# DT0139
# Design tip

## Decision tree generation

| Main components | | | |
|---|---|---|---|
| **Part Number** | **Market** | **Family** | **Machine Learning Core (MLC) capability** |
| LSM6DSOX | Consumer | iNEMO inertial modules (IMU) | 256 Nodes |
| LSM6DSO32X | Consumer | iNEMO inertial modules (IMU) | 256 Nodes |
| LSM6DSRX | Consumer | iNEMO inertial modules (IMU) | 512 Nodes |
| ISM330DHCX | Industrial | iNEMO inertial modules (IMU) | 512 Nodes |
| IIS2ICLX | Industrial | 2-axis accelerometer | 512 Nodes |

## Purpose and benefits

Machine learning is a field of study in computer algorithms where algorithms can be built using data without the need to explicitly derive the mathematical model. There are four different types of machine learning approaches that are widely used:

- Supervised learning
- Unsupervised learning
- Semi-supervised learning
- Reinforcement learning

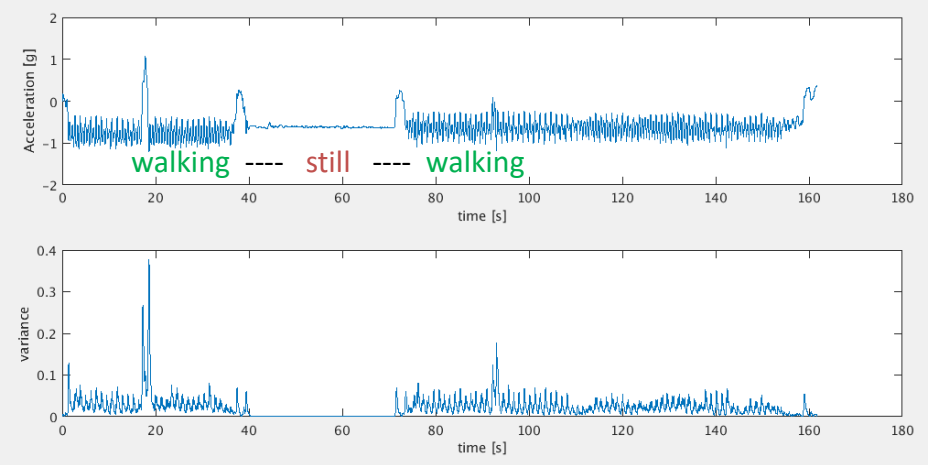
Supervised learning uses training data for machine learning algorithms that have been tagged (or labeled) with relevant real-world information. An example is accelerometer data that can be collected from a data collection device mounted on the wrist of a test subject and this information can be labeled for different activities such as stationary, walking, running, swimming, or biking. Unsupervised machine learning algorithms do not require labeled data and can create the model by determining patterns in the data. Semi-supervised learning uses a mix of labeled and unlabeled data to train and improve accuracy. Reinforcement learning algorithms derive and improve the model through interaction with the real world. More details about machine learning algorithms can be found at Machine learning.

## Description

In this document, we will discuss one supervised learning algorithm. The latest generation of ST MEMS sensors features a built-in machine learning core (MLC) based on decision tree classifiers. These products are easily recognizable with an X in the suffix (e.g. LSM6DSO**X**).This MLC can execute a programmed decision tree in the sensor with extremely low power consumption. More details about the machine learning core in these devices can be found in the related application notes (LSM6DSOX with AN5259, LSM6DSO32X with AN5656, LSM6DSRX with AN5393, ISM330DHCX with AN5392, IIS2ICLX with AN5536).

A sensor with a Machine Learning Core can process the sensor data and compute 12 different types of features on a window of data, such as mean, variance, energy in the band, peak-to-peak value, zero-crossings (positive and negative), peak detection (positive and negative) on raw and filtered sensor data. If distinguishing patterns are observed in the selected features, then a decision tree can be used to perform classification. Consider the example of acceleration data shown in Figure 1.

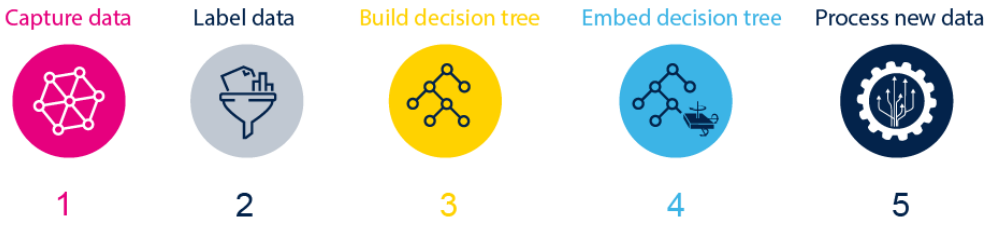
**Figure 1. Acceleration data and its variance**




The plot of variance of the acceleration norm (the feature in this case) over a specific window length is shown in Figure 1 above. It is evident from the second sub-plot that the variance feature shows distinctly different values between the two classes of “walking” and “still.” In this case, we can design a decision tree with variance as one of the features to distinguish these two different types of motion.

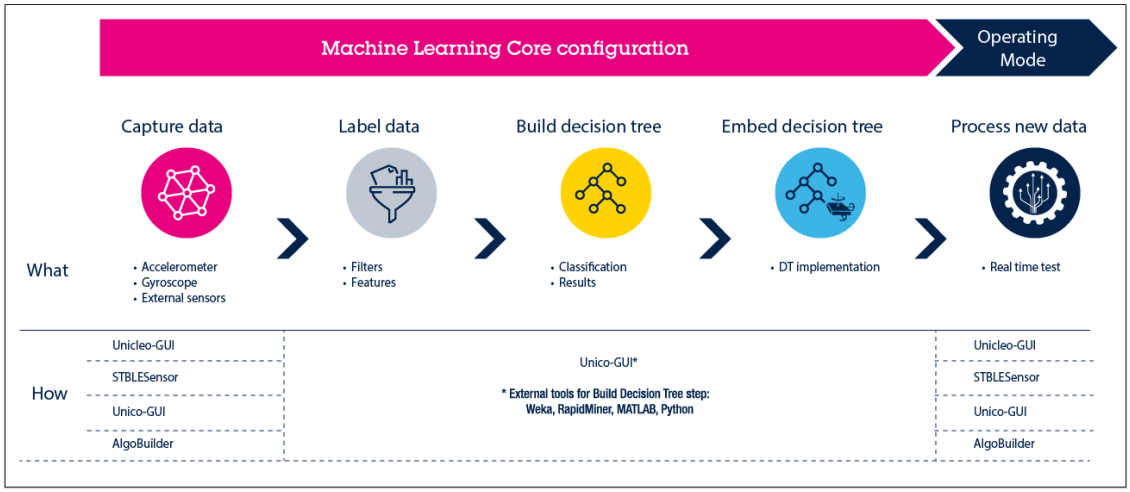
The usual flow for developing a machine learning classification algorithm is shown in Figure 2.

**Figure 2. Basic process of developing a machine learning algorithm**




Data collection and data labelling (for supervised learning) are the first two steps. Then, the model is trained using the training data (a subset of the whole data collection). The model is finally transferred and implemented in the device with MLC core in order to perform real-time validation testing using the embedded hardware.

**Figure 3. Specifying the process of implementing the Machine Learning Core**



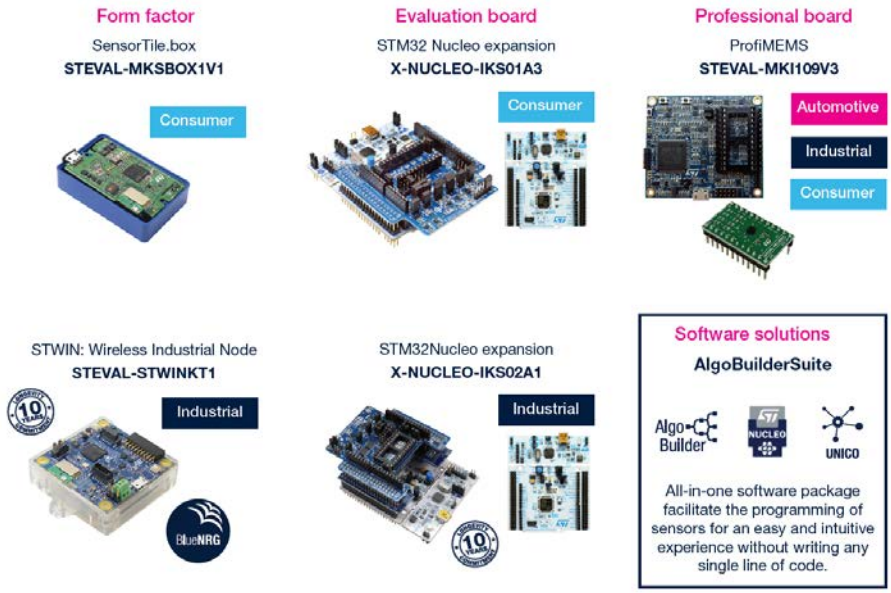

The processing steps of MLC and some of the available tools are shown in Figure 3. In this case, the first two steps can be addressed with, for example, ST Unico-GUI to collect data, label data and store each class in a specific log file. Then a decision tree model can be computed using any of the available tools, such as WEKA, RapidMiner, MATLAB, Python (e.g. sklearn) or the Unico-GUI built-in decision tree generation feature.

Data collection and pre-processing will be covered in the first section of this document. Section 2 describes how to further manage the training dataset and how to set up the system (features selection, window length definition, sensor ODR, etc.) in order to build the model. Details about decision tree generation are presented in Section 3. Methods to evaluate the model and the approaches to optimize the classification performance are presented in Section 4.

# 1. Data collection and pre-processing

The first step for any ML classification problem is to collect the data. Sensor data can be collected and labelled using various applications such as Unico-GUI on PC, ST BLE Sensor app, AlgoBuilder, FP-SNS-DATALOG1 etc., along with different hardware device options such as the ProfiMEMS board, the SensorTile.Box, the Nucleo Boards and STWIN.

**Figure 4. Development kits and GUIs for machine learning core experience**




A general guideline for a proper data collection campaign is based on the following points:

- **Class definition.** The classification problem must be defined as along with classes to be recognized. In addition, it should be clarified what happens when an input does not belong to any of the defined classes. In this case, think about the inclusion of an “other/idle” class to add to the class list.
- **Sensor.** Some of the classification problems can be solved by using an accelerometer signal, some with a gyroscope signal, and some require both sensor inputs. It is recommended to enable logging from both sensors to investigate which configuration can provide the best accuracy in case of 6-axis inertial module products. A 3-axis magnetometer can also be accessed through the sensor hub capability in the 6-axis MLC family of sensors.
- **Sensor configuration.** In the Machine Learning Core configuration tool (Unico-GUI), all the MLC parameters can be set, including sensor rates (ODR) and full scales (FS):
  - Make sure the sensor data is collected at a sufficient ODR and bandwidth to capture the needed information. Bandwidth, which represents the maximum frequency signal captured by the system, is generally half of ODR (BW ~ ODR/2). In case when exact ODR selection is not feasible in advance, it is preferable to set a high ODR (e.g. 104 Hz) maximum bandwidth and then utilize decimation/resample methods. Consider also that the defined configuration must be maintained for the entire process, and that the ML algorithm will process the field data in the same manner.

- Full Scale (FS) should be defined in accordance with the application requirement. For example, the gyro FS can be very different if the sensor is used for vehicle or wrist applications.

The sensor configuration can be changed to optimize the solution.

- **Data consistency and balancing.** Collect sufficient data according to the application requirements and that represent the actual use case. If the application is based on a wrist device, do not include data recorded on smartphones. **It is also important to collect a similar amount of data for each class. The machine learning training algorithm will try to achieve maximum total accuracy and consequently, classes with a small amount of data will get lower weighting and the model may be biased toward classes with a large amount of data.** For example, if class ‘A’ has 90% data and class ‘B’ has 10% data, then the model assigns all the output to class ‘A’ and the accuracy is 90%.
- **Data logging format.** Data must be logged in text file (.txt) with the same convention defined in Unico-GUI (For more details, refer to the application note AN5259) if the data logging is done using some other tools.

After the first phase of Data collection, a series of operations are needed to set up the classification problem. Data should be validated and cleaned in order to avoid outlier and poor labeling. Visualizing data is a good practice for data validation, selecting features, labelling and many more.  In the next section we discuss these operations.

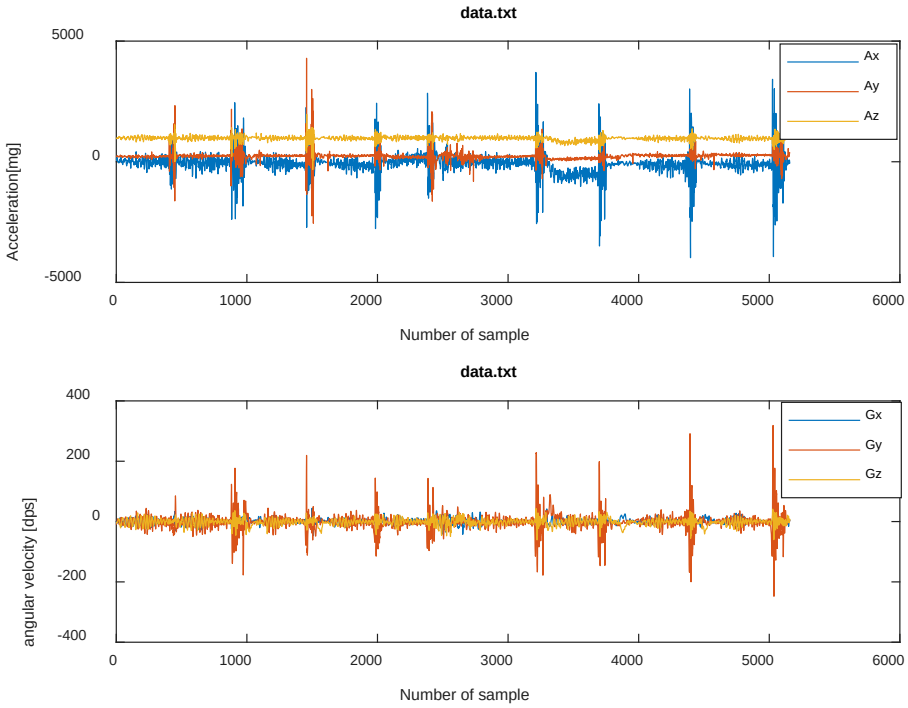
## 1.1 Data visualization

This is one of the most important steps in the development of the ML model. Data visualization can be used in data pre-processing and analysis. It can also be used in understanding the relationship between different features and variations across different classes. We can divide this process into time and frequency domains:

### 1.1.2 Analyze signal in time domain

There are several options available to visualize the collected data. If we collect accelerometer data in [m*g*] and gyroscope data in [dps], we can use MATLAB/Python/Excel/R to generate plots as shown in the following example.

**Figure 5. Visualizing data in the time domain**




The analysis of the signal in time domain allows:

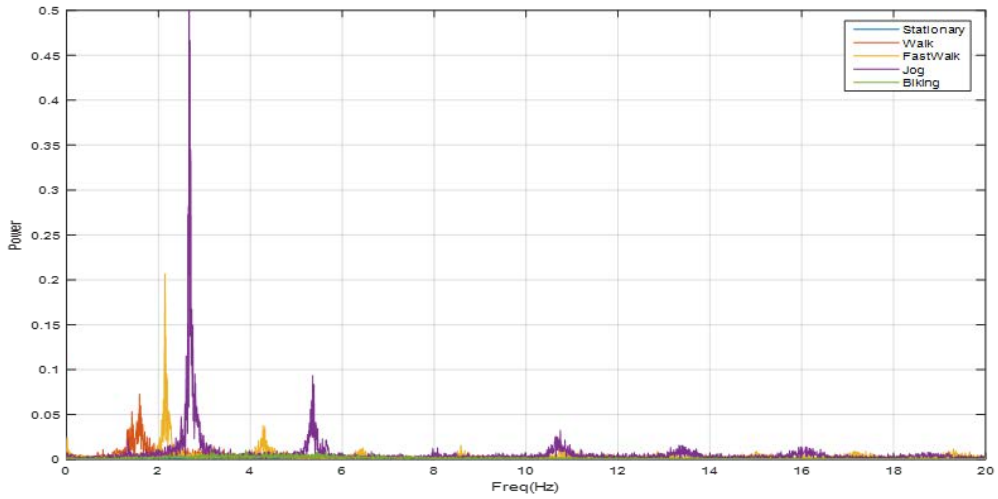
- Visually inspecting any poor quality or outliers.
- Understanding if a pattern exists between different classes.
- Selecting the informative features (mean, peak-to-peak, peak, etc.).

### 1.1.3 Analyze data in frequency domain

The frequency-domain analysis could be helpful to complement the time domain analysis. The same tools (MATLAB/Python/Excel/R) can be used to compute and produce FFT or spectrogram plots. Such plots can provide insights about applying filters on the signal before it is processed to compute features. Or they can be useful to evaluate if a specific recorded log is correctly referenced. For example, think about a data log for a “walking” use case. With an FFT plot is possible to verify the dominant frequency associated with the processed signal. If this frequency is too high (e.g. 2.5 Hz), we can assume that the log should be referenced as “running” instead of “walking”.

Figure 6 shows the frequency spectrum for different human activities (walking, fast walking, jogging, biking) considered in a classification problem.

**Figure 6. Frequency domain analysis of different activities**




From the plot shown in Figure 6, this information is derived. Each activity has a distinct dominating frequency component, thus appropriate filters can be applied to extract informative signals for each class. For example, if we apply a low-pass filter at 1.8 Hz, we can separate the walking class from fast walking and jogging classes characterized by higher energy at higher frequencies.

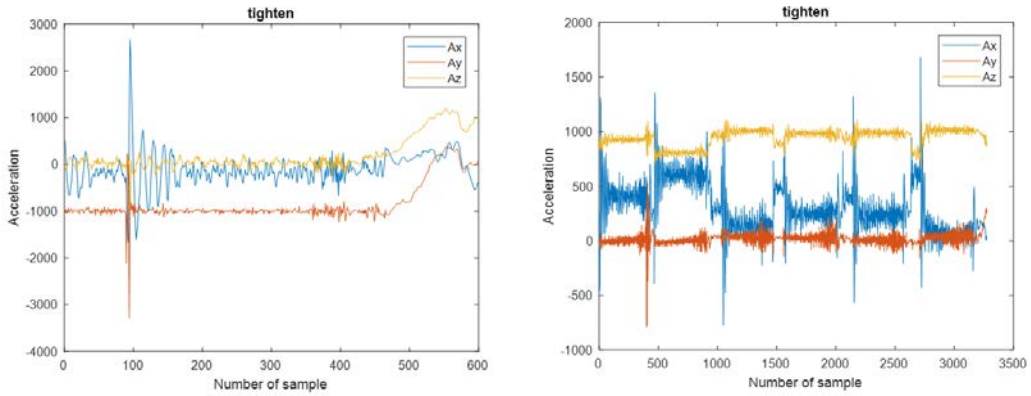
## 1.2 Data cleaning

Labeling of sensor data is the starting point for machine learning algorithm development. The best practice is to conduct sanity checks for validating the collected data before starting the ML training exercise. Data cleaning is the process of fixing or dropping corrupted, incorrectly formatted, or incomplete data. Users can easily visualize and check, for instance, mean, variance, minimum, and maximum values. If a computed feature is significantly different from logs in the same labeled class (but they are supposed to be similar in that application), then labelled data may be not clean.

In most projects, we know and collect the ground truth when we are recording data. Usually is possible to start and stop the log with only a single class registered. But it might happen that different classes are included in a single recorded log file, for various reasons. In such cases, data should be truncated to have one single class per one single file before the log file is considered ready for ML training.

A common mistake that users usually make is to collect sensor data logs with multiple classes in the same file. In supervised learning, the data cleaning is really important because any data which does not belong to the corresponding class will affect the trained model performance. For example, the following plots (Figure 7) show data from the accelerometer fixed on a drilling machine when collecting data for the “tighten” class. Here the objective is to collect labelled data for classes: idle, tighten, loosen, and drill.

**Figure 7. Example of data collected from drilling machine**




The left-hand side plot shows that the logging session does not stop when the drilling machine motion stops. There is redundant data at the end of the plot when the drilling motion is not there. The right-hand side plot shows that the log contains data for tightening multiple screws and different sessions are logged in the same file. The orientation of the drilling machine is varying, Ax, Az are changed in the [500, 1000] second interval. All these issues could affect the generated decision tree and the performance of the MLC result in real time. A few suggestions:

- The units of labelled data should be consistent in each file. We use [m*g*] for acceleration and [dps] for angular velocity.
- Each session of data collection must be truncated for the proper period.
- Separate data based on the class, using any available tool such as Python, MATLAB, R
- Remove data which are not part of any class or use it for an “other/idle” class.
- Correct scaling if needed.

# 2. Window size and Feature selection

After the completion of pre-processing of collected data, the next step is to decide the window length and start the process of feature selection.

## 2.1 Window size

Window length is strongly dependent on the application. We should make sure the window can capture all the information for the decision tree to separate different classes. A short window length could make the response quick since the prediction is made for each window length of data. However, it could also make the precision lower if the window length is too small to capture enough information. The size of the window should be carefully selected by the user after data analysis and visualization. If the motion is repetitive at some frequency, the ideal window size should be long enough to capture the entire motion pattern. Please also consider the slowest varying signal. For example, if we want to distinguish between class A which has a periodicity 0.9 Hz and class B which has periodicity 1.9 Hz, we should decide the window size based on class A and set window size to be (1/0.9 Hz) ≈ 1.1 s or longer.

## 2.2 Feature selection

### 2.2.1 Observation

We can select the informative and relevant features by observing the motion. Consider the example, where we are going to detect the head gestures, such as “nod” when the sensors are mounted on the headset with the y-axis pointing toward Earth (gravity direction) and the x-axis is aligned with user heading direction. Here, we might choose the x-axis and y-axis acceleration with variance, mean, minimum, and maximum because these features of the x-axis and y-axis will change when the user is nodding.

Take another example with features that are not orientation specific. If we attach sensors on the drilling machine, we do not want to constrain the use of machine for the orientation in which the drilling machine has to be used. In this case, we should NOT apply orientation-dependent features, such as, x, y, z, mean values, minimum, maximum. We might choose variance, zero-crossing, peak-detection, energy of norm (the magnitude of x, y, z-axis). These features are independent of the orientation of the drilling machine and these features are most applicable for detecting vibration.

### 2.2.2 Leverage model training

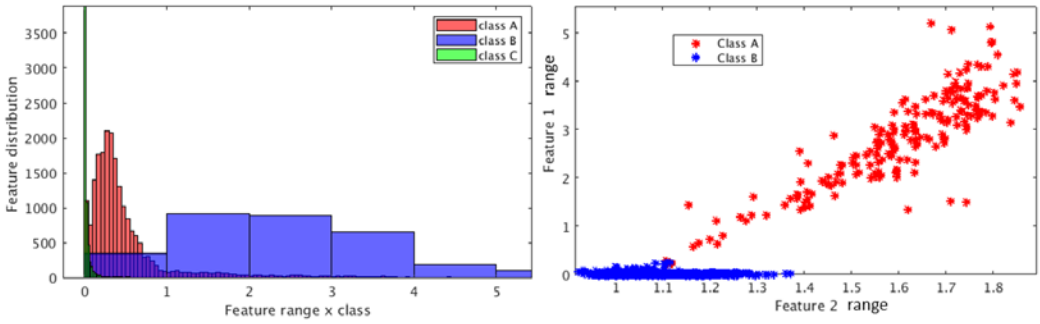
One approach to derive a first possible pool of candidate features is by selecting all the available features and train the model according to the classifier selected. In our case, the available features are pre-defined in the MLC and the classifier is a decision tree that needs to be trained. While training the model, the decision tree learning algorithm will select the features that can better discriminate the classes. It is also recommended to select features which are intuitive to solve the problem. These features may then be selected as the only features to use for the next steps in the development of the machine learning model.

There are some drawbacks to this approach, such as the possibility to get a model that is overfit due to the usage of too many features or features that are not actually relevant to the detection of the specific classes. Therefore, it is always recommended to also rely on the other methods for feature selection and to possibly avoid using all features in the first place. A better approach is to first identify a set of potentially good features and then train the model in order to see which ones it uses.

### 2.2.3 Feature visualization

Visualization may help in selecting and dropping features to optimize the performance of the model. The information contribution from a single feature can be evaluated with a measure of how much different classes are separated. The analysis can be visually performed with 1D/2D representations as shown in Figure 8.

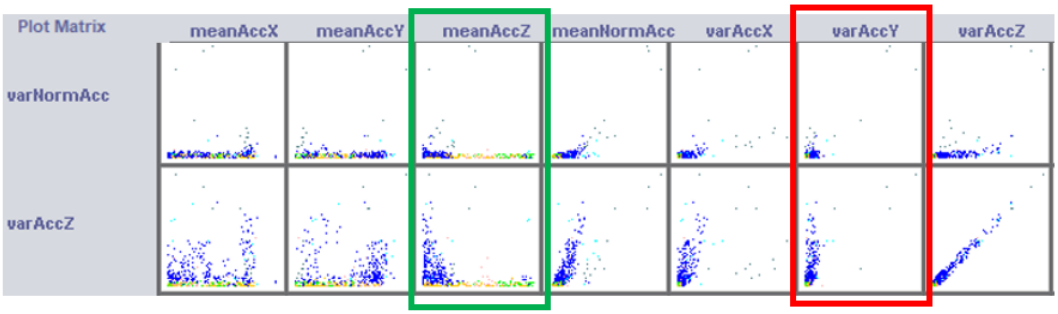
**Figure 8. Histogram of the computed values of a single feature for three different classes. These are reasonably separated which means that the selected feature is valuable in discriminating the different classes Features and Class visualization.**




It may be useful, when possible, to plot the Cartesian diagram of 2 features, like in the picture above (on the right), where is shown a 2D plot related to a 2 classes classification problem and a selection of 2 different features. In this case, the strict separation is evident and thus the information from combining the selected two features is visible. This kind of plot is helpful to also evaluate the relationship between different features and classes. The figure on left side is a histogram of a single feature for each class. The distribution of the feature value for each class is very different and even using a simple threshold, we can see that the class can be separated.

The same approach is supported by ML tools such as WEKA. The pair plots shown below represent the relationship between different features and classes. In particular, we can understand that the feature combinations in the green block show separability, while the ones in the red block are not.

**Figure 9. Visualization of features in WEKA**

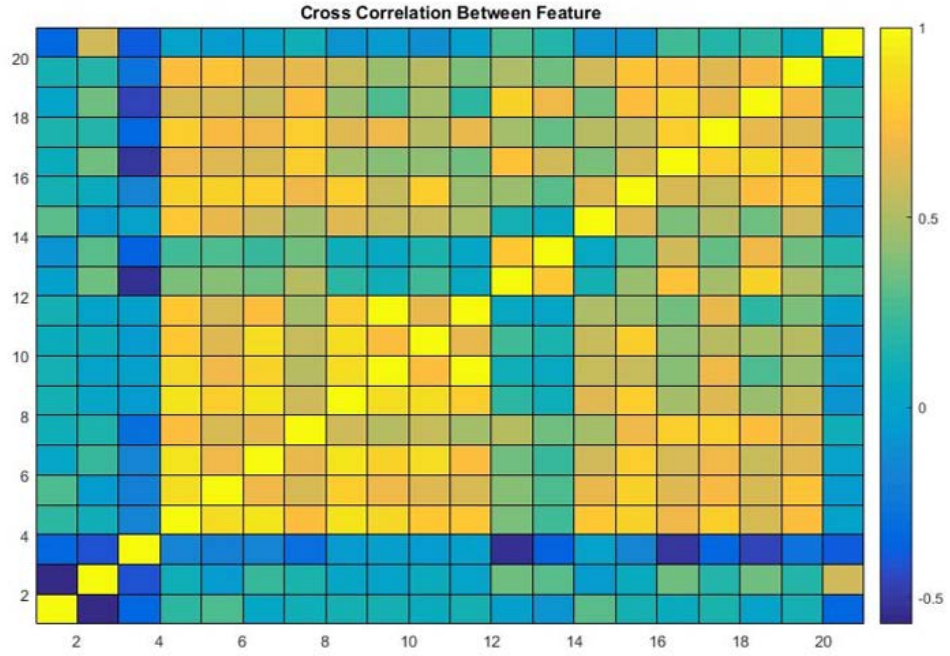



The following things to look at before we include the feature in the final computation:

- If the feature has a linear relationship or is strongly correlated with another feature, the information is already captured by another feature and hence this feature can be dropped. If visualization is difficult, the covariance matrix can be computed to identify the most correlated features, as shown with an example in Figure 10. In this plot we can see there are many non-diagonal elements which are strongly correlated and hence they can be dropped.

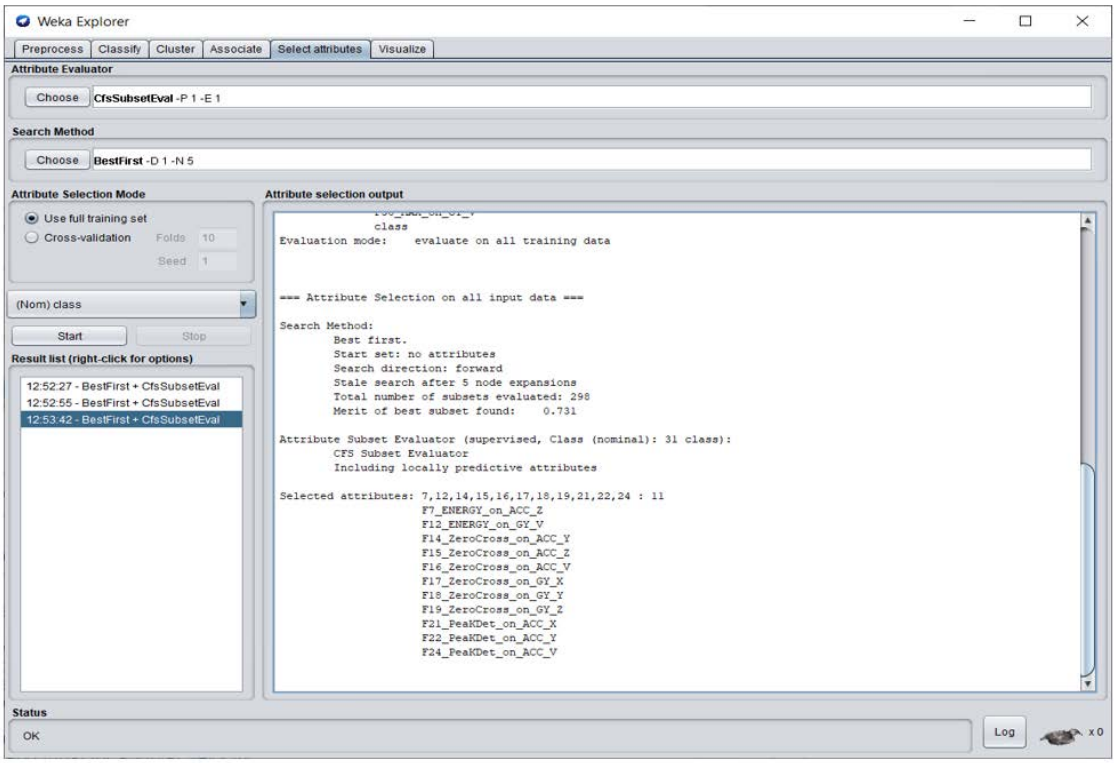
- Variability: if the feature does not show much variation between different classes, the feature may not help in separating classes.

**Figure 10. Cross-correlation between features**




Finally, WEKA provides some built-in tools to identify the best subset of informative features. In Figure 11 an example is shown for the process to get the features ranking in Weka. In the example shown, the ARFF file has 30 features and by using feature ranking logic, Weka selects only 11 important features for the current dataset.

**Figure 11. Feature selection method available in Weka**

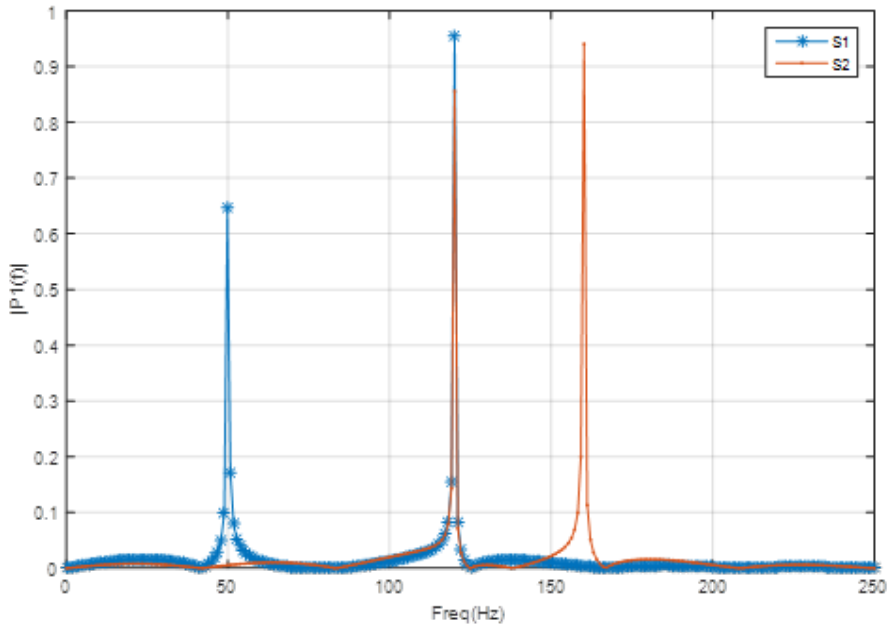



## 2.3 Filter selection

The basic features such as mean, variance, and energy contain direct information but may not be able to separate classes with different dominating frequencies. Applying a filter on the signal allows isolating specific information from the signal and may allow separating the class successfully. One can select specific frequency regions by applying a dedicated filter to the raw signal. For human activity detection, we can distinguish precisely if the user is walking (typically 1 - 2 Hz) versus jogging (2.5 - 4 Hz) through the use of appropriate filters. Filter selection should be done by doing Fourier analysis and observing Power spectral density of signal. The Fourier analysis provides insight on which frequency region is dominating for each class.

This process involves the selection of a proper filter (low pass, high pass, bandpass) and defines the cutoff frequencies. The selection of filter and frequency of interest is done by finding significant energy in the non-overlapping region between classes. The following Figure 12 illustrates the overlapping region and frequency region of interest for class S1 and S2.

**Figure 12. Frequency analysis for two different classes**




If we compute total energy (total power over all frequencies (0 - 250 Hz)), it will not provide information about the energy at a specific frequency (e.g. 50 Hz). Hence, it may not be sufficient to separate 2 classes if the total energy for different classes are similar but have different power at different frequencies. The user has to apply an appropriate filter to remove the signal from a certain frequency region before computing the energy. Applying a filter will allow the signal to pass through the threshold that is lower (in a low-pass filter) than the cutoff frequency. If we take the example in Figure 12 and apply a low-pass filter at 60 Hz, the signal S2 will be completely filtered out and part of S1 which is below 55 Hz will pass through. Now if we compute the energy on both signals after filtering, S1 will have higher energy compared to S2 because the S2 signal is completely filtered out. The recommended filter in the above case can be a band-pass filter at 50 Hz and 150 Hz or low-pass filter at 55-100 Hz and 125-150 Hz.

## 2.4 Train-test split

Once we have decided on the window length, then the next step is to decide how to divide data into a training set and testing/validation set. For every machine learning model, it is important to split the data in testing, training, and validation (if needed). The split of data is required in selecting the model, checking the quality (underfitting, overfitting) of the final model. The training data is used to train the model, testing data is used to evaluate the quality/accuracy of a trained model and validation data is used if we have more than one model to select the appropriate model. The accuracy of testing and training data can be useful in deciding if the model is underfit, overfit or well balanced. For example, if the accuracy of the trained data is high (>80%) and low on test data (<70%), the model is likely overfit and the model may require some pruning or adding more diverse data. On the other hand, if the accuracy of trained and testing data is low (< 60%), the model is likely underfit and may require adding more features or reducing pruning. More information can be found in Section 4.

Usually, the ratio between training and testing samples is about 80:20 or 70:30 if we are evaluating a single model. In case we are interested in evaluating multiple models (from different training algorithms J48, CART or different feature sets), then we should split the data into training, validation and testing in a 60:20:20 ratio. The validation data should only be used to select the best model and test data should be used to evaluate the final accuracy of selected model. The selection of the ratio for splitting the data is problem-specific and can be changed according to the application requirements.

Python code is shown to divide whole data into training data and testing data as follows.

```
from sklearn.model_selection import train_test_split
# split train : test = 70 : 30
X_train, X_test, y_train, y_test =
  train_test_split(wholeData, label, test_size=0.3)

# split train : test : validate = 70: 15 : 15
X_test, X_validate, y_test, y_validate =
    train_test_split(X_test, y_test, test_size=0.5)
```

The above listed Python code will first separate the whole dataset into training and testing with the ratio 70:30. Then it will split the testing dataset in half as the validation dataset. The final ratio between training, testing, and validations can be 70%, 15%, and 15%.

There is a simple rule to determine if we need to split a validation dataset or not. If there is one model, then split the data between training and test (30%-70% or 20%-80%) and evaluate the model trained on training data on test datasets. If there are multiple models, then split the data between testing, validations, and training data sets (15%-15%-70% or 20%-20%-60%). We can perform cross validation on training and validation. Then, we select the model using validation and evaluate the final model on the test dataset.

# 3 Decision tree

A decision tree is a binary tree structure made of nodes and leaves (the nodes that do not have children). A decision node has two branches. The leaf node produces the classification output or the decision. At each decision node, a split on the data is performed based on the threshold of one of the input features. Usually, less than the threshold samples would choose left node otherwise right node is chosen. While traversing through the decision nodes, more and more splits on the data are made, and this process stops at a certain leaf with the decision. Each leaf is associated with a class label and the paths from the root to leaf produce the classification rules.

## 3.1 Different types of decision trees

There are different algorithms to generate decision tree classifiers.

The conceptual difference between different algorithms is not the way the splitting at node is done but it also differs in feature importance, pruning criteria, and stopping criteria. The basic steps of decision tree training are:

a. Splitting (node): The splitting criteria such as Gini, Information Gain, or Entropy allow for maximizing the information (maximizing separating classes) using one of the features. The process involves selecting the best feature and corresponding threshold. Two examples are shown in Figure 14 which illustrates the distribution of a feature for different classes. If we select this feature and appropriate threshold, we can see the example on the right side would not separate classes easily as compared to the example on the left side. The criteria, such as Gini, entropy, information gain, reflect the separation between classes for a given threshold.

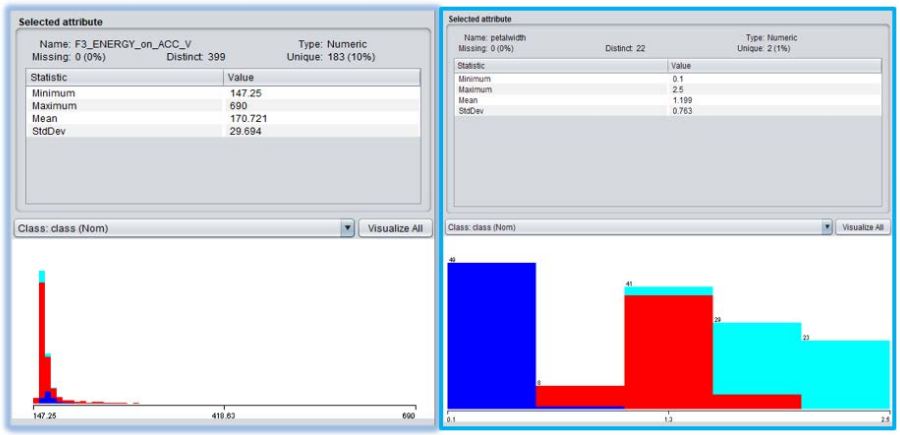
b. Pruning: The process of decision tree pruning involves removing certain nodes in the final decision tree to reduce the size and avoid overfitting. We will present one example of a pruning method in Section 4.2.4.

c. Stopping criteria: The decision tree can grow in depth to achieve 100% accuracy, but it leads to an overfitting problem. Every algorithm uses different criteria to stop the further splitting of nodes. The criteria can be a minimum number of samples in each node or a minimum gain for further splitting. The following table summarizes the overall differences in different types of decision tree algorithms.

**Figure 13. Different decision tree algorithms**

| Algo | Splitting criterion | Pruning criterion | Other features |
|---|---|---|---|
| **CART** | • Gini<br>• Twoing | Cross-validation post-pruning | 1. Regression/Classification<br>2. Nominal/ numeric attributes<br>3. Missing values<br>4. Oblique splits<br>5. Nominal splits grouping |
| **ID3** | Information Gain | Pre-pruning | 1. Classification<br>2. Nominal attributes |
| **C4.5** | • Information Gain<br>• Information Gain Ratio | Statistical based post-pruning | 1. Classification<br>2. Nominal/numeric attributes<br>3. Missing values<br>4. Rule generator<br>5. Multiple nodes split |

**Figure 14. Features distribution for different classes**

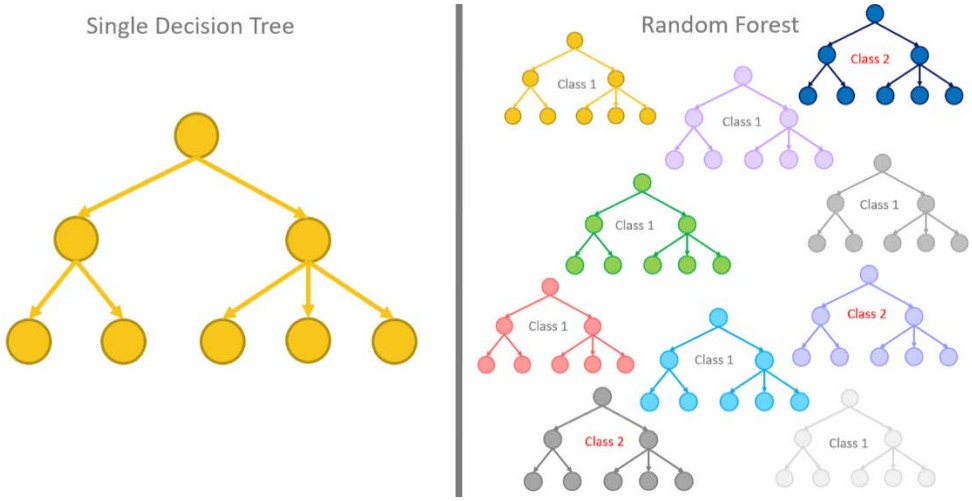



Each decision tree training algorithm has certain advantages, but C4.5 is the most recent from the algorithms mentioned and it is robust against outliers in datasets and generates a balanced decision tree.

## 3.2 Decision Tree vs. Random Forest

Random Forest is another classification method. Many decision trees can produce more accurate predictions than just one single decision tree because it is generated using different criteria. In other words, the Random Forest builds K (preselected, maximum number of trees) slightly differently trained decision trees and merges them to get more accurate and stable predictions, compared with a single decision tree. The following figure illustrates the concept.

**Figure 15. Single decision tree versus Random Forest**




We usually use 70% of the dataset as training data and the rest of it (30%) as test data. One implementation of Random Forest is described as: for each decision tree that is used in Random forest, we use the boosting or in place random sampling to select the dataset and different sets of features to get a different decision tree and the final output will be an

average of outputs from different decision trees. Hence, Random Forest does not suffer from overfitting and we can get a better, more accurate and stable prediction when using Random Forest. With the MLC we can replicate a similar behavior since it can allow for running up to 8 decision trees simultaneously to classify the same problem. Then, we can merge the results on a microcontroller which can process the outputs from these trees.

## 3.3 Build and visualize decision tree

In Python, we can visualize the decision tree after training. The following is the code which can be used to implement the process steps including:

- Splitting data in training and testing data
- Building the decision tree
- Visualization of the tree

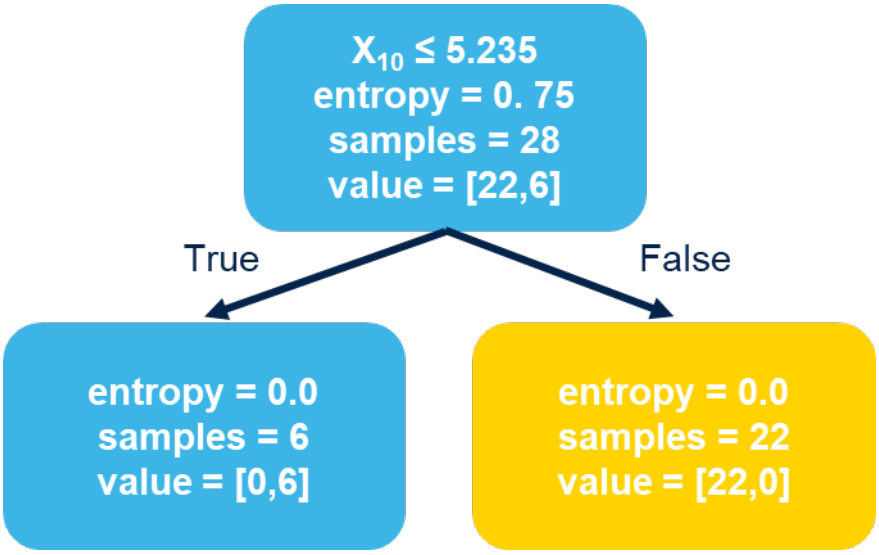
```
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.externals.six import StringIO
from sklearn.tree import export_graphviz

X_train, X_test, y_train, y_test = train_test_split(
    segm, labels, test_size = 0.3, random_state = 101)

clf_entropy = DecisionTreeClassifier(criterion =
    "entropy", random_state = 100)
dot_data = StringIO()
export_graphviz(clf_entropy, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True)
graph = pydotplus.graph_from_dot_data(
    dot_data.getvalue())
```

This is quite a simple example with only one node and two leaves. The code above will generate a graphical representation of the decision tree shown in Figure 16.

**Figure 16. Example of decision tree plot in Python**



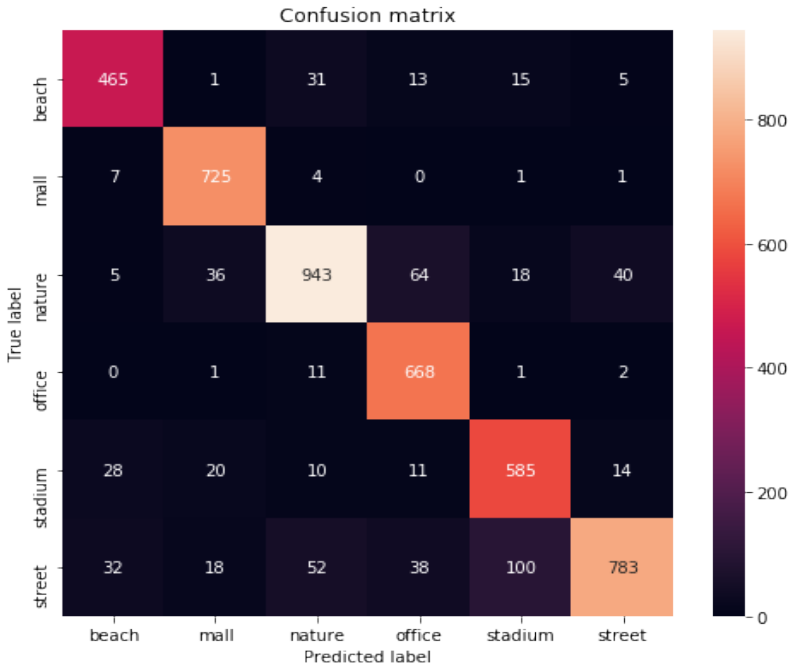

The first line in the node represents the condition on a certain feature to split the data. The next line about entropy represents the variation of response (classes). If there is a single class, then the entropy field will be zero. The samples field represents the total samples reach to a given node or leaf. The value field contains the information about the sample in an individual class. We should notice that entropy after each split is reduced, and the higher the reduction, the higher the importance of the node.

If the decision tree is small, visualization may allow understanding the rules/conditions in the nodes. In a few cases, it may allow us to understand if the rules are set as expected.

## 3.4 Make prediction and confusion matrix

A confusion matrix is used for summarizing and displaying the classification performance of a decision tree. This matrix provides an overview of correct classifications and confusion between multiple classes. Since there can be different definitions to express the accuracy of a classification algorithm, the approach to use a confusion matrix is the most straightforward way to avoid misunderstanding about “accuracy.” Let’s take an example as the following figure.

**Figure 17. Confusion matrix for audio spatial environment classification**




We can set the rows as the true classes and the columns as the predicted classes. For a given class, the output of classification algorithm is entered in each cell of the corresponding row. This generates mapping of the decision tree algorithm between the ground truth and the prediction result. Here, we can see that the algorithm is confused between the “street” class and “stadium” class. The fifth cell of the last row in the confusion matrix shows that the algorithm incorrectly predicted 100 samples of the “street” class into the “stadium” class. The following Python code can allow us to build the confusion matrix as shown in Figure 17.

```
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.externals.six import StringIO
from sklearn.tree import export_graphviz
from sklearn.metrics import confusion_matrix

def confusion_matrix(y_test, y_pred, class_list):
    confusion_matrix = metrics.confusion_matrix(y_test,
                y_pred)
    fig_confuse = plt.figure(figsize=(4, 3.5))
          sns.heatmap(confusion_matrix, xticklabels=
          sorted(class_list), yticklabels = sorted(class_li
          st), annot=True, fmt="d");
    plt.title("Confusion matrix")
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show();

class_list = ["beach", "mall", "nature", "office",
                    "stadiu  m", "street"]
y_pred_entropy = clf_entropy.predict(X_test)
confusion_matrix(y_test, y_pred_entropy, class_list)
```

# 4. Model evaluation

How can we know the decision tree model we have trained is good enough? There are multiple methods available to measure model performance. First, we must implement the confusion matrix with the testing dataset. To select a stable model, the K-fold (cross-validation) can be implemented. The following section explains how to determine the quality of the model and how to fix any issue in the model.

Cross-validation is also known as out-of-sampling technique. We often split the data randomly in testing and training to train and test the model. The training data is assumed to have captured diverse data and is sufficient for model training exercise. There are 3 specific reasons for using cross-validation:

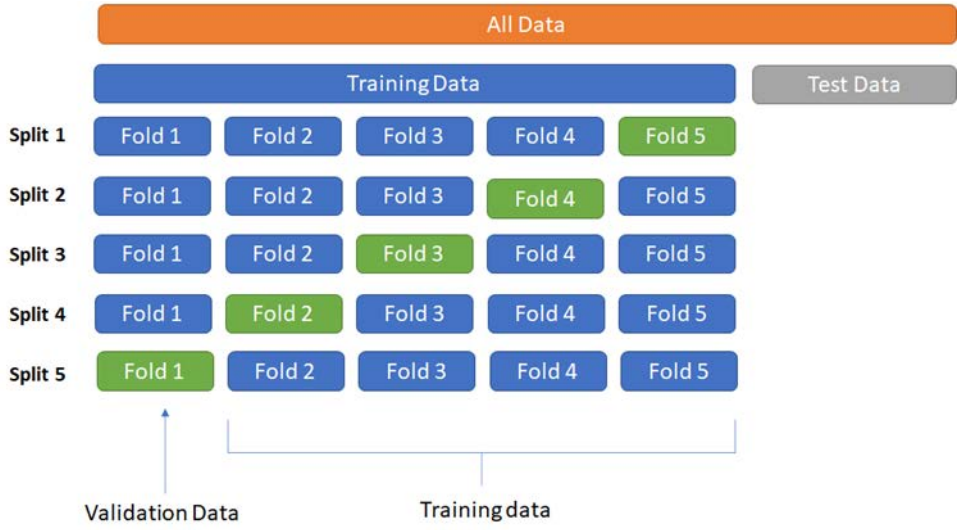
- First, for some specific problems, a random sampling method may select all the data of a particular use case which is a part of testing data and is not part of training data. In this case, the model will perform poorly since the model is not trained for the specific use case. Consider the case where data variation is seasonal and random sampling fails to split the data and make it part of training and testing data.
- Second, since we are performing a random split, the accuracy of the model will keep changing after every random split.
- Third, the data used in validation are just used to validate and select the model; once the mode is selected, we are not able to use this data to improve the accuracy of model.

There are various cross-validation techniques that are available. Each of them has certain advantages and disadvantages. The standard cross-validation techniques are:

- Leave one out
- Leave p out
- Repeated random sub-sampling
- K-fold
- Holdout

The most used method is K-fold cross-validation (specifically 10-fold). In K-fold cross-validation, training data should be divided in k-splits and training will be done k times as demonstrated in the following Figure 18.

**Figure 18. 5-fold cross validation**



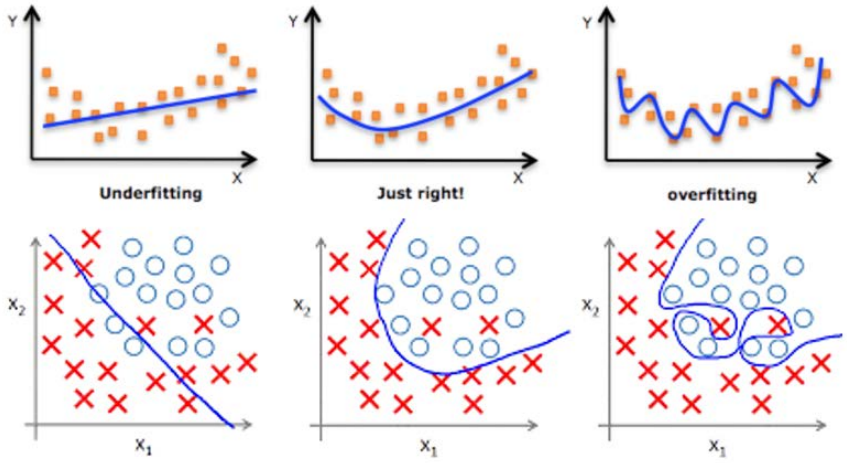

One of the subsets is used as the validation set and the other k-1 subsets are used as training sets. In this manner each subset will be used k-1 times for training and 1 time for validation. The benefit of this approach is that the error estimation is averaged over all k trials to get the total effectiveness of the model. At the end of K-fold validation, the final model will be prepared utilizing all the training data.

When we perform K-fold cross validation, its validation summarizes various errors through statistical measures such as mean squared error, root mean squared error or absolute median deviation. These parameters are useful in selecting the appropriate model in determining the quality of the model (underfitting, overfitting etc.). In the next subsection we will discuss in more detail about the quality of the model.

## 4.1 Underfitting

Underfitting happens when a machine learning model is not complex enough or does not have sufficient information (features) to capture the relationship and information contained in the dataset. The concept of underfitting, good fitting (Robust) and overfitting is illustrated in Figure 19.

**Figure 19. Concept of underfitting, good fitting (Robust), and overfitting**




How to determine if the model is underfitting? When the model is underfitting it fails to capture data, patterns contained in training data. In other words, if the trained model performs poorly both on training data and test data, then it can be due to underfitting. In this scenario, the selected features may not be sufficient to capture the information for the model to fit the data.

Underfitting is easy to handle and the standard solution is to add extra features and train the model again. The other option is to reduce the number of classes. In certain cases, if data between two classes look similar, the model may not be able to do a decent job in classifying it. Adding diverse data may also be useful for addressing the underfitting issue because the model may be able to increase the complexity and learn the patterns that were missed in the previous training.

## 4.2 Overfitting

Unlike underfitting, overfitting is a modeling error that occurs when the model performs well on trained data points but performs poorly on unseen data. It would have good accuracy on training dataset but low accuracy during testing or on another newly collected dataset. The overfitting model generally takes the form of making an overly complex model to capture features in data used in training. This situation arises when the model has a lot of flexibility and freedom in fitting as many as points as possible by interpreting noise as actual pattern. K-fold validation or any cross-validation generally helps in identifying the overfitting. When we run the model on the training dataset, and it predicts 95% or higher accuracy but on test data, it shows 50-70% accuracy, then it is a good identification of an overfitting problem. The RMSE (Root Mean Square Error) in K-fold validation is a cumulative error on the various folds on validation data. If the final model accuracy is good but RMSE is high (>0.2), then it is a good sign of overfitting.

The following methods can help to address overfitting.

### 4.2.1 Collect more logs

This is usually a straightforward solution for an overfitting issue. New logs should be collected in a different situation with greater diversity. The collected data should capture various use cases that the model would encounter in a real-world application.

### 4.2.2 Data augmentation

Sometimes, we do not have multiple users to collect logs or it is not easy to have diverse environments / conditions. We can consider adding noise or rotate the data at different angles. Take an example for a head gesture detection project. The objective of this project is to detect the following classes [“nod”, “shake “,” stationary”, “walk”, “swing”] and the sensor is attached on a headset. We expect that the y-axis points towards Earth (same orientation as gravity) and the x-axis is the user heading direction.

However, if test data has only a limited number of users with a limited orientation setting, then the model could suffer from two issues. First, the decision tree that is trained on this data would get tuned for the people who collect the logs and show poor performance on other unseen users. Second, the model is tuned for a specific orientation and performance could be affected by the user’s preference for the orientation of the headset. To overcome these issues, we can:

- Inject noise as mentioned earlier to handle any small variation in data.
- Rotate the existing data at different angles which are possible (during usage) and concatenate the rotated data with the training dataset. The trained model will cover most of the user’s preferred orientation and the solution would be robust enough against sensor orientation.

### 4.2.3 Reduce number of features

Selecting a large number of features during training effort may allow more flexibility and freedom to fit the model to all the training data, but it can cause overfitting. Removing features that are not very important can help in lowering the complexity of the model and solve the problem of overfitting. As we mentioned in the feature visualization section, plotting and visualizing features helps in understanding the relationship between feature and class. If the feature does not make sense, the user can remove it from model to reduce the complexity.

### 4.2.4 Pruning

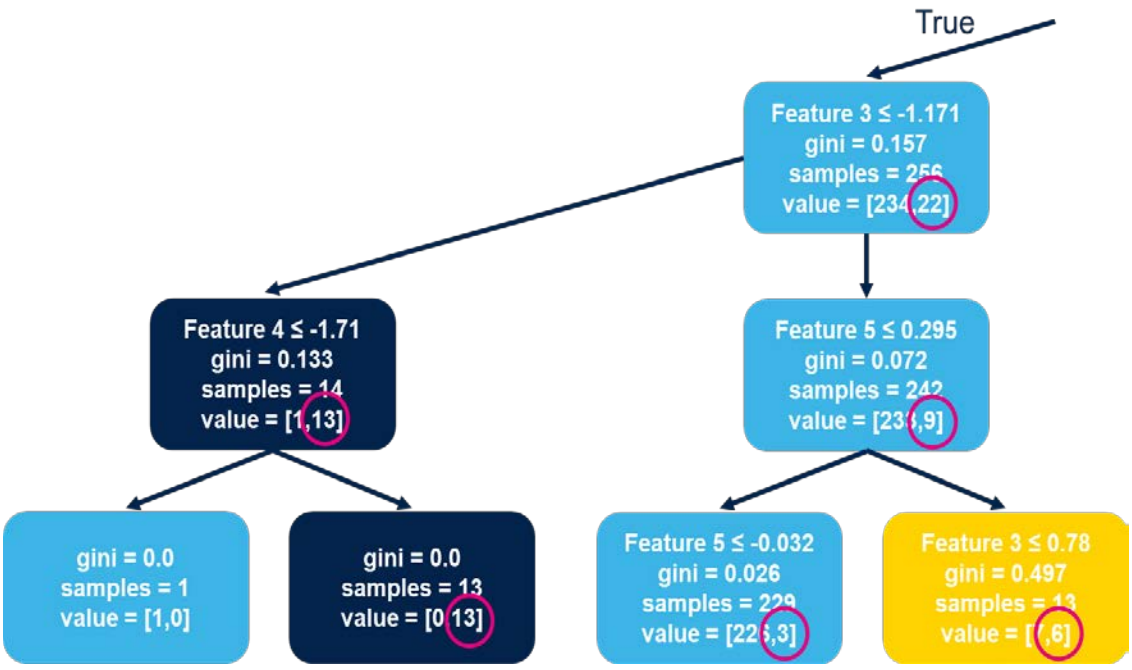
Pruning means simplifying the decision tree by removing sections of the tree that are not critical. This can correspond to the nodes which are not classifying too many data points. You can think of this as an early stopping point of growing the model. We stop the branching of any node at this point. Different types of decision trees have different criteria to prune the tree as shown in Figure 13. There are two types of pruning:

- Pre-pruning: Pre-pruning stops growing the tree before it continues to classify the subset.
- Post-pruning: After building the decision tree, logic decides about trimming a certain node or subtree.

Most of the time, post-pruning is preferred since it is not easy to know the precise moment to stop growing the tree. In post-pruning, the logic first grows the complete decision tree and then, using criteria (such as information gain, or imbalance of classes), starts removing the non-critical part of the decision tree. Here, we provide a practical example for the user to understand post-pruning. If there are certain nodes in the decision tree that only classify a few samples of a particular class and remaining large number of samples belong to other classes, then replace the node with the leaf which classifies all of the samples that belong to “other” classes. The performance of the resulting decision tree will not be adversely affected by this operation.

Pruning reduces the leaves which are part of the decision tree that may cause overfitting. That is the reason pruning can help in reducing overfitting and make the solution robust. Let’s take the nodes shown in the following Figure 20 as an example.

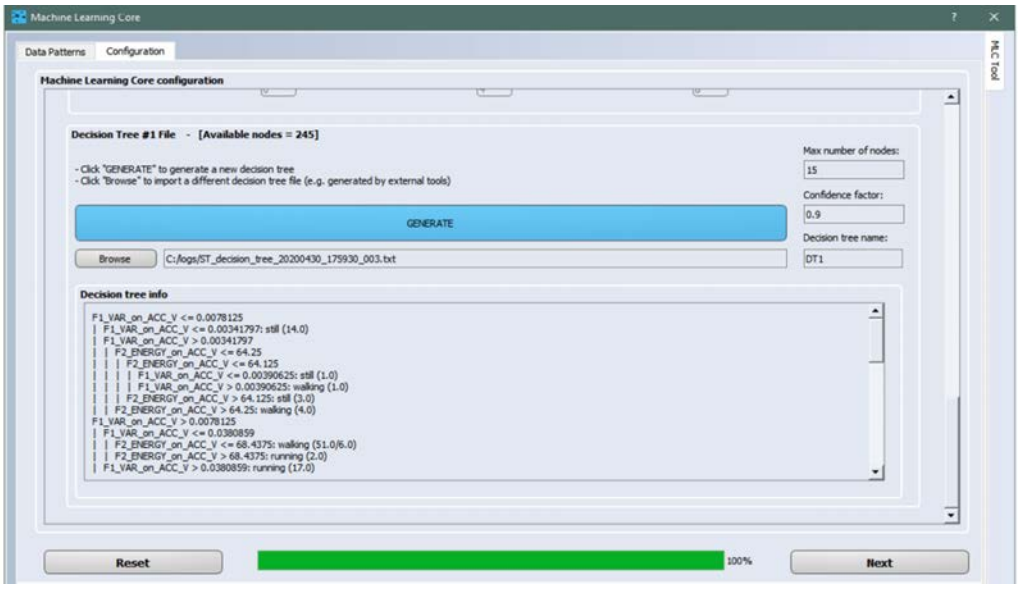
**Figure 20. Example of pruning**



The left-hand side node (Features 4 ≤ -1.71) of the above decision tree is only classified as one exception sample but all other thirteen samples belong to another class. In this case, it could be better to make this node as a leaf.

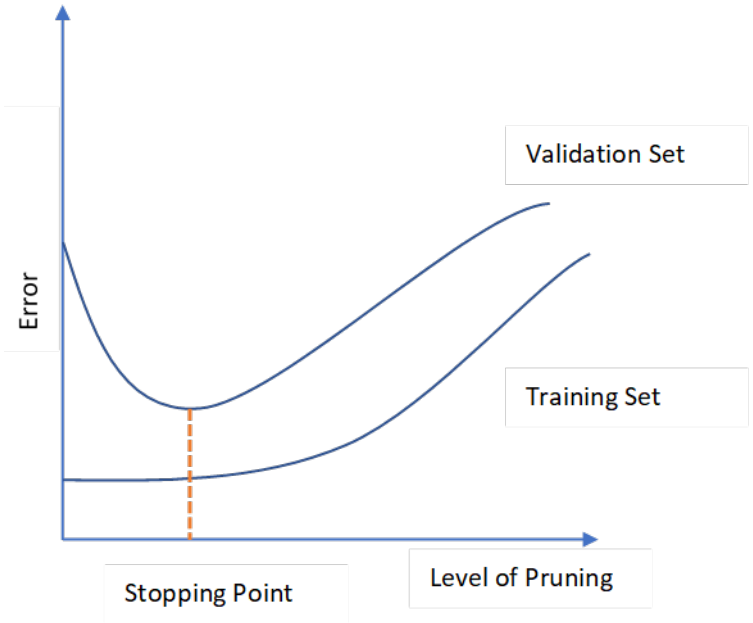
The decision tree algorithm inside Unico-GUI has the option for pruning. We can set up a limited number of nodes and the confidence factor would prune until we meet the condition.

**Figure 21. MLC default decision tree algorithm in Unico-GUI**




The approach to determine optimum pruning is through monitoring the validation and training accuracy. If we plot the accuracy/error of both validation and training datasets against the level of pruning, we tend to get a plot as shown in Figure 22 below.

**Figure 22. Stopping point of pruning**



Without pruning, the model error on training data will be lowest but the error on the validation set can be high due to overfitting. As we increase the level of pruning, the error in the training set will increase, but the error on the validation set will start reducing because some of the nodes were overfit to the noise. As we further increase the pruning, the error on both validation and training sets will increase because pruning will remove nodes that are classifying correctly. This is the point to stop the pruning process.

### 4.3 Meta-classifier

A meta-classifier provides an additional method for the machine learning model to improve the accuracy, and in particular the stability of the output from a computed model. A meta-classifier can be applied to the classification algorithm (not constrained to the decision tree form), acting as a smoothing filter (such as maximum voting) on the output of the decision tree. There is a detailed description of the meta-classifier in the respective MLC application notes.

The following example illustrates the advantage of using a meta-classifier for a use case where the transition between classes is not frequent. Consider the case of activity detection for two classes: “awake and “sleeping” when the sensor is installed in a smartwatch. A slight movement of wrist could trigger the classification output to “awake” when the user is actually “sleeping”. A meta-classifier can smooth the output from the classification algorithm and remove spurious ”awake” classification from the output.

**Supporting Material**

| Part Number | Market | Family | Application Note |
|---|---|---|---|
| LSM6DSOX | Consumer | iNEMO inertial modules (IMU) | AN5259 |
| LSM6DSO32X | Consumer | iNEMO inertial modules (IMU) | AN5656 |
| LSM6DSRX | Consumer | iNEMO inertial modules (IMU) | AN5393 |
| ISM330DHCX | Industrial | iNEMO inertial modules (IMU) | AN5392 |
| IIS2ICLX | Industrial | Accelerometer | AN5536 |

**Machine Learning Resources**

| | |
|---|---|
| GitHub projects for MLC | **https://github.com/STMicroelectronics/STMems_Machine_Learning_Core** |
| ST MLC Ecosystem | **www.st.com/mems-sensors-ml** |
| ST MEMS Community for ML | **community.st.com/s/group/CollaborationGroup** |
| MEMS and Sensors Q&A | **Mems-and-Sensors Q&A** |

## Revision history

| Date | Version | Changes |
|---|---|---|
| 03-Nov-2020 | 1 | Initial release |
| 08-Feb-2021 | 2 | Improvements throughout the document, added Figure 4, updated Figure 16, Figure 20, added “Machine Learning Resources” |
| 20-Apr-2021 | 3 | Added LSM6DSO32X and its application note AN5656 |