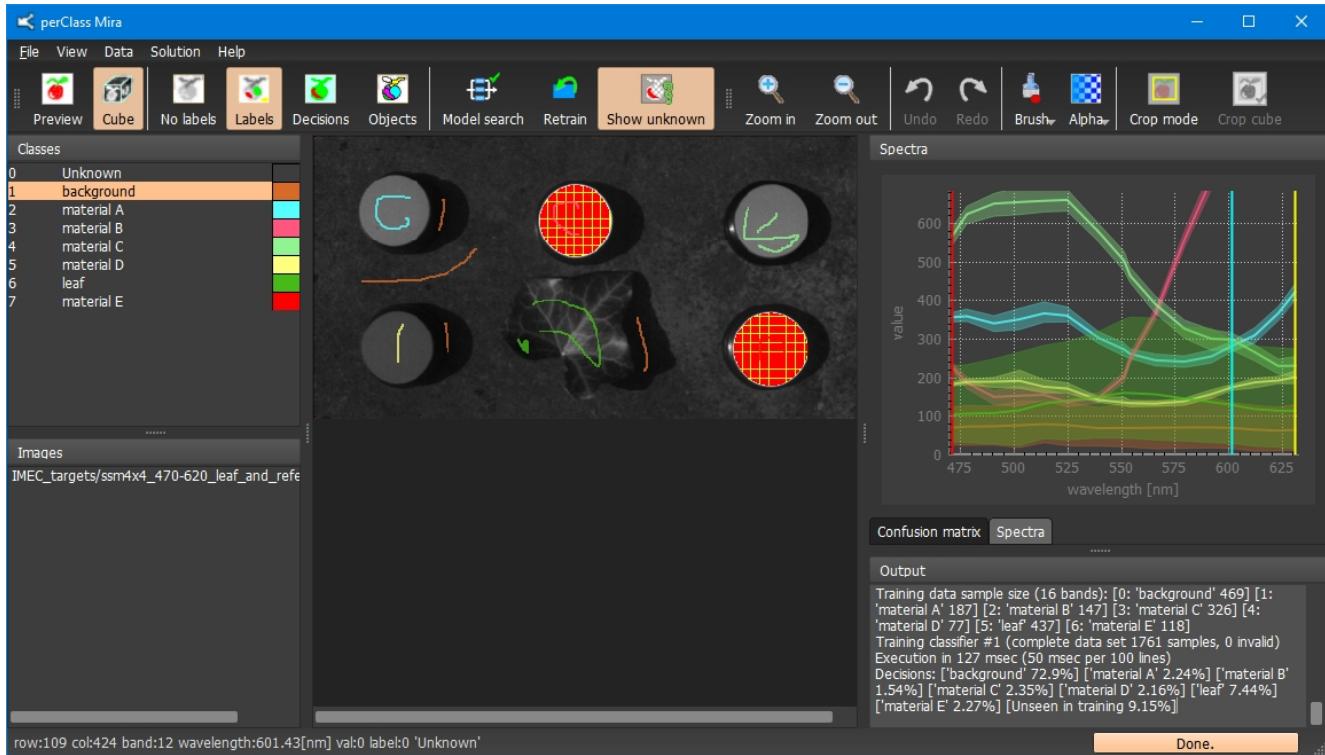


The yellow arrow above the image indicates the position of mouse pixel and the respective spectrum is given as white solid line in the spectral pane.

The *Show unknown data* tool highlights areas that were not seen in model training. Although, we may get good quality results for these examples, these are only extrapolations for situations not known to the model.

One insight from this result is that we need to label more background examples in dark areas. Also the "material C" target highlighted by the yellow error needs better labeling.

Here is example of improved labeling:



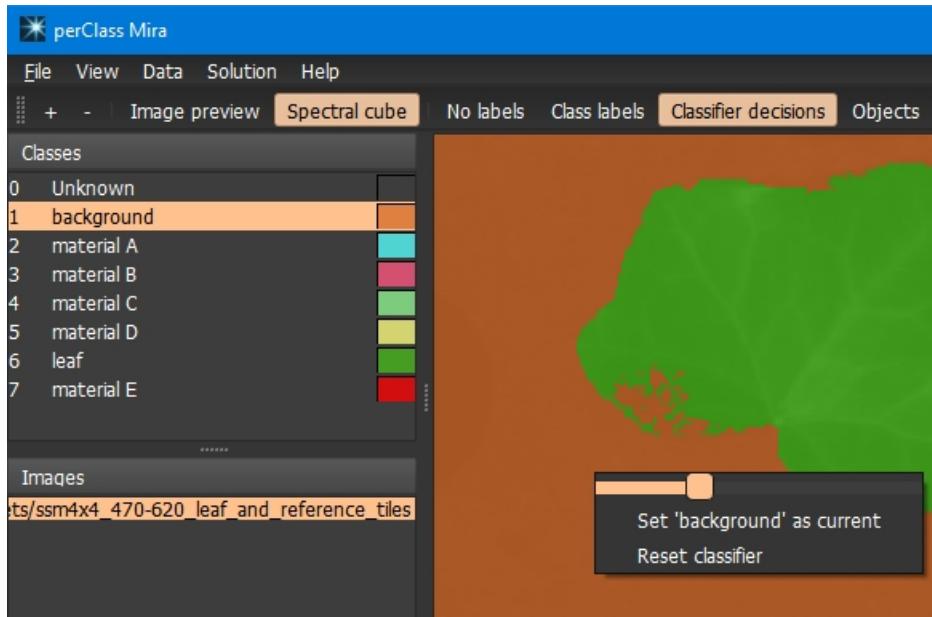
We may now wish to improve also the "material B" definition and maybe label target rims as background or introduce another class for this material.

TIP: Although perClass Mira can accomodate multi-modal classes such as the background, it is beneficial to define as many clearly distinct material categories as possible. This helps the machine learning engine to find better overall solutions.

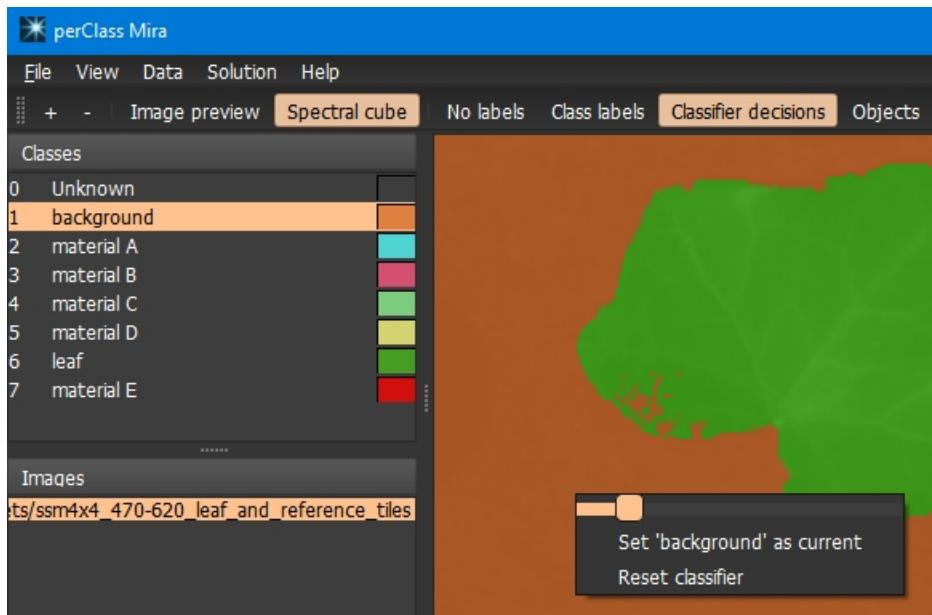
Tuning model performance

Alternative way to improve classifier decisions is to fine-tune model to desired performance.

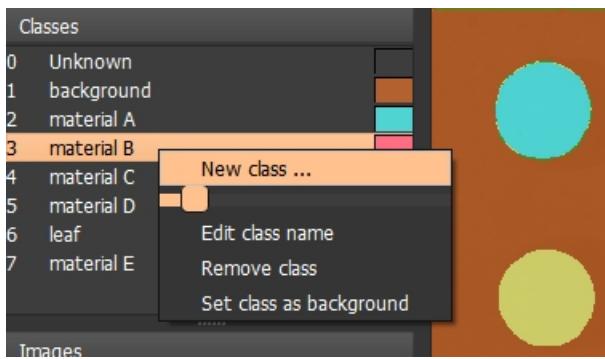
This can be achieved by right-clicking on any pixel of a class of interest in the decision image. The context menu will show a slider for the class under cursor.



Moving this slider, we can adjust different performance trade-off for the given class. For example, by reducing slider value on background, we express that it is 'less important' and, as a consequence will see the whole on the leaf diminishing in extent. This happens live as we move the slider.



TIP: Sometimes, classes of interest may not be easy to pin-point by mouse or entirely missing from the decision image. In these situations, it is easier to right-click on the desired class in the class list (while the decision layer is active) and use the slider:



TIP: To remove any performance trade-off adjustments, use *Reset classifier* command in the right-click context menu on the image layered with decisions.

Limiting spectral range

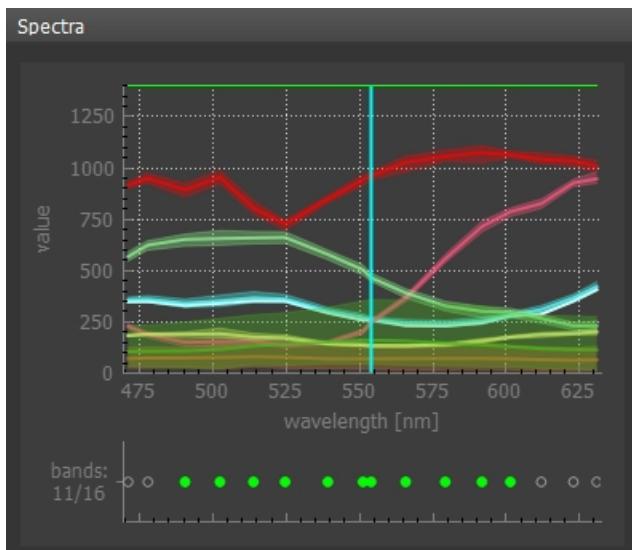
By default, all available spectral wavelengths are used to build classifiers. We may want to adjust the used spectral range from several reasons:

1. In this way, we may quickly validate whether specific range is sufficient to separate classes of interest
2. We may discard noisy spectral measurements at the start or end of the spectral range.

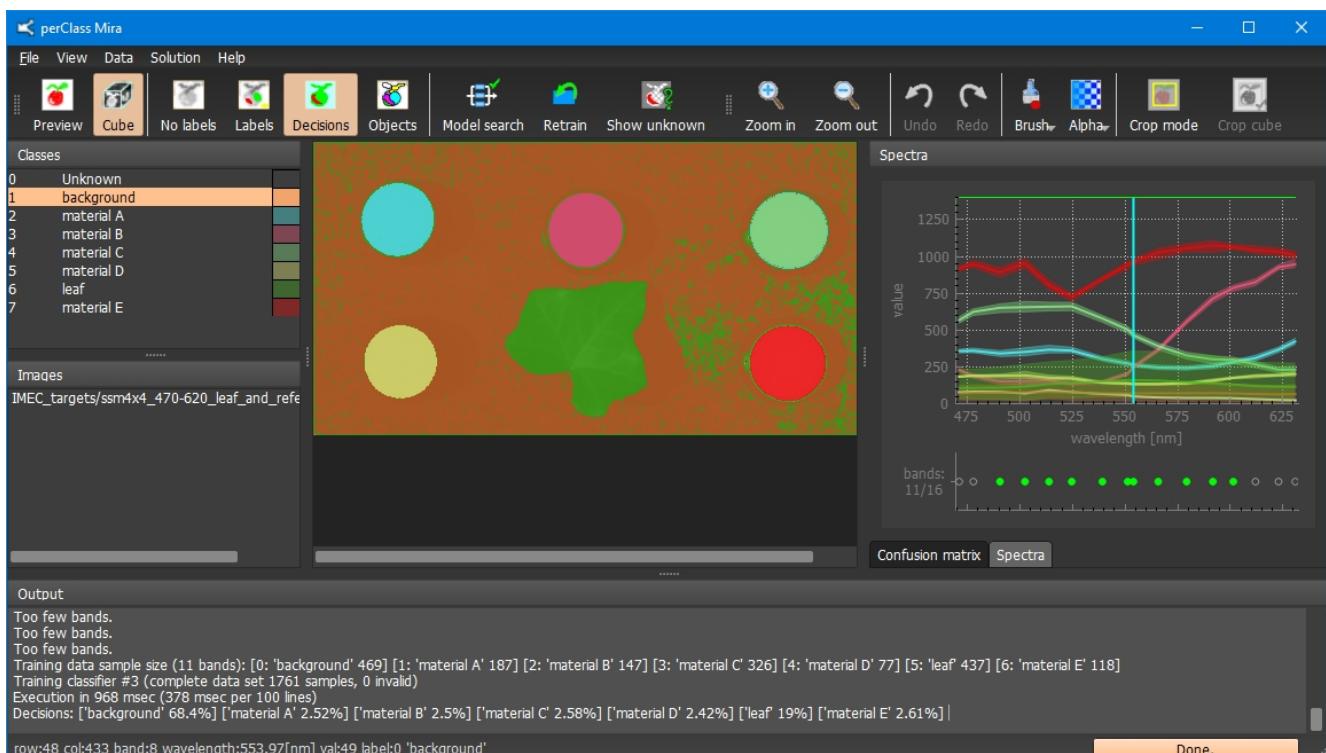
To adjust start and end of used spectral range, simply toggle spectral bands in the band selection bar under the spectral plot:



We disable bands at the start and end of the spectral range:



When retraining the classifier, we can observe that the "leaf" and "background" classes are more difficult to separate.



Executing on new images

To execute on new images, use *Add images* command in the image list. If decision layer is active, the current classifier will be directly applied to the selected image.

Note, that in one project, we can only work with images describing the same spectral range with identical number of spectral bands.

Exporting solutions for deployment

Once we are happy with the classification solution, we may export it for deployment outside of perClass Mira interface, e.g. in custom application.

This is accomplished with *Export classifier...* commands in *Solution* menu.

The result of the export is a .ppl file that can be loaded by perClass Runtime library (perclass.dll). This library

needs to be integrated in custom application to process new images.

For the latest perClass Runtime 5.4, it is possible to choose single or double precision floating values. Older runtimes require double precision spectral data on input.

Existing integrations include:

- IMEC HSIViewer contains perClass Runtime component that allows to execute solutions from within the Viewer interface

- Cubert Utilities application has perClass Runtime build in. Therefore, any solution exported from perClass Mira can be directly run on new images from Cubert Touch interface. Also, live spectral video can be processed by the perClass Mira classifier in real-time

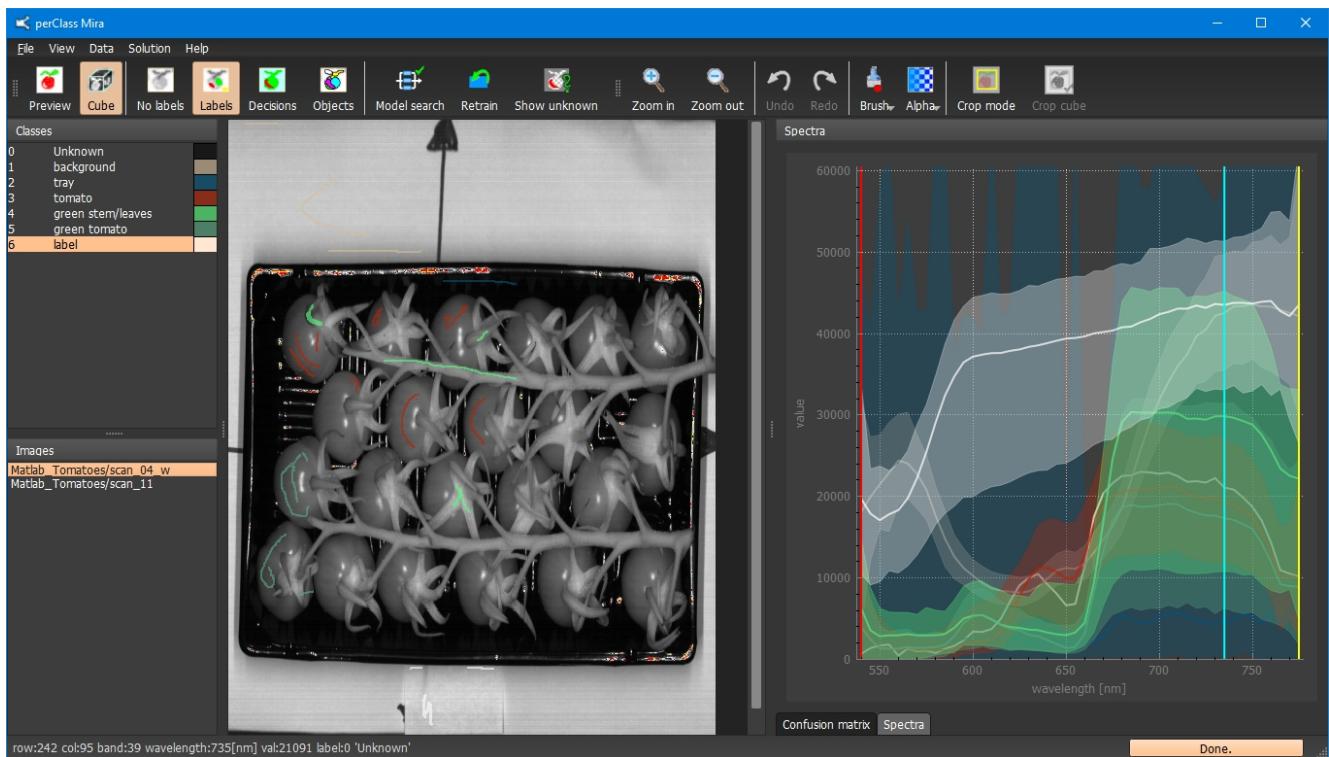
For details on embedding perClass Runtime library to custom application, see [online documentation](#)

Performance optimization

When designing a classification system, one needs to understand its performance. perClass Mira provides a confusion matrix offering a detailed performance break-down per class.

Confusion matrix highlights how labeled examples of different classes (ground truth) maps to classifier output (decisions).

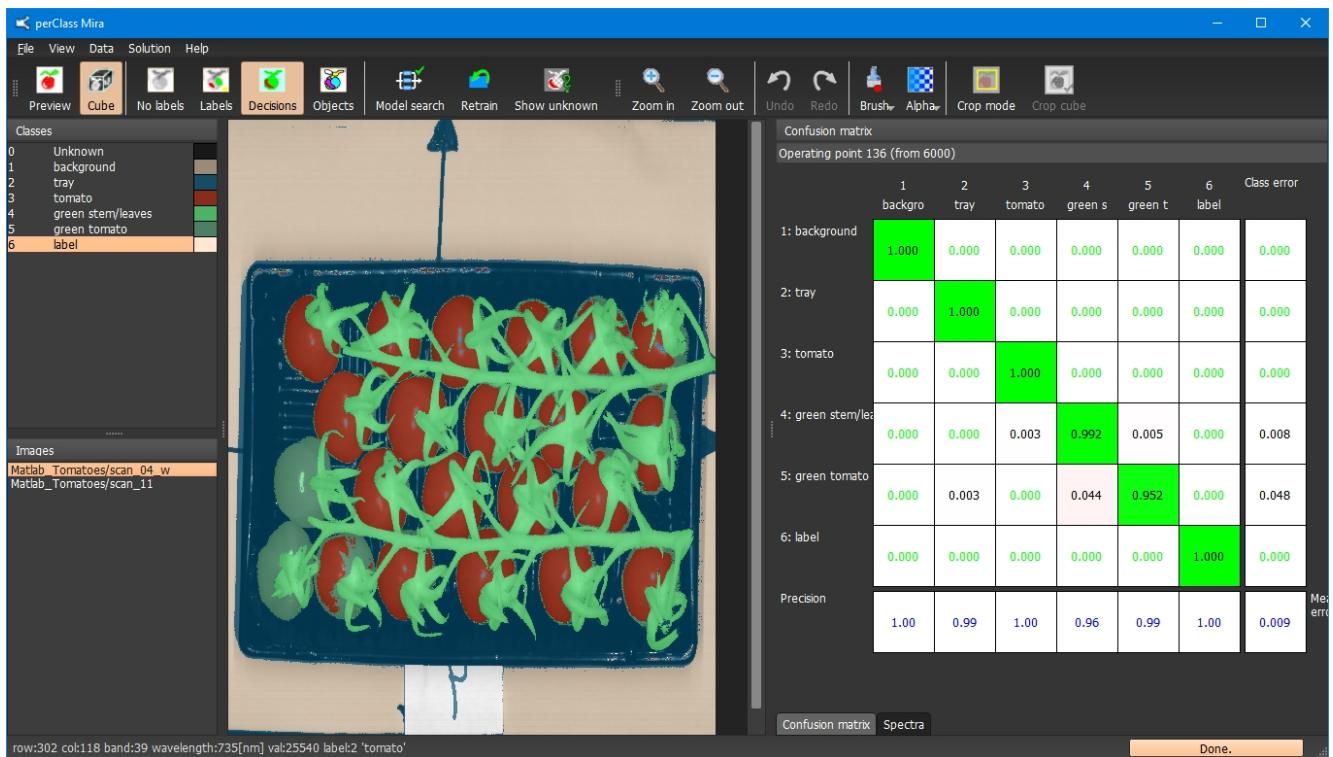
In this example, we have an image with a set of tomatoes in a box.



We have labeled six distinct classes.

When we build a classification model, we may switch to the *Confusion matrix* docked panel.

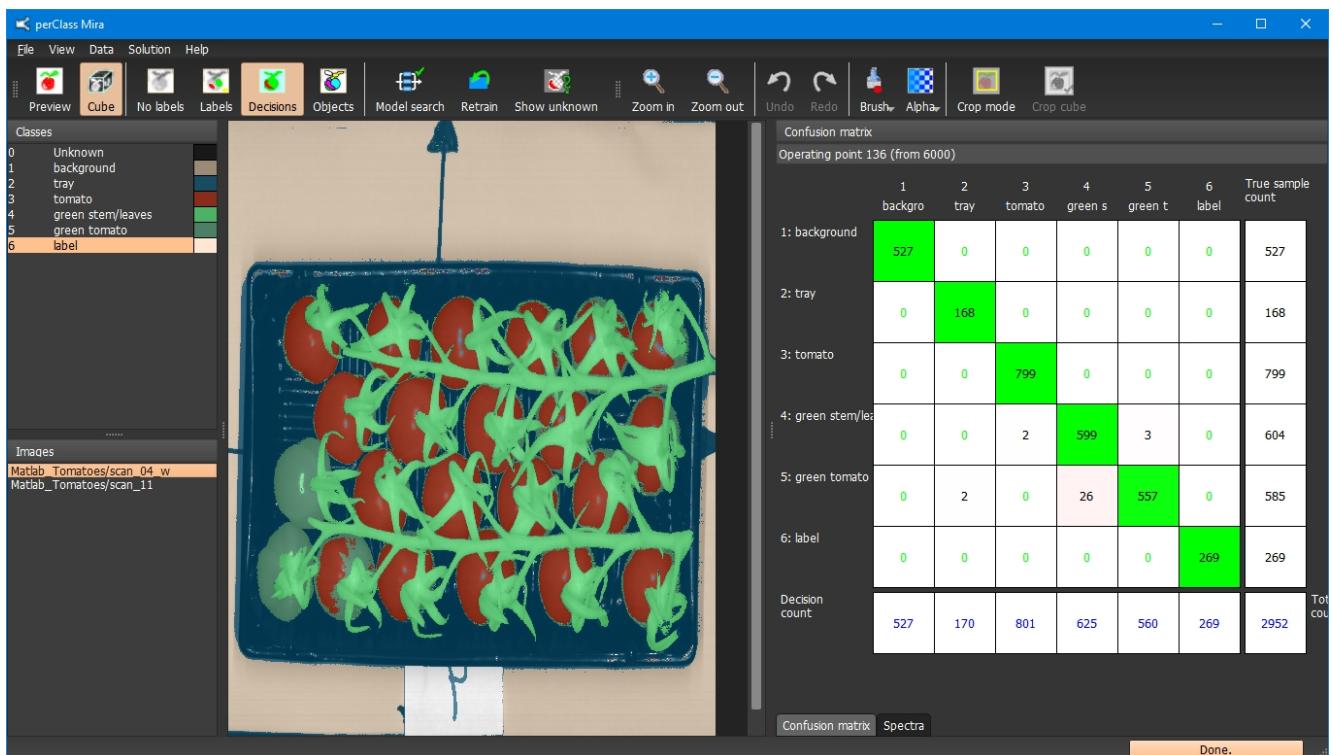
TIP: You may quickly switch to confusion matrix by pressing 'c' key and to spectral plot by pressing 's' key



The confusion matrix shows true class labels in rows and decisions in columns. Therefore, the diagonal represent correctly classifier examples and off-diagonal elements the errors.

By default, confusion matrix is normalized by sum of each row (total number of true class examples) to provide class errors.

To view the absolute, not normalized, values, toggle the *Show normalized matrix* command in the right-click context menu or press Shift+N key.



The right-most column shows total number of class samples, or (on normalized confusion matrix) the per class error rate.

The last row shows number of decisions per class, or (on the normalized confusion matrix), the precisions. Precision is a total number of correctly classified samples divided by the total decisions of this class. We wish to

have precision of 1.00 which means that all each decision for this class is correct (pure).

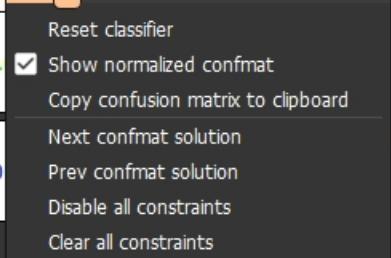
Observing the confusion matrix in our example, we can see that the *green tomato* and *green stem/leaves* classes are not well separable. This probably caused by overlap of these classes in our data.

Although we cannot fully separate them, we may fine-tune the respective error trade-off. **perClass Mira provides fully interactive confusion matrix.**

We may right-click on any field of interest, for example the true *green tomato* vs *green stem/leaves*. The context menu shows a slider which may be used to tune the respective trade-off.

By moving the slider, we may see that the error is lowered and iage decisions change to reflect this new setting. Note that our statistical model does not change, we only tune the importance of specific class in our application. Therefore, the error removed from one confusion matrix field will move to another one. In our case, there will be higher error between true *green stem/leaves* and *green tomato* decision.

		1 backgro	2 tray	3 tomato	4 green s	5 green t	6 label	Class error
		1: background	2: tray	3: tomato	4: green stem/leaves	5: green tomato	6: label	Precision
1: background	1.000	0.000	0.000	0.000	0.000	0.000	0.000	
2: tray	0.000	1.000	0.000	0.000	0.000	0.000	0.000	
3: tomato	0.000	0.000	1.000	0.000	0.000	0.000	0.000	
4: green stem/leaves	0.000	0.000	0.003	0.843	0.154	0.000	0.157	
5: green tomato	0.000	0.003	0.000	0.017	0.979	0.000	0.021	
6: label	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
Precision	1.00	0.99	1.00	0.00	0.00	0.00	0.00	



When exporting the trained model for execution using perClass Runtime, the current operating point (performance setting) is used. The deployed classifier should, therefore, reflect the situation in the confusion matrix.

Performance constraints

To express application-specific requirements perClass Mira allows definition of constraints in confusion matrix.

By double-clicking on any field of the matrix, we create respective constraint for the current value. For error (off-diagonal) elements the means that only solutions with error less or equal than current value are allowed. For accuracies (diagonal) elements it is values higher or equal.

Confusion matrix							
Operating point 411 (5178 valid from 6000)							
	1 backgro	2 tray	3 tomato	4 green s	5 green t	6 label	Class error
1: background	1.000	0.000	0.000	0.000	0.000	0.000	0.000
2: tray	0.000	1.000	0.000	0.000	0.000	0.000	0.000
3: tomato	0.000	0.000	1.000	0.000	0.000	0.000	0.000
4: green stem/leav	0.000	0.000	0.002	0.997	0.002 ≤ 0.029	0.000	0.003
5: green tomato	0.000	0.000	0.000	0.041	0.959	0.000	0.041
6: label	0.000	0.000	0.000	0.000	0.000	1.000 ≥ 1.000	0.000
Precision	1.00	1.00	1.00	0.96	1.00	1.00	0.007
							Mean error

Each field with a constrain shows a small square in its left upper corner. Note, that due to equal sign in constrain definition our current solution does not change by creating a constrain. We only limit a subset of admissible solutions (see the text in the upper part of the confusion matrix widget showing that now 5178 solutions from the total 6000 are valid.)

We may install multiple constraints by double clicking. To remove a constrain, double click again.

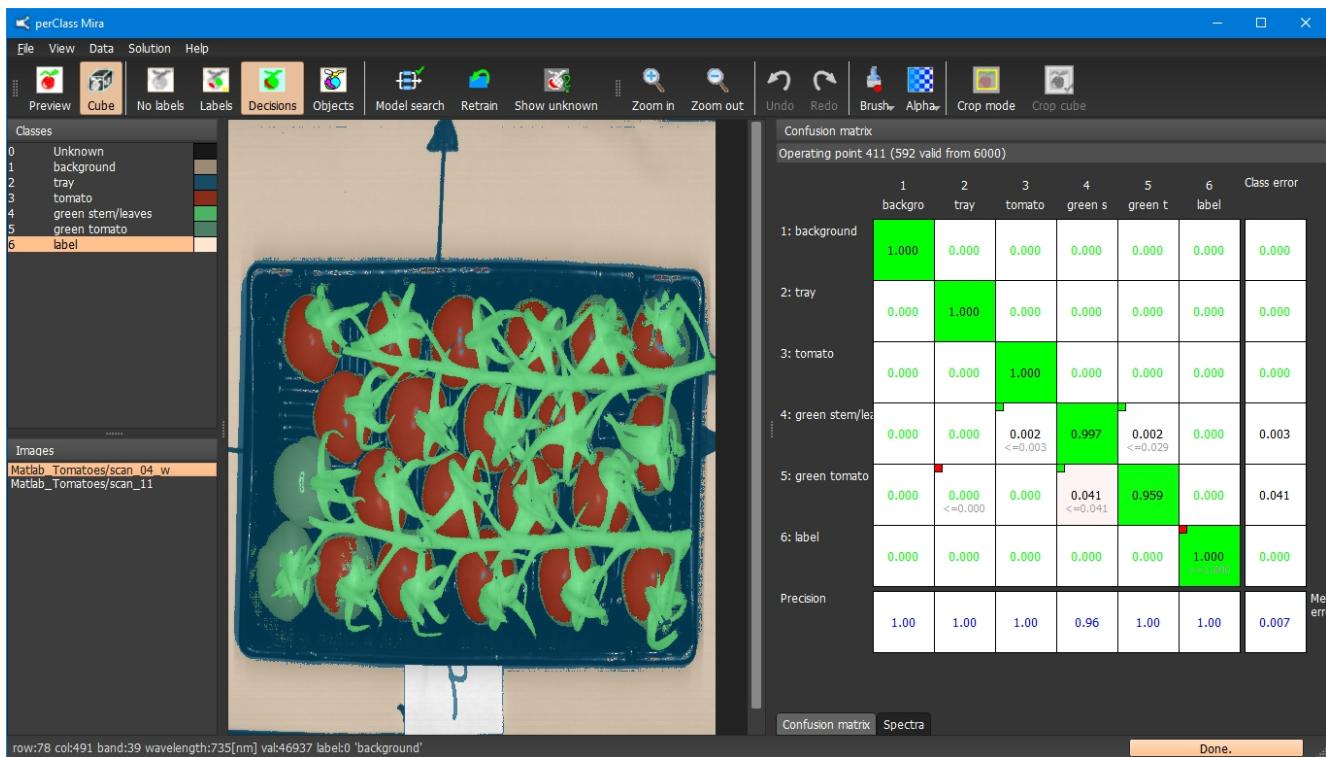
To change any constrain value, we may use Ctrl+mouse wheel on the specific field. This may lead to a new, better, operating point.

Note, that we may not minimize all errors simultaneously. Lowering the error between class 4 (*green stem/leaves*) and decision 5 (*green tomato*) the opposite error (class 5 vs decision 4) will increase. If we also install the constrain on class 5 vs decision 4 we may reach the situation where this constrain cannot be lowered further (by doing so, there would be no more solutions left). To explore fully these situations, constraints may be *disabled* by clicking on the small square in the left-upper corner. Only the constraints with green square are used in performance optimization, not the red square constraints.

Best practice procedure:

- Install constraints of interest by double-clicking
- Tune the constraints' values using Ctrl+mouse wheel
- If the errors cannot be lowered further as desired:
 - either increase value of other constraints (Ctrl+mouse wheel)
 - or disable other constraints and lower the error value of the more important ones

Example of multiple constraints:



Note, that re-training a model does not change the optimization options (operating points) only updates the model and selects one of existing points.

When selecting a new model via *Model search*, also new set of operating points (solutions) are created.

In both cases, there may be no solution that fulfills all enabled constraints. In this situation, all constraints are disabled. You may re-enable some of them and/or update value of others to find a desired solution.

Marking images for testing

Images can be marked for testing using context menu in the image list and *Set image for test* command or by pressing (Ctrl+T)

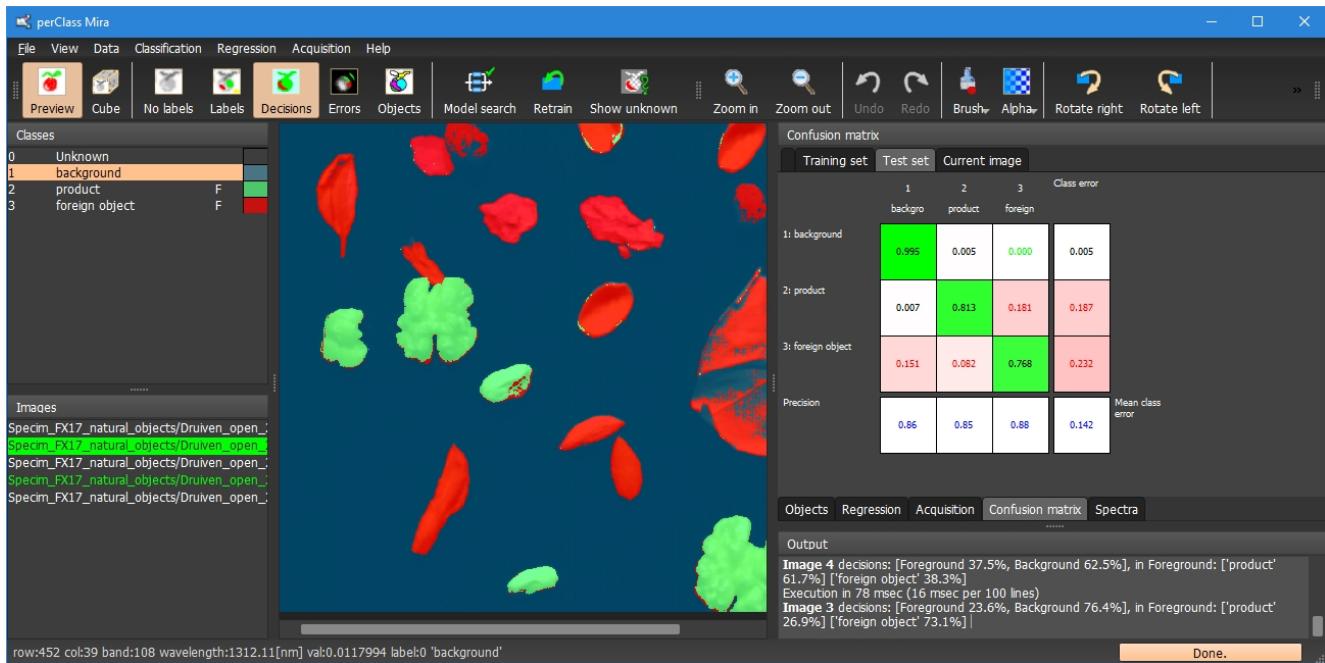
Images marked for testing are not included in the training set for building pixel classifiers.

The model needs to be retrained in order to put new train/test split into effect. This holds both for classification and for regression modeling.

Both, Model search and Retrain commands always use the current setting of training and test images.

Separate test set confusion matrix is also available with a detailed break-down of performance on labels in test images.

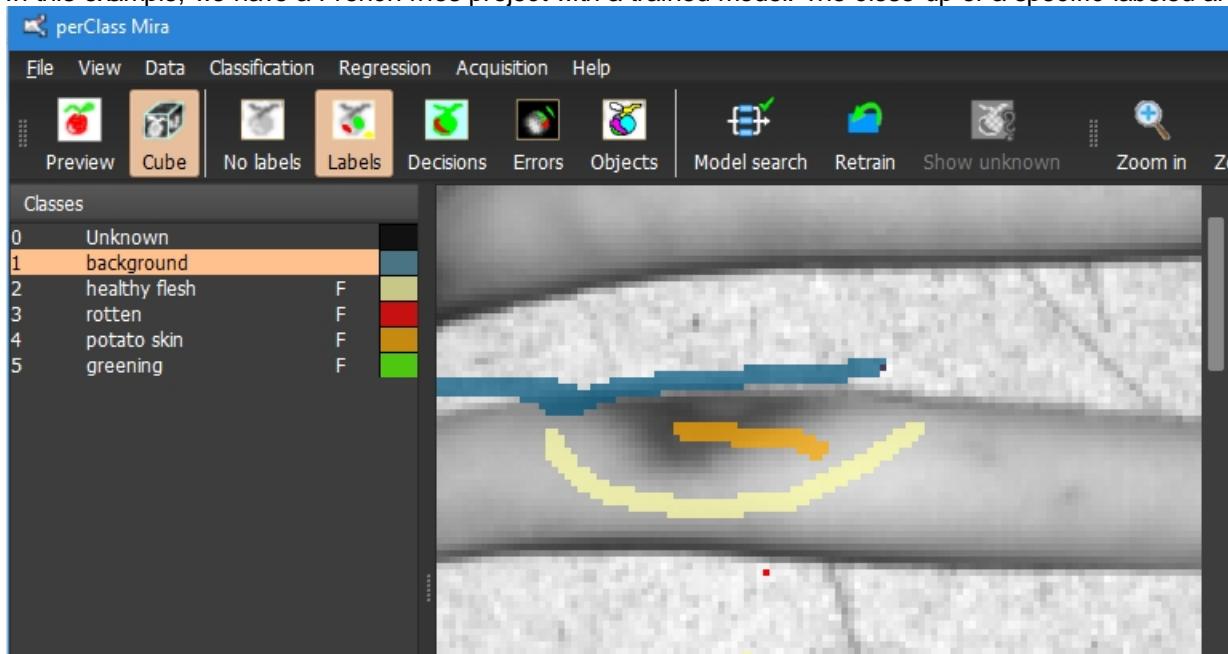
This enables us to understand how the model performs on unseen data.



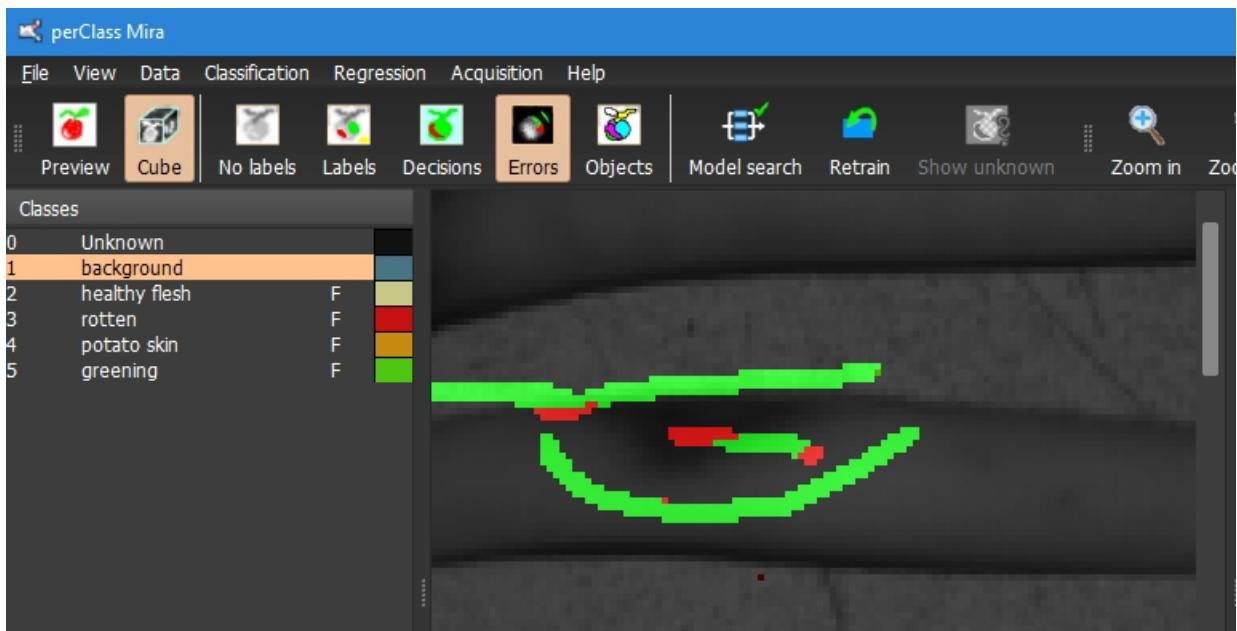
Error visualization

perClass Mira 2.0 brings error visualization. It enables the user to understand where the classifier fails and facilitates the model improvements.

In this example, we have a French fries project with a trained model. The close-up of a specific labeled area:

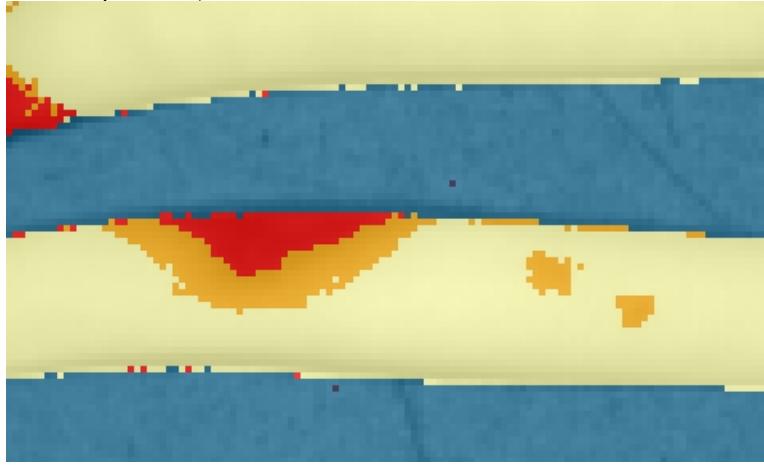


When we select the *Errors* mode in the toolbar (or press E key), the visualization will highlight correct model decisions in the labeled areas in green and incorrect decisions in red:



We can see, that there are three regions where the current model misclassifies the labeled information.

For completeness, we show the decisions of the model:



It is clear that the central area is classified as rotten while our middle label stroke assigns it to the *potato skin* class.

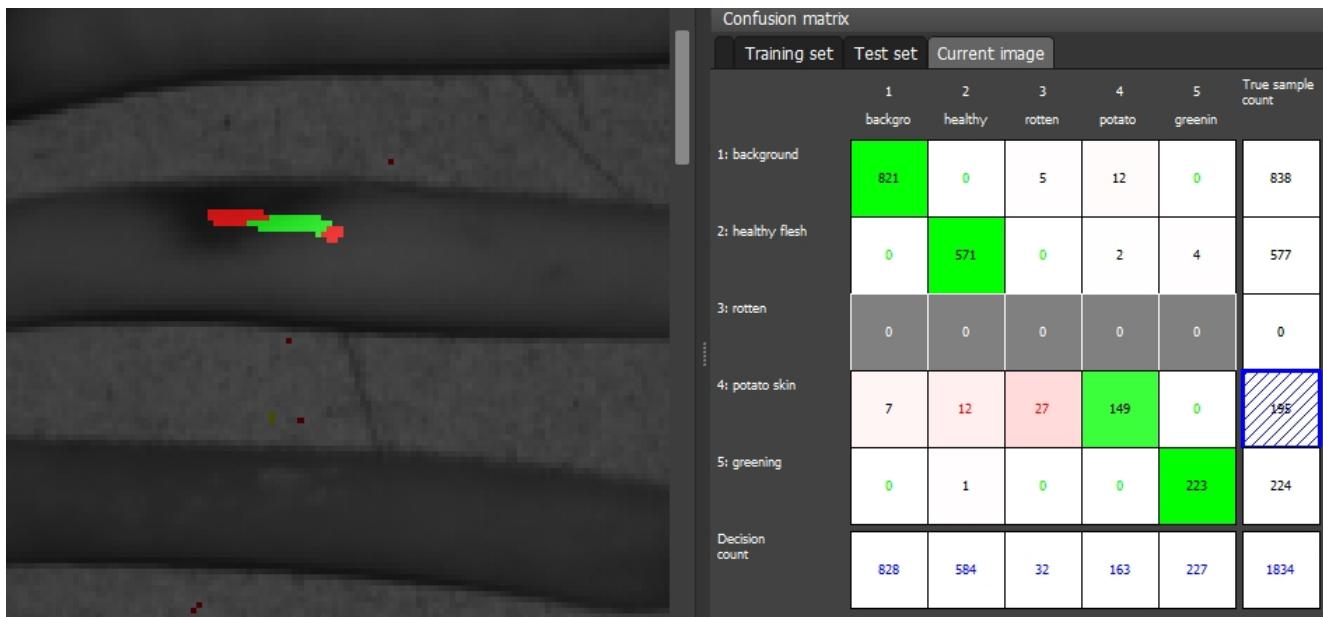
The background label stroke also reaches into the potato piece resulting into incorrect classification.

Interactive error visualization in image confusion matrix

By default, the error mode shows all labeled pixels in an image. In order to gain deeper insight into model performance, it is possible to limit the visualization to specific type of errors in a confusion matrix.

Hovering the mouse over the *image confusion matrix*, we visualize only the corresponding subset of pixels.

Using the French fries [example from previous page](#), we visualize only examples labeled a *potato skin*:



When we move the mouse to the entry *potato skin - rotten* (the value of 27), we can quickly identify the part of the stroke where the current models mislabels skin as rot defect.



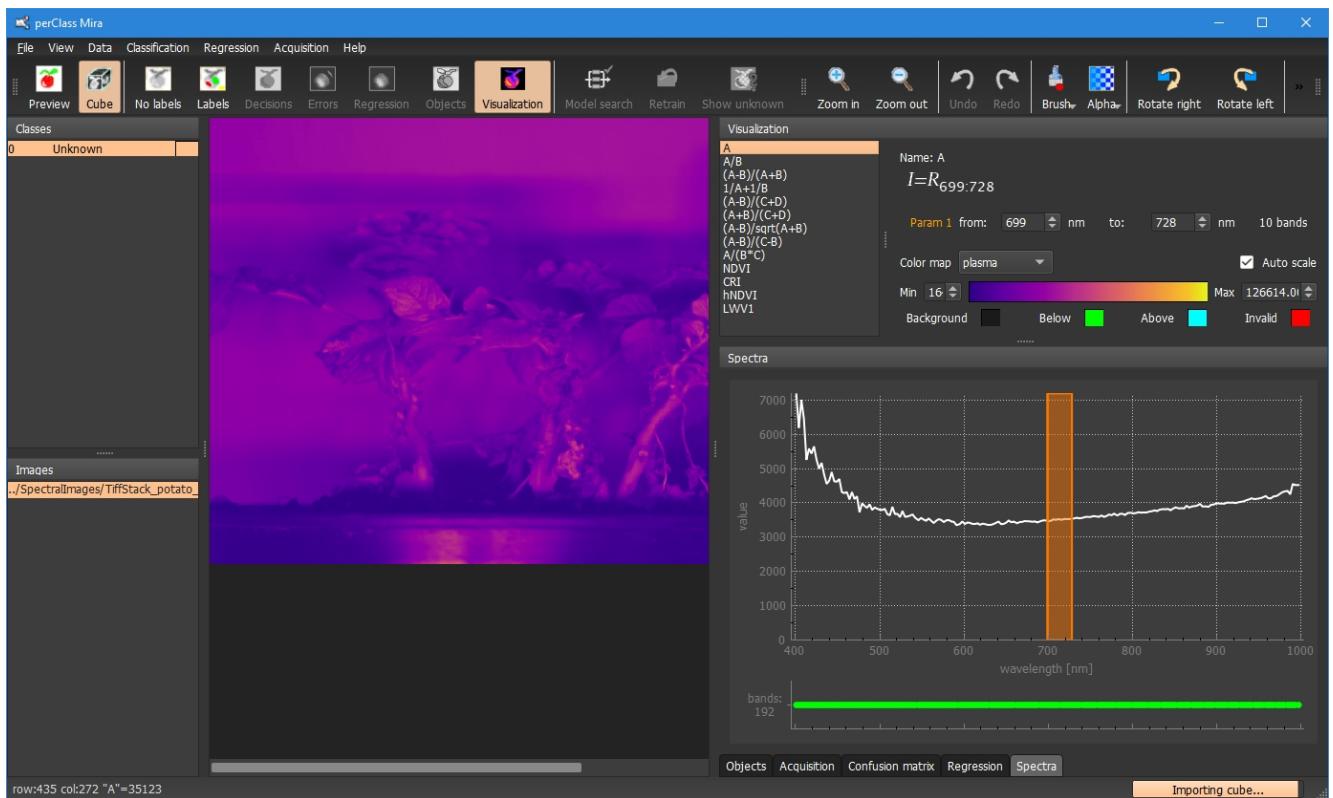
Important: Please note, that the green/red pixel visualization corresponds to the current model. There may be several reasons why we observe incorrect labeling in the left part of the stroke:

1. Our labeling may not be precise and the error suggests which part is incorrectly labeled. In this most common case, removing the *potato skin* on the left and the right end of the stroke would be advisable to improve the training labels. Leaving incorrect labels confuses machine learning models.
2. If we observe model errors when inspecting a test image (unused in model building), it may be that the areas highlighted as errors are, in fact, valid examples not represented in the training set. In such a case, it is advisable to include similar material examples to our training set.

Visualization

perClass Mira provides an interactive visualization tool that allows us to quickly show bands or band ranges but also results of more complex common equations such as NDVI.

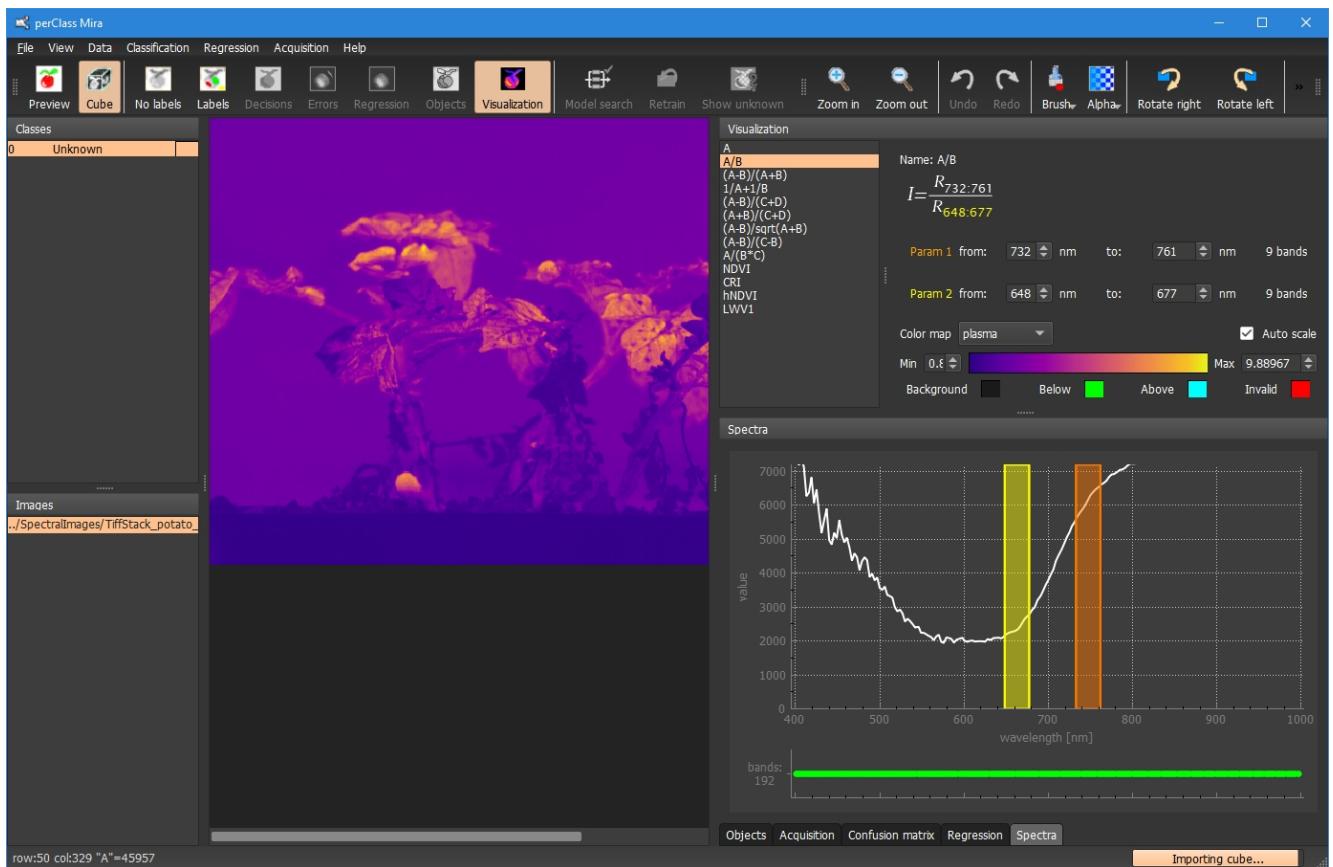
The functionality is configurable in a separate *Visualization* pannel. As the visualization equations are defined in terms on spectral bands or band ranges, it is useful to place both *Visualization* and *Spectra* pannels next to each other, as shown on the screenshot:



The *Visualization* pannel lists number of common equations at its left side. When an equation is selected, details are shown on the right.

In our example, only a single band or band range is shown. Note, that the band range is represented by a rectangle in the spectral plot.

Equations may have multiple parameters. For example, if we select A/B equation, we will compute ratio of values of the two bands or band ranges per pixel. Parameters may be changed by defining minimum and maximum wavelengths in nanometers or simply by dragging the corresponding rectangles in the spectral plot.

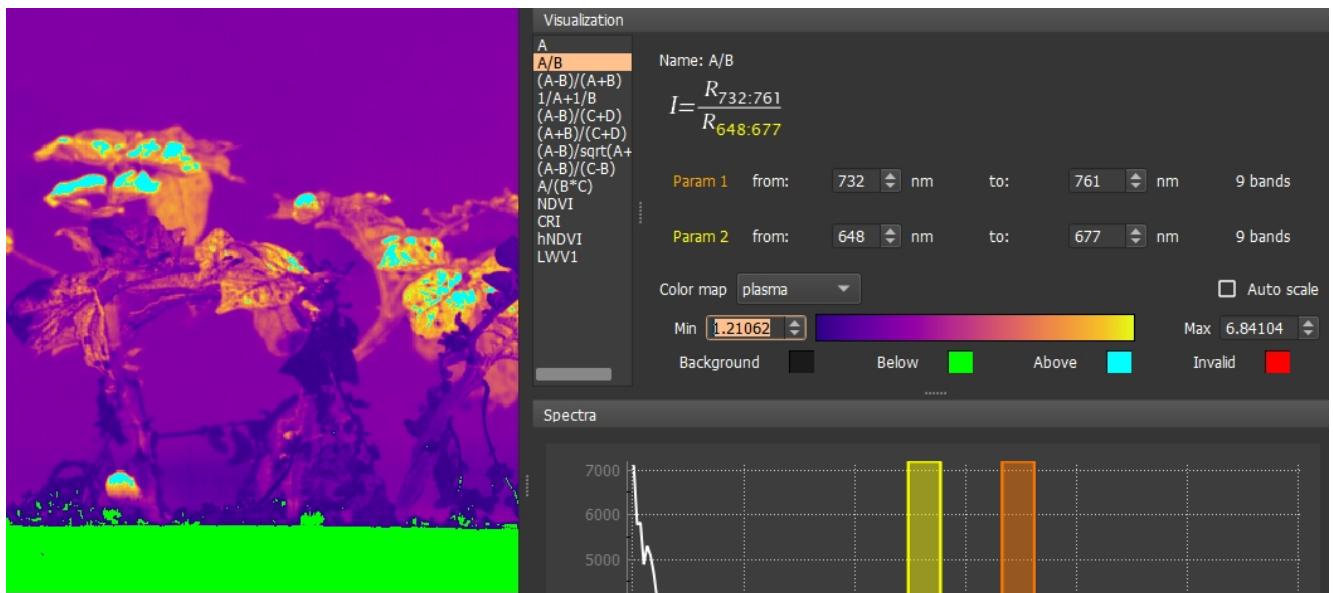


Scaling

Visualization output is, by default, auto-scaled based on min and max values of the entire image.

We may switch the auto-scale using the check box or by directly adjusting min and max value.

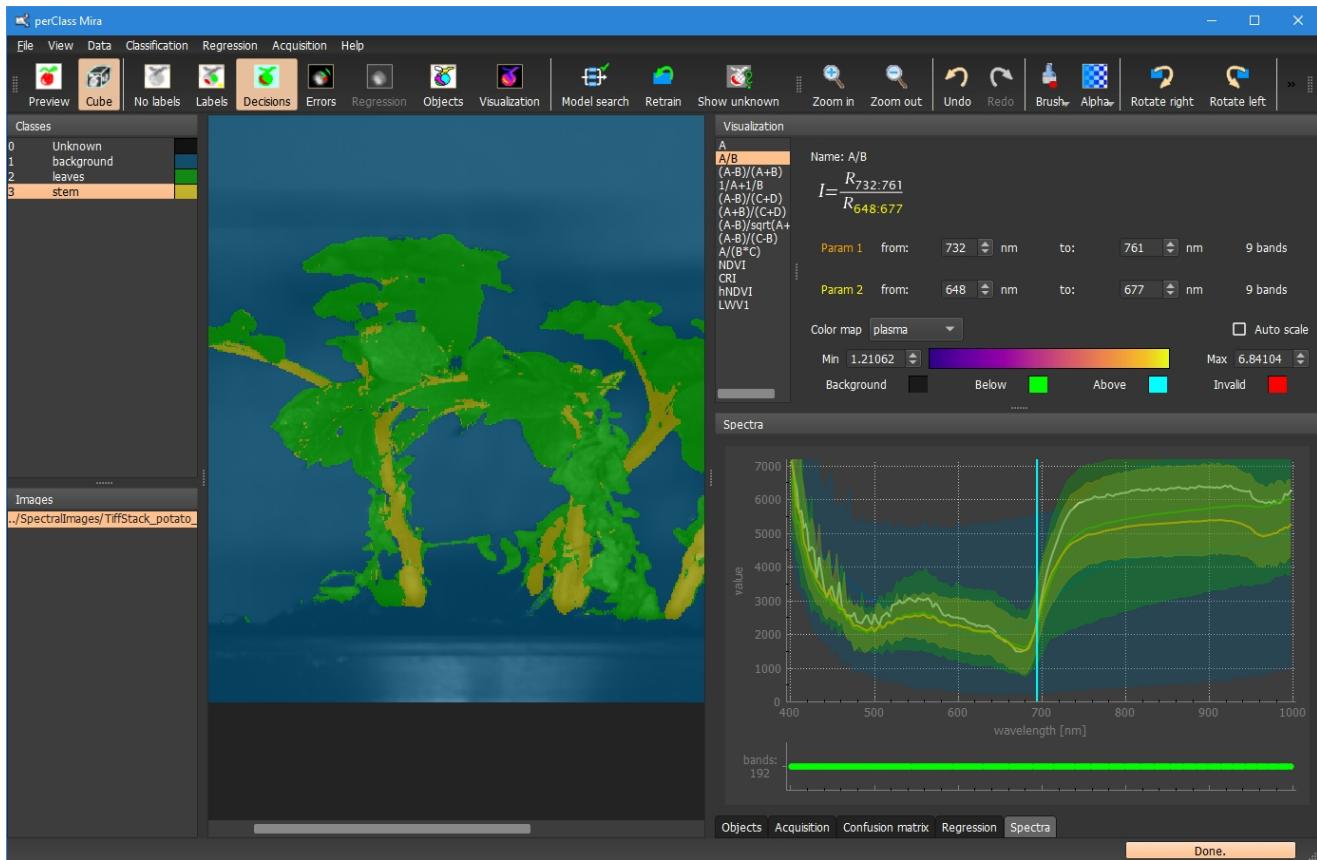
The pixels out of range are rendered using below/above colors, respectively:



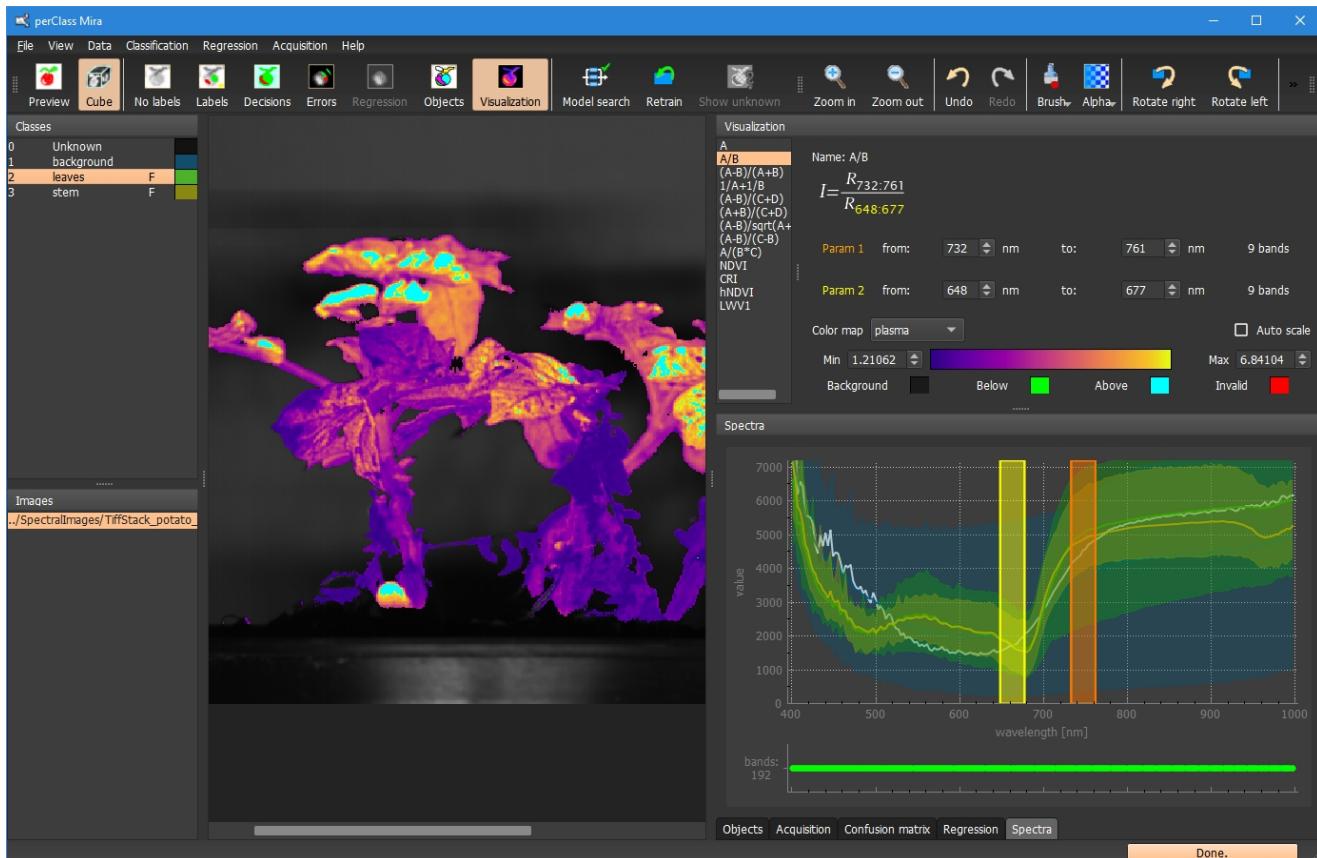
Constraining visualization to foreground

Typically, we are not interested in spectral index value for the entire image but only for objects of interest. For example, in plant phenotyping, we want to characterize plant health using NDVI index.

In perClass Mira, we can easily construct a classification model for leaves and stem of the plant.



When we flag only the classes of interest as foreground, the visualization will be applied only to the respective pixels.



Note that also data auto-scaling will also use only foreground pixels. This allows us to avoid reflections or

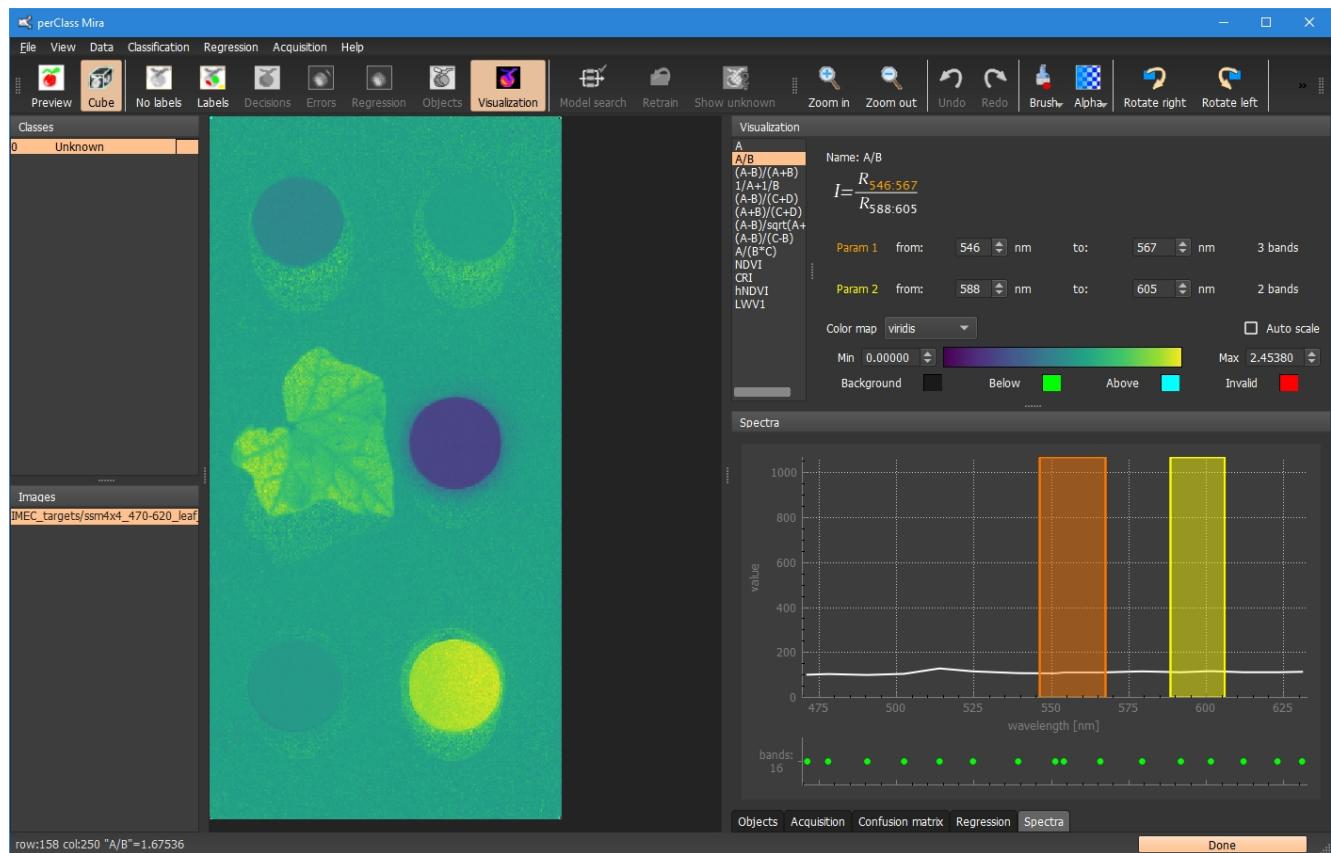
uninformative areas of the background.

Mapping wavelengths to cube bands

Visualization parameters are defined by wavelengths or wavelength ranges in nanometers.

Next to each parameter, we may see indication how many bands are covered by its definition.

In this example, we can see a spectral cube from a sensor with 16 spectral bands.

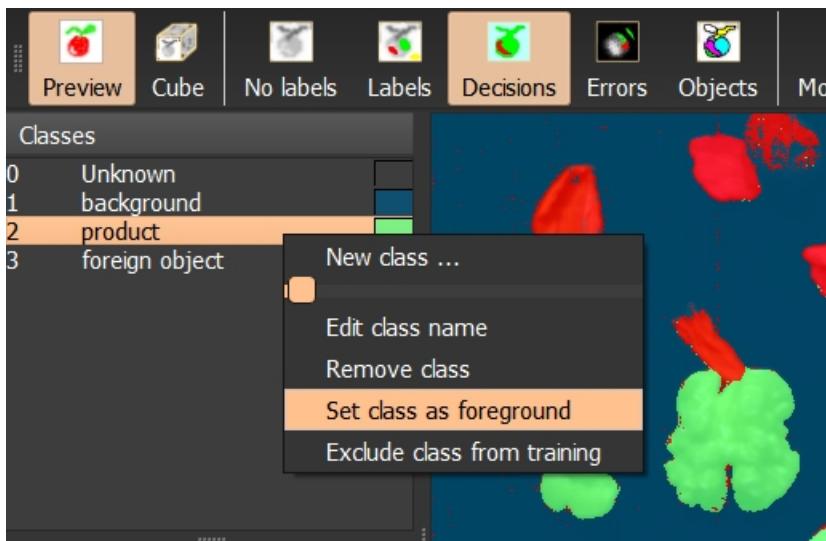


If there are no bands available to fit the parameter definition, the visualization output is rendered as "invalid", by default in red.

Object segmentation

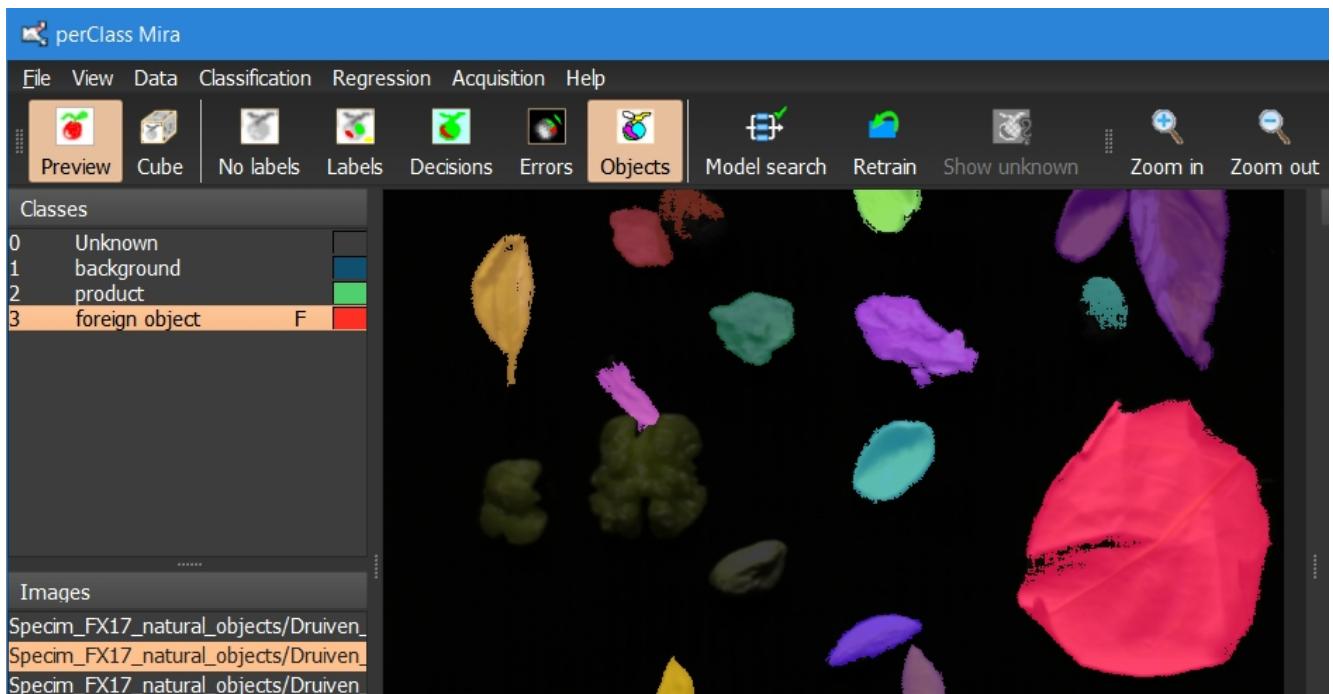
Apart of pixel classification perClass Mira provides object segmentation. In order to segment objects (connected components), one needs to set one or more classes as "foreground":

1. In the class list, use right click to open the context menu on the desired class and select *Set class as foreground* (or press F key)



2. On the toolbar, press the *Objects* button (or press O key)

Selected class/classes will be segmented out and each connected component will be visualized with a randomly selected color.



You may notice that small objects got removed from the segmentation result. The minimum size of accepted objects is adjustable in the Objects pannel or via *Classification / Set minimum objects size* menu.

Object labels and object decisions

By default, object segmentation shows object labels. This means, that each connected component is assigned into a unique category.

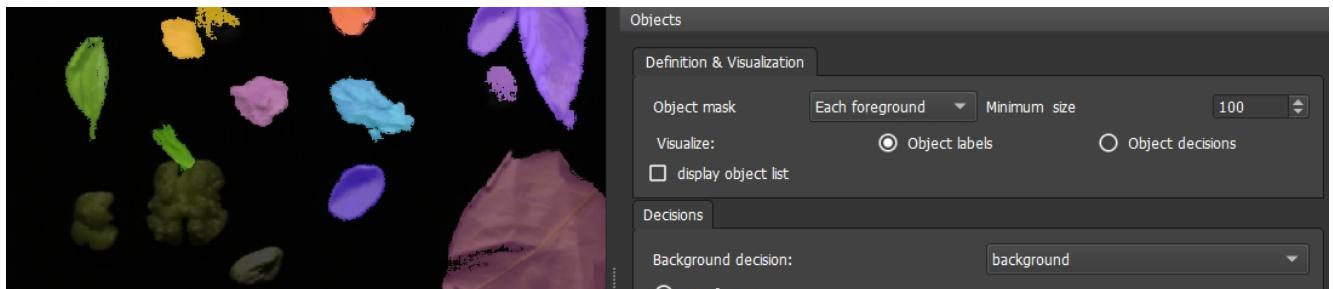
When hovering the mouse over the objects, the object label is displayed on the status bar



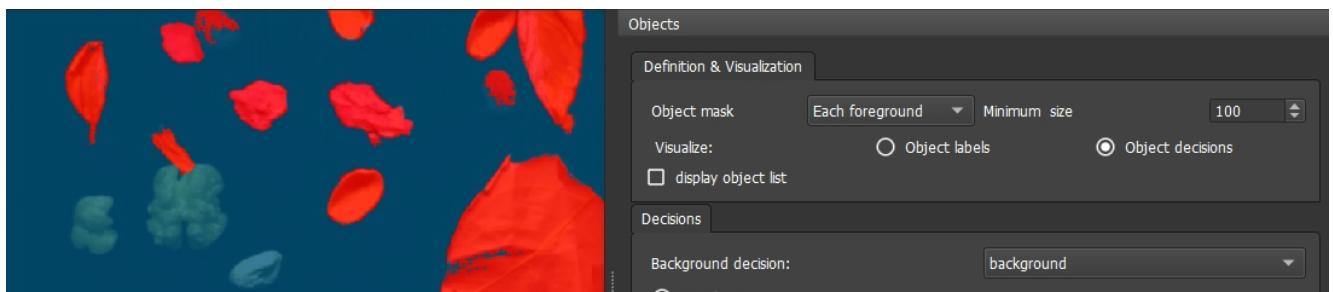
The object pannel provides more options on how to deal with detected objects.

Instead of object labels, we may visualize **object decisions**. It means, that each object is assigned into one of the user-defined classes.

Object labels output:



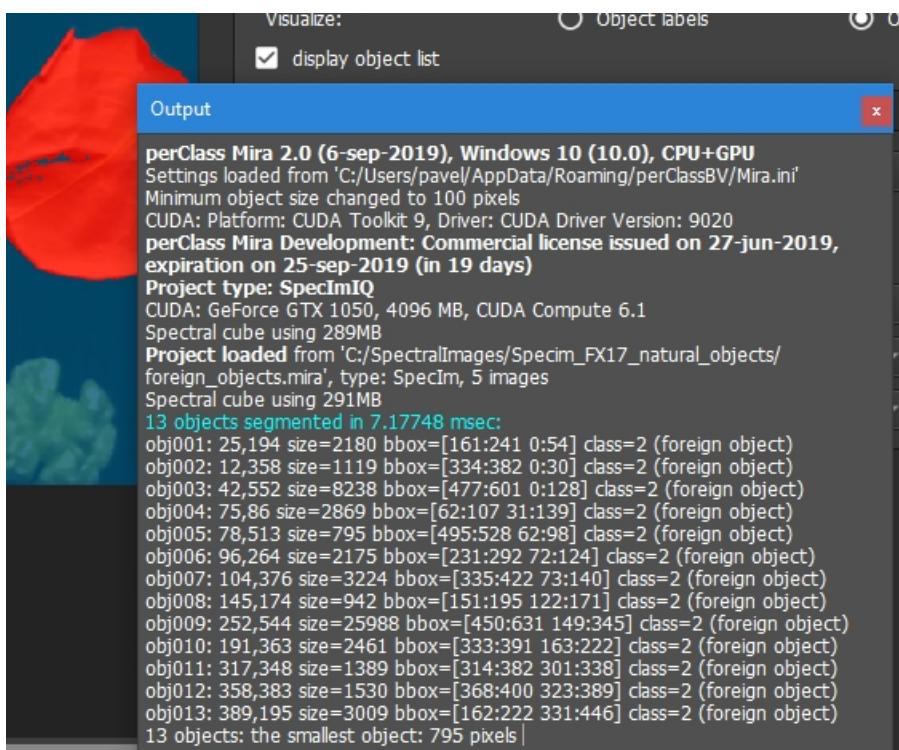
Object decisions output (when only the *foreign object* class is considered as foreground):



Note, that multiple classes may be considered as foreground. In such a case, perClass Mira can handle touching objects of different classes.

Displaying object list

Object list can be displayed in the output window by enabling the *Display object list* check box.



The object list summarizes:

- object label
- center of gravity (row, column)
- size in pixels

- bounding box (columns, rows)
- class

This information can be delivered by perClass Mira Runtime processing a live stream of spectral data.

Object modes

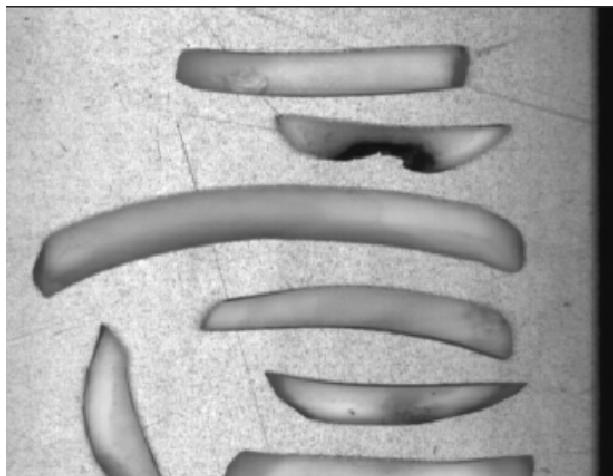
perClass Mira offers two object-definition modes, depending on the mask construction process:

1. **Object mask: Each foreground.** Each object is composed of a single class (or material). Example: Object is either a nut or a shell or a stone. This approach results in [object detection useful in sorting applications](#).
2. **Object mask: All foregrounds.** An object is composed of multiple materials (classes). For example, potato chips sorting needs to remove potato pieces (objects) that contain any rotten or green part. In general, this approach is fitting to [object classification use-cases](#)

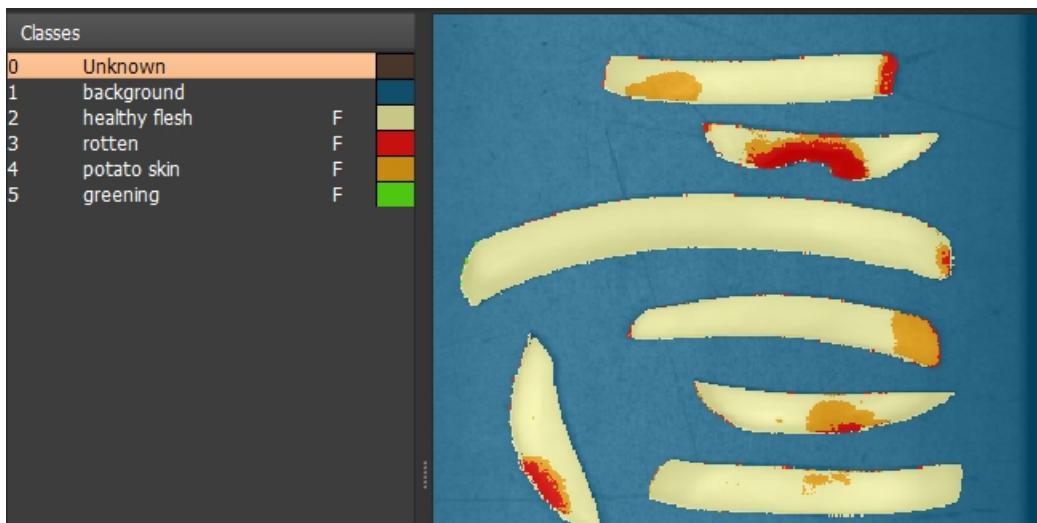
Example of both modes on a French fries data set

Pieces of French fries are to be sorted to remove objects containing defects. Within each piece of a potato, there can be healthy potato flesh or skin (both OK for the consumer) but also rot or greening defects than must be avoided.

Single band image



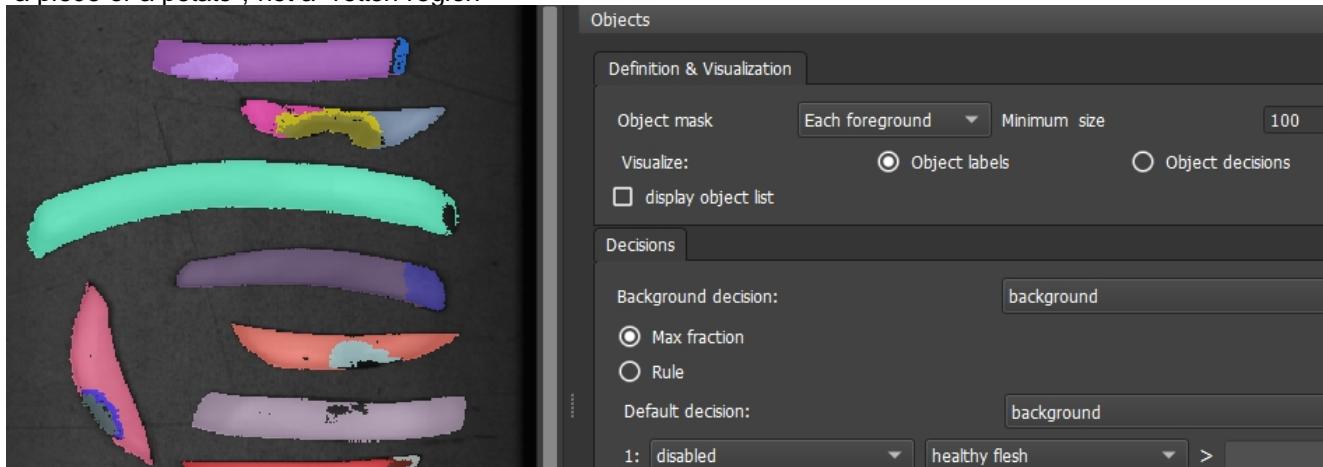
Decisions of a classifier



Object definition with "mask by each foreground" mode

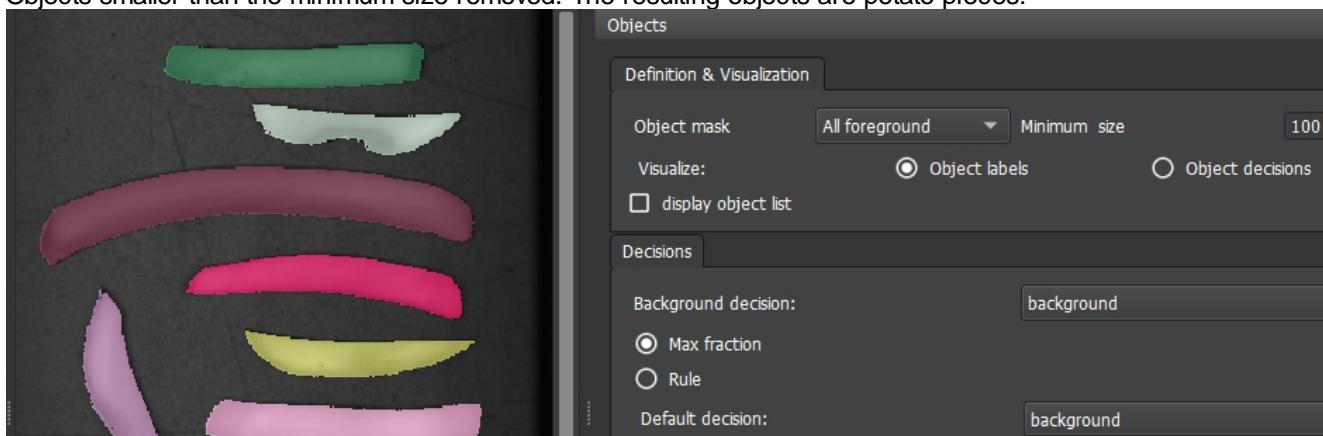
Using the "Each foreground" setting, each foreground class (healthy, skin, rot and green) is handled separately.

Small objects are removed. The resulting object labels will show a colorful patchwork of individual regions. This approach does not help us to build a sorting problem as "an object" in the French fries sorting application is "a piece of a potato", not a "rotten region"



Object definition with "mask by all foreground" mode

Using the "All foreground" approach, a union of all foreground classes is considered as a mask for segmentation. Objects smaller than the minimum size removed. The resulting objects are potato pieces:



Object classification

Object classification use-case based on "All foreground" masking allows us to define specific way how to decide class on individual objects. The object pannel provides two approaches:

1. Maximum fraction (majority voting)
2. Rule-based classification

Maximum fraction object classification

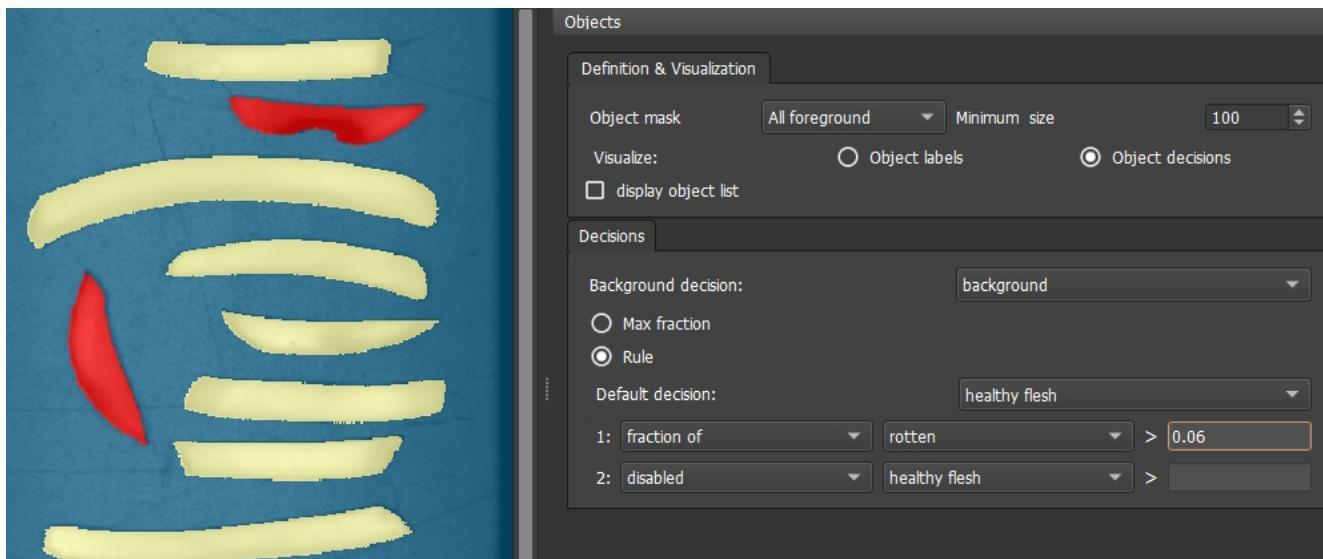
Object is classified to a foreground class with the highest number of pixels within an object. Background is assigned to a class defined by the "*Background class*" combo box.

This option is useful in situations where we distinguish several spectrally different classes but the entire object is either one or the other. For example, in a fruit sorting application, we may train a "*firm fruit*" and *hard fruit* classes. The maximum fraction object classifier assigns the fruit piece to the most frequent class.

Rule-based object classification

In some sorting use-cases, the presence of even small amount or fraction of a specific class mandates rejection of an object. For example, more than 6% or a rotten potato may result in a piece rejection as defective. Object pannel allows us to define rules based on size (in pixels) or fraction (0..1) of an object.

The default decision is also available. It is applicable when no other rules trigger on a given object.



Regression

Regression modeling is used to estimate a numerical value instead of making a decision. It is often adopted in quality estimation.

perClass Mira implements object-centered regression modelling. Individual objects can be annotated with numerical meta-data. A regression model is build that can be applied to a new object.

This enables practical applications such as:

- Estimate brix (sugar) content in a tomato
- Estimate a dry-matter content of a plant leaf
- Estimate moisture content in biscuits

Regression example: Powder mixture

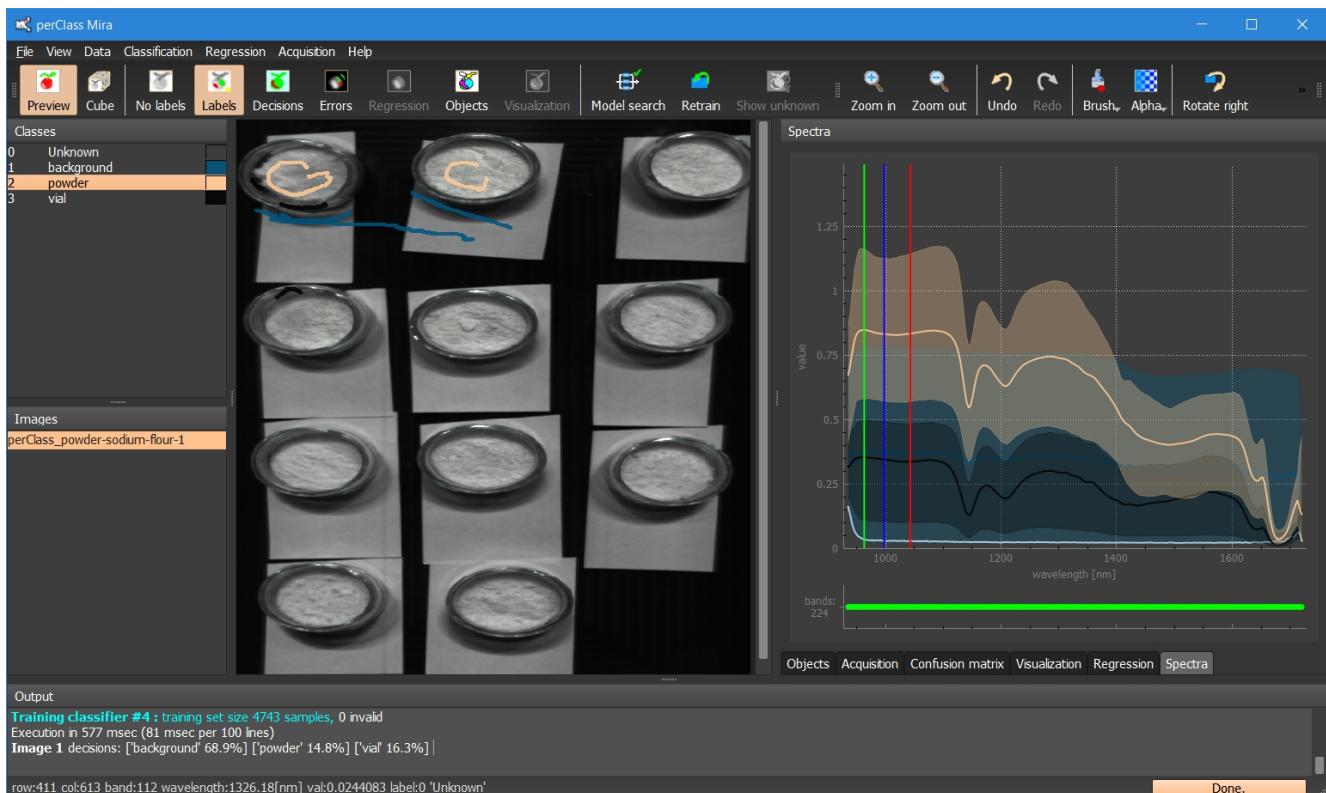
In this example, we will use a mixture of sodium carbonate and flour. A set of vials with different mixing proportions is scanned using Specim FX17 camera in the range between 900 and 1700nm.

In order to estimate mixing proportion we:

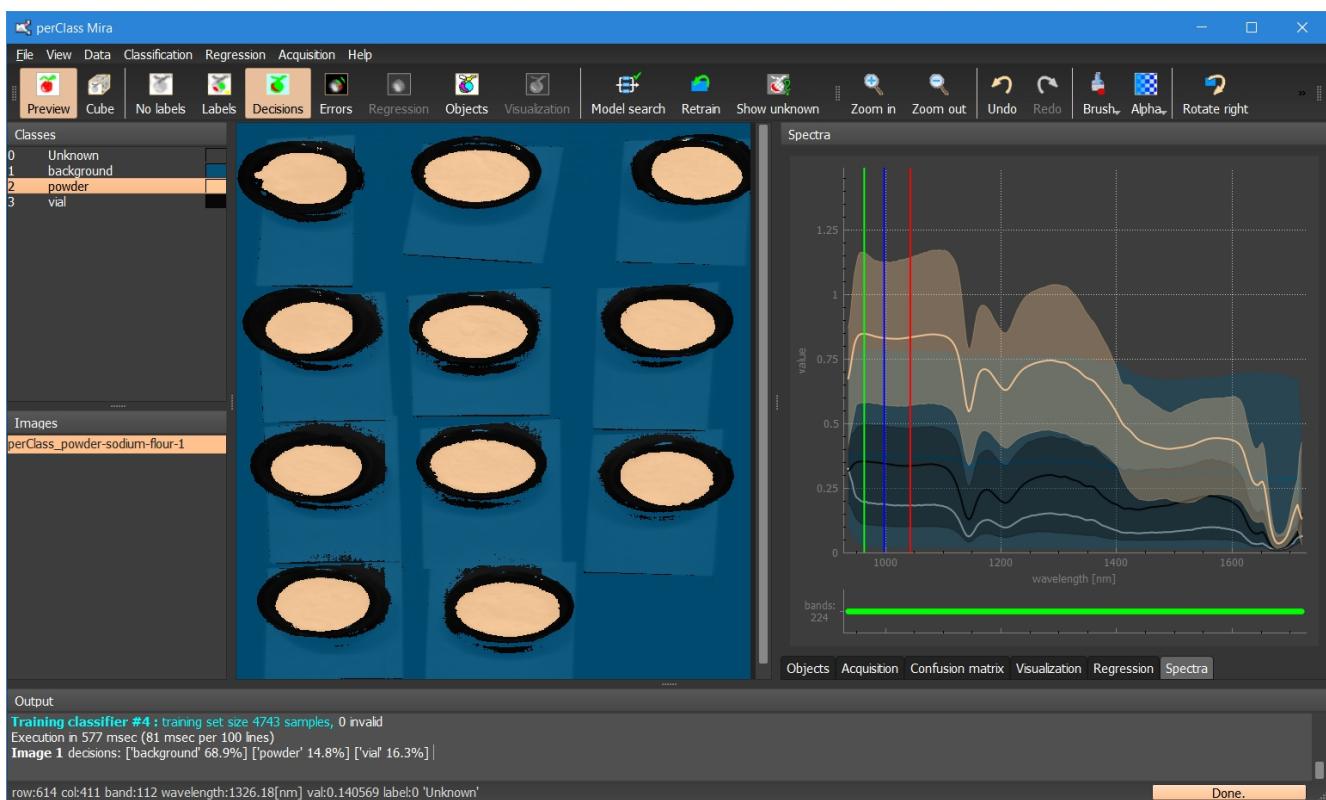
1. Build a classification model that identifies area of interest for regression
2. Define the class of interest as foreground so that objects can be segmented out
3. Annotate number of objects with known ground-truth values for regression
4. Build a regression model
5. Flag some of the images for testing only to judge generalization performance
6. Improve the model
7. Apply the solution to a new hyperspectral scan (objects get identified and for each a value is estimated)

Step 1: Pixel classifier

Step 1: We build a pixel classifier separating the powder from the background and vials

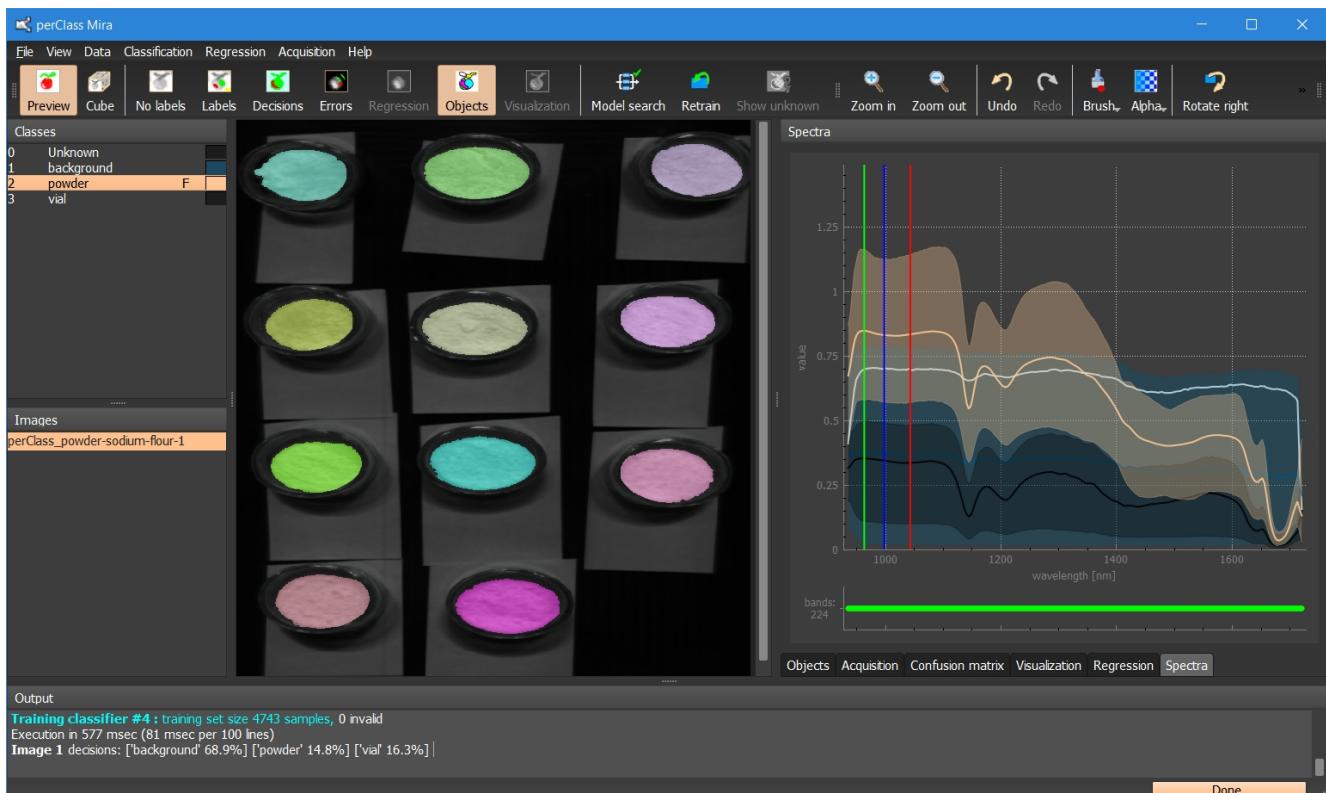


Decisions:



Step 2: Object segmentation

We now need to flag the "powder" class as foreground (by right-clicking on the class name in the class list and selecting *Set class as foreground* or by pressing 'F' key)



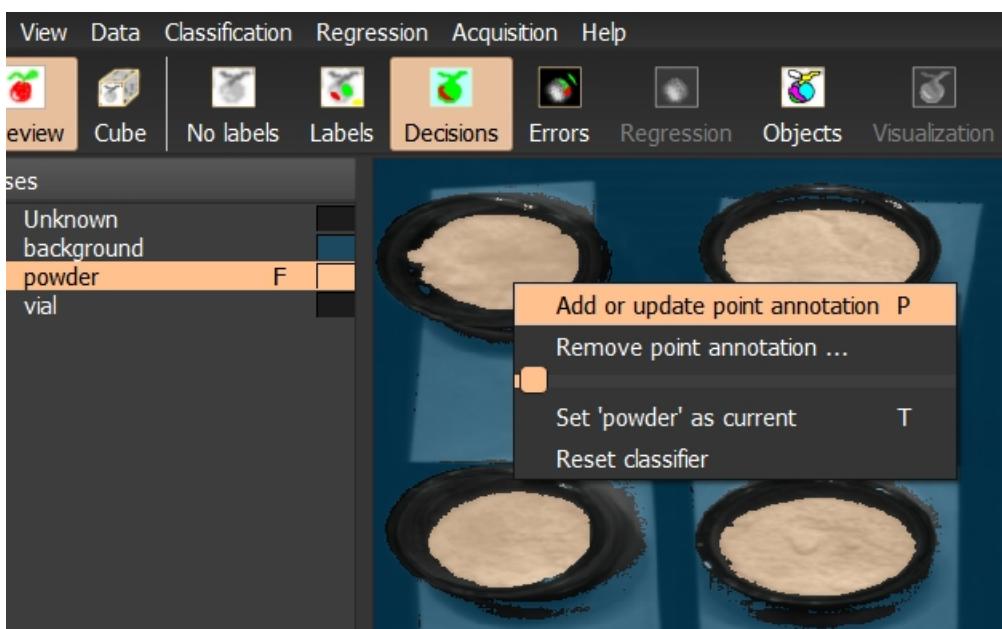
Step 3: Point annotation

In order to build a regression model, we need extra annotation connecting known ground-truth numerical values to objects.

In our case, we know for each vial what mixing proportion is present. We will use percentage of Sodium as the numeric target.

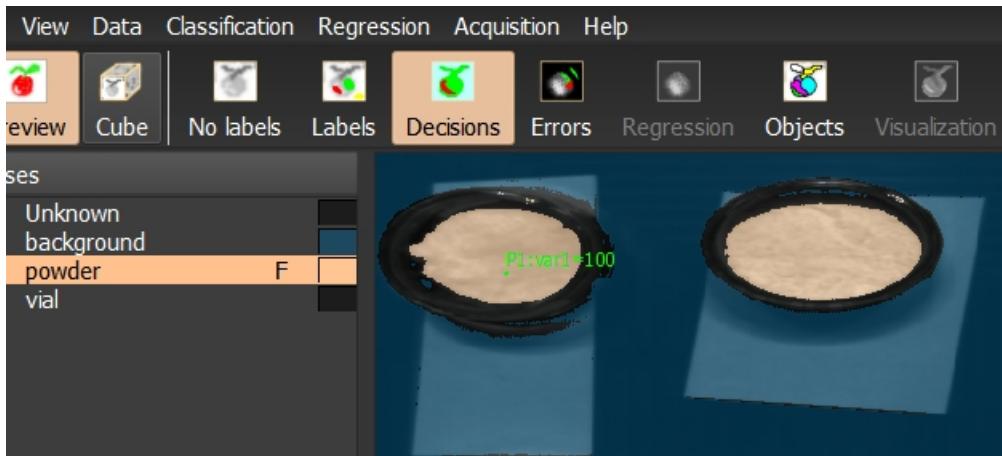
Because the regression modelling is performed on the object level, we need sufficient number of annotated objects. How many exactly are needed will depend on the project. The best guidance is by building the regression model and comparing its performance on the training and test subset of objects. We will discuss construction of a test set further in our example.

We can add a point annotation by right-click and *Add or update point annotation* command:



A dialog will appear where we can enter numeric value.

A new point will appear at the location indicating the value.

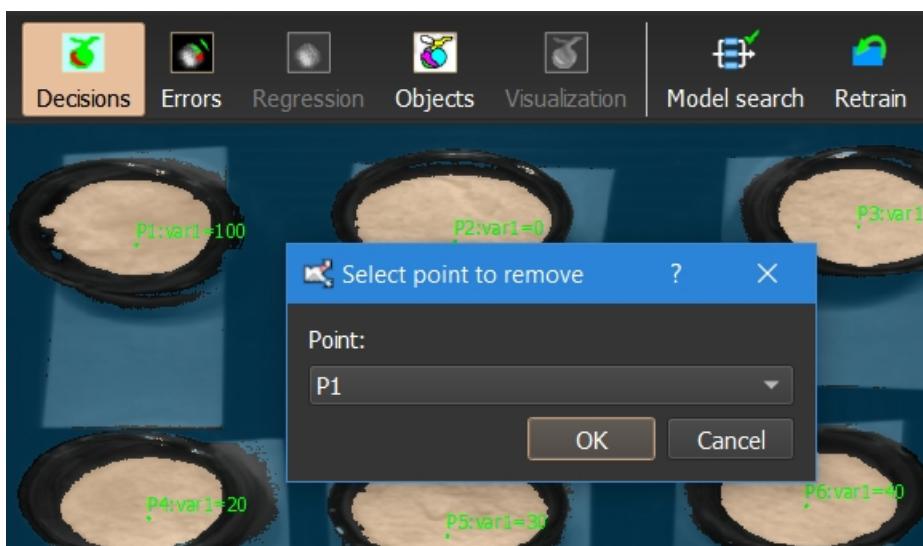


In order to edit the point, move mouse over it and use the same context menu item.

Point annotations need to be placed inside the objects in order to be used for regression. You may drag the point to adjust its position.

Step 3a: Removing point annotations

To delete points use *Remove point annotation* command. A dialog will appear with a list of points in the current image.



Each point has a unique number that is identifying him in the project.

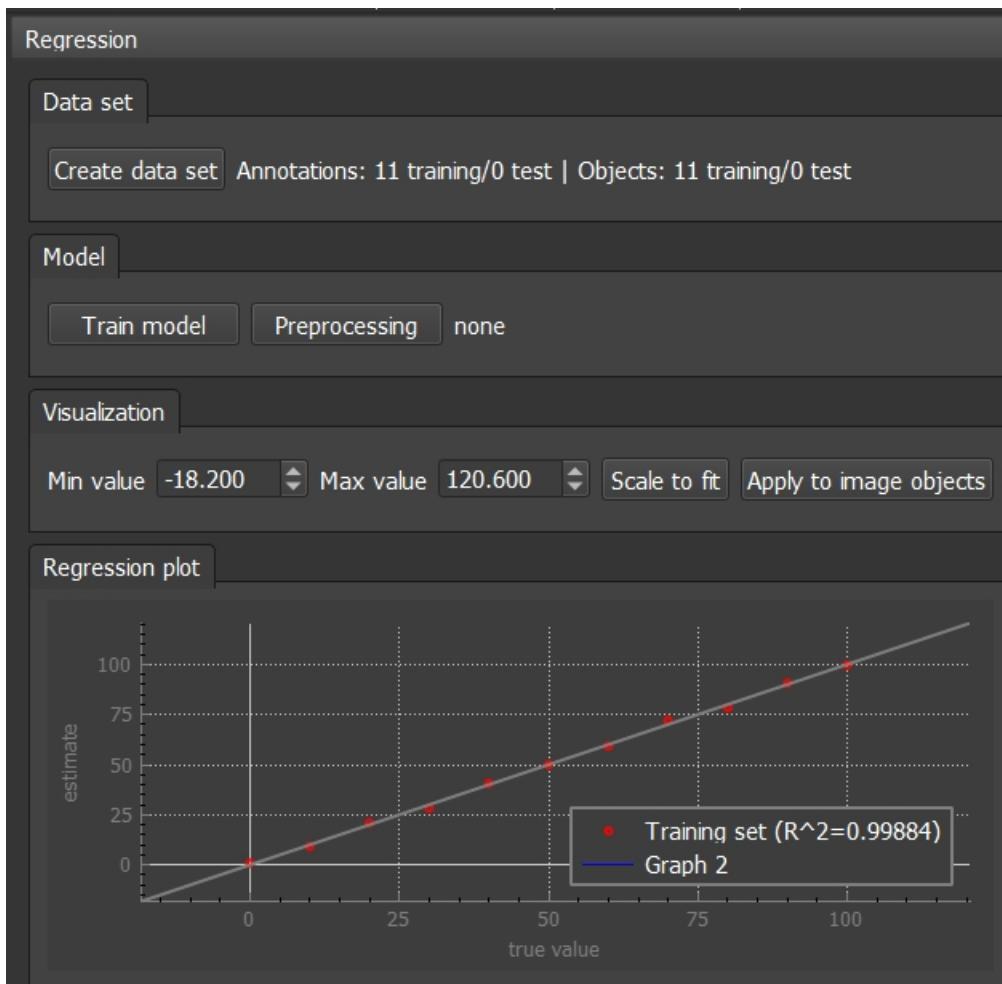
In order to remove all points in an image, use *Remove all point annotations* from the *Regression* menu.

TIP: If you wish to remove points in several images, simply select the images (holding shift for continuous selection or Ctrl for arbitrary selection) and use the same command.

Step 4: Build regression model

In order to build a regression model, we first need to construct a regression data set.

This can be done using the *Create data set* button in the *Regression* panel:



Next to the *Create data set* button, number of annotations and objects is listed. Annotations refer to placed point annotations while objects to the number of point annotations representing specific objects used for training or as independent test.

We can only train a regression model, if there are training objects listed.

By clicking *Train model* we, we build a regression model and regression plot will show true (horizontal axis) versus predicted (vertical axis) values for each object.

We can adjust minimum and maximum value or scale the plot to fit.

To improve regression, we may change data preprocessing. By clicking *Preprocessing* button, a dialog opens with a list of options.

By default, no preprocessing is applied. We may also select spectral smoothing, 1st or 2nd derivative with different window sizes (in bands).

After changing preprocessing, it is necessary to explicitly retrain the regression model.

When to recreate regression data set? Regression data set needs to be recreated if point values are edited, new points are added, points removed or images flagged for training/test. It is not necessary to recreate the data set if points are only moved within their containing object or when a different set of features is selected.

When to retrain regression model? Regression model needs to be retrained when points' values are edited, points were added, removed, images were flagged differently for train/test. Retraining is also needed when different preprocessing is used or feature subset is selected.

Step 5: Adding test examples

To properly validate performance of regression models, we need to apply the regression model to objects unseen

in training.

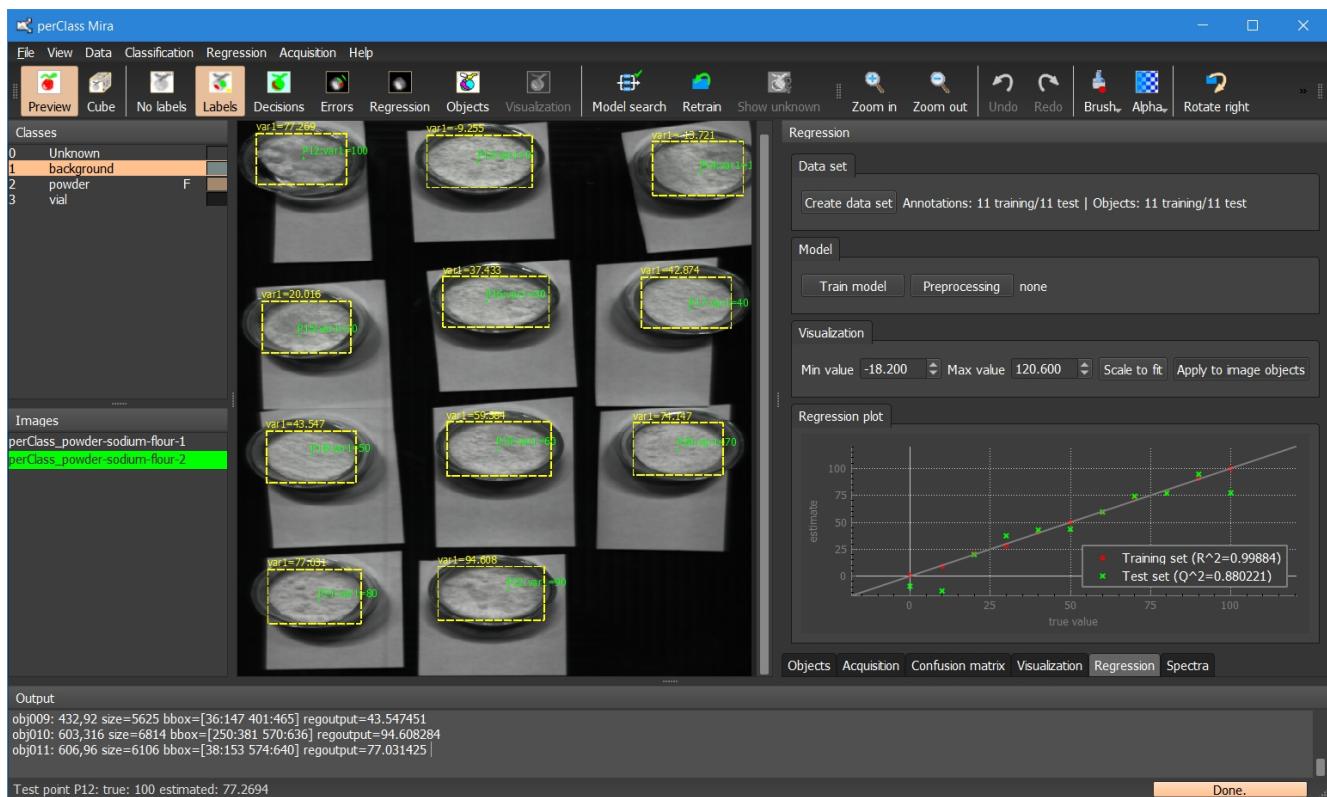
In perClass Mira, we may flag images for testing only (right-click in the image list or press **Ctrl+T** when image is selected). Test images are displayed in green.

We still need to place object annotations on the test images, because we need the ground-truth to compare with estimated values.

In this example, we added a new scan, annotated the vials with proper ground truth. Then, we recreated the data set by clicking *Create data set* in Regression pannel. Note, that 11 objects are listed for testing.

We then retrained our model. The regression plot shows a new set of green points referring to the test objects. The red points always refer only to the training objects.

Finally, we clicked on *Apply to image object* button. This applies our pixel classifier to the image, segmented objects and applies our regression model to each one. The result (estimated fraction of sodium) is displayed next to the yellow bounding box.



Although there is a good match of the regression trend, estimated values for two test points fall far from the ground truth. We may hover the mouse over the points in the regression plot to inspect which point is the closest to the mouse. The point number, ground truth and the estimated value are listed in the status bar at the bottom of the window.

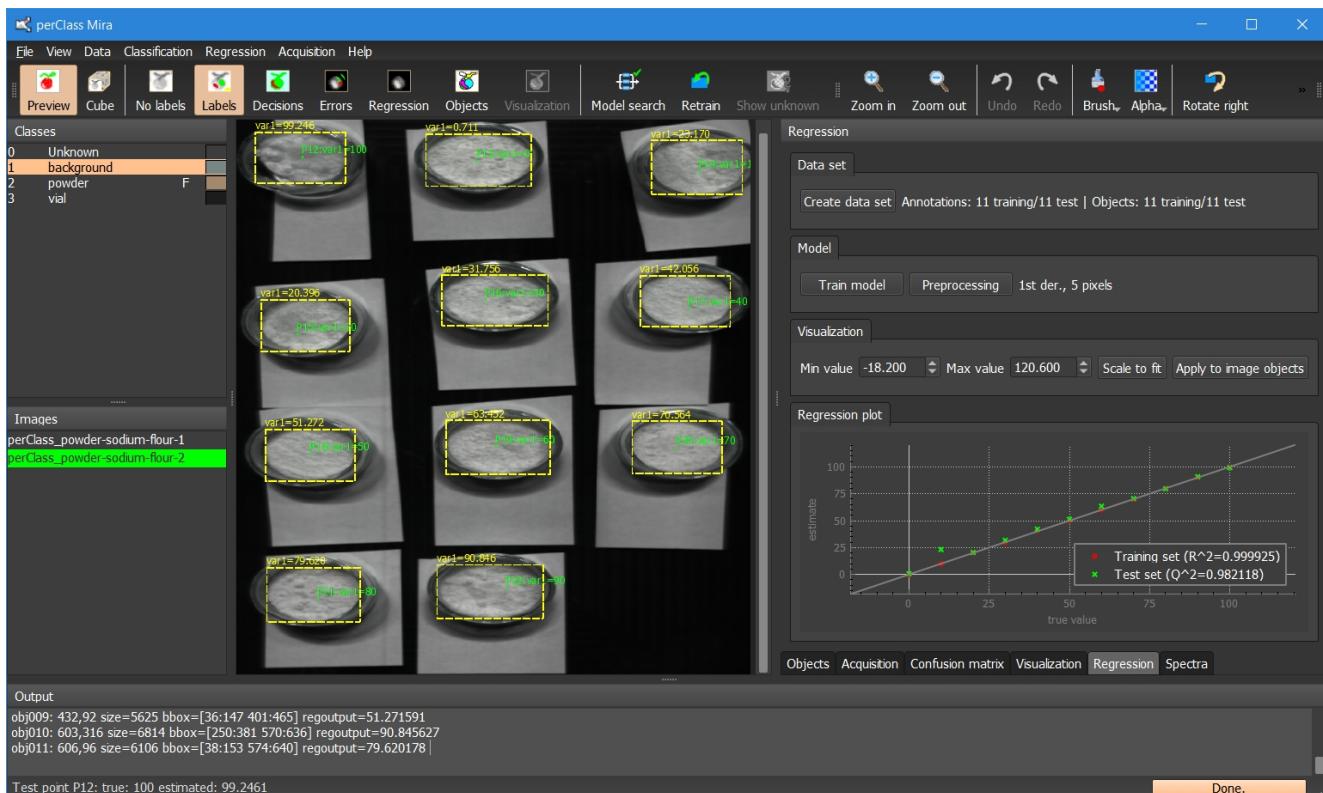
We were placing the mouse pointer over the green point in the upper right part of the plot. We can see that the marker refers to point P12 with the true fraction of 100 and the estimated value of 77.2. Note the corresponding object in the left upper corner of our scan - the bounding box shows the same 77.2 estimated value.

Improving regression models

How do we improve regression model model?

Preprocessing

For example, we may preprocess spectral data. In our example, we may use 1st spectral derivative and retrain the model. Updated model shows better fit:



perClass Mir provides R^2 and Q^2 statistics quantifying model fit in the regression plot legend. The R^2 refers to training objects and Q^2 to test objects. Our goal is to get as good test set fit as possible, meaning the highest Q^2 statistic.

Feature subset

In some problems, it may be beneficial to avoid certain parts of spectrum in regression modeling. Typically, start and end of the spectral range exhibit higher noise and may be better avoided. We may simply select desired set of bands in the spectral pannel.

perClass Mira allows to use different feature subset for the pixel classifier and different for the regressor. Both models adopt the selected bands when trained.

Limiting spatial neighborhood

In non-homogeneous objects, it may be useful to limit the object area used for building regression model. We may do that by defining "point radius". Use the *Set point radius* command from *Regression* menu. By default, the radius is 0 which means that all spectral of each object are used to build regression model. By limiting the radius to, for example 12 pixels, only the spectra inside the displayed circle will be utilized.



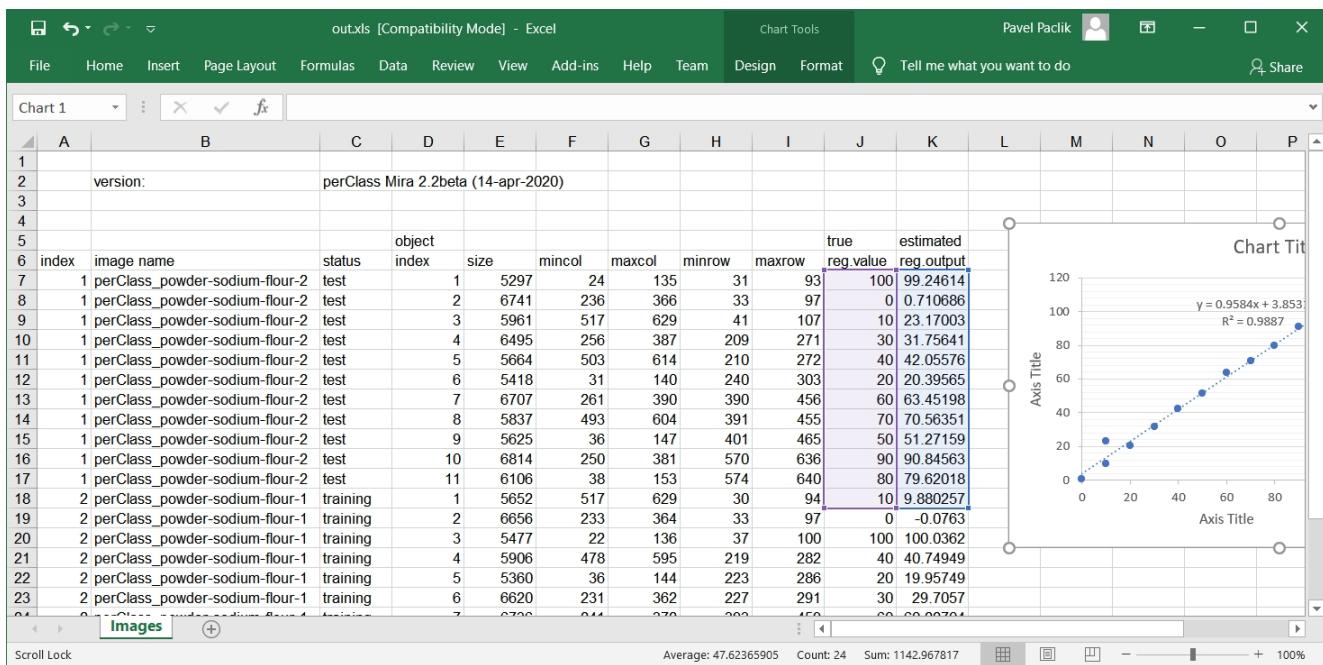
When point radius is adjusted, regression data set needs to be re-created with *Create data set* button.

Note, that the point radius is applied only in building the training and test sets. When applying the model to a new image using *Apply to new image* entire object is used.

Exporting regression results to Excel

For selected images, regression results may be exported to Excel. Select *File* menu, *Export - Export regression results* command.

For each selected image, all detected objects are reported with their indices, sizes, bounding box and true/estimated regression values.

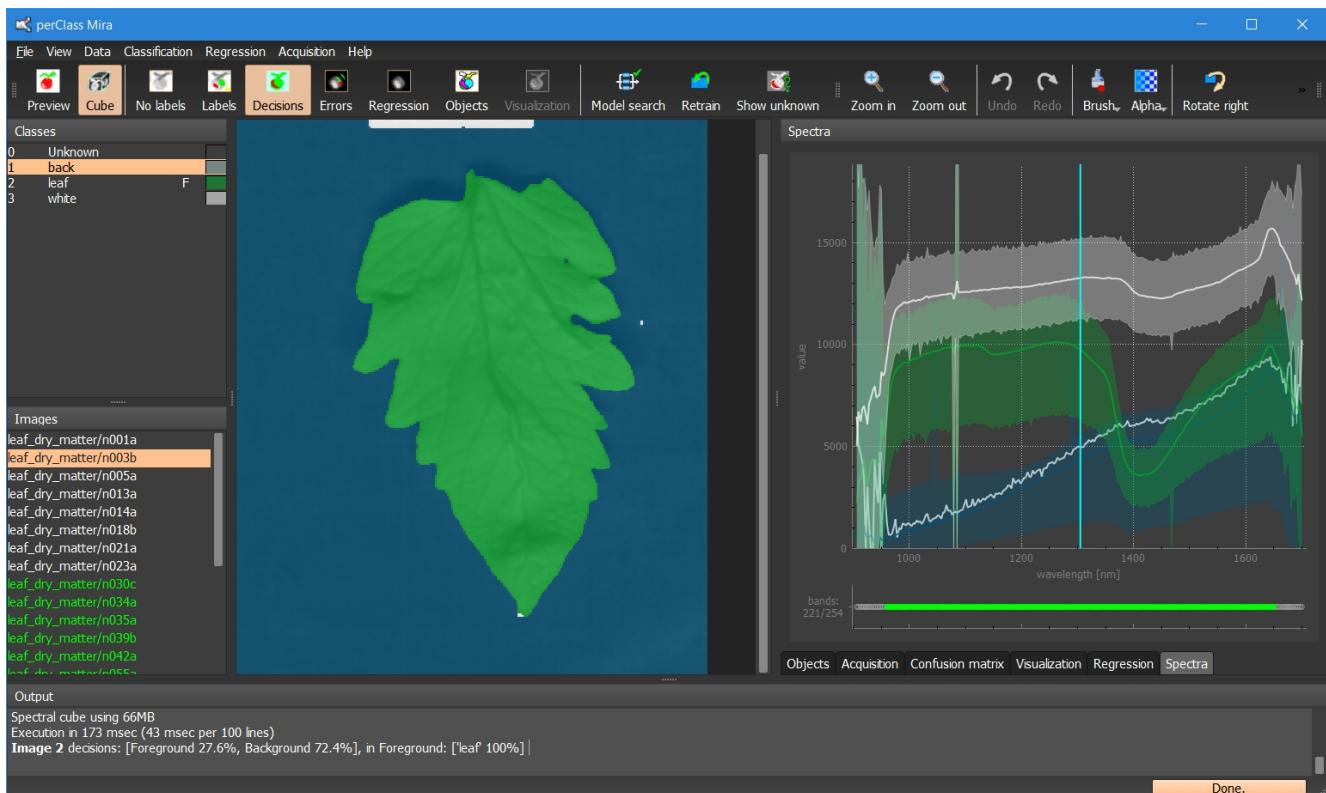


Importing annotations from Excel

In many regression problems, large number of annotated objects is needed to build good models. perClass Mira provides import of point annotations from Excel.

To demonstrate, we will use a different data set with leaves. The goal is to estimate dry-matter-content (DMC) per leaf.

We have added several scans to perClass Mira project and build a pixel classifier separating leaf from background. The "leaf" class is flagged as foreground.



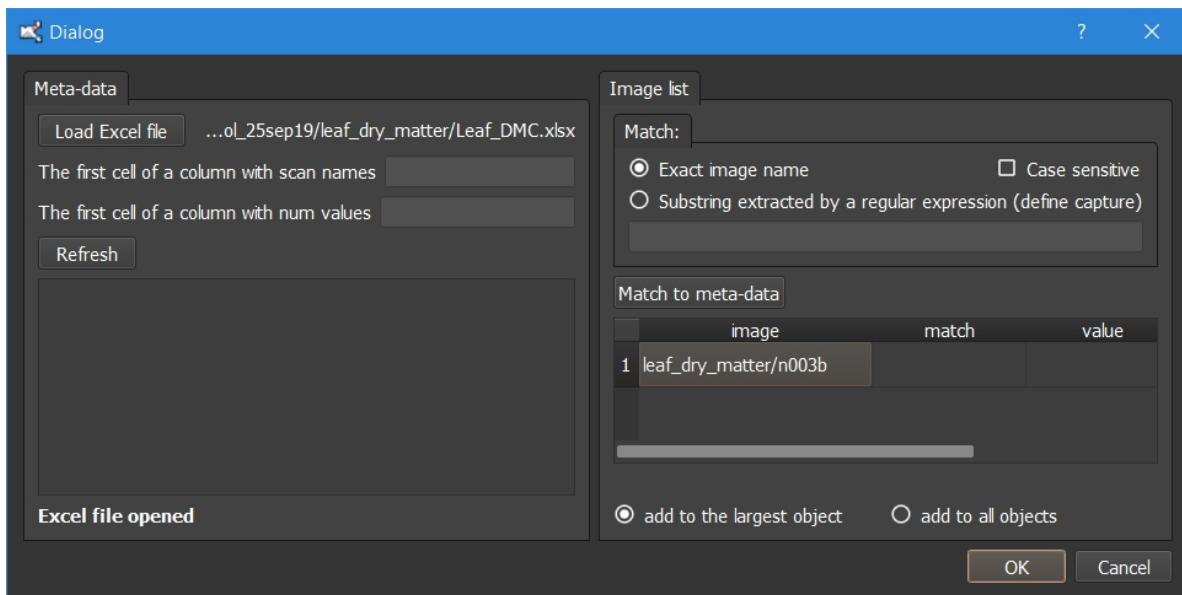
The ground-truth annotation is available in a separate Excel file:

The screenshot shows a Microsoft Excel spreadsheet titled "Leaf_DMC.xlsx - Excel". The ribbon menu includes File, Home, Insert, Page Layout, Formulas, Data, Review, View, Add-ins, Help, Team, Tell me what you want to do, and Share. The active sheet is "Sheet1".

	A	B	C	D	E	F	G	H	I	J
1	relationship	genotype	treatment	file_VIS	file_NIR	leaf_fruit	remark	DMC		
2	1	F	2	V001	N001	leaf	height 1: top	0.1480		
3	1	F	2	V002	N002	leaf	height 1: top	0.1578		
4	1	F	2	V003	N003	leaf	height 2: midd	0.1210		
5	1	F	2	V004	N004	leaf	height 2: midd	0.1108		
6	1	F	2	V005	N005	leaf	height 3: low	0.0887		
7	1	F	2	V006	N006	leaf	height 3: low	0.0895		
8	1	F	2	V007	N007	leaf	old leaf	0.0888		
9	1	F	2	V008	N008	leaf	old leaf	0.0905		
10	1	F	2	V009	N009	leaf	old leaf	0.0947		
11	1	F	2	V010	N010	leaf	old leaf	0.0883		
12	1	F	2	V011	N011	leaf	old leaf	0.0868		
13	1	F	2	V012	N012	leaf	old leaf	0.0852		

To import object annotation from this Excel file, we need to match scan name to dry-matter-content (DMC) value.

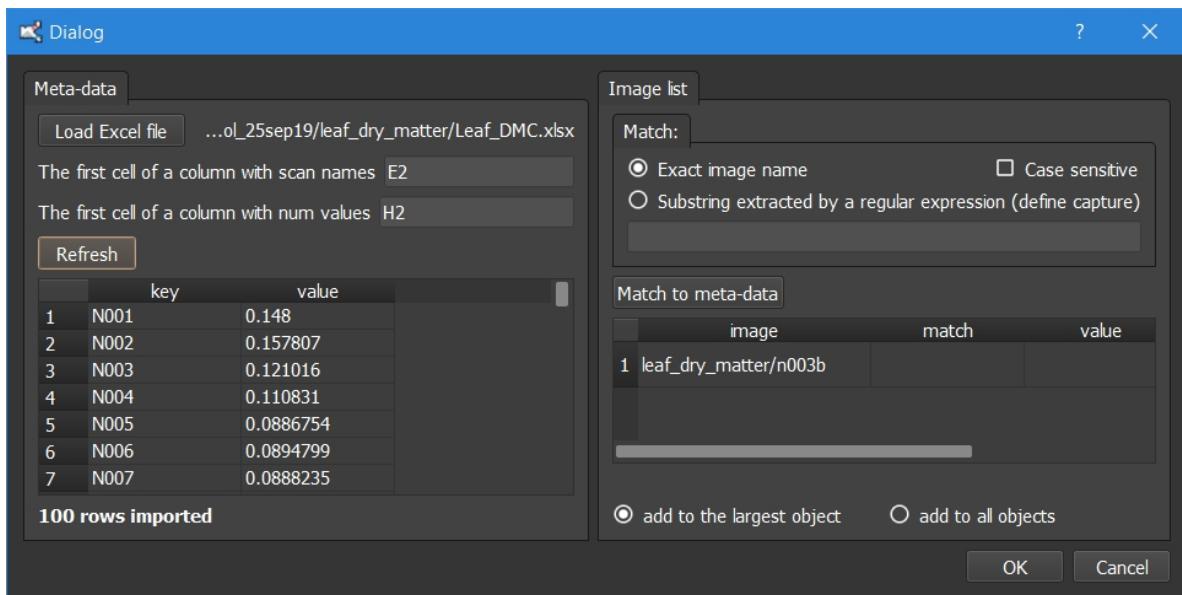
We will select *Import point meta-data* from *Regression* menu. A dialog will appear:



We have pressed *Load Excel file* button and selected the file containing meta-data.

Now we need to specify the first cell of a column containing scan names and numerical values.

In our example, it will be E2 for scan names and H2 for numerical values, respectively. We can then click on *Refresh* button. scan-value pairs will load in the meta-data pannel on the left.



Now we need to match scan names of selected images (single **leaf_dry_matter/n003b** image in our example) to the meta-data values.

In a simple situation, where the file name exactly matches the field in the Excel file, we can keep the default *Exact image name* option and press *Match to meta-data* button.

Often, the scan names contain additional characters: For example, we have the enclosing directory name ('leaf_dry_matter/') before the scan name and also extra letter after.

In perClass Mira, we can use regular expression to extract the scan name only. Regular expression is a text pattern describing how to match or extract substring from a string.

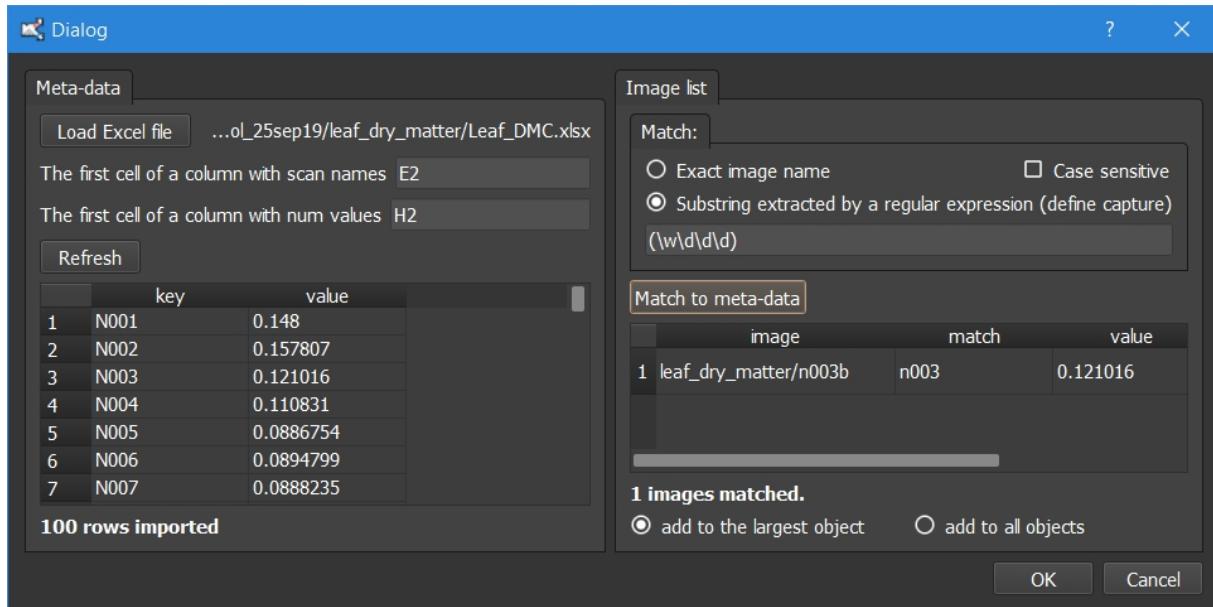
If you are not familiar with regular expressions, simply update your Excel file to list exact scan names. **TIP:** If you set the top-level data directory exactly above the scans, you will not see the enclosing directory as a part of the scan name ("leaf_dry_matter" in our example). This will simplify the task.

In our situation, we use a regular expression: `(\w\d\d\d)`

This means: There is a letter (`\w`) followed by three digits (`\d`). The parentheses are defining a "capture" i.e. part of the string returned. We need the parentheses to capture the scan name without the trailing letter.

To make sure this pattern does not match anywhere earlier, we could also use: `leaf_dry_matter/(\w\d\d\d)`

We click on *Match to meta-data* button and see that our selected image is matched to 0.121016:



Now we can press OK. By default, a point annotation will be created inside the largest object in the scan. We can also annotate all objects (larger than minimum object size).

Note, that the existing point annotations are not removed automatically. If needed, you can remove point annotations from the images prior to import, as [described here](#).

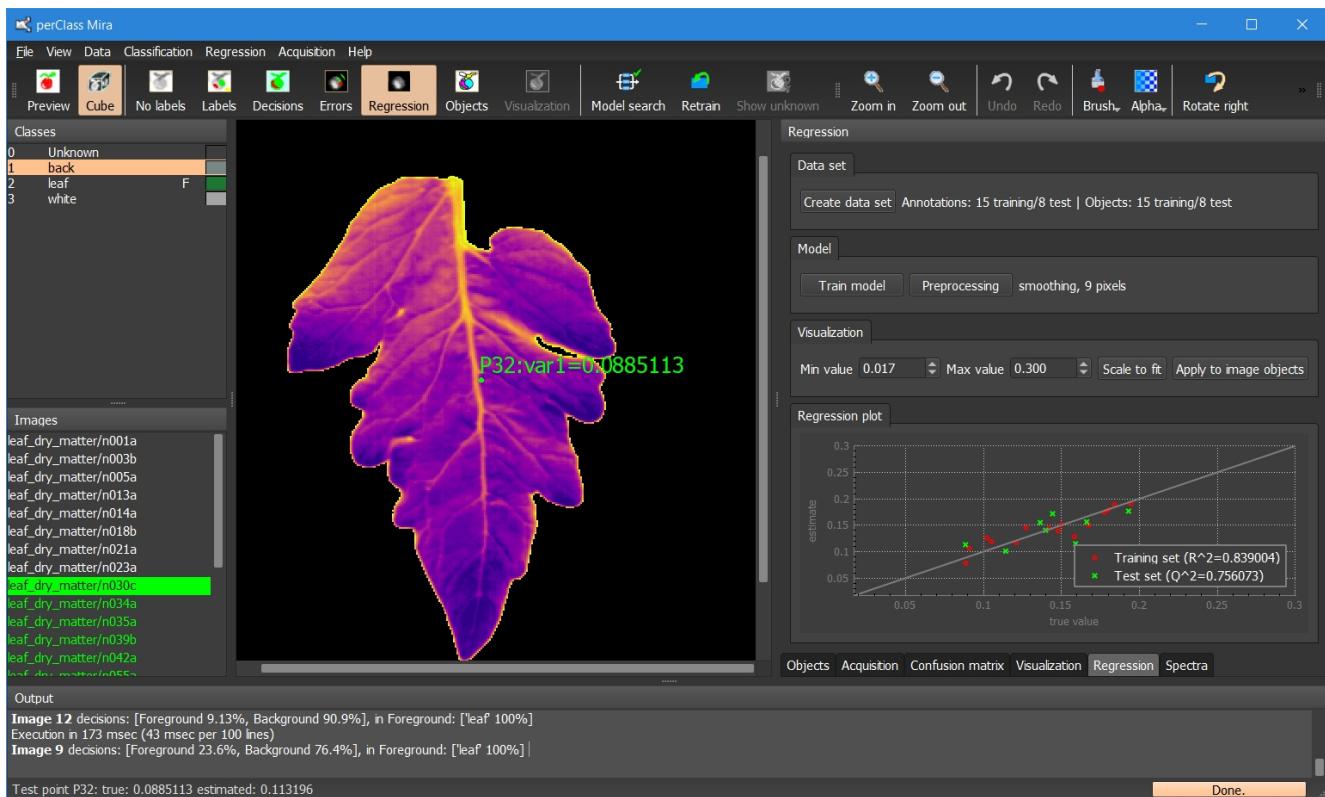
Naturally, the import dialog can process multiple scans at once. To do that, just select desired scans in the image list.

The import dialog remembers the settings in one perClass Mira session. So you may just return to it later and annotate several more images from the same Excel file with one click.

Per-pixel regression output

Apart of per-object regression, perClass Mira also provides per-pixel regression output.

When a regression model is trained, press the *Regression* button on the toolbar.



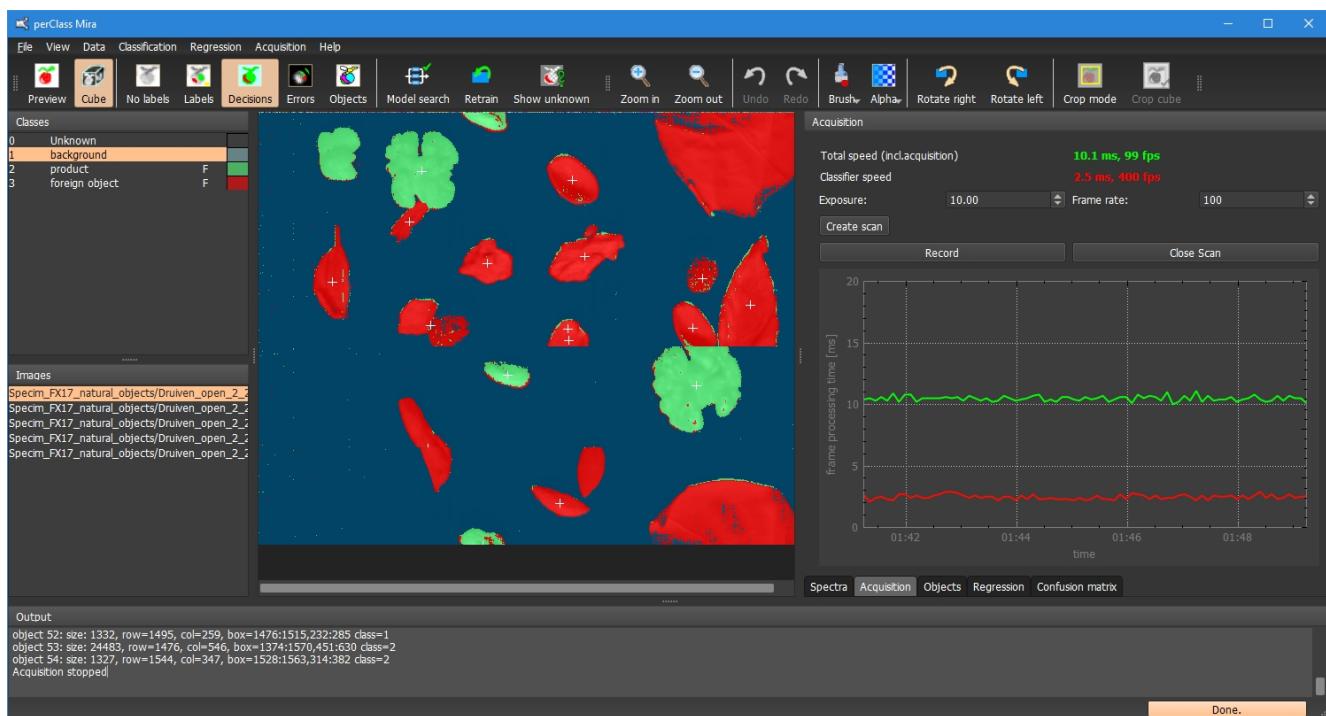
The image shows a false-color representation in the minimum and maximum value range defined by the values in the *Regression* pannel.

TIP: Inspect predicted value per pixel by hovering the mouse over the image.

As with other visualizations, regression rendering can be exported for selected images using *Export visualization* command in image-list right-click context menu.

Live data acquisition

Live acquisition functionality allows users with Specim cameras and Specsensor SDK installed apply models directly on data stream from a camera or a file via filereader interface.



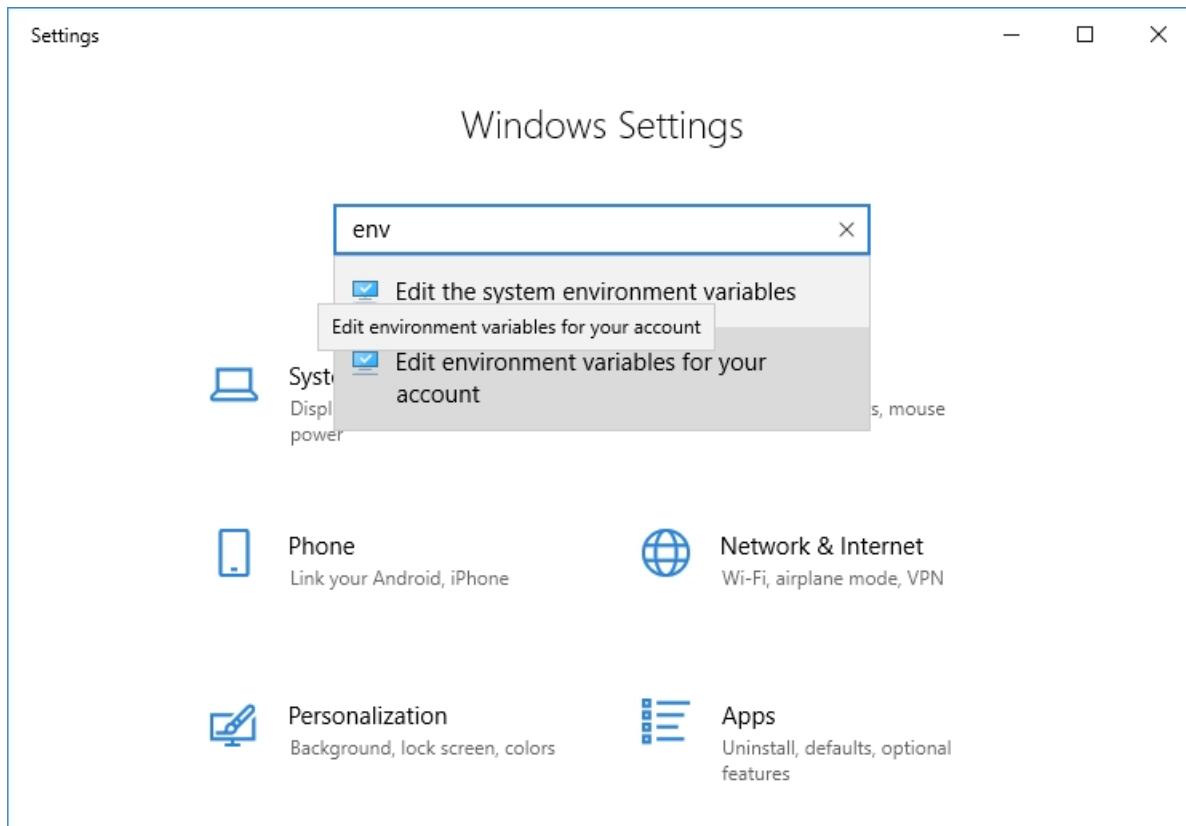
Installation of Specsensor SDK

Specim Specsensor SDK needs to be installed on the system running perClass Mira.

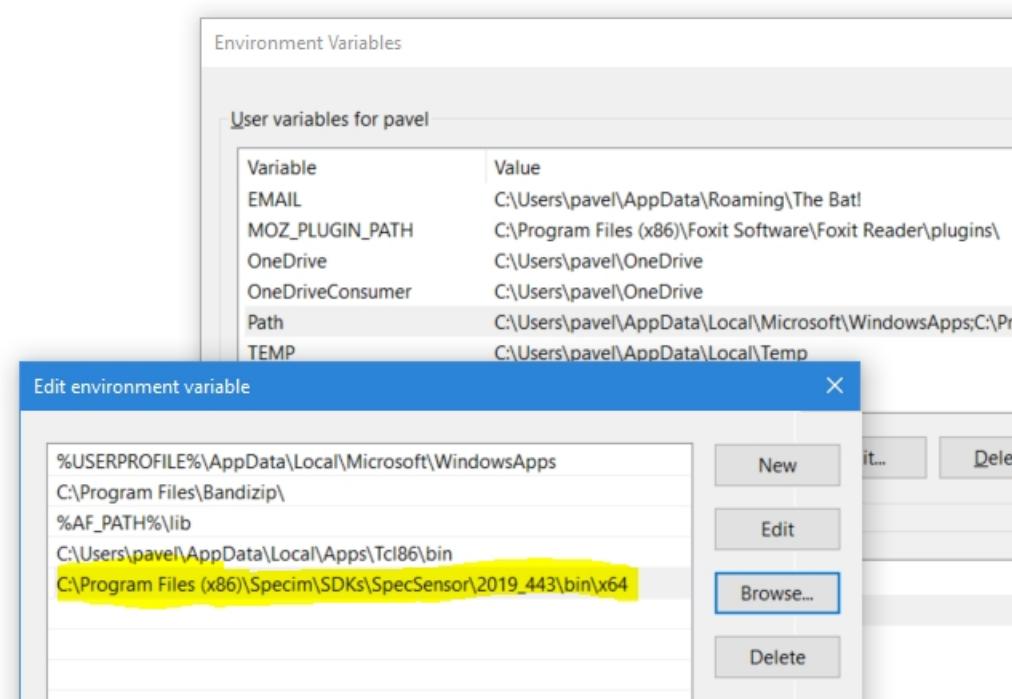
The Specsensor version needs to be at least 2018_415 (if you still use 2017 SDK release, please update). In general, it is advisable to update to the latest Specsensor release.

After Specsensor installation, make sure that the system Path environment variable contains the path to Specsensor

1. Open System Settings dialog



2. Open "Edit environmental variables for your account" dialog

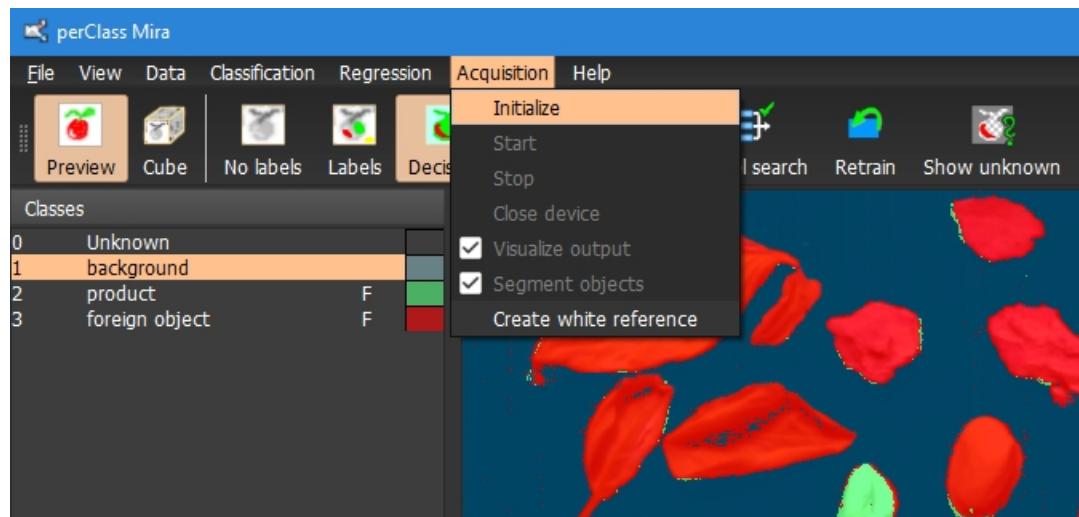


3. Make sure the path to points to the bin\x64 sub directory of the SDK

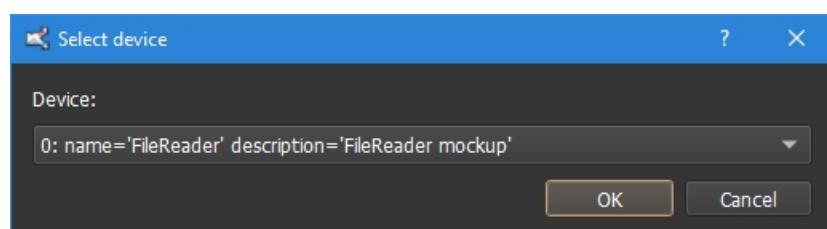
Restart of the machine is not needed, only restart of perClass Mira application.

Initializing acquisition

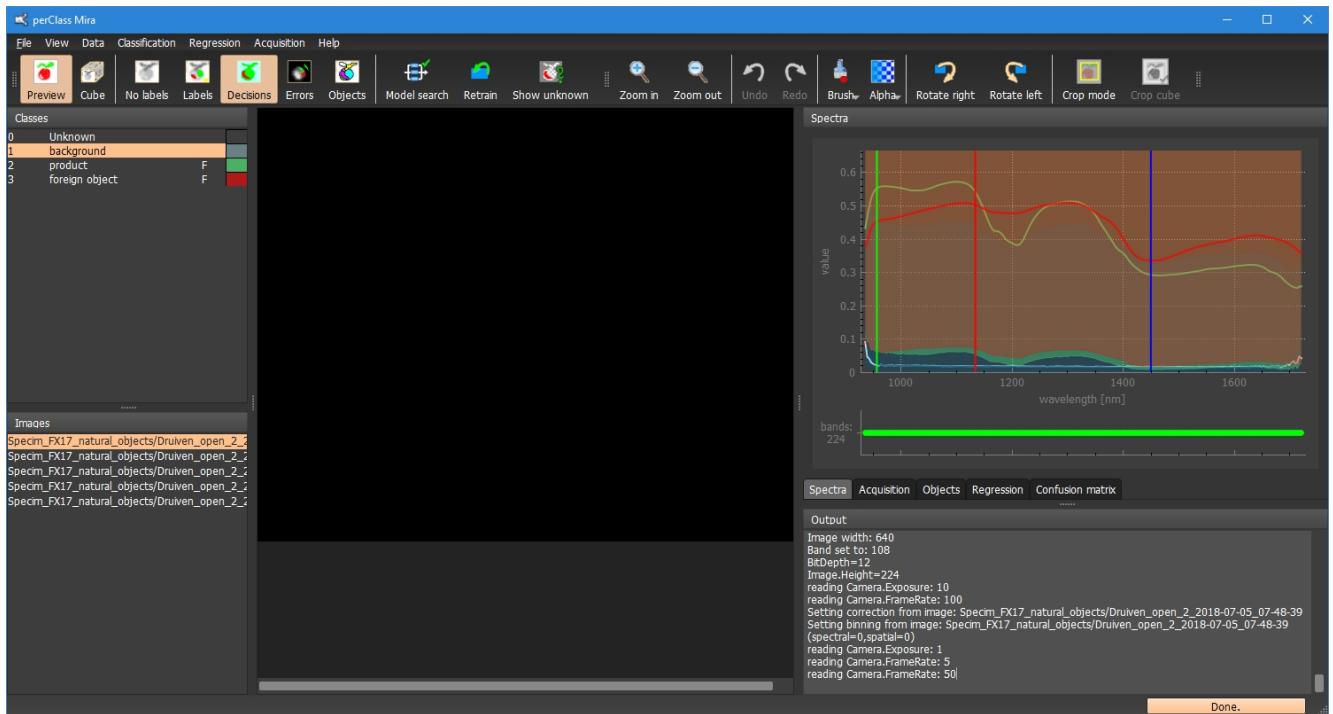
1. Create or open a Specim FX project
2. Train a classifier (a model must exist in order to process live data)
3. In the *Acquisition* menu, select *Initialize*



4. A dialog will appear listing all available devices accessible to Specsensor SDK



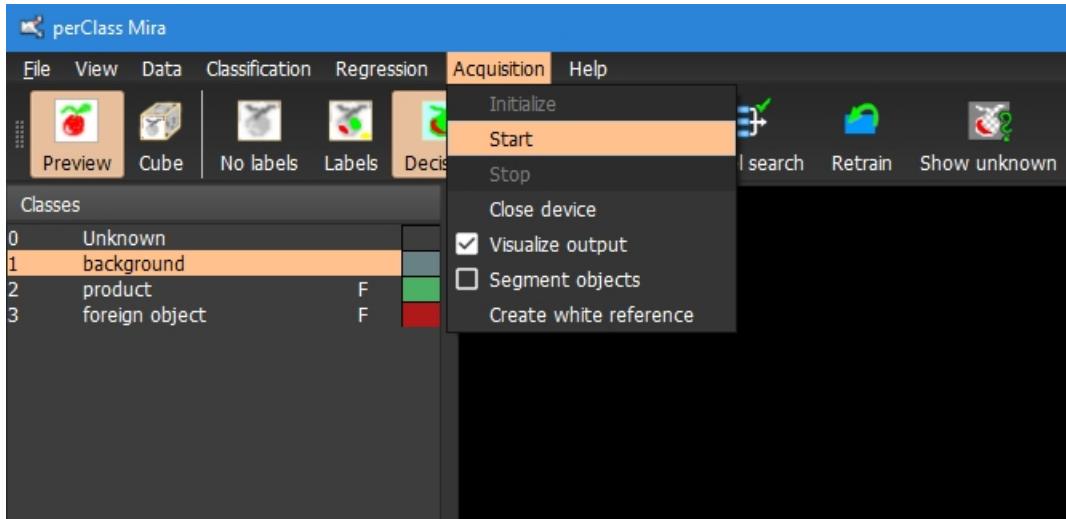
5. Select the camera or the FileReader interface and click OK
6. If using FileReader an Open File dialog will appear where you can point to the **raw spectral cube** simulating sensor acquisition stream. Select an existing Specim FX scan directory and the raw cube header file under the capture sub-directory
7. The window should show an empty buffer



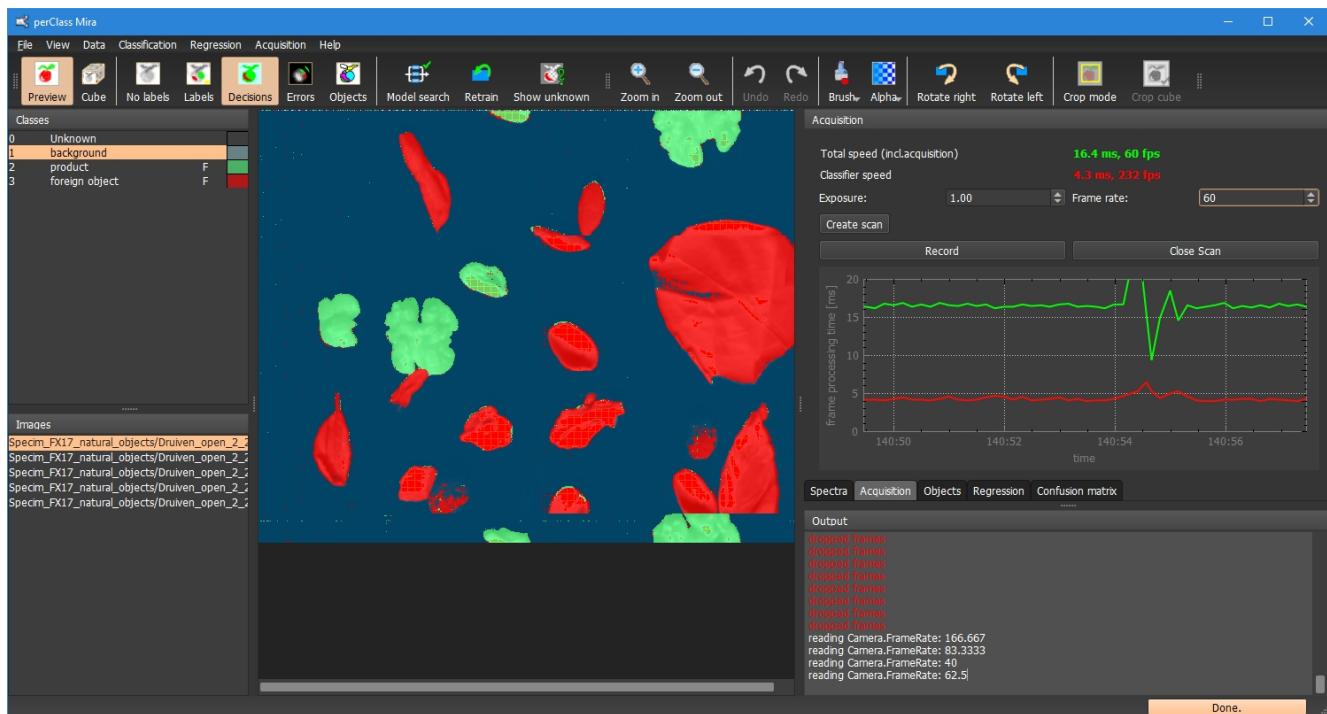
8. Process the data stream can now be started

live data processing

Select *Start* in *Acquisition* menu.



You should see new acquisition arriving in the window



On the right side of the screenshot you can see opened acquisition pannel. It shows the speed of the classifier in red and the total speed of the system in green.

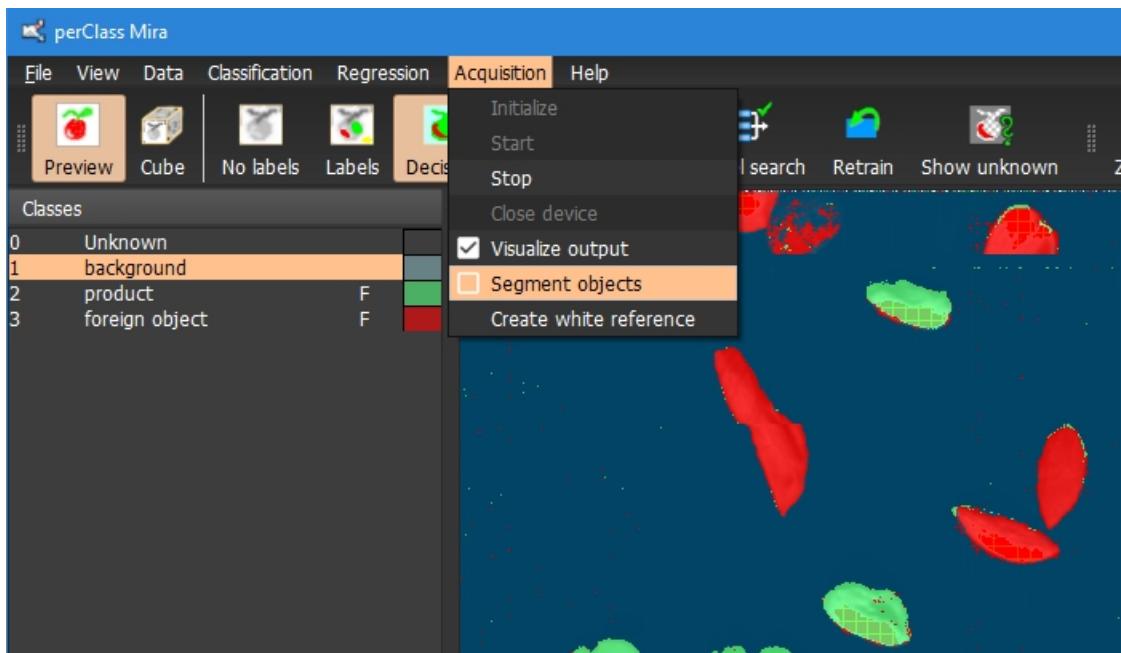
You can change the exposure and frame rate of the sensor from the pannel. By increasing the frame rate, you can speed up acquisition. If the classifier cannot process data at a given speed, you will see red message in the output window pointing out that frames are dropping.

Please note that some combinations of exposure and frame rate are invalid i.e. you cannot have high frame rate and high exposure at the same time.

Segmenting out objects

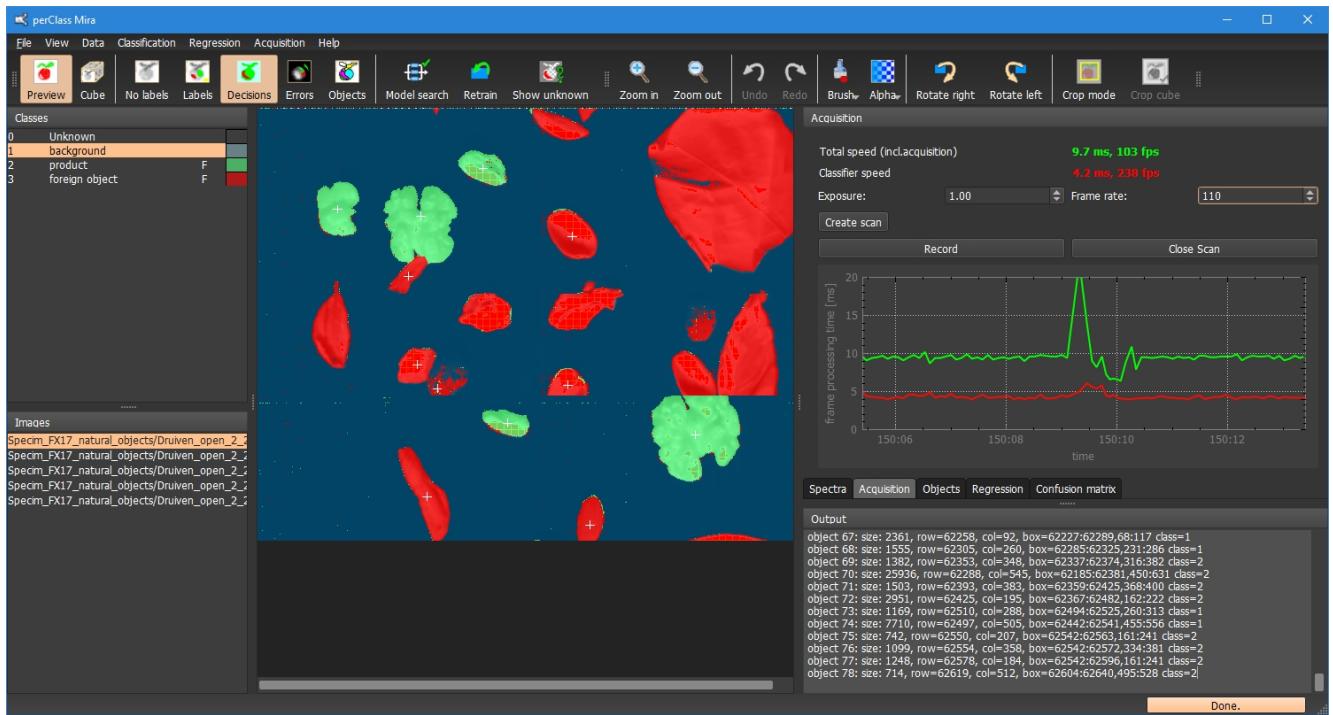
To segment objects on the live data stream:

1. Make sure that one or more classes are flagged as foreground in the class list
2. Switch on *Segment objects* in the *Acquisition* menu



Segmented objects will be marked with crosses at their center of gravity.

The output windows also shows object sizes, bounding boxes and classes (if multiple classes are used for segmentation)



The same per-object information is accessible from perClass Mira Runtime API from custom applications.

Closing live acquisition mode

Use *Stop* and *Close device* in the *Acquisition* menu to disable acquisition mode.

Reference information

Project types

perClass Mira offers several project types:

1. Cubert UH185

- Images are represented by multi-layer tiff files
- User needs to point to .cu3 file (50x50 spatial pixels)
- High-resolution panchromatic image is used for preview
- Multiple files can be loaded at once

2. Cubert ENVI

- Images are ENVI cubes saved from Cubert Utils
- This project type can be used for upsampled cubes from UH185 or Ultris

3. Cubert Tiff

- Images are native multi-layer tiff files from Cubert Ultris or UH185
- Preview is generated from the spectral cube

4. IMEC

- Images are ENVI cubes
- User needs to point to .hdr files
- It is assumed, that the data cube is stored in a file with the same name as .hdr file and with .raw extension
- There is no preview image expected

5. Headwall

- Images are ENVI cubes
- User needs to point to .hdr files
- It is assumed, that the data cube is stored in a file with the same name as .hdr file and without any extension
- No preview

6. Hyspex

- Images are ENVI cubes
- User needs to point to .hdr files
- It is assumed, that the data cube is stored in a file with the same name as .hdr file and without any extension
- A preview in a jpeg file is assumed to be available

7. Matlab

- Images are stored in .mat files
- Data cube is a 3D matrix in uint8, uint16, single or double precision
- User may select which variable to use upon load of the first image
- Optionally, a wavelength vector in single or double precision may be provided

8. SpecIm FX (LUMO scanner)

- Images are ENVI files
- User needs to point to the entire directory saved from LUMO scanner
- Preview image is assumed to be present
- Raw image cube is loaded and corrected by included white and dark standard images

9. SpecIm iQ (iQ Studio)

- Images are ENVI files

- User needs to point to the specific .hdr file representing already corrected cube (such as the reflectance cube in results directory)
- No preview is assumed
- Spectral cube is assumed to have .raw extension

10. Tiff stack

- A scan is a directory with individual band images in .tiff format
- Specific band image naming is expected (example: 231001_1_ch_0007_w_0419.tif - the scan name 231001_1, the 1-based channel index 7 and the wavelength in nm is 419)

11. Senop

- ENVI images with .dat extension for the spectral cube
- No reflectance correction is applied

Supported image formats

Supported image formats in perClass Mira are:

ENVI Files

- BIL, BIP or BSQ layout
- uint8, int16, uint16, single or double precision float
- Support for header offset
- Wavelengths are loaded from the wavelength field

Matlab

- 3D cubes as uint8, uint16, float/single and double precision
- BSQ layout is assumed with two spatial dimensions followed by the spectral dimension
- Separate vector of wavelengths in single or double precision can be provided when loading the first image to use proper wavelength range in nanometers.

TiffStack

- Image is a directory with a stack of tiff images, each representing one band.
- Channel index and wavelength is stored in a file-name
- Example: Band image for 7th channel (1-based indexing) representing wavelength 419nm. The first part of the filename is the image name (also the name of the directory)
 - 231001_1_ch_0007_w_0419.tif

Image zoom

- To zoom, use + and - toolbar buttons
- Alternative is to hold *Ctrl* key and use the mouse scroll-wheel

Keyboard shortcuts

In image view:

Ctrl+n	new class, ask name
n	new class, fill automatic name
0 (zero)	set "Unknown" class as current (removing labels)
1..9	set specific class as current
+-	zoom increase/decrease
Ctrl+1	zoom 100%

PgUp/PgDown	select next/previous spectral band
Ctrl+s	save project
Space	switch to the last label layer
Ctrl+Space	switch between preview and spectral cube
. (dot)	randomly change color of class under cursor
t	set class under cursor as current
m	run model search
r	retrain algorithm found in the last model search
shift-r	re-run the existing classifier on the image (to measure execution speed)
u	show data unseen in training (active learning)
shift+l	switch to no labels mode
l	switch to label mode
d	switch to decisions mode
e	switch to errors mode
o	find connected components (objects)
F1	open help
F11	toggle full screen mode
hold mouse & move	paint using current class
Shift+hold mouse	paint removing labels under cursor
Ctrl+mouse wheel	zoom image
Shift+mouse wheel	select next/previous spectral band
c	switch to confusion matrix
s	switch to spectral plot
x	mark class as excluded from training
b	mark class as background (for object definition)

In spectral plot (after clicking on spectral plot)

up/down	increase/decrease max display value
Shift+mouse wheel	select next/previous spectral band

In confusion matrix

Double click	add or remove constrain
Ctrl+mouse wheel	adjust constrain value live
Shift+N	toggle between normalized confusion matrix and absolute values (sample counts)
] (square bracket)	next valid solution
[previous valid solution

perClass Mira Runtime API

Initialization and cleanup

- [mira_Init](#)
- [mira_Release](#)

Error handling

- [mira_GetErrorCode](#)
- [mira_GetErrorMsg](#)

Computational device selection

- [mira_RefreshDeviceList](#)
- [mira_GetDeviceCount](#)
- [mira_GetDeviceName](#)
- [mira_SetDevice](#)

Loading model

- [mira_LoadModel](#)
- [mira_LoadCorrection](#)

Data processing - querying input data parameters

- [mira_GetInputWidth](#)
- [mira_GetInputHeight](#)
- [mira_GetInputBands](#)
- [mira_GetInputDataType](#)
- [mira_GetInputDataLayout](#)

Data processing

- [mira_SetMinObjSize](#)
- [mira_SetSegmentation](#)
- [mira_StartAcquisition](#)
- [mira_ProcessFrame](#)
- [mira_ProcessCube](#)
- [mira_StopAcquisition](#)

Processing results

- [mira_GetFrameDecisions](#)
- [mira_GetDecCount](#)
- [mira_GetDecName](#)
- [mira_GetDecColor](#)
- [mira_GetMaskType](#)
- [mira_GetObjCount](#)
- [mira_GetObjDataInt](#)
- [mira_GetObjDataClassSize](#)
- [mira_GetObjDataClassFrac](#)
- [mira_SaveImage](#)

[mira_Init](#)

Initialize runtime environment.

```
mrkernel* mira_Init(const char* path)
```

Input: Path to a directory with a license file

Output: Runtime environment pointer

Description:

The `mira_Init` function initializes runtime environment. It returns a pointer used for any other API function that interacts with the runtime.

The input is a path to a directory where a license file with `.lic` extension can be found. Pass `".` for the current directory.

After initialization, `mira_GetErrorMsg` provides welcome string listing software version or error message.

Example:

```
mrkernel* pmr=mira_Init( "." );
printf("Init: %s", mira_GetErrorMsg(pmr));
if( pmr==NULL ) return;
```

Error codes

- 101 Passing NULL pointer
- 102 `mira_GetDeviceName`: Device index out of bounds
- 103 `mira_StartAcquisition`: Project not loaded
- 104 `mira_StartAcquisition`: Classifier model not loaded

- 110 `mira_LoadModel`: Error loading model from file
- 111 `mira_LoadModel`: Wrong file format
- 112 `mira_LoadModel`: Internal error when loading
- 113 `mira_LoadModel`: File cannot be opened

- 120 `mira_SaveImage`: Label image does not exist

- 130 `mira_LoadCorrection`: Loading meta-data from the correction scan failed.
- 131 `mira_LoadCorrection`: Error loading dark reference data
- 132 `mira_LoadCorrection`: Dark and White reference images have different width or band count.
- 134 `mira_LoadCorrection`: Both dark and white reference scans need to be loaded.
- 135 `mira_LoadCorrection`: Reference file not present
- 136 `mira_LoadCorrection`: Unsupported data layout or data type

- 140 Error switching to the computation device
- 141 `mira_RefreshDeviceList`: Error setting CUDA backend
- 142 `mira_RefreshDeviceList`: Error setting OpenCL backend
- 143 `mira_RefreshDeviceList`: listNVIDIA and listOpenCL must be specified as 0 or 1 values

- 150 Feature does not exist
- 151 Wrong feature type requested

- 160 Max number of objects per frame reached
- 161 `mira_GetObjData*`: Object index out of bounds
- 162 `mira_GetObjData*`: Class index out of bounds (0..9)
- 163 `mira_GetObjDataClassSize`: Segmentation not set to required 'All foreground' mode.
- 164 `mira_GetMaskType`: Object segmentation not defined

- 170 `mira_StartAcquisition`: Acquisition already running
- 171 `mira_StopAcquisition`: Acquisition not running
- 172 `mira_StartAcquisition`: Object segmentation cannot proceed

- 180 `mira_GetDecName`: Decision index out of bounds

- 190 `mira_SetForegroundClass`: Foreground class index out of bounds

- 201 `mira_ProcessCube`: Line-scan project type cannot process cubes
- 202 `mira_StartAcquisition`: Label image dimension mismatch

- 203 `mira_ProcessCube`: Missing image geometry description

- 204 `mira_ProcessFrame`: Line-scan processing requires BIL layout

- 205 `mira_GetInputDataLayout`: Undefined data layout

- 206 mira_GetInputDataType: Undefined data type
- 207 mira_ProcessCube: Unsupported project type
- 208 mira_ProcessFrame: Unsupported data type
- 208 mira_ProcessCube: Unsupported data type or data layout

mira_GetVersion

Return runtime version

```
const char* mira_GetVersion()
```

Input: None

Output: Version string

Description:

`mira_GetVersion` returns version string together with the release date. For example "2.1 26-mar-2020".

mira_GetErrorCode

Return error code

```
int mira_GetErrorMsg(mrkernel* pmr)
```

Input: Runtime environment pointer

Output: Error code

Description:

`mira_GetErrorCode` returns error code. For a specific string description, [mira_GetErrorMsg](#)

mira_GetErrorMsg

Returns error message.

```
const char* mira_GetErrorMsg(mrkernel* pmr)
```

Input: Runtime environment pointer

Output: Error message string

Description:

`mira_GetErrorMsg` returns error message as string. For a specific error code, use [mira_GetErrorCode](#)

mira_RefreshDeviceList

Returns error message.

```
int mira_RefreshDeviceList(mrkernel *pmr, int listNVIDIA, int listOpenCL)
```

Input:

- Runtime environment pointer pmr
- Flag whether to search for and list NVIDIA devices (0: no, 1: yes)
- Flag whether to search for and list OpenCL devices (0: no, 1: yes)

Output: Result: MIRA_OK or error code

Description:

`mira_RefreshDeviceList` searches for computational devices (NVIDIA or OpenCL). After calling

`mira_RefreshDeviceList`, one can get device count using `mira_GetDeviceCount` and names with `mira_GetDeviceName`.

Example:

```
MIRA_CHECK( mira_RefreshDeviceList(pmr,1,0) );
const int devCount=mira_GetDeviceCount(pmr);
printf("\n%d devices:\n",devCount);
for(int i=0;i<devCount;i++) {
    printf("%d : %s\n",i, mira_GetDeviceName(pmr,i));
}
int deviceInd=atoi(argv[1]);
MIRA_CHECK( mira_SetDevice(pmr,deviceInd) );
printf("Device selected: %d '%%
s'\n",deviceInd,mira_GetDeviceName(pmr,deviceInd));
```

The `MIRA_CHECK` macro checks the output result. If error occurs, the program flow is terminated. See `perclass_mira.h` definition.

[mira_GetDeviceCount](#)

Returns the number of GPU devices found

```
int mira_GetDeviceCount(mrkernel* pmr)
```

Input: Runtime environment pointer

Output: Number of devices found

Description:

`mira_GetDeviceCount` returns the number of devices found by [mira_RefreshDeviceList](#).

[mira_GetDeviceName](#)

Returns the name of a specific computational device

```
const char* mira_GetDeviceName(mrkernel* pmr,int deviceInd)
```

Input:

- Runtime environment pointer
- Device index

Output: String name of a device

Description:

`mira_GetDeviceName` returns the name for a specific device. Before using `mira_GetDeviceName` or `mira_GetDeviceCount`, the device list needs to be constructed by [mira_RefreshDeviceList](#).

[mira_SetDevice](#)

Sets specific computational device

```
int mira_SetDevice(mrkernel* pmr,int deviceInd)
```

Input:

- Runtime environment pointer
- Device index

Output: Result code (MIRA_OK or `error`)

Description:

`mira_SetDevice` sets specific computational device. The device list needs to be constructed by [mira_RefreshDeviceList](#).

Example:

```
MIRA_CHECK( mira_RefreshDeviceList(pmr,1,0) );
const int devCount=mira_GetDeviceCount(pmr);
printf("\n%d devices:\n",devCount);
for(int i=0;i<devCount;i++) {
    printf("%d : %s\n",i, mira_GetDeviceName(pmr,i));
}
int deviceInd=atoi(argv[1]);
MIRA_CHECK( mira_SetDevice(pmr,deviceInd) );
printf("Device selected: %d '%
s'\n",deviceInd,mira_GetDeviceName(pmr,deviceInd));
```

The `MIRA_CHECK` macro checks the output result. If error occurs, the program flow is terminated. See `perclass_mira.h` definition.

mira_LoadModel

Loads classification model

```
int mira_LoadModel(mrkernell *pmr, const char* filename)
```

Input:

- Runtime environment pointer
- Filename (.mira project file)

Output: Result code (MIRA_OK or [error](#))

Description:

`mira_LoadModel` loads a classification model from .mira project file.

mira_LoadCorrection

Loads white and dark correction data from disk

```
int mira_LoadCorrection(mrkernell *pmr, const char* dirname, const char *scancode);
```

Input:

- Runtime environment pointer
- Dirname - a name of a directory containing a scan directory
- Scancode - a name of a scan directory

Output: Result code (MIRA_OK or [error](#))

*Description:***For Headwall project type:**

Correction information is assumed to be in whiteReference and darkReference scans. To load references, pass the path to a directory containing the whiteReference and darkReference ENVI scans. The third argument is NULL.

Header files must have .hdr extensions. Both spectral cubes can have arbitrary extensions. Reference cubes must be in BIL data layout. Both uint16 and float data types are supported.

Example:

```
res = mira_LoadCorrection(pmr, "path_to_dir_with_references",NULL);
```

In this way, both reference files are loaded at the same time.

For Specim project type:

`mira_LoadCorrection` loads dark and white correction information from `dirname` directory. The assumption

is the a scancode is a name of a directory inside the dirname directory and that it conforms Specim LUMO scanner directory structure. This means that inside scancode directory is a capture sub-directory. Inside the capture sub-dir, the following files are needed:

- WHITEREF_scancode.hdr
- WHITEREF_scancode.raw
- DARKREF_scancode.hdr
- DARKREF_scancode.raw
- scancode.hdr

The scancode.hdr defines wavelengths, band count and pixel count of a scan. Note, that the scancode.raw is not needed when loading correction.

All ENVI cubes are supposed to be in BIL data layout and use uint16 data type.

If `mira_LoadCorrection` is not called, the assumption is that the input data stream is already reflectance corrected and in float data type. This can be checked using `mira_GetInputDataType`.

[mira_SetMinObjSize](#)

Set the minimum object size for segmentation

```
int mira_SetMinObjSize(mrkernel *pmr, int minSize);
```

Input:

- Runtime environment pointer
- minSize - minimum object size in pixels

Output: Result code (MIRA_OK or [error](#))

Description:

`mira_SetMinObjSize` sets the minimum object size in pixels. Objects with size larger or equal than minSize are reported.

[mira_SetSegmentation](#)

Set the minimum object size for segmentation

```
int mira_SetSegmentation(mrkernel *pmr, int enable);
```

Input:

- Runtime environment pointer
- enable - flag is to enable (1) or disable (0) object segmentation

Output: Result code (MIRA_OK or [error](#))

Description:

`mira_SetSegmentation` enables or disables object segmentation. For all type of projects it is disabled by default (Note: before 2.3, it was enabled by default for line-scan projects).. Use before starting the acquisition. Note, that the model needs to have some class or classes flagged as foreground to perform segmentation.

[mira_GetInputWidth](#)

Get the expected width of the input image stream

```
int mira_GetInputWidth(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Input image width if > 0 or an error code if < 0

The image width in the line scan use case is the number of pixels of one line i.e. the pixels across the belt.

mira_GetInputHeight

Get the expected height of the input image stream

```
int mira_GetInputHeight(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Input image height if > 0 or an error code if < 0

This call is only meaningful in snapshot use-case where entire spectral cube is to be processed with mira_ProcessCube function.

mira_GetInputBands

Get the expected number of spectral bands of the input image stream

```
int mira_GetInputBands(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Input band count if > 0 or an error code if < 0

This call returns the number of spectral bands expected in each pixel.

mira_GetInputDataType

Get the expected data type in the input image stream

```
int mira_GetInputDataType(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Input data type if >= 0 or an error code if < 0

MIRA_DATATYPE_UNKNOWN	0
MIRA_DATATYPE_UINT16	1
MIRA_DATATYPE_FLOAT	2
MIRA_DATATYPE_UINT8	3

This call returns the data type expected in the input image stream based on the loaded model.

mira_GetInputDataLayout

Get the expected data layout of the input image stream

```
int mira_GetInputDataLayout(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Input data layout if ≥ 0 or an error code if < 0

```
MIRA_DATALAYOUT_UNKNOWN 0
MIRA_DATALAYOUT_BIP      1      /* spectrum-by-spectrum (dimensions: bands-width-height) */
MIRA_DATALAYOUT_BIL      2      /* frame-by-frame (dimensions: width-bands-height) */
MIRA_DATALAYOUT_BSQ      3      /* spatial frame by frame (dimensions: width-height-bands) */
```

This call returns the data layout expected in the input image stream based on the loaded model.

mira_GetMaskType

Get the mask type of the loaded object segmentation model

```
int mira_GetMaskType(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Input data layout if ≥ 0 or an error code if < 0

```
MIRA_MASK_EACH_FOREGROUND 1
MIRA_MASK_ALL_FOREGROUND 2
```

This call returns the mask type for the loaded model. The 'each-foreground' type is used for single material per object situations (object detection). The 'all-foreground' mask is used for complex objects composed of multiple materials where the union of classes defines object mask (object classification). An example: A potato can have healthy flesh or rotten defect - these are the trained classes. The segmentation mask is set of 'all-foreground' and, therefore, entire piece of potato is segmented out. For each object, perClass Mira Runtime provides pixel count for each foreground class. This allows object sorting based on composition.

mira_StartAcquisition

Starts the acquisition

```
int mira_StartAcquisition(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Result code (MIRA_OK or [error](#))

Description:

`mira_StartAcquisition` starts the acquisition process. Computational device must be defined, model loaded and correction information defined.

Individual frames can then be processed using [mira_ProcessFrame](#)

mira_ProcessFrame

Process a single individual raw spectral frame from a line-scan camera

```
int mira_ProcessFrame(mrkernel* pmr, void* pData);
```

Input:

- Runtime environment pointer
- Pointer to external buffer with raw spectral frame data

Output: Result code (MIRA_OK or [error](#))

Description:

`mira_ProcessFrame` passes data of a raw spectral frame from a line-scan camera. Acquisition process need to be running (started using `mira_StartAcquisition`).

The input data stream from a line scan camera is expected to be in BIL layout (pixels on the spatial line times spectral bands). The expected geometry and data type are defined by the loaded solution. This information can be queried by the `mira_GetInputWidth`, `mira_GetInputBands`, and `mira_GetInputDataType`.

After a frame is processed, per-pixel decisions may be read out using `mira_GetFrameDecisions` or object information extracted using `mira_GetObj*` functions.

[mira_ProcessCube](#)

Process a single spectral cube

```
int mira_ProcessCube(mrkernel* pmr, void* pData);
```

Input:

- Runtime environment pointer
- Pointer to external buffer with raw spectral frame data

Output: Result code (MIRA_OK or [error](#))

Description:

`mira_ProcessCube` passes data of a entire spectral cube. Acquisition process need to be running (started using `mira_StartAcquisition`).

The expected geometry and data type are defined by the loaded solution. This information can be queried by the `mira_GetInputHeight`, `mira_GetInputWidth`, `mira_GetInputBands`, `mira_GetInputDataLayout` and `mira_GetInputDataType`.

After a cube is processed, per-pixel decisions may be read out using `mira_GetFrameDecisions` or object information extracted using `mira_GetObj*` functions.

[mira_StopAcquisition](#)

Starts the acquisition

```
int mira_StopAcquisition(mrkernel* pmr)
```

Input:

- Runtime environment pointer

Output: Result code (MIRA_OK or [error](#))

Description:

`mira_StopAcquisition` stops the acquisition process. Statistics on number of processed frames, speed per frame and frame-rate is available via a subsequent `mira_GetErrorMsg` call.

Example:

Average alg time: 2.79676 ms/frame (357.557 fps), processed 1500 frames, first 500 skipped for warm-up.

[mira_GetFrameDecisions](#)

Returns a pointer to pixel decisions on the last processed line

```
const unsigned char *mira_GetFrameDecisions(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Pointer to decisions on the last processed line

Description:

`mira_GetFrameDecisions` returns pointer to decisions at the last processed line. The values are zero-based indices. Class name corresponding to each index can be obtained using `mira_GetDecName`

mira_GetDecCount

Returns the number of decisions provided by the classifier

```
int mira_GetDecCount(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Number of decisions provided by the classifier

Description:

`mira_GetDecCount` returns the number of decisions provided by the classifier. The decision index, returned for each pixel by `mira_GetFrameDecisions` is a value smaller than decision count (zero-based indexing).

mira_GetDecName

Returns decision (class) name given decision index

```
const char* mira_GetDecName(mrkernel* pmr, int decInd);
```

Input:

- Runtime environment pointer
- Decision index (0 to number of decisions - 1)

Output: String name of a class (decision)

Description:

`mira_GetDecName` returns the name for a specific decision index.

Example:

```
printf("classifier decisions:\n");
const int decCount=mira_GetDecCount(pmr);
for(int i=0;i<decCount;i++) {
    printf("%d : %s\n",i,mira_GetDecName(pmr,i));
}
```

Output:

```
classifier decisions:
0 : background
1 : product
2 : foreign object
```

mira_GetDecColor

Returns R,G,B color of a given decision

```
const char* mira_GetDecColor(mrkernel* pmr, int decInd, unsigned char* R, unsigned
char* G, unsigned char* B);
```

Input:

- Runtime environment pointer
- Decision index (0 to number of decisions - 1)
- pointer to red, green and blue color

Output: String name of a class (decision)

Description:

`mira_GetDecColor` returns R,G, and B colors for a given decision

mira_GetObjCount

Returns the number of objects found after processing a frame

```
int mira_GetObjCount(mrkernel* pmr);
```

Input:

- Runtime environment pointer

Output: Number of objects found after processing a given frame

Description:

`mira_GetObjCount` returns the number of objects found after processing a given frame. If non-zero, the object information can be read using `mira_GetObjData*` functions, [see this example](#).

Note, that the object-specific information is replaced after next frame processing.

mira_GetObjDataInt

Read information on segmented out objects

```
int mira_GetObjDataInt(mrkernel* pmr, int entryInd, int** ppObjData);
```

Input:

- Runtime environment pointer
- entryID - zero-based index of an object
- ppObjData - pointer to a pointer to a table with object information

Output: Result (MIRA_OK or [error](#))

Description:

`mira_GetObjDataInt` returns details on a specific object found. The first parameter is a zero-based object index (0..number of objects found -1). The second parameter represents an object table. The example below illustrates that we declare a pointer to int called pObjData and initialize its value to NULL. In an acquisition loop, after processing a frame, if an object is found, we call `mira_GetObjDataInt` in a for loop extracting object information. Note, that we pass address of the pObjData to the `mira_GetObjDataInt`.

The object table:

MIRA_OBJECT_ID	0	Unique object identifier
MIRA_OBJECT_FRAME	1	Frame index for the object centroid
MIRA_OBJECT_POS	2	Position of the object centroid across the belt
MIRA_OBJECT_MINFRAME	3	Bounding box coordinates:
MIRA_OBJECT_MAXFRAME	4	
MIRA_OBJECT_MINCOL	5	
MIRA_OBJECT_MAXCOL	6	
MIRA_OBJECT_SIZE	7	Object size in pixels
MIRA_OBJECT_CLASS	8	Object class index

Example:

```
int objCount=0;
int* pObjData=NULL; /* pointer to object table data */
while( frameInd<frameCount ) {
```

```

mira_ProcessFrame(pmr, pFrame);

objCount=mira_GetObjCount(pmr);
if( objCount>0 ) {
    for(int i=0;i<objCount;i++) {
        /* we pass address of a pointer to receive object table
         * allocated by the runtime */

        mira_GetObjDataInt(pmr, i, &pObjData);

        /* pObjData allows us to access object details */
        printf("\n obj%d : %d, %d, %d, %d",
               pObjData[MIRA_OBJECT_ID],
               pObjData[MIRA_OBJECT_FRAME], /* along the belt */
               pObjData[MIRA_OBJECT_POS] /* across the belt */
        } /* end of for loop */
    } /* end of if objects */
} /* end of frame acquisition */

```

mira_GetObjDataClassSize

For complex object segmentation, returns number of class pixels within the object

```
int mira_GetObjDataClassSize(mrkernell* pmr, int entryInd, int classInd);
```

Input:

- Runtime environment pointer
- entryInd - Object index (zero-based)
- classInd - Class index (zero-based)

Output: Number of pixels of specific class within specific object or error

Description:

`mira_GetObjDataClassSize` returns the number of pixels of specific class within an object. This function is applicable when object segmentation mask is defined as "all foreground" i.e. when complex objects are segmented.

mira_GetObjDataClassFrac

For complex object segmentation, returns the fraction of class pixels within the object

```
float mira_GetObjDataClassFrac(mrkernell* pmr, int entryInd, int classInd);
```

Input:

- Runtime environment pointer
- entryInd - Object index (zero-based)
- classInd - Class index (zero-based)

Output: Fraction (0.0 to 1.0) inclusive of pixels of specific class within specific object or error code

Description:

`mira_GetObjDataClassFrac` returns the fraction of specific class within an object. This function is applicable when object segmentation mask is defined as "all foreground" i.e. when complex objects are segmented.

If error occurs the negative error code value is returned.

mira_SaveImage

Save internal segmentation buffer as PNG image

```
int mira_SaveImage(mrkernel* pmr, const char* filename)
```

Input:

- Runtime environment pointer
- Filename

Output: Result (MIRA_OK or error)

Description:

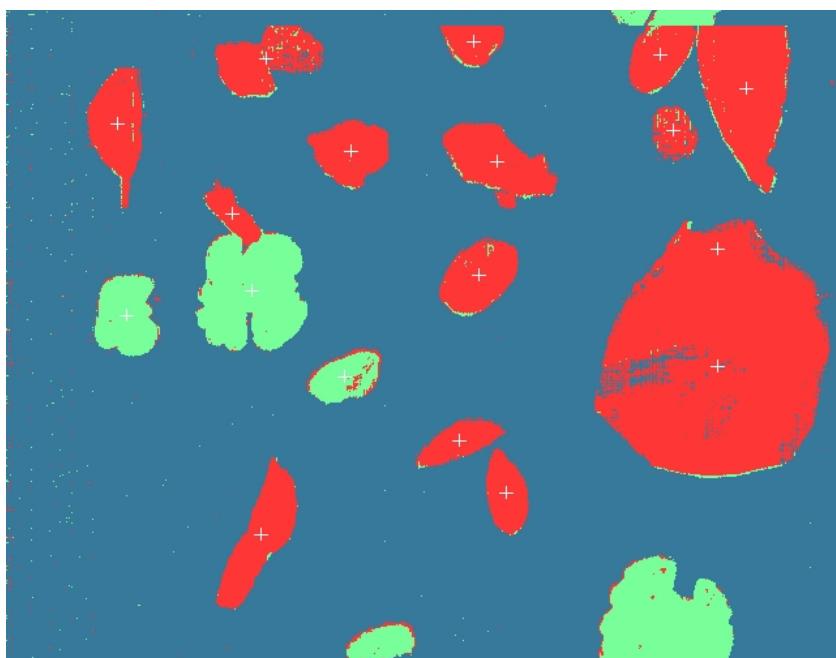
`mira_SaveImage` saves internal segmentation buffer into a PNG file. It is intended as a quick visualization of what the classifier can "see" in a deployed system.

Segmented objects are highlighted by white crosses.

Example:

```
MIRA_CHECK( mira_StopAcquisition(pmr) );
printf( "\n %s",mira_GetErrorMsg(pmr));
/* We can save the content of the internal buffer. */
mira_SaveImage(pmr,"out.png");
mira_Release(pmr);
```

Output: Content of out.png file



mira_Release

Release runtime internal session and clean resources.

```
void mira_Release(mrkernel* pmr)
```

Input:

- Runtime environment pointer

Output: None

Description:

`mira_Release` end the session and releases all memory allocated by the runtime.