SUPERVISED_CLASSIFICATION PLUGIN

The **Supervised Classification** plugin is a QGIS plugin that can classify land cover classes in satellite images using different methods of supervised and unsupervised classification.

In this notebook, the main python file "*supervised_classification_dialog.py*" will be explained and commented, so that anyone who wants to modify it, expand it or improve it, will be able to properly understand what was done, why, and where.

As any code, in the first part we define the import of libraries and modules that we are going to need during the implementation of the code. These include, as an example, a variety of standard python libraries, machine learning modules (sklearn), qgis data management (gdal), and plugin aiding modules (PyQt, qgis).

```
from qgis.PyQt.QtCore import QSettings, QTranslator, QCoreApplication
from qgis.PyQt.QtGui import QIcon
from qgis.PyQt.QtWidgets import QAction, QFileDialog, QMessageBox,
QDialog, QProgressBar
from qgis.PyQt.QtWidgets import QVBoxLayout, QWidget, QSizePolicy
from qgis.core import QgsProject, Qgis
from qgis.gui import QgsMessageBar
from PyQt5 import QtGui
import time
import numpy as np
from osgeo import gdal
from sklearn.cluster import MiniBatchKMeans
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import LinearSVC
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import RidgeClassifier
from joblib import dump, load
# Initialize Qt resources from file resources.py
from .resources import *
# Import the code for the dialog
from .supervised_classification_dialog import
SupervisedClassificationDialog
from .probar_dialog import Ui_Form
import os.path
import webbrowser
from datetime import datetime
```

We then go on to define a class named SupervisedClassification, which represents the implementation of the QGIS plugin.

```
class SupervisedClassification:
```

This code initializes the *Supervised Classification* class, by setting up:

1. QGIS interface reference
2. Plugin directory
3. Local settings
4. Translation
5. Instance attributes

It lays down the foundation for the plugin to interact with QGIS.

```
## The init method is the constructor of the class.
## It takes an iface parameter - an instance of the QgsInterface class
that
## provides a hook for interacting with the QGIS app
def __init__(self, iface):
        """Constructor.
        :param iface: An interface instance that will be passed to
this class
            which provides the hook by which you can manipulate the
QGIS
            application at run time.
        :type iface: QgsInterface
        """
        ## In this part, the iface parameter is saved as an instance
attribute self.iface,
        ## which allows access to the QGIS interface throughout the
plugin.
        self.iface = iface

        ## The plugin directory is initialized by getting the
directory path of the current file using os.path.dirname(__file__).
        ## This allows the plugin to reference files and resources
relative to its location.
        self.plugin_dir = os.path.dirname(__file__)

        ## This retrieves the user's local settings. It extracts the
first two characters
        ## to get the language code.
        locale = QSettings().value('locale/userLocale')[0:2]
        ## this sets the local path by joining the 3 arguments.
        locale_path = os.path.join(
            self.plugin_dir,
            'i18n',
            'SupervisedClassification_{}.qm'.format(locale))

        ## If the translation file exists at the path that we have
just created,
        ## we create a instance of QTranslator and load it with the
translation
```

```
        ## file. Then we call to install the translator, so that the
user inter-
        ## face elements may be translated
        if os.path.exists(locale_path):
            self.translator = QTranslator()
            self.translator.load(locale_path)
            QCoreApplication.installTranslator(self.translator)

        ## Declares instance attributes. self.actions is an empty list
that will
        ## store the plugin's actions. self.menu will translate the
plugin's menu
        self.actions = []
        self.menu = self.tr(u'&Supervised Classification')

        ## This checks if the plugin was already started for the first
time in
        ## the current QGIS session and needs to perform some
initialization tasks.
        ## It will be set later on, in the initGui() method, in order
to survive
        ## plugin reloads
        self.first_start = None
```

The **tr** function provides a method to translate strings within the user interface, by using the Qt translation API.

```
 # noinspection PyMethodMayBeStatic
    def tr(self, message):
        """Get the translation for a string using Qt translation API.

        We implement this ourselves since we do not inherit QObject.

        :param message: String for translation.
        :type message: str, QString

        :returns: Translated version of message.
        :rtype: QString
        """
        # noinspection PyTypeChecker,PyArgumentList,PyCallByClass
        return QCoreApplication.translate('SupervisedClassification',
message)
```

The **add_action** function gives a way to create and add actions to the plugin's user interface, like toolbar icons and menu items. It allows the customization of various parameters related to the action and handles the processes to integrate the action in the GUI.

Not much will be said about this, as it a standard implementation

```
 def add_action(
        self,
        icon_path,
        text,
        callback,
        enabled_flag=True,
        add_to_menu=True,
        add_to_toolbar=True,
        status_tip=None,
        whats_this=None,
        parent=None):
        """Add a toolbar icon to the toolbar.

        :param icon_path: Path to the icon for this action. Can be a
resource
            path (e.g. ':/plugins/foo/bar.png') or a normal file
system path.
        :type icon_path: str

        :param text: Text that should be shown in menu items for this
action.
        :type text: str

        :param callback: Function to be called when the action is
triggered.
        :type callback: function

        :param enabled_flag: A flag indicating if the action should be
enabled
            by default. Defaults to True.
        :type enabled_flag: bool

        :param add_to_menu: Flag indicating whether the action should
also
            be added to the menu. Defaults to True.
        :type add_to_menu: bool

        :param add_to_toolbar: Flag indicating whether the action
should also
            be added to the toolbar. Defaults to True.
        :type add_to_toolbar: bool

        :param status_tip: Optional text to show in a popup when mouse
pointer
            hovers over the action.
        :type status_tip: str

        :param parent: Parent widget for the new action. Defaults
None.
```

```
        :type parent: QWidget

        :param whats_this: Optional text to show in the status bar
when the
            mouse pointer hovers over the action.

        :returns: The action that was created. Note that the action is
also
            added to self.actions list.
        :rtype: QAction
        """

        icon = QIcon(icon_path)
        action = QAction(icon, text, parent)
        action.triggered.connect(callback)
        action.setEnabled(enabled_flag)

        if status_tip is not None:
            action.setStatusTip(status_tip)

        if whats_this is not None:
            action.setWhatsThis(whats_this)

        if add_to_toolbar:
            # Adds plugin icon to Plugins toolbar
            self.iface.addToolBarIcon(action)

        if add_to_menu:
            self.iface.addPluginToMenu(
                self.menu,
                action)

        self.actions.append(action)

        return action
```

This part defines the **initGui** function, which creates the menu entries and toolbar icons in the GUI.

This part of the code also comes automatically with Plugin Builder.

```
def initGui(self):
        """Create the menu entries and toolbar icons inside the QGIS
GUI."""
        icon_path = ':/plugins/supervised_classification/icon.png'
        self.add_action(
            icon_path,
            text=self.tr(u'Supervised Classification'),
            callback=self.run,
            parent=self.iface.mainWindow())
```

```
        # will be set False in run()
        self.first_start = True
```

The **unload** function is responsible for the removal of the plugin item and icon from QGIS. It basically removes the plugin when one decides to unload it / deactivate it.

This too is automatically provided by Plugin Builder.

```
def unload(self):
        """Removes the plugin menu item and icon from QGIS GUI."""
        for action in self.actions:
            self.iface.removePluginMenu(
                self.tr(u'&Supervised Classification'),
                action)
            self.iface.removeToolBarIcon(action)
```

The **selectClassifiedFile** function opens the file dialog, allowing the user to select a file where the output will be saved after the classification process.

More information can be found below the code

```
def selectClassifiedFile(self):
        filename, _filter = QFileDialog.getSaveFileName(self.dlg,
"Select output file ","", '*.tiff')
        self.dlg.leClassName.setText(filename)
```

QFileDialog.getSaveFileName opens the file dialog to save the file.

The parameters called are the following:

- *self.dlg* is the parent widget for the file dialog, which ensures that it stay on top of the plugin's main dialog
- *Select output file* is the title of the dialog
- "" is the empty string which is given as the initial directory for the file dialog. It allows to navigate any directory.
- *.tiff* restricts the selectable items to only TIFF ones.

The method retrns a tuple, which contains the selected *filename* and *filter*. Since the latter is not used in the code, it is assigned to a variable *_filter* which is ignored in the code.

filename is then set as the text of *leClassName* in the plugin dialog - *self.dlg*, which is used to display the chosen path.

The **selectModelFile** function opens a file dialog, enables the user to select the file where to save the trained model and updates the line_edit widget with the chosen file path.

```
def selectModelFile(self):
    modelname, _filter = QFileDialog.getSaveFileName(self.dlg, "Select
output file ", "", '*.joblib')
    self.dlg.leModelName.setText(modelname)
```

QFileDialog.getSaveFileName opens the file dialog to save the file.

The parameters called are the following:

- *self.dlg* is the parent widget for the file dialog, which ensures that it stay on top of the plugin's main dialog
- *Select output file* is the title of the dialog
- *""* is the empty string which is given as the initial directory for the file dialog. It allows to navigate any directory.
- *.joblib* restricts the selectable items to only .joblib ones.

The method returns a tuple, which contains the selected *modelname* and *filter*. Since the latter is not used in the code, it is assigned to a variable *_filter* which is ignored in the code.

modelname is then set as the text of *leModelName* in the plugin dialog - *self.dlg*, which is used to display the chosen path.

The **selectExistentModel** opens a file dialog, enables the user to select an existing model file and then updates the line_edit with the chosen file path.

```
    def selectExistentModel(self):
        modelname, _filter = QFileDialog.getOpenFileName(self.dlg,
"Open Image","", "*.joblib")
        self.dlg.leModelNameEx.setText(modelname)
```

QFileDialog.getOpenFileName opens the file dialog to save the file.

The parameters called are the following:

- *self.dlg* is the parent widget for the file dialog, which ensures that it stay on top of the plugin's main dialog
- *Open Image* is the title of the dialog
- *""* is the empty string which is given as the initial directory for the file dialog. It allows to navigate any directory.
- *.joblib* restricts the selectable items to only .joblib ones.

The method returns a tuple, which contains the selected *modelname* and *filter*. Since the latter is not used in the code, it is assigned to a variable *_filter* which is ignored in the code.

modelname is then set as the text of *leModelNameEx* in the plugin dialog - *self.dlg*, which is used to display the chosen path.

The `changeparameters()` function updates the value of a line_edit widget, based on the selected classifier option in the plugin's dialog.

Basically, depending on which method is selected, its default parameter values are shown, through a series of `elif` statements.

These default parameters can be found on the online documentation of scikit-learn.

```
def changeparameters(self):
        if self.dlg.cSVM.isChecked():

self.dlg.leParameters.setText("l2;squared_hinge;False;0.0001;1.0;ovr;T
rue;1;None;0;None;1000")
        elif self.dlg.cGaussianNB.isChecked():
            self.dlg.leParameters.setText("None;1e-9")
        elif self.dlg.cNearestN.isChecked():

self.dlg.leParameters.setText("5;uniform;auto;30;2;minkowski;None;None
")
        elif self.dlg.cNeuralNetwork.isChecked():

self.dlg.leParameters.setText("100;relu;adam;0.0001;auto;constant;0.00
1;0.5;200;True;None;0.0001;False;False;0.9;True;False;0.1;0.9;0.999;1e
-08;10;15000")
        elif self.dlg.cLogReg.isChecked():

self.dlg.leParameters.setText("l2;False;0.0001;1.0;True;1;None;None;lb
fgs;100;auto;0;False;None;None")
        elif self.dlg.cRandForest.isChecked():

self.dlg.leParameters.setText("100;gini;None;2;1;0.0;sqrt;None;0.0;Tru
e;False;None;None;0;False;None;0.0;None")
        elif self.dlg.cKmeans.isChecked():
            self.dlg.leParameters.setText("8;k-means+
+;100;1024;0;True;None;0.0;10;None;auto;0.01")
```

The `opendocumentationpage()` function enables the user to open the documentation page of the selected machine learning method.

It checks if a method is clicked and sets its documentation page web address to the variable url, which is later opened with `webbrowser.open`.

The argument `new=0` opens the new page in the existing browser window, and `autoraise=True` specifies that the URL should be brought to the front, if possible.

```
    def opendocumentationpage(self):
        if self.dlg.cGaussianNB.isChecked():
            url =
"https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes
.GaussianNB.html#sklearn.naive_bayes.GaussianNB"
        elif self.dlg.cSVM.isChecked():
            url
="https://scikit-learn.org/stable/modules/generated/sklearn.svm.Linear
SVC.html#sklearn.svm.LinearSVC"
        elif self.dlg.cLogReg.isChecked():
            url
="https://scikit-learn.org/stable/modules/generated/sklearn.linear_mod
```

```
el.LogisticRegression.html#sklearn.linear_model.LogisticRegression"
        elif self.dlg.cNearestN.isChecked():
            url
="https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.
KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier"
        elif self.dlg.cRandForest.isChecked():
            url
="https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.R
andomForestClassifier.html#sklearn.ensemble.RandomForestClassifier"
        elif self.dlg.cNeuralNetwork.isChecked():
            url
="https://scikit-learn.org/stable/modules/generated/sklearn.neural_net
work.MLPClassifier.html#sklearn.neural_network.MLPClassifier"
        elif self.dlg.cKmeans.isChecked():
            url =
"https://scikit-learn.org/stable/modules/generated/sklearn.cluster.Min
iBatchKMeans.html#sklearn.cluster.MiniBatchKMeans"
        webbrowser.open(url, new=0, autoraise=True)
```

The `cleantraindata` function cleans the training data, removing certain specific values.

The method takes four arguments:

- `arrX`: training input features as numpy array
- `arrY`: training target values as numpy array
- `trainnband`: humber of bands/features in the training data
- `nval`: training target value, to be excluded from training data.

What happens is the following:

1. `arrX` is reshaped, or flattened, preserving the number of bands as the first dimension.
2. `arrY` is reshaped to be a single column array
3. `arrX` and `arrY` are stacked together horizontally, to create a new array `t`. `t` has input features and corresponding target values side by side.
4. We filter out rows of `t` where the target is equal to `nval`.
5. The new filtered array `t` is transformed into an `int` type.
6. `t` is split back into `arrX` and `arrY`.
7. The function returns the newly cleaned `arrX` and `arrY`.

```
    def cleantraindata(self,arrX,arrY,trainnband,nval):
        arrX = np.transpose(arrX.reshape(trainnband,-1))
        arrY = arrY.reshape(-1,1)
        t = np.hstack((arrX,arrY))
        t = t[t[:,-1] != nval]
        t = t.astype(int)
        arrX,arrY = np.hsplit(t,[trainnband])
        return arrX,arrY
```

The `splitclass()` function takes the input testing data and classifies it using the provided model (`model`). It splits the input into chunks - or tiles - and performs the classification by using the `model.predict()` method.

The output is then appended to the `test_output` array.

If the `cbRuleImage` option is checked, the functions also calculates for each tile the rule image. The output of this depends by the classification model used. For example, it uses `mode.decision_function()` and `model.transform()` for SVM and K-Means, while it uses `model.predict_proba()` for the other methods. The output is stored into `rule_output`.

The progress then gets updated and this value is returned along with `test_output` and `rule_output` and the signal is pushed to the ProgressBar.

```
def
splitclass(self,test_input,nc,model,totimp,gx,gy,p,nodatax,nodatay,par
):

        # classification
        if self.dlg.cSVM.isChecked():
            rule_output = np.empty((0,len(model.classes_)), dtype =
np.float32)
        if self.dlg.cKmeans.isChecked():
            rule_output = np.empty((0,int(par)), dtype = np.float32)
        else:
            rule_output = np.empty((0,len(model.classes_)), dtype =
np.uint16)

        nodataindex = np.where(np.all(test_input == nodatax, axis=1))
        x =
np.asarray(np.delete(np.linspace(0,test_input.shape[0],nc+1),
[0,nc]),int)
        test_output = np.empty(0,dtype=int)
        totClassifiedChunks=0

        #list of chunks
        y = np.split(test_input,x)
        for i in range(nc):
            chunkoutput = model.predict(y[i])
            totClassifiedChunks+=1
            print('Classified chunks: ',
totimp*nc+totClassifiedChunks,'/',gx*gy*nc,sep='', end='\r')
            test_output = np.hstack((test_output,chunkoutput))
            #rule image
            if self.dlg.cbRuleImage.isChecked():
                if self.dlg.cSVM.isChecked():
                    rulechunk =
model.decision_function(y[i]).astype(np.float32)
                elif self.dlg.cKmeans.isChecked():
```

```
                    rulechunk =
model.transform(y[i]).astype(np.uint16)
                else:
                    rulechunk = model.predict_proba(y[i])
                    rulechunk = (rulechunk*10000).astype(np.uint16)
                rule_output = np.vstack((rule_output,rulechunk))

            #updates progress
            p = p + (38/(gx*gy*nc))
            self.dlg.progressBar.setValue(int(p))

        test_output = np.ravel(test_output)
        test_output[nodataindex] = nodatay
        return test_output, rule_output, p
```

Now we enter into the main function of this plugin, `supervisedclass()`.

In this function all the previous functions are called and the real processing work happens.

First of all, we need to pass to `supervisedclass` the argument `my_msg_bar` in order to be able to see the different error or status messages that being pushed in the code, directly in the GUI mask.

```
def supervisedclass(self, my_msg_bar):
```

The function then clears the log file, and sets the progressBar back to 0. This way, it is possible to start again from scratch with the new classification.

```
        self.dlg.ptLog.clear()
        prog = 0
        self.dlg.progressBar.setValue(prog)
```

We then set the starting time, which gets appended to the Log.

```
        # Current time
        string = "Started at: " + datetime.now().strftime("%H:%M:%S")
        self.dlg.ptLog.appendPlainText(string)
        startTime = time.time()
```

We import the variables from the GUI, specifically the null value and the output file name.

```
        # Import variables from GUI
        nullval = self.dlg.sbNullValue.value() #null value of training
image
        filename = self.dlg.leClassName.text() #output name
```

If the input image list is empty, or the output classification image name is empty, we push a critical error into the GUI.

```
        if len(self.dlg.TestInputList.checkedItems()) == 0:
            my_msg_bar.pushMessage("Error", "Missing input image
bands.", Qgis.Critical)
```

```
            return
        if len(filename) == 0:
            my_msg_bar.pushMessage("Error", "Missing output
classification image name.", Qgis.Critical)
            return
```

We get the number of rows and columns from the GUI (if they were selected), otherwise we keep them at value 1. We also check that the number chosen is positive, else we push a message error.

The same thing is done for the classification tiles (chunks).

```
        # nColumns e nRows variables
        if self.dlg.cImageGrid.isChecked():
            gridX = self.dlg.sbGridX.value()
            gridY = self.dlg.sbGridY.value()
        else:
            gridX = 1
            gridY = 1
        if gridX <= 0 or gridY <=0:
            my_msg_bar.pushMessage("Error", "The number of rows and
columns must be larger than or equal to 1.", Qgis.Critical)
            return

        # Chunks variables
        if self.dlg.cClassChunks.isChecked():
            nchunks = self.dlg.sbChunks.value()
        else:
            nchunks = 1
        if nchunks <=0:
            my_msg_bar.pushMessage("Error", "The number of tiles must
be larger than or equal to 1.", Qgis.Critical)
            return
```

This segment deals with opening the image bands selected in the GUI, in the `TestInputList` part.

It iterates over these and performs the following operations:

1. retrieves the layer by name, and maps it in the QGIS project
2. uses `gdal.Open()` to open the layer and appends it to the `openLayers` list.
3. checks if it is the first band that is being processed. If so, it creates a new copy of it and assigns it to `dataset`, while also retrieving the size of the raster.
4. updates the progress value for the progressBar
5. updates the Log

Some additional comments are added in the code below.

```
        # Open image bands
```

```
        #creates a list to store the opened image bands
        openLayers = []
        #flag that indicates the first band
        firstband = True

        #iterates over the checked items from the list
        for testband in self.dlg.TestInputList.checkedItems():

            #gets the layer by name
            layer = QgsProject.instance().mapLayersByName(testband)
            layer = layer[0].dataProvider().dataSourceUri()

            #uses gdal to open the layer and adds it to openLayers
            openLayers.append(gdal.Open(layer,0))

            if firstband == True:

                #creates a new copy of the first band
                driver = gdal.GetDriverByName('GTiff')
                dataset = driver.CreateCopy(filename,openLayers[0])

                #gets the sixe of the first band's raster
                xSize = openLayers[0].RasterXSize
                ySize = openLayers[0].RasterYSize

                #sets the flag to false
                firstband = False


            prog = prog +
(10/(len(self.dlg.TestInputList.checkedItems())))
            self.dlg.progressBar.setValue(int(prog))

        string = 'Input image band: ' +
str(self.dlg.TestInputList.checkedItems())
        self.dlg.ptLog.appendPlainText(string)
```

Here we define some variables related to no-data values/tile size

1. `nodataval` - creates an array of no-data values with the same length as the number of image bands being processed
2. `nodataclass` - stores the value of the no-data class
3. `xStep` and `yStep` - represent the height/width, calculated using the raster size by # of tiles.
4. `xRes` and `yRes` - the residual height/width after the division into tiles. They are those that don't perfectly fit into the division.
5. `totImportedChunks` - initialized variable to keep track of the imported tiles.

```
        # No data values
        nodataval = np.repeat(self.dlg.sbNoDataValue.value(),
len(openLayers))  # Array of no data values for each band
        nodataclass = self.dlg.sbNoDataClass.value()  # No data class
value

        xStep = int(xSize / gridX)  # Width of each chunk
        yStep = int(ySize / gridY)  # Height of each chunk
        xRes = xSize - xStep * gridX  # Residual width after dividing
the raster into chunks
        yRes = ySize - yStep * gridY  # Residual height after dividing
the raster into chunks

        totImportedChunks = 0  # Counter to keep track of the total
imported chunks
```

This part handles the case if which an existing classification model is chosen.

It checks if the checkbox was clicked, then retrives the model entered. It then checks if the model name is empty and, if so, pushes an error message.

Afterwards it loads the model using the `load` function and is assigned to the variable `mod`.

It checks whether the number of image bands corresponds to the number of features expected. If not, it pushes an error message. If they do, the code displays in the log a success message.

```
        # Use an existing model
        if self.dlg.cbUseModel.isChecked():
            modelname = self.dlg.leModelNameEx.text()
            if len(modelname) == 0:
                my_msg_bar.pushMessage("Error", "Missing
classification model name", Qgis.Critical)
                return
            # Load model and check
            mod = load(modelname)
            if len(self.dlg.TestInputList.checkedItems()) !=
mod.n_features_in_:
                message = "The loaded model expects {} features, but
the image to be classified has{}".format(mod.n_features_in_,
len(self.dlg.TestInputList.checkedItems()))
                my_msg_bar.pushMessage("Error", "message",
Qgis.Critical)
                return
            else:
                string = "Model successfully loaded: " + modelname
                self.dlg.ptLog.appendPlainText(string)
```

The following code, however, is executed only when the user chooses to create a model from the uploaded training dataset.

It records the starting time of the training process, initializes `algorithm` to `None`, retrieves the user entered parameters (or the automatically preset ones) and creates an empty array (`trainx`) to store the training input data in. `trainy` is the array where the training output data is stored. It has shape (0,1) to store the class labels.

```
        # Create Model from training dataset
        else:
            startTimeTrain = time.time()
            algorithm = None
            # Parameters list
            parameters = self.dlg.leParameters.text().split(';')
            # Create empty arrays for training
            trainx = np.empty((0, len(openLayers)), dtype=int)
            trainy = np.empty((0, 1), dtype=int)
```

The following code takes into account the latest addition of an unsupervised method to the plugin (previously built to only contain supervised methods).

After checking that KMeans is indeed clicked, it assigns the name to the variable algname.

To handle parameters, it does as follows: it checks if there are some parameters set to None and converts them to the actual `None`, while checking if `parameters[10]` is set to neither auto nor warn. It so, it converts the value to an integer.

The algorithm then creates an instance of the selected `MiniBarchKMeans`, which can fit the model in tiles. It iterates through each tile, creates an empty `testBandsChunks` list to store the data in, and determines the window size based on the current chunk's position.

Afterwards, it loops over all the image bands, appending the corresponding tile to `testBandsChunk`. The data gets stacked into `testChunkX` and then reshaped, to match the expected format. The tile is eventually passed to `algorithm.partial_fit()` which updates the model with the current tile. The progressBar gets updated accordingly.

After the training is done, we calculate the training time, which get shown into the Log, together with other information about the algorithm/parameters.

```
        # Training for K-means
        if self.dlg.cKmeans.isChecked():
            algname = 'K-Means'
            # Parameters
            if parameters[6] == 'None':
                parameters[6] = None
            else:
                parameters[6] = int(parameters[6])
            if parameters[9] == 'None':
                parameters[9] = None
            else:
                parameters[9] = int(parameters[9])
            if parameters[10] != 'auto' and parameters[10] != 'warn':
                parameters[10] = int(parameters[10])
```

```
            algorithm = MiniBatchKMeans(n_clusters=int(parameters[0]),
init=parameters[1], max_iter=int(parameters[2]),
batch_size=int(parameters[3]), verbose=int(parameters[4]),
compute_labels=eval(parameters[5]), random_state=parameters[6],
tol=float(parameters[7]), max_no_improvement=int(parameters[8]),
init_size=parameters[9], n_init=parameters[10],
reassignment_ratio=float(parameters[11]))
            # Fitting the model in chunks
            for j in range(gridY):
                for i in range(gridX):
                    testBandsChunk = []
                    if i == (gridX - 1) and j == (gridY - 1):
                        winX = xRes + xStep
                        winY = yRes + yStep
                    elif j == (gridY - 1):
                        winX = xStep
                        winY = yRes + yStep
                    elif i == (gridX - 1):
                        winX = xRes + xStep
                        winY = yStep
                    else:
                        winX = xStep
                        winY = yStep
                    for k in range(len(openLayers)):

testBandsChunk.append(openLayers[k].ReadAsArray(xStep * i, yStep * j,
winX, winY))
                    testChunkX = np.stack(testBandsChunk)
                    testChunkX =
np.transpose(testChunkX.reshape(testChunkX.shape[0], -1))
                    mod = algorithm.partial_fit(testChunkX)
                    prog = prog + (40 / (gridX * gridY))
                    self.dlg.progressBar.setValue(int(prog))

            trainTime = time.time() - startTimeTrain
            string = "Training completed in: " + str(int(trainTime /
60)) + ":" + str(int(trainTime % 60)) + " minutes"
            self.dlg.ptLog.appendPlainText(string)
            string = 'Algorithm: ' + algname + ', parameters: ' +
str(self.dlg.leParameters.text())
            self.dlg.ptLog.appendPlainText(string)
            my_msg_bar.pushMessage("Status", "Training completed.",
Qgis.Info)
            prog = 50
            self.dlg.progressBar.setValue(prog)
```

This code handles the case where the user chooses to create a model from the training dataset.

At first, it checks if the training output is selected. If it is not, an error message is displayed, and the execution terminated. If it is selected, it proceeds to open the training output layer using `gdal.Open()`.

Secondly, it populates the training arrays by iterating over each tiles, and storing the data of the current chunk into testBandsChunk. The code then determines the window size based on the current tile position, iterates over the image bands and appends the corresponding tile to testBandsChunk.

The data from the input image and the output layers is read as an array with `ReadAnArray()`.

We then call `cleantraindata()` to remove null values, and then we append the data tile to `trainx` and `trainy.

Finally, we create the algorithm using the name and parameters indicated in the GUI. This is only shown for the Gaussian Naive-Bayes method, but it is pretty much the same for all the remaining, which are thusly not shown.

In this case:

- it checks if the algorithm is selected by the use (`algorithm.isChecked()`)
- if so, it sets the algname to the algorithm name
- it check the parameters value for priors, and if there are any None, it converts them into `None`.
- at last, an instance of the algorithm is created with the algorithm name and specified parameters.

```
else:
    # Check if training output is selected by the user
    if self.dlg.TrainOutput.currentLayer() is None:
        my_msg_bar.pushMessage("Error", "Missing output classification
image.", Qgis.Critical)
        return
    # Open training output
    layer =
self.dlg.TrainOutput.currentLayer().dataProvider().dataSourceUri()
    opentrainy = gdal.Open(layer)
    string = 'Input training data: ' +
str(self.dlg.TrainOutput.currentLayer())
    self.dlg.ptLog.appendPlainText(string)

    # Populate training arrays removing null values
    for j in range(gridY):
        for i in range(gridX):
            testBandsChunk = []
            if i == (gridX - 1) and j == (gridY - 1):
                winX = xRes + xStep
                winY = yRes + yStep
            elif j == (gridY - 1):
```

```
                winX = xStep
                winY = yRes + yStep
            elif i == (gridX - 1):
                winX = xRes + xStep
                winY = yStep
            else:
                winX = xStep
                winY = yStep
            for k in range(len(openLayers)):
                testBandsChunk.append(openLayers[k].ReadAsArray(xStep
* i, yStep * j, winX, winY))
            testChunkX = np.stack(testBandsChunk)
            testChunkY = opentrainy.ReadAsArray(xStep * i, yStep * j,
winX, winY)
            testChunkX, testChunkY = self.cleantraindata(testChunkX,
testChunkY, testChunkX.shape[0], nullval)
            trainx = np.vstack((trainx, testChunkX))
            trainy = np.vstack((trainy, testChunkY))
            prog = prog + (20 / (gridX * gridY))
            self.dlg.progressBar.setValue(int(prog))

    # Create an algorithm with the parameters
    # Gaussian Naive-Bayes
    if self.dlg.cGaussianNB.isChecked():
        algname = 'Gaussian Naive-Bayes'
        if parameters[0] == 'None':
            parameters[0] = None
        algorithm = GaussianNB(priors=parameters[0],
var_smoothing=float(parameters[1]))
```

This code segment performs the training process of the model.

It firstly prepares the training data, flattening it with `np.ravel()`.

Secondly, it fits the model by passing it `trainx` and `trainy` as input training data. The trained model is stored in `mod`.

Afterwards, we log the time that it has taken and we update the progressBar as well.

```
# Fit the model
trainy = np.ravel(trainy)
mod = algorithm.fit(trainx, trainy)

# Training time
trainTime = time.time() - startTimeTrain
string = "Training completed in: " + str(int(trainTime / 60)) + ":" +
str(int(trainTime % 60)) + " minutes"
self.dlg.ptLog.appendPlainText(string)
string = 'Algorithm: ' + algname + ', parameters: ' +
str(self.dlg.leParameters.text())
self.dlg.ptLog.appendPlainText(string)
```

```
my_msg_bar.pushMessage("Status", "Training completed.", Qgis.Info)

prog = 50
self.dlg.progressBar.setValue(prog)
```

This segment checks the accuracy of the trained model and then performs the classification.

It firstly checks if in the GUI the accuracy option was selected, and if so it proceeds to calculate it.

Then we use the `train_test_split()` function to split the input data into training and testing subset, choosing the test_size and a random_state, which ensures the reproducibility of the split. We then assign the subsets to the X and y training and testing variables.

To calculate the model accuracy, we first fit the algorithm, predicting the class labels for the test subset. We compute the number of mislabeled points against the true labels using `(np.transpose(y_test) != y_pred).sum()`. We then compute the percentage of accuracy. We also add a log entry with the # of mislabeled points and the percentage of accuracy.

Finally, we update the progressBar and delete the data from memoryn with `del`.

```
# Check accuracy of the model and classify
if self.dlg.cAccuracy.isChecked():
    X_train, X_test, y_train, y_test = train_test_split(trainx,
trainy, test_size=0.3, random_state=0)
    y_pred = algorithm.fit(X_train, y_train).predict(X_test)
    string = "Number of mislabeled points out of a total " +
str(X_test.shape[0]) + " points: " + str((np.transpose(y_test) !=
y_pred).sum())
    self.dlg.ptLog.appendPlainText(string)
    string = "Model accuracy: " + ("{:.2f}".format(((X_test.shape[0]
- ((np.transpose(y_test) != y_pred).sum())) / X_test.shape[0]) *
100))) + " %"
    self.dlg.ptLog.appendPlainText(string)
    prog = 57
    self.dlg.progressBar.setValue(prog)
    del(X_train, X_test, y_train, y_test, y_pred)
```

This part deals with saving the model in .joblib format.

It checks if cKmeans hasn't been selected, and then check is cSaveModel has. If both are met, it retrieves the model name (if there's no model name, it pushes an error). Else, the `dump()` function is used to save the model in .joblib format. A log entry shows that the model was correctly saved.

The memory is then freed and the progress updated.

```
# Save model in .joblib format
if self.dlg.cKmeans.isChecked() != True:
    if self.dlg.cSaveModel.isChecked():
        modelname = self.dlg.leModelName.text()
        if modelname is None:
            my_msg_bar.pushMessage("Error", "Missing classification
model name.", Qgis.Critical)
            return
        else:
            dump(mod, modelname)
            string = "Model saved successfully in: " + modelname
            self.dlg.ptLog.appendPlainText(string)

# Delete training arrays
del(trainx, trainy)

prog = 60
self.dlg.progressBar.setValue(prog)
```

This part deals with the creation of the rule image .tif file.

First of all, it checks if the relevant option was selected. If so, it then checks the classification algorithm selected to determine the data type & # of bands.

If SVM is selected, the rule image data type is set to `gdal.GDT_Float32`. In all other cases it is set to `gdal.GDT_UInt16`.

While generating the rule image name and dataset, the filename is split based on the '.' character, in order for the file name to be extracted without the extension. We add as a suffix "_rule_image.tif". With driver.Create() function creates the new tif file with the specified inputs.

```
# Create .tif file for rule image
if self.dlg.cbRuleImage.isChecked():
    if self.dlg.cSVM.isChecked():
        ruletype = gdal.GDT_Float32
        nbands = len(mod.classes_)
    elif self.dlg.cKmeans.isChecked():
        ruletype = gdal.GDT_UInt16
        nbands = int(parameters[0])
    else:
        ruletype = gdal.GDT_UInt16
        nbands = len(mod.classes_)
    rulename = filename.split('.')
    rulename = rulename[0] + "_rule_image.tif"
    ruledataset = driver.Create(rulename, xsize=xSize, ysize=ySize,
bands=nbands, eType=ruletype)
```

The code segment handles the classification process and performs the following steps:

1. Two loops iterate over the grid of image chunks defined by gridX and gridY. For each chunk, the code determines the window size (winX and winY) based on its position in the grid.

2. The code initializes an empty list testBandsChunk to store the image bands for the current chunk. Another nested loop iterates over the available image bands (openLayers). The corresponding bands for the current chunk are extracted using the ReadAsArray() method and appended to testBandsChunk. The resulting bands are stacked together along the first axis to create the testChunkX array. The array is reshaped by transposing it and flattening it to have the shape (num_samples, num_features).

3. The chunk is classified using the mod model obtained from training or loaded. The splitclass() method is called to classify the chunk, generate the rule image chunk (ruleChunk), and update the progress (prog) value. The total count of imported chunks (totImportedChunks) is incremented. The classified labels (testChunkY) are reshaped to have the shape (winY, winX).

4. The classified chunk (testChunkY) is written to the output raster file (dataset) at the corresponding position (xStep*i, yStep*j) using WriteArray(). If the rule image chunk (ruleChunk) is not empty, it is reshaped to have the shape (winY, winX, num_classes). The rule image bands are iterated over (l) and each band is written to the rule image dataset (ruledataset) at the corresponding position (xStep*i, yStep*j) using WriteArray().

5. The progress value is updated to 98%. The classification time is calculated by subtracting the start time (startTimeClass) from the current time. A log entry is added with information about the classification time, the number of image pieces, the number of chunks per piece, and the total time taken for classification. Another log entry indicates the path where the classified image is saved (filename).

In summary, this code segment iterates over image chunks, reads the test data for each chunk, classifies the chunk using the model, writes the classified and rule image chunks to respective datasets, and updates the progress and logs the classification time and output file path.

```
# Classification
        startTimeClass = time.time()
        for j in range(gridY):
            for i in range(gridX):
                testBandsChunk = []
                if i==(gridX-1) and j==(gridY-1):
                    winX = xRes+xStep
                    winY = yRes+yStep
                elif j==(gridY-1):
                    winX = xStep
                    winY = yRes+yStep
                elif i==(gridX-1):
                    winX = xRes+xStep
```

```
                    winY = yStep
                else:
                    winX = xStep
                    winY = yStep
                for k in range(len(openLayers)):

testBandsChunk.append(openLayers[k].ReadAsArray(xStep*i,yStep*j,winX,w
inY))
                testChunkX = np.stack(testBandsChunk)
                testChunkX =
np.transpose(testChunkX.reshape(testChunkX.shape[0],-1))
                # Classify chunk
                testChunkY, ruleChunk, prog =
self.splitclass(testChunkX,nchunks,mod,totImportedChunks,gridX,gridY,p
rog,nodataval,nodataclass,parameters[0])
                totImportedChunks+=1
                testChunkY = testChunkY.reshape((winY,winX))
                # Write classified chunk

dataset.GetRasterBand(1).WriteArray(testChunkY,xStep*i,yStep*j)
                # Write rule image chunk
                if ruleChunk.shape[0] != 0:
                    ruleChunk =
ruleChunk.reshape((winY,winX,ruleChunk.shape[1]))
                    for l in range(ruleChunk.shape[2]):

ruledataset.GetRasterBand(l+1).WriteArray(ruleChunk[:,:,l],xStep*i,ySt
ep*j)
                    print("rule written")


        prog = 98
        self.dlg.progressBar.setValue(prog)
        # Calculate classification time
        classTime = time.time() - startTimeClass
        string = "Image classified in: " + str(int(classTime/60)) +
":" + str(int(classTime%60)) + " minutes, loading the image in " +
str(gridX*gridY) + " pieces, and classifying every piece divided in "+
str(nchunks) + " chunks"
        self.dlg.ptLog.appendPlainText(string)
        string = "Classified Image saved in: " + filename
        self.dlg.ptLog.appendPlainText(string)
```

This part save the Log messages into a text file.

At first it checks if the option is selected, then it splits the name considering the '.' character and adds _log.txt at the end of it. It then open the file, writes the Log on it and closes it.

```
# Save Log in txt file
        if self.dlg.cbSaveLog.isChecked():
```

```
            logname=filename.split('.')
            logname =logname[0]+"_log.txt"
            logfile = open(logname, "w")
            logfile.write(self.dlg.ptLog.toPlainText())
            logfile.close()
```

```
  File "<ipython-input-1-00a7807416a9>", line 2
    if self.dlg.cbSaveLog.isChecked():
    ^
IndentationError: unexpected indent
```

This part clears all the variables at the end of the execution, removing the layers that have been opened, the dataset in input. If the rule image was computed, it save in the log the name of the path. It then flushes it.

```
# Clear variables
        del(testChunkX, testChunkY)
        openLayers = None
        dataset.FlushCache()
        dataset = None
        if self.dlg.cbRuleImage.isChecked():
            string = "Rule image saved in: " + rulename
            self.dlg.ptLog.appendPlainText(string)
            ruledataset.FlushCache()
            ruledataset = None
```

This part adds the new output layer to the already present ones in QGIS, using the addRasterLayer function.

It then sets the progressBar value to 100, signaling that the whole procedure is now completed.

```
        # Add output layer in the current QGIS Project
        layerName = filename.split('/')
        layerName = layerName[-1].split('.')
        self.iface.addRasterLayer(filename,layerName[0])
        self.dlg.progressBar.setValue(100)
```

```
---------------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
<ipython-input-2-e472098efd33> in <cell line: 2>()
      1 # Add output layer in the current QGIS Project
----> 2 layerName = filename.split('/')
      3 layerName = layerName[-1].split('.')
      4 self.iface.addRasterLayer(filename,layerName[0])
      5 self.dlg.progressBar.setValue(100)

NameError: name 'filename' is not defined
```

In the `resetUI()` function, the different widgets and checkboxes in the GUI are reset, after the completion of the procedure. It also repopulates the selection widgets.

```
    def resetUI(self):
        self.dlg.TestInputList.clear()
        self.dlg.leClassName.clear()
        self.dlg.leModelName.clear()
        self.dlg.leModelNameEx.clear()
        self.dlg.cbUseModel.setChecked(False)
        self.dlg.cImageGrid.setChecked(False)
        self.dlg.cClassChunks.setChecked(False)
        self.dlg.cbCustomParameters.setChecked(False)
        self.dlg.cAccuracy.setChecked(False)
        self.dlg.cSaveModel.setChecked(False)
        self.dlg.cbSaveLog.setChecked(False)
        self.dlg.cbRuleImage.setChecked(False)
        self.dlg.cGaussianNB.setChecked(True)
        self.dlg.sbGridX.setValue(1)
        self.dlg.sbGridY.setValue(1)
        self.dlg.sbChunks.setValue(1)
        self.dlg.sbNullValue.setValue(-1)
        self.dlg.sbNoDataValue.setValue(0)
        self.dlg.sbNoDataClass.setValue(0)
        # Populate the comboBox with names of all the loaded layers
        layers = QgsProject.instance().layerTreeRoot().children()
        self.dlg.TestInputList.addItems([layer.name() for layer in
layers])
        self.dlg.leParameters.setText("None;1e-9")
        self.dlg.ptLog.clear()
        self.dlg.progressBar.setValue(0)
```

Lastly, the run() function is the one function that recalls all the others, performing all the work.

It creates the dialog with its elements.

If it is the dialog first time being started, it sets it to false and connects all the various buttons and widgets.

It fetches the layers present in QGIS and populates the widgets with the layer names.

Here the messageBar is also implemented as an element of the GUI, and it is then passed as an argument to`supervisedclass()`.

If OK was clicked, the whole process starts, as the user has selected in the GUI.

```
    def run(self):
        """Run method that performs all the real work"""
        # Create the dialog with elements (after translation) and keep
reference
        # Only create GUI ONCE in callback, so that it will only load
```

```
when the plugin is started
        if self.first_start == True:
            self.first_start = False
            self.dlg = SupervisedClassificationDialog()

self.dlg.pbBrowseClass.clicked.connect(self.selectClassifiedFile)
            self.dlg.pbModelName.clicked.connect(self.selectModelFile)

self.dlg.pbBrowseModel.clicked.connect(self.selectExistentModel)
            self.dlg.cSVM.clicked.connect(self.changeparameters)

self.dlg.cGaussianNB.clicked.connect(self.changeparameters)
            self.dlg.cLogReg.clicked.connect(self.changeparameters)
            self.dlg.cNearestN.clicked.connect(self.changeparameters)

self.dlg.cNeuralNetwork.clicked.connect(self.changeparameters)

self.dlg.cRandForest.clicked.connect(self.changeparameters)
            self.dlg.cKmeans.clicked.connect(self.changeparameters)

self.dlg.pbHelp.clicked.connect(self.opendocumentationpage)
            self.dlg.buttonBox.accepted.disconnect()
            self.dlg.buttonBox.accepted.connect(self.run)
            self.dlg.pbReset.clicked.connect(self.resetUI)
            # Fetch the currently loaded layers
            layers = QgsProject.instance().layerTreeRoot().children()
            # Clear the contents of the comboBox and Lists from
previous runs
            self.dlg.TestInputList.clear()
            self.dlg.leClassName.clear()
            self.dlg.leModelName.clear()
            self.dlg.leModelNameEx.clear()
            # Populate the comboBox with names of all the loaded
layers
            self.dlg.TestInputList.addItems([layer.name() for layer in
layers])
            self.dlg.leParameters.setText("None;1e-9")

        #insert message bar in layout
        current_layout = self.dlg.layout()

        #create a container widget
        widget = QWidget(self.dlg)
        #current_layout.addWidget(widget)

        #creates message bar
        my_msg_bar = QgsMessageBar(self.dlg)
        #my_msg_bar = QgsMessageBar(widget)

        #sets size policy for message bar
```

```
        size_policy = QSizePolicy(QSizePolicy.Expanding,
QSizePolicy.Fixed)
        my_msg_bar.setSizePolicy(size_policy)

        #add message bar to container widget
        current_layout.addWidget(my_msg_bar)
        # show the dialog
        self.dlg.show()
        # Run the dialog event loop
        result = self.dlg.exec_()

        # See if OK was pressed
        if result:
            self.supervisedclass(my_msg_bar)
```