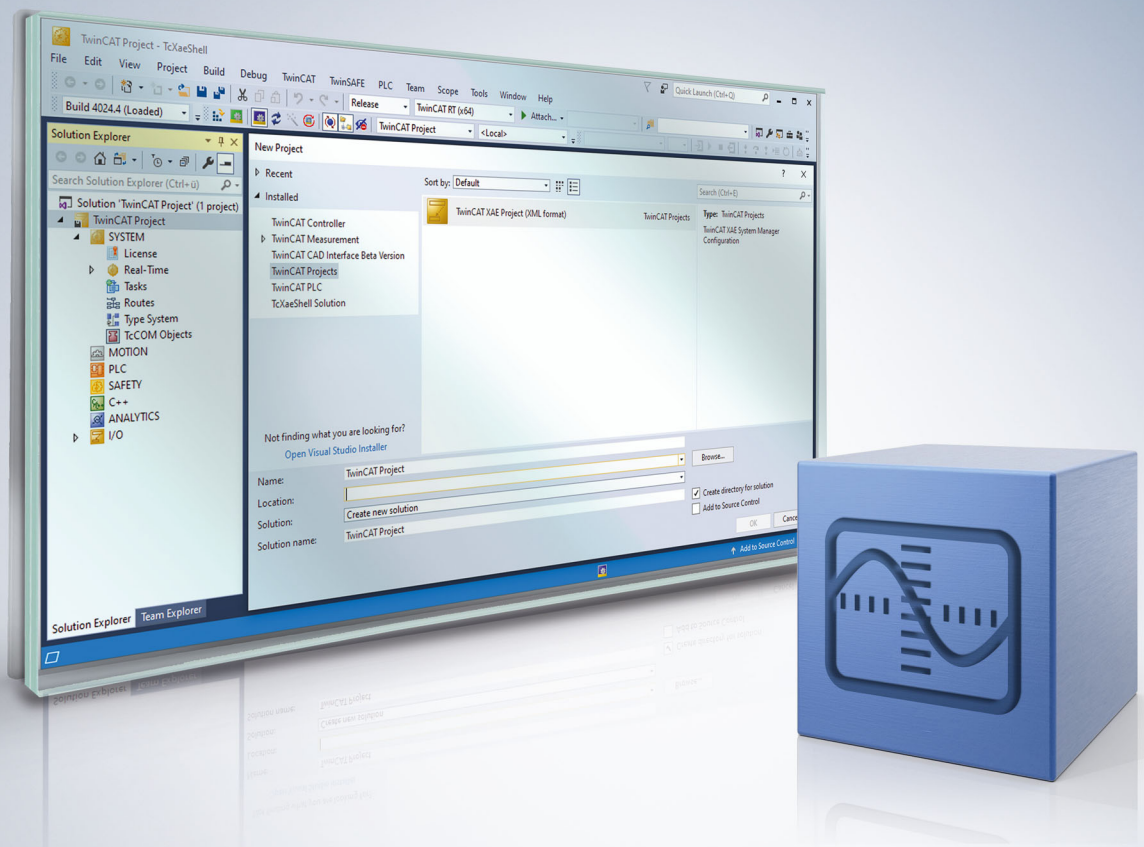
BECKHOFF New Automation Technology

Manual | EN

# TF3800, TF3810

## TwinCAT 3 | Machine Learning- und Neural Network Inference Engine




2023-10-13 | Version: 1.6.4

# Table of contents

# 1 Foreword

## 1.1 Notes on the documentation

This description is intended exclusively for trained specialists in control and automation technology who are familiar with the applicable national standards.
For installation and commissioning of the components, it is absolutely necessary to observe the documentation and the following notes and explanations.
The qualified personnel is obliged to always use the currently valid documentation.

The responsible staff must ensure that the application or use of the products described satisfies all requirements for safety, including all the relevant laws, regulations, guidelines, and standards.

### Disclaimer

The documentation has been prepared with care. The products described are, however, constantly under development.
We reserve the right to revise and change the documentation at any time and without notice.
No claims to modify products that have already been supplied may be made on the basis of the data, diagrams, and descriptions in this documentation.

### Trademarks

Beckhoff®, TwinCAT®, TwinCAT/BSD®, TC/BSD®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® and XPlanar® are registered and licensed trademarks of Beckhoff Automation GmbH.
If third parties make use of designations or trademarks used in this publication for their own purposes, this could infringe upon the rights of the owners of the said designations.

### Patents

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents:
EP1590927, EP1789857, EP1456722, EP2137893, DE102015105702
and similar applications and registrations in several other countries.

EtherCAT®

EtherCAT® is registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany

### Copyright



## 1.2 For your safety

### Safety regulations

Read the following explanations for your safety.
Always observe and follow product-specific safety instructions, which you may find at the appropriate places in this document.

### Exclusion of liability

All the components are supplied in particular hardware and software configurations which are appropriate for the application. Modifications to hardware or software configurations other than those described in the documentation are not permitted, and nullify the liability of Beckhoff Automation GmbH & Co. KG.

### Personnel qualification

This description is only intended for trained specialists in control, automation, and drive technology who are familiar with the applicable national standards.

### Signal words

The signal words used in the documentation are classified below. In order to prevent injury and damage to persons and property, read and follow the safety and warning notices.

### Personal injury warnings

**⚠ DANGER**

Hazard with high risk of death or serious injury.

**⚠ WARNING**

Hazard with medium risk of death or serious injury.

**⚠ CAUTION**

There is a low-risk hazard that could result in medium or minor injury.

### Warning of damage to property or environment

***NOTICE***

The environment, equipment, or data may be damaged.

### Information on handling the product

This information includes, for example:
recommendations for action, assistance or further information on the product.

## 1.3 Notes on information security

The products of Beckhoff Automation GmbH & Co. KG (Beckhoff), insofar as they can be accessed online, are equipped with security functions that support the secure operation of plants, systems, machines and networks. Despite the security functions, the creation, implementation and constant updating of a holistic security concept for the operation are necessary to protect the respective plant, system, machine and networks against cyber threats. The products sold by Beckhoff are only part of the overall security concept. The customer is responsible for preventing unauthorized access by third parties to its equipment, systems, machines and networks. The latter should be connected to the corporate network or the Internet only if appropriate protective measures have been set up.

In addition, the recommendations from Beckhoff regarding appropriate protective measures should be observed. Further information regarding information security and industrial security can be found in our https://www.beckhoff.com/secguide.

Beckhoff products and solutions undergo continuous further development. This also applies to security functions. In light of this continuous further development, Beckhoff expressly recommends that the products are kept up to date at all times and that updates are installed for the products once they have been made available. Using outdated or unsupported product versions can increase the risk of cyber threats.

To stay informed about information security for Beckhoff products, subscribe to the RSS feed at https://www.beckhoff.com/secinfo.

# 2 Overview

### Introduction

The idea behind machine learning is to learn a generalized correlation between inputs and outputs on the basis of example data. Accordingly, a certain amount of training data is required, on the basis of which a **model** is trained. In the training of the model, parameters of the model are adapted automatically to the training data by means of a mathematical process. In machine learning, the user has a large number of different models at his disposal. The selection and design of the models is part of the engineering process. Different types or designs of models fulfill different tasks, wherein the most important subdivisions are the classification and regression.

*Classification*: The model receives an input (an image, one or more vectors, etc.) and assigns it to a class. The output is correspondingly a categorized variable. These classes could be, for example, good part or bad part. Distinction can also be made between several classes, for example quality classes A, B, C, D.

*Regression*: The model receives an input and generates a continuous output. Not only are directly learned inputs assigned to directly learned outputs (as with a lookup table), but in addition the model is able to interpolate or, respectively, extrapolate non-learned inputs, provided it generalizes well. A functional correlation is learned.

Once a model has been trained, it can be used for the learned task, i.e. the model is used for **inference**.

### TC3 Machine Learning Runtime

Beckhoff supplies components for the inference of models in the TwinCAT XAR with its products **TF3800 TwinCAT 3 Machine Learning Inference Engine** and **TF3810 Neural Network Inference Engine**. A common software basis is used for both products, which is referred to in the following as the Machine Learning Runtime (ML Runtime for short).

The ML Runtime is a module (TcCOM) integrated in TwinCAT 3 and executed in the TwinCAT XAR. It is thus possible to access the model interface (model inputs and model outputs) as well as to execute the model loaded in the ML Runtime in hard real-time.

The TF3800 and TF3810 have different licenses. The license required depends on the ML model to be loaded. In principle, the TF3800 is required for loading and executing classic ML models. The TF3810 license is required for the loading of neural networks. The TF3810 includes the TF3800 license.

Additional information on supported models and required licenses. [▶ 20]

### Workflow

Basically, the process of machine learning and integration into TwinCAT 3 consists of three phases:

1. The collection of data
2. The training of a model
3. The deployment in the TwinCAT XAR

A large number of TwinCAT products are available for the **collection of data** from the controller: see TwinCAT Scope, TwinCAT Database Server, TwinCAT Analytics Logger, TwinCAT IoT, etc.

ML models can be trained in a large number of software tools. **ML models are usually created** in programming environments such as Python or R. Various open source and free tools exist that are suitable for the creation of ML models, such as PyTorch, Keras and Scikit-learn. Trained models can be exported from these tools in a standardized format as an ONNX file. The ONNX file is a standardized description of the trained ML model. This file is first converted into a format that is conditioned for TwinCAT (XML or BML file).

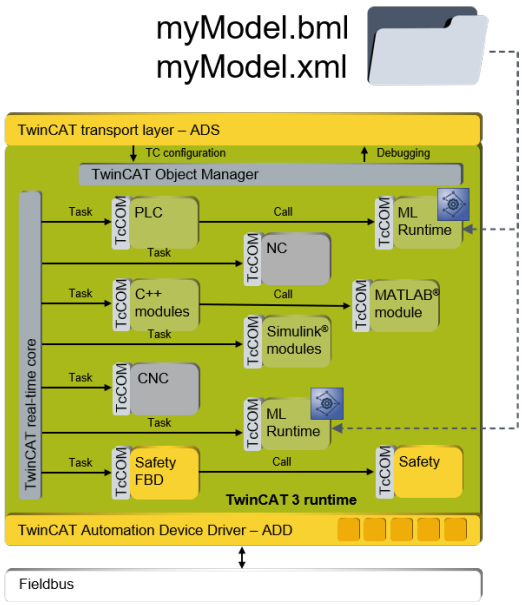
**Further Information:**

- Creation of ONNX files [▶ 61]
- Conversion of ONNX files [▶ 63]

TwinCAT offers two methods for **deploying the model**:

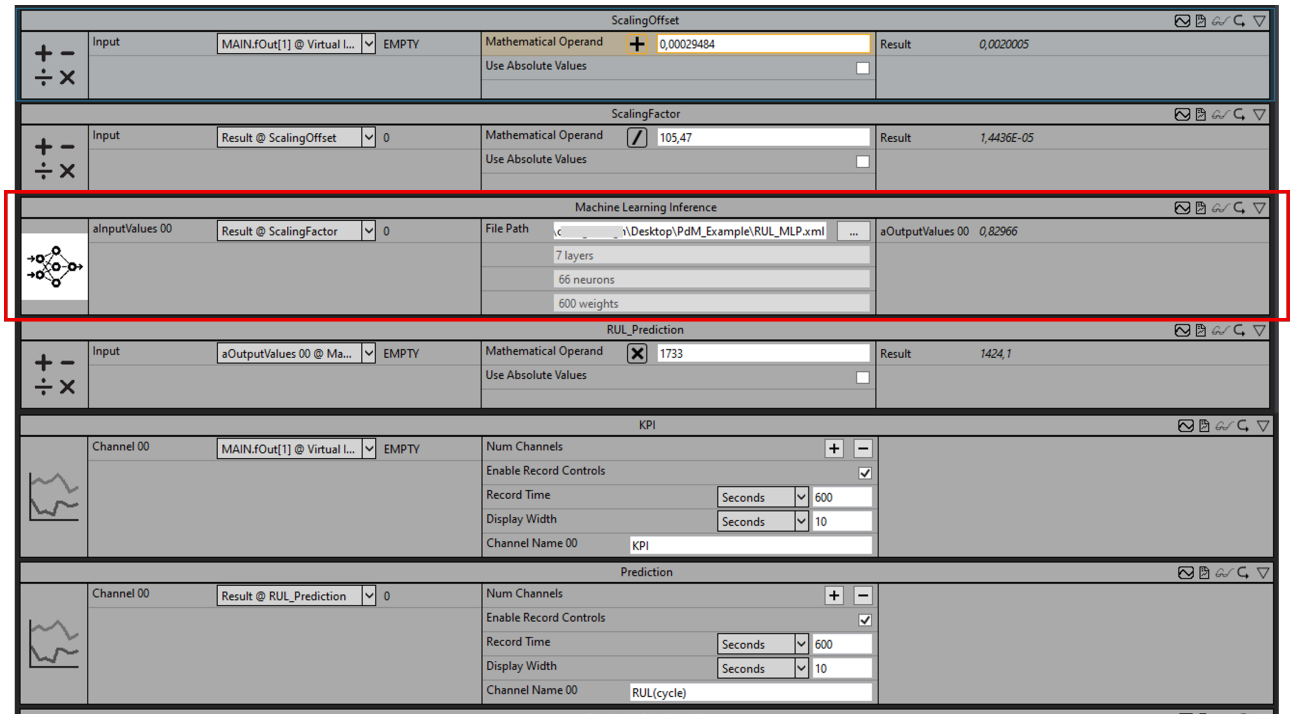
1. The library TC3_MLL is provided for use in the PLC environment. ML models can be loaded asynchronously via a method call and subsequently executed cyclically in the PLC program by calling a further method. Additional information on the PLC API [▶ 80]
2. A simple method of machine learning without programming effort: a TcCOM object that can be inserted in the TwinCAT object tree in the development environment and configured. On starting the system, the TcCOM loads the configured model and executes it in the assigned cycle time. Additional information on the ML TcCOM [▶ 78]

The picture below illustrates the deep integration of the Machine Learning Runtime in the TwinCAT XAR. Like all TwinCAT Runtime objects, the module is a TcCOM and is accordingly anchored deep in the hard real-time.




### Integration of machine learning into TwinCAT Analytics

The products Machine Learning Inference Engine and Neural Network Inference Engine can also be integrated into the TwinCAT Analytics workflow. Refer to the TwinCAT Analytics documentation for detailed information.



# 3 Installation

## 3.1 System requirements

**Runtime**

| Technical data | Description |
|---|---|
| Operating system | Windows 7 64-bit, Windows Embedded Standard 7 64-bit, Windows 10 64-bit, TwinCAT/BSD |
| Target platform | PC architecture (x64) |
| TwinCAT version | TwinCAT 3.1 Build 4024.0 or higher |
| Required TwinCAT setup level | TwinCAT 3 XAR |
| Required TwinCAT license | TF3800 TC3 Machine Learning Inference Engine or<br>TF3810 TC3 Neural Network Inference Engine |

The required license depends on the loaded ML model, see Machine learning models supported [▶ 20].

**Engineering**

| Technical data | Description |
|---|---|
| TwinCAT version | TwinCAT 3.1. Build 4024.0 and higher |
| Required TwinCAT setup level | TwinCAT 3 XAE |

The setup is to be executed both on the Engineering PC and on the Runtime PC.
7-day trial licenses can be generated for the runtime

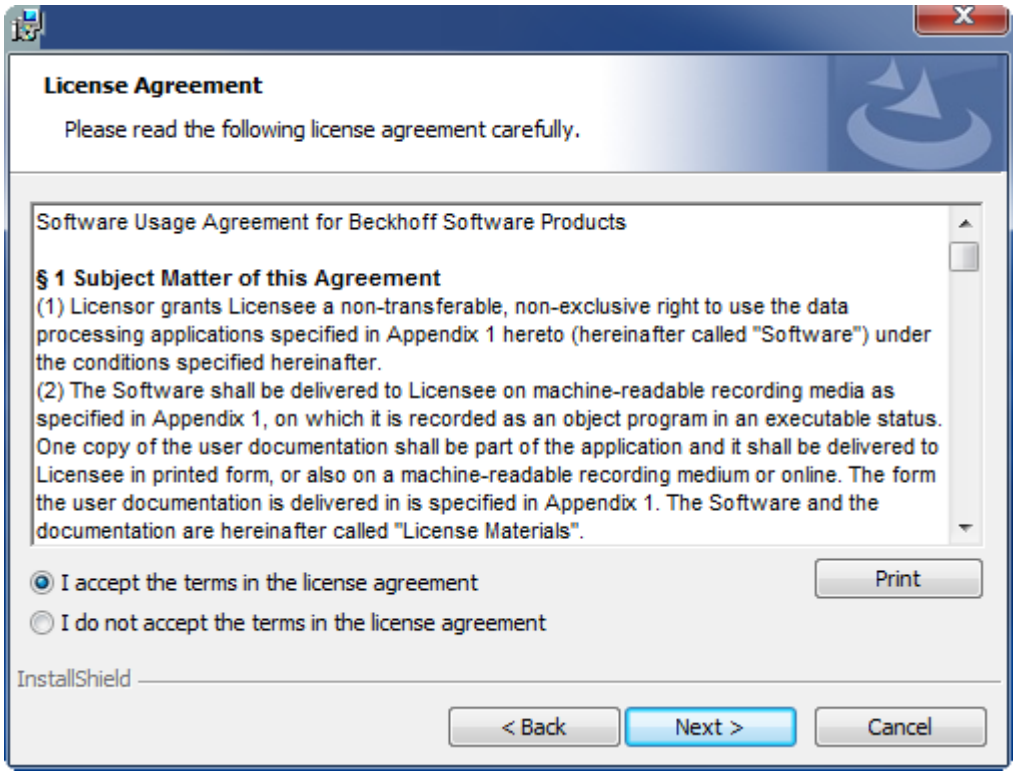
## 3.2 Installation

The following section describes how to install the TwinCAT 3 Function for Windows-based operating systems.

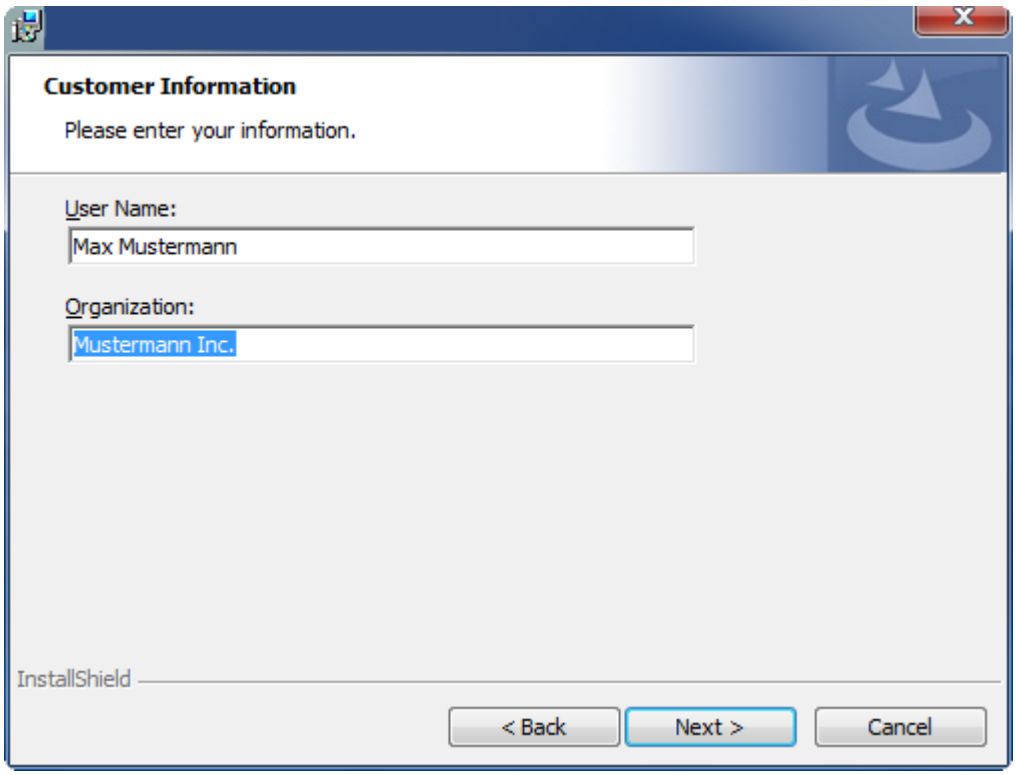
✓ The TwinCAT 3 Function setup file was downloaded from the Beckhoff website.

1. Run the setup file as administrator. To do this, select the command **Run as administrator** in the context menu of the file.
   ⇨ The installation dialog opens.

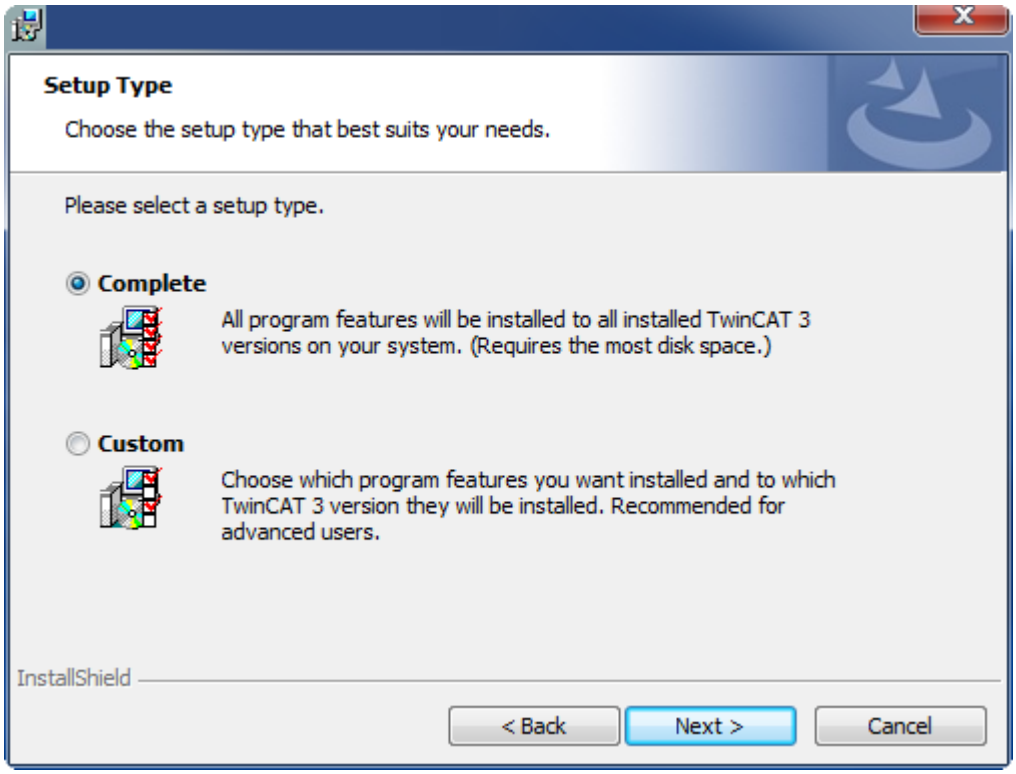
2. Accept the end user licensing agreement and click **Next**.



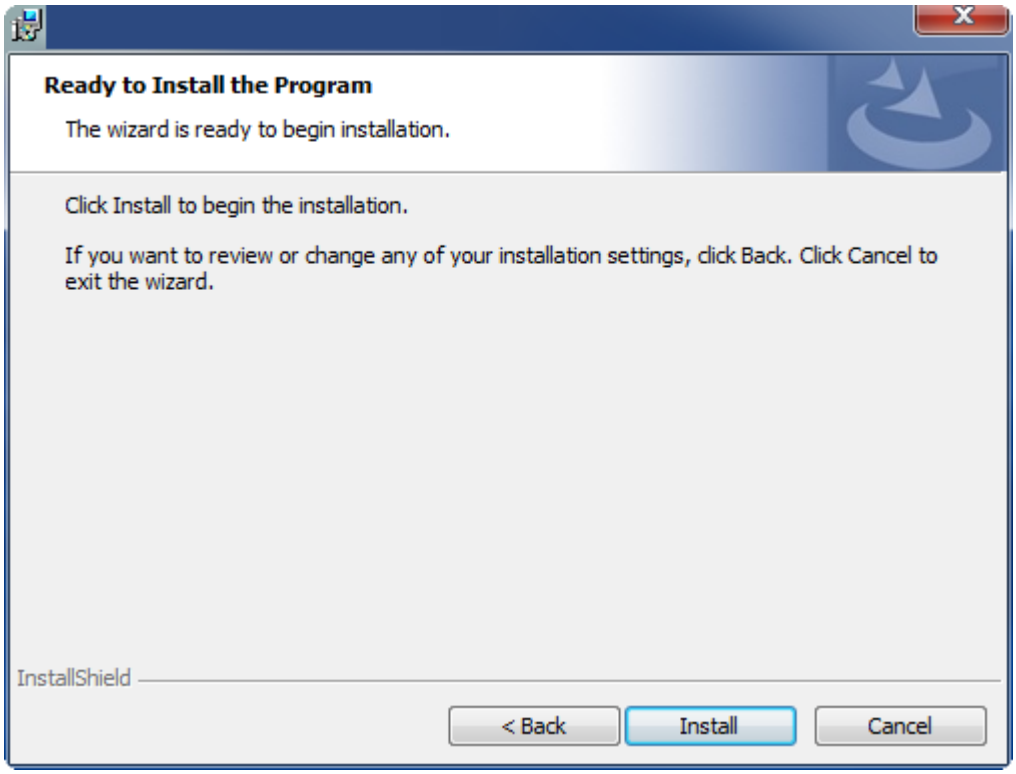

3. Enter your user data.



4. If you want to install the full version of the TwinCAT 3 Function, select **Complete** as installation type. If you want to install the TwinCAT 3 Function components separately, select **Custom**.

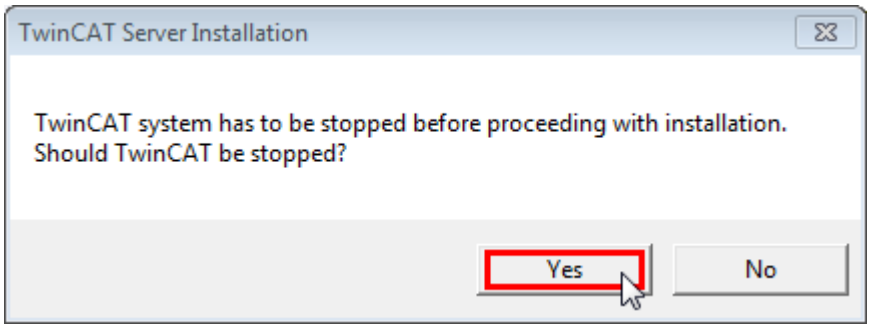



5. Select **Next**, then **Install** to start the installation.

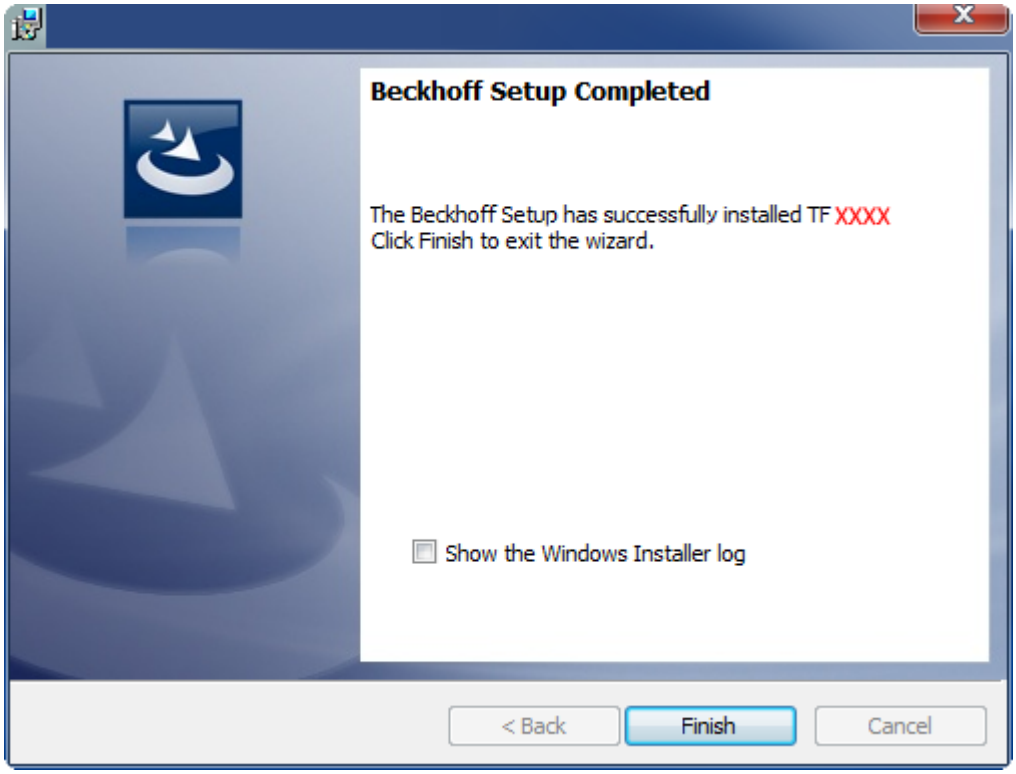



⇨ A dialog box informs you that the TwinCAT system must be stopped to proceed with the installation.

6. Confirm the dialog with **Yes**.




7. Select **Finish** to exit the setup.




⇨ The TwinCAT 3 Function has been successfully installed and can be licensed (see Licensing [▶ 13]).

## 3.3 Licensing

The TwinCAT 3 function can be activated as a full version or as a 7-day test version. Both license types can be activated via the TwinCAT 3 development environment (XAE).

### Licensing the full version of a TwinCAT 3 Function

A description of the procedure to license a full version can be found in the Beckhoff Information System in the documentation "TwinCAT 3 Licensing".

### Licensing the 7-day test version of a TwinCAT 3 Function

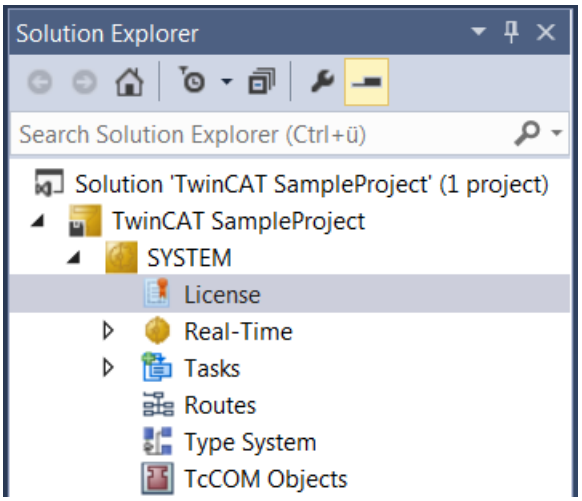
A 7-day test version cannot be enabled for a TwinCAT 3 license dongle.

1. Start the TwinCAT 3 development environment (XAE).
2. Open an existing TwinCAT 3 project or create a new project.

3. If you want to activate the license for a remote device, set the desired target system. To do this, select the target system from the **Choose Target System** drop-down list in the toolbar.

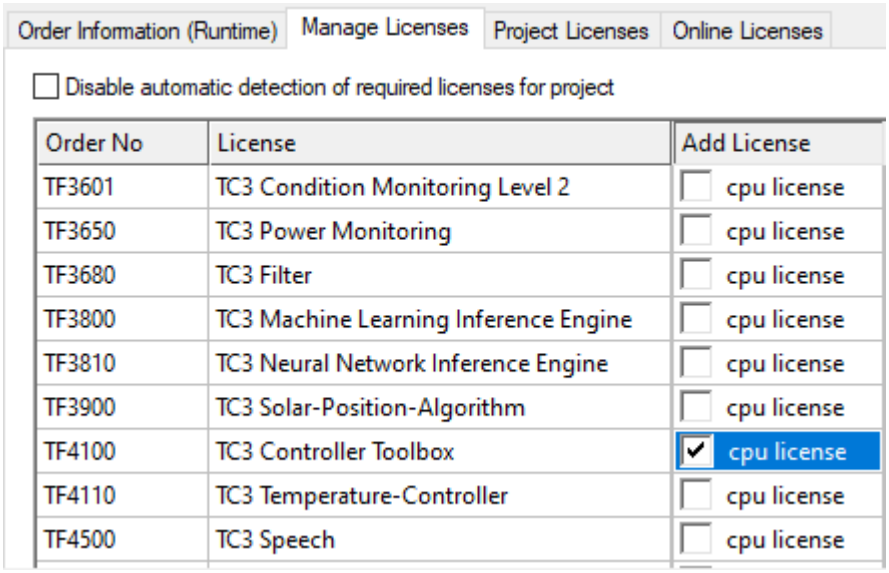
   ⇨ The licensing settings always refer to the selected target system. When the project is activated on the target system, the corresponding TwinCAT 3 licenses are automatically copied to this system.

4. In the **Solution Explorer**, double-click **License** in the **SYSTEM** subtree.




   ⇨ The TwinCAT 3 license manager opens.

5. Open the **Manage Licenses** tab. In the **Add License** column, check the check box for the license you want to add to your project (e.g. "TF4100 TC3 Controller Toolbox").

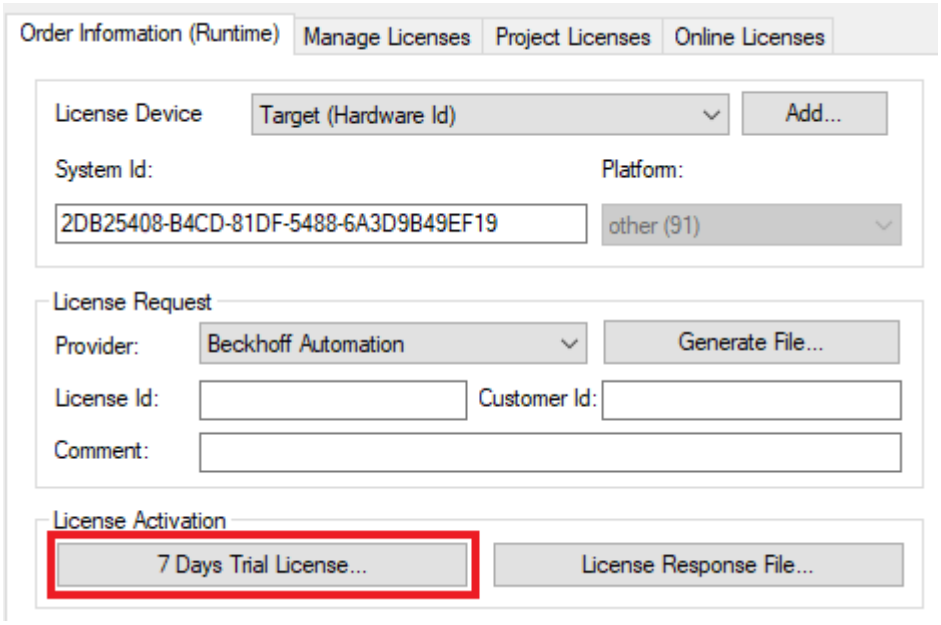



| Order No | License | Add License |
|---|---|---|
| TF3601 | TC3 Condition Monitoring Level 2 | cpu license |
| TF3650 | TC3 Power Monitoring | cpu license |
| TF3680 | TC3 Filter | cpu license |
| TF3800 | TC3 Machine Learning Inference Engine | cpu license |
| TF3810 | TC3 Neural Network Inference Engine | cpu license |
| TF3900 | TC3 Solar-Position-Algorithm | cpu license |
| TF4100 | TC3 Controller Toolbox | cpu license |
| TF4110 | TC3 Temperature-Controller | cpu license |
| TF4500 | TC3 Speech | cpu license |

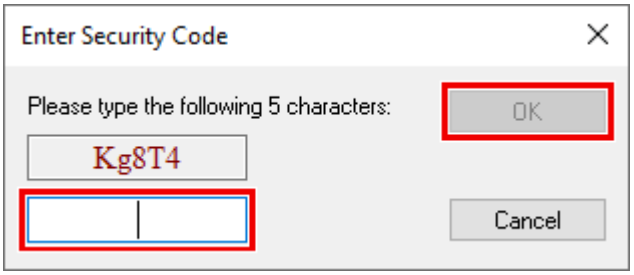
6. Open the **Order Information (Runtime)** tab.

   ⇨ In the tabular overview of licenses, the previously selected license is displayed with the status “missing”**.**

7. Click **7-Day Trial License...** to activate the 7-day trial license.




⇨ A dialog box opens, prompting you to enter the security code displayed in the dialog.




8. Enter the code exactly as it is displayed and confirm the entry.
9. Confirm the subsequent dialog, which indicates the successful activation.
   ⇨ In the tabular overview of licenses, the license status now indicates the expiry date of the license.
10. Restart the TwinCAT system.

⇨ The 7-day trial version is enabled.

# 4 Quick start

### Convert model (optional)

An ML description file (*KerasMLPExample_cos.XML*) is provided in the ZIP of the https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8746884875/.zip. This file can be used directly for integration into TwinCAT, or optionally converted to a binary format (BML).

The method of converting a description file is shown below by way of an example. The same procedure also applies to the more usual case where an ONNX file is to be converted.

✓ The Machine Learning Model Manager [▶ 64] is open.
Menu bar: (Extensions)* > TwinCAT > Machine Learning > Machine Learning Model Manager
* Only with Visual Studio 2019

1. Use the Convert tool.
2. Click **Select file** and select the file *KerasMLPExample_cos.XML*.
3. In the drop-down menu, select **Convert to *.bml**
4. Click **Convert files.**

⇨ The corresponding BML appears under *<TwinCATpath>\Functions\TF38xx-Machine-Learning\ConvertToolFiles*

You can change the default path for saving converted ML description files in the Machine Learning Model Manager with *Select Target Path*. The change is persistent.

You can also convert several files at once by means of multi-selection.
The converted files are saved by default on the XAE system in the folder *<TwinCATpath>\Functions\TF38xx-Machine-Learning\ConvertToolFiles*.

### Integration in TwinCAT via the PLC API

The procedure to load the ML description file into TwinCAT and to run it cyclically is described below. The PLC API [▶ 80] is dealt with first.

- First of all, create a TwinCAT project and a PLC project
- Add the PLC library Tc3_MLL under the References node

In the **Declaration**, please create an instance of FB_MllPrediction. In this simple case, the description file contains an MLP with one input and one output of the type FP32; accordingly, variables for input and output are created as REAL. A more generally accepted possibility to handle the inputs and outputs can be found in the Samples for PLC API [▶ 93].

In addition, create a string variable that contains the file name incl. path to the ML description file (path on the target system). Copy the corresponding file to this location on the target system (FTP, RDP, Shared Folder, …).
Further information on this step can be found here [▶ 75].

**NOTICE**

**Path of the description file on the target system**

Pay attention to the settings of the File Writer and the writing rights on the target system.

```
PROGRAM MAIN
VAR
   fbPredict   :  FB_MllPrediction;      // ML Interface
   nInputDim, nOutputDim : UDINT := 1;
   fDataIn, fDataOut : REAL;
   sModelName    : T_MaxString  := 'C:/TwinCAT/3.1/Boot/ML_Boot/KerasMLPExample_cos.xml';
   hrErrorCode   : HRESULT;
   bLoadModel    : BOOL;
   nState        : INT := 0;
END_VAR
```

In the **Implementation part** you create a state machine, for example, which enables you to switch between the Idle state, Config state, Predict state and Error state.

In the first state you initially wait for the command to load a description file. Subsequently, the Configure method [▶ 84] of the `fbPredict` function block is called until `TRUE` is returned. This is present for one cycle and means that the configuration has been completed. A check is then done to establish whether an error occurred. If no error occurred, the state is switched to the next state, which is the Predict state. The state machine remains in the Predict state as long as no error occurs or a (new) model is to be loaded.

```
CASE nState OF
    0: // idle state
      IF bLoadModel THEN
         bLoadModel := FALSE;
         nState := 10;
      END_IF
   10: // Config state
      fbPredict.stPredictionParameter.MlModelFilepath := sModelName; // provide model path and name
      IF fbPredict.Configure() THEN // load model
         IF fbPredict.bError THEN
            nState := 999;
            hrErrorCode := fbPredict.hrErrorCode;
         ELSE // no error -> proceed to predict state
            nState := 20;
         END_IF
      END_IF
   20: // Predict state
      fbPredict.Predict(
         pDataInp := ADR(fDataIn),
         nDataInpDim := nInputDim,
         fmtDataInpType := ETcMllDataType.E_MLLDT_FP32_REAL,
         pDataOut := ADR(fDataOut),
         nDataOutDim := nOutputDim,
         fmtDataOutType := ETcMllDataType.E_MLLDT_FP32_REAL,
         nEngineId := 0,
         nConcurrencyId := 0);

      IF fbPredict.bError THEN // error handling
         nState := 999;
         hrErrorCode := fbPredict.hrErrorCode;
      ELSIF bLoadModel THEN // load (updated) model
         bLoadModel := FALSE;
         nState := 10;
      END_IF;
   999: // Error state
       // add error handling here
END_CASE
```

The Predict method [▶ 89] of `fbPredict` is used in the Predict state. This runs the loaded model. The method is informed of the input variables via the first three parameters of the method – pointer to a PLC variable, number of inputs and associated data type. The same is to be specified for the output variables (parameters 4 to 6). `nEngineId` and `nConcurrencyId` are not required in this simple example and are always transferred with the value zero. Details for these parameters can be found in the samples Detailed example [▶ 93] and Parallel, non-blocking access to an inference module [▶ 93].

Before activating the project on a target, you must select the TF3810 license **manually** on the Manage Licenses tab under System>License in the project tree, as you wish to load a multi-layer perceptron (MLP).

You can now activate the configuration. Log into the PLC and start the program. By setting the `bLoadModel` variable in the online view to `TRUE`, the model is now loaded and begins with the prediction. You can manipulate the input variable `fDataIn` and view the results in the output `fDataOut`. The multi-layer perceptron loaded approximately maps a cosine function in the input range of [-pi, pi] to the value range [-1,1]. Outside of the range [-pi, pi] the function increasingly diverges from the cosine function.

TwinCAT_Project5.Untitled1.MAIN

| Expression | Type | Value |
|---|---|---|
| fbPredict | FB_MllPrediction | |
| nInputDim | UDINT | 1 |
| nOutputDim | UDINT | 1 |
| fDataIn | REAL | 3.14 |
| fDataOut | REAL | -1.018022 |
| sModelName | T_MaxString | 'C:/TwinCAT/3.1... |
| hrErrorCode | HRESULT | 16#00000000 |
| bLoadModel | BOOL | FALSE |
| nState | INT | 20 |

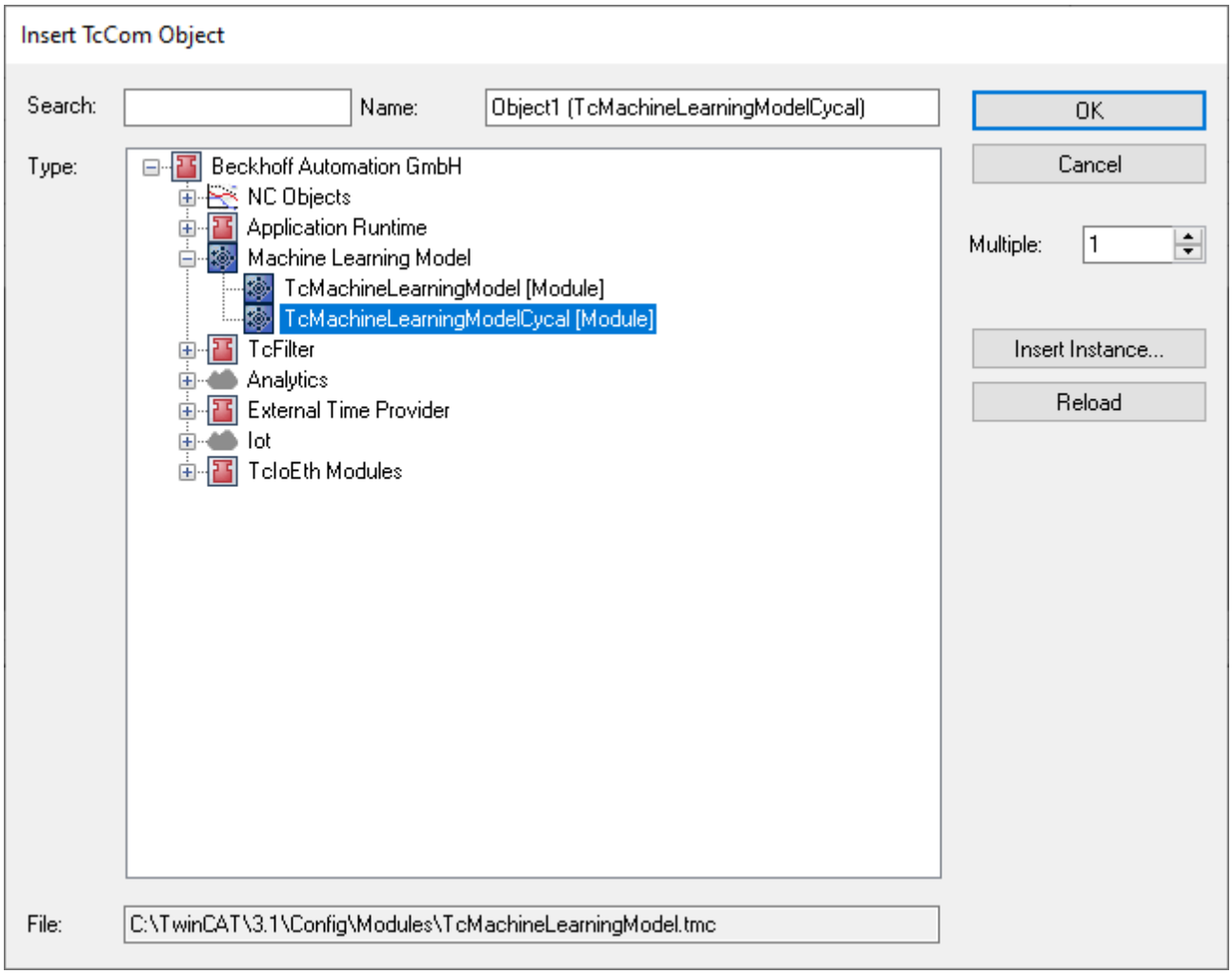
You can download the sample described above here [▶ 93].

### Incorporation of a model by means of TcCOM object

This section deals with the execution of machine learning models by means of a prepared TcCOM object. A detailed description can be found here [▶ 78]. This interface offers a simple and clear way of loading models, executing them in real-time and generating appropriate links in your own application by means of the process image.

### Generate a prepared TcCOM object TcMachineLearningModelCycal

1. To do this, select the node **TcCOM Objects** with the right mouse button and select **Add New Item…**

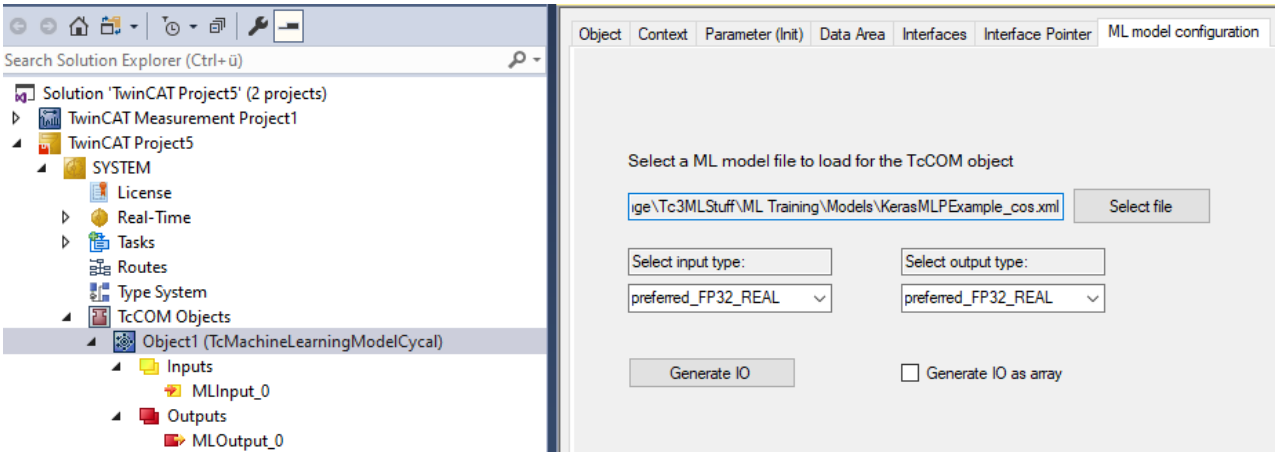



**Under Tasks, generate a new TwinCAT task and assign this task context to the newly generated instance of TcMachineLearningModelCycal**

2. To do this, open the **Context** tab of the generated object.
3. Select your generated task in the drop-down menu.

⇨ The instance of TcMachineLearningModelCycal has a tab called **ML model configuration** where you can load the description file of the ML algorithm (XML or BML) and the available data types for the inputs and outputs of the selected model are then displayed.

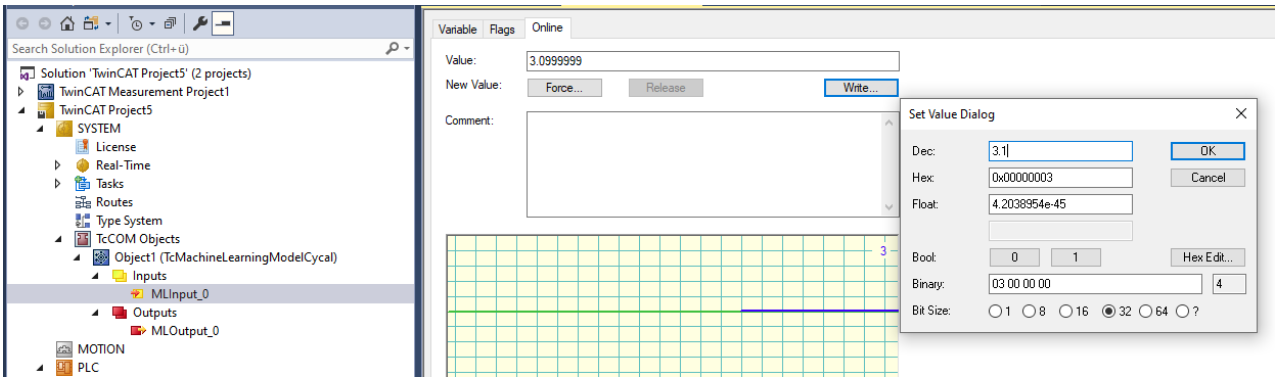
- The file does not have to be on the target system. It can be selected from the development system and is then loaded to the target system on activating the configuration.
  - A distinction is made between preferred and supported data types. The only difference is that a conversion of the data type takes place at runtime if a non-preferred type is selected. This may lead to slight losses in performance when using non-preferred data types.
- The data types for inputs and outputs are initially set automatically to the preferred data types. The process image of the selected model is created by clicking **Generate IO**. Accordingly, by loading *KerasMLPExample_cos.xml*, you get a process image with an input of the type REAL and an output of the type REAL.




### Activating the project on the target

1. Before activating the project on a target, you must select the TF3810 license manually on the **Manage Licenses** tab under System>License in the project tree, as you wish to load a multi-layer perceptron (MLP).
2. Activate the configuration.

⇨ You can now test the model by manually writing at the input.



# 5 Machine Learning Models and file formats

This chapter provides an overview of supported Machine Learning models [▶ 20] and lists guiding principles for the selection of a suitable model [▶ 58]. In addition, it contains for each algorithm an example-based description of how you can export trained models from a Python environment as an ONNX file [▶ 60]. The exported ONNX file must be converted into a TwinCAT-specific XML or BML file [▶ 63]. For this purpose, several interfaces to a converter are available (CLI, API and GUI), with which the file management process can be integrated into your existing software landscapes.

## 5.1 Machine learning models supported

The table below lists all supported model types, including the required license and software version.

**Selecting a model**

An introduction explaining which criteria you should consider when selecting a model can be found here: Machine Learning Cheat Sheet: selection of models [▶ 58].

**ONNX export for supported model types**

Python examples of the ONNX export from different frameworks are given for all supported model types in the section Samples of ONNX export [▶ 61].

**Required license for supported model types**

The required TwinCAT license differs depending on the model type that is loaded into the Machine Learning Runtime. Note that the TF3810 license contains the TF3800 license, which means that if the TF3810 license is valid, all models that require a TF3800 or TF3810 license can be loaded.

**Supported models**

For licensing, refer also to: System requirements [▶ 10].

| Model type | License | Available from setup version |
|---|---|---|
| Support vector machine [▶ 27] (SVM) | TF3800 | 3.1.42.0 |
| Principal Component Analysis [▶ 34] (PCA) | TF3800 | 3.1.57.0 |
| k-Means [▶ 33] | TF3800 | 3.1.57.0 |
| Random Forest [▶ 43] | TF3800 | 3.1.58.0 |
| Multi-layer perceptron [▶ 20] (MLP) | TF3810 | 3.1.42.0 |
| Decision Tree [▶ 36] | TF3800 | 3.1.62.0 |
| Extra Tree [▶ 38] | TF3800 | 3.1.62.0 |
| Extra Trees [▶ 41] | TF3800 | 3.1.62.0 |
| Gradient Boosting [▶ 46] | TF3800 | 3.1.62.0 |
| Hist Gradient Boosting [▶ 48] | TF3800 | 3.1.62.0 |
| XGBoost [▶ 50] | TF3800 | 3.1.62.0 |
| LightGBM [▶ 54] | TF3800 | 3.1.62.0 |

### 5.1.1 Multi-layer perceptron

A multi-layer perceptron (MLP) can be used both for classification and for regression [▶ 60]. The basic idea of an MLP is the linking of the smallest units – so-called neurons – in a network. Each neuron takes up information from previous neurons or directly via model inputs and processes it. A directional flow of information takes place through this network from inputs to outputs.

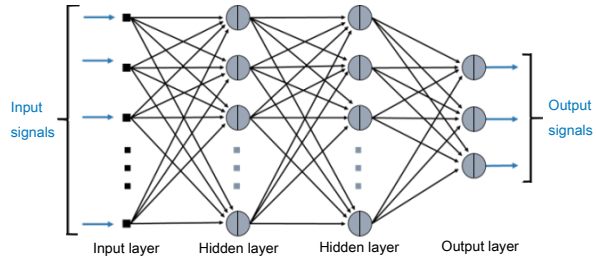
A neuron processes its inputs **x** as a weighted sum plus an ordinate value and transforms the intermediate result with an activation function.

$$y = f_{\text{act}}(\mathbf{wx} + w_0)$$

Neurons are usually arranged in layers, which are then linked one after the other. If a network has more than one layer between inputs and outputs, then it is referred to as a multi-layer perceptron.

The structure is illustrated in the figure below.

- **Input layer:** Has no neurons of its own. Serves as an input layer and defines the number and data type of the inputs.
- **Hidden layer:** Layer with its own neurons. The layer is characterized by the number of neurons as well as the selected activation function. Any number of hidden layers can be layered one after the other.
- **Output layer:** Layer with its own neurons. The number of neurons and their activation function are oriented to the application to be implemented.




### Supported properties

### ONNX support

The following ONNX operators are supported:

- MatMul (matrix multiplication) followed by ADD (add)
- GEMM (general matrix multiplication)

In addition, the following activation functions are supported:

| Activation function | Description |
|---|---|
| tanh | Hyperbolic tangent (-1.1) |
| sigmoid | Sigmoid function – an exponential function (0.1) |
| softmax | Softmax – a normalized exponential function – often used for classification (0.1) |
| sine | Sine function (-1.1) |
| cosine | Cosine function (-1.1) |
| relu | "Rectifier" – positive portion is linear – good learning properties in case of deep networks (0, inf) |
| abs | Absolute value of the input (0, inf) |
| linear/id | Linear identity – simple linear function f(x) = x (-inf, inf) |
| exp | A simple exponential function e^(x) (0, inf) |
| logsoftmax | Logarithm of softmax – sometimes more efficient than softmax in the calculation (-inf, inf) |
| sign | Sign function (-1.1) |
| softplus | Sometimes better than relu due to the differentiability (0, inf) |
| softsign | Conditionally better convergence behavior than tanh (-1.1) |

Samples of the ONNX support for MLPs from Pytorch, Keras and Scikit-learn can be found here: ONNX export of an MLP [▶ 22].

### Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 32 (E_MLLDT_FP32-REAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

**Further comments**

There are no limits on the software side with regard to the number of layers or the number of neurons. With regard to the calculation duration and memory requirement, the limits of the employed hardware are to be observed.

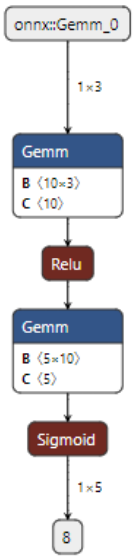
#### 5.1.1.1 ONNX export of an MLP

**Download Python samples**

A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

**MLP Regressor with PyTorch**

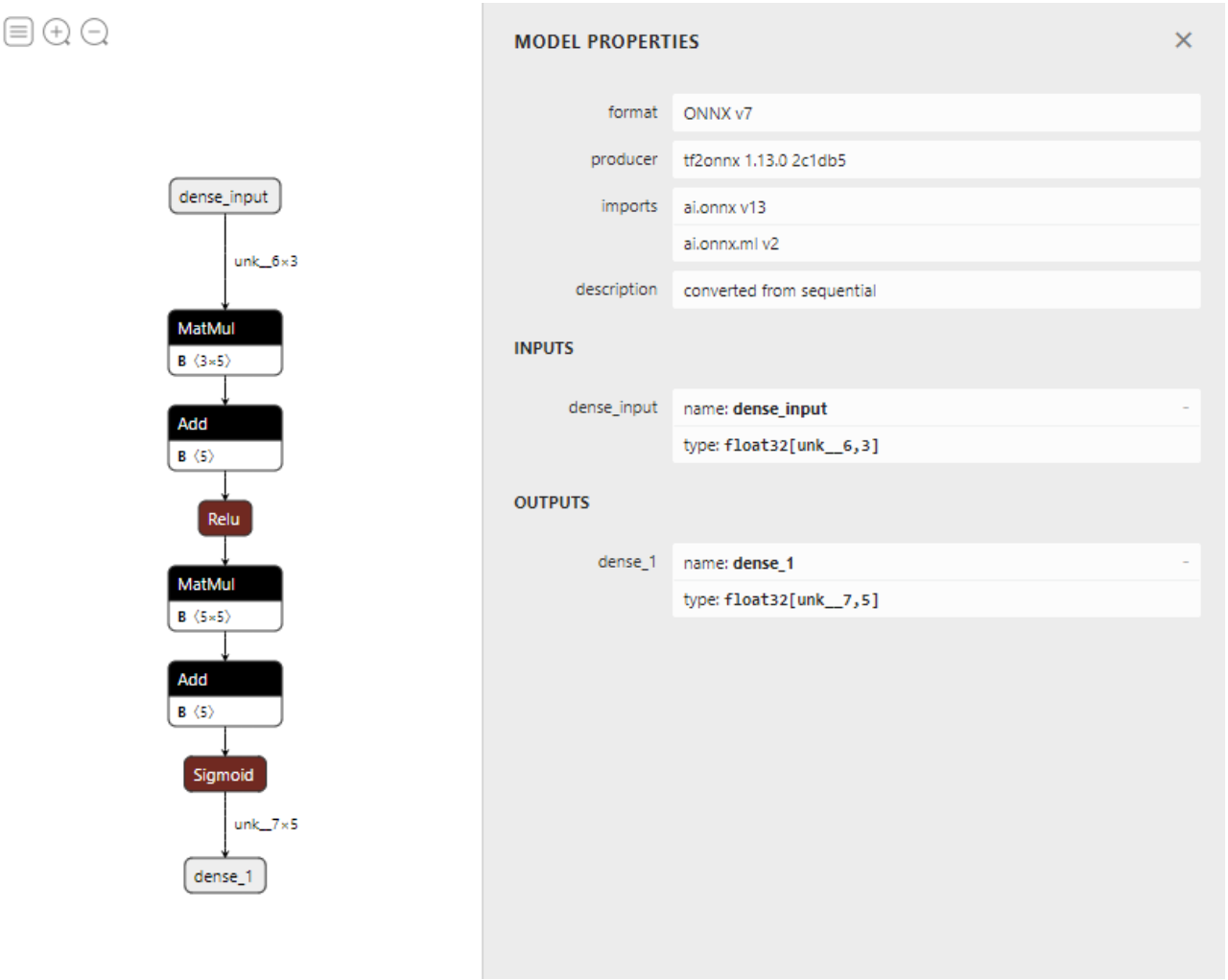
```
import torch
import numpy as np

input_dim = 3
output_dim = 5
n_samples = 100
dummy_input = np.random.random((n_samples, input_dim))
dummy_output = np.random.random((n_samples, output_dim))
tensor_in = torch.FloatTensor(dummy_input)
tensor_out = torch.FloatTensor(dummy_output)
train_dataset = torch.utils.data.TensorDataset(tensor_in, tensor_out)
train_loader = torch.utils.data.DataLoader(train_dataset, shuffle=True, batch_size=32)
class MLP_Net(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = torch.nn.Linear(input_dim, 10)
        self.fc2 = torch.nn.Linear(10, output_dim)

    def forward(self, x):
        x = torch.nn.functional.relu(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        return x
mlp_net = MLP_Net()
optimizer = torch.optim.Adam(mlp_net.parameters(), lr =0.001)
n_epochs = 5
for epoch in range(n_epochs):
    for batch in train_loader:
        input, output = batch
        mlp_net.zero_grad()
        pred_out = mlp_net(input)
        criterion = torch.nn.MSELoss()
        loss = criterion(pred_out, output)
        loss.backward()
        optimizer.step()
onnx_file = 'pytorch_mlp.onnx'
tensor_input_size = torch.FloatTensor(np.random.random((1,input_dim))) # First dimension must be 1
torch.onnx.export(mlp_net, tensor_input_size, onnx_file, verbose=True)
```





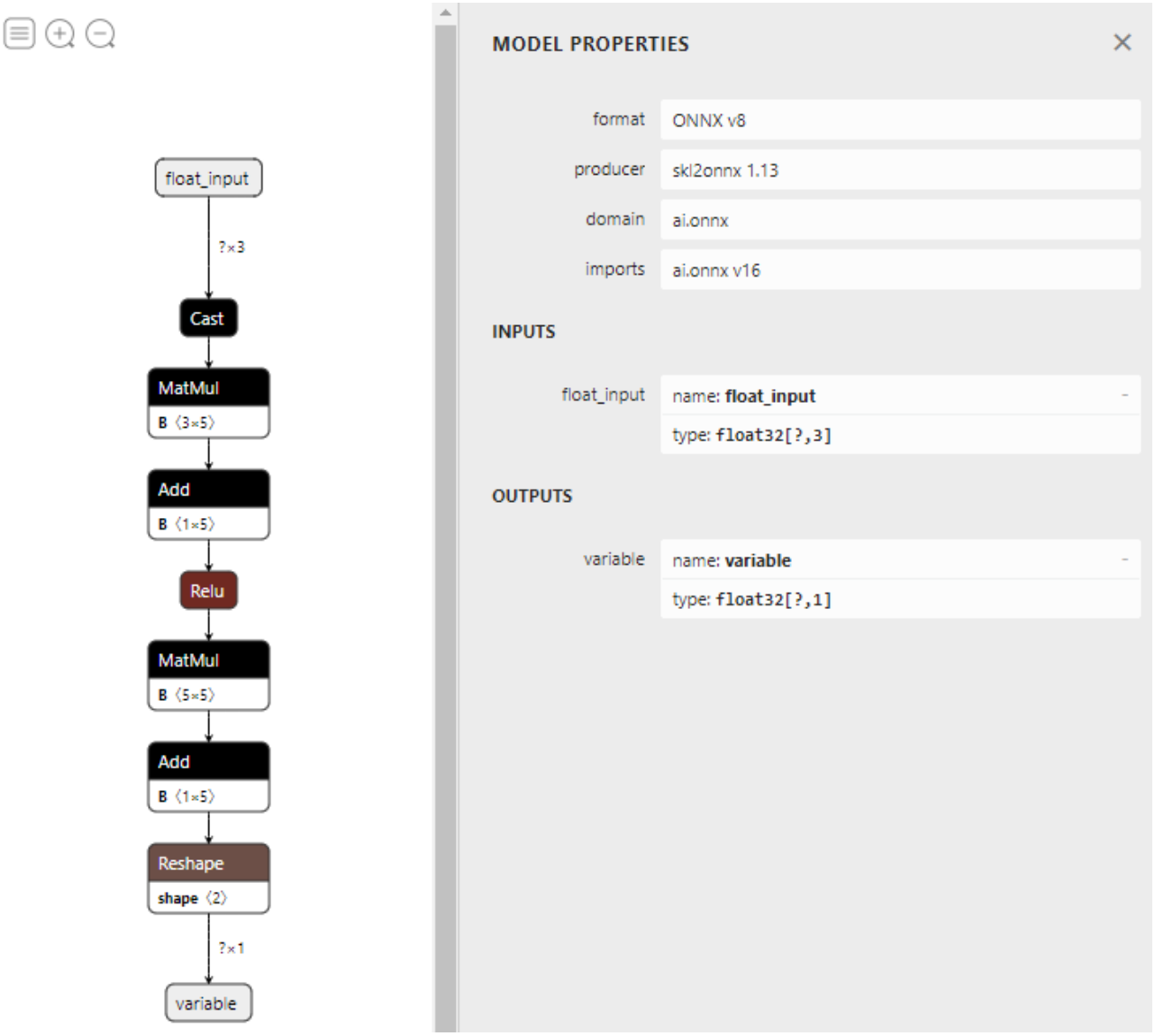
### MLP Regressor with Keras

```
import tensorflow as tf
import numpy as np
import tf2onnx

input_dim = 3
output_dim = 5
n_samples = 100

dummy_input = np.random.random((n_samples, input_dim,))
dummy_output = np.random.random((n_samples, output_dim))

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(5, input_shape = (input_dim,), activation=tf.keras.activations.relu,
 use_bias = True))
model.add(tf.keras.layers.Dense(output_dim, activation='sigmoid'))
model.build()
model.summary()
learning_rate = 0.001
loss = 'mean_squared_error'
optimizer = tf.keras.optimizers.Adamax(lr=learning_rate)
model.compile(optimizer=optimizer, loss=loss)
model.fit(x=dummy_input, y=dummy_output, batch_size=32 ,epochs=5,verbose=1, shuffle=True)

filename = 'tf_keras_mlp'
onnx_model, _ = tf2onnx.convert.from_keras(model)
with open(filename+'.onnx','wb') as f:
    f.write(onnx_model.SerializeToString())
```




### MLP Regressor with Scikit-learn

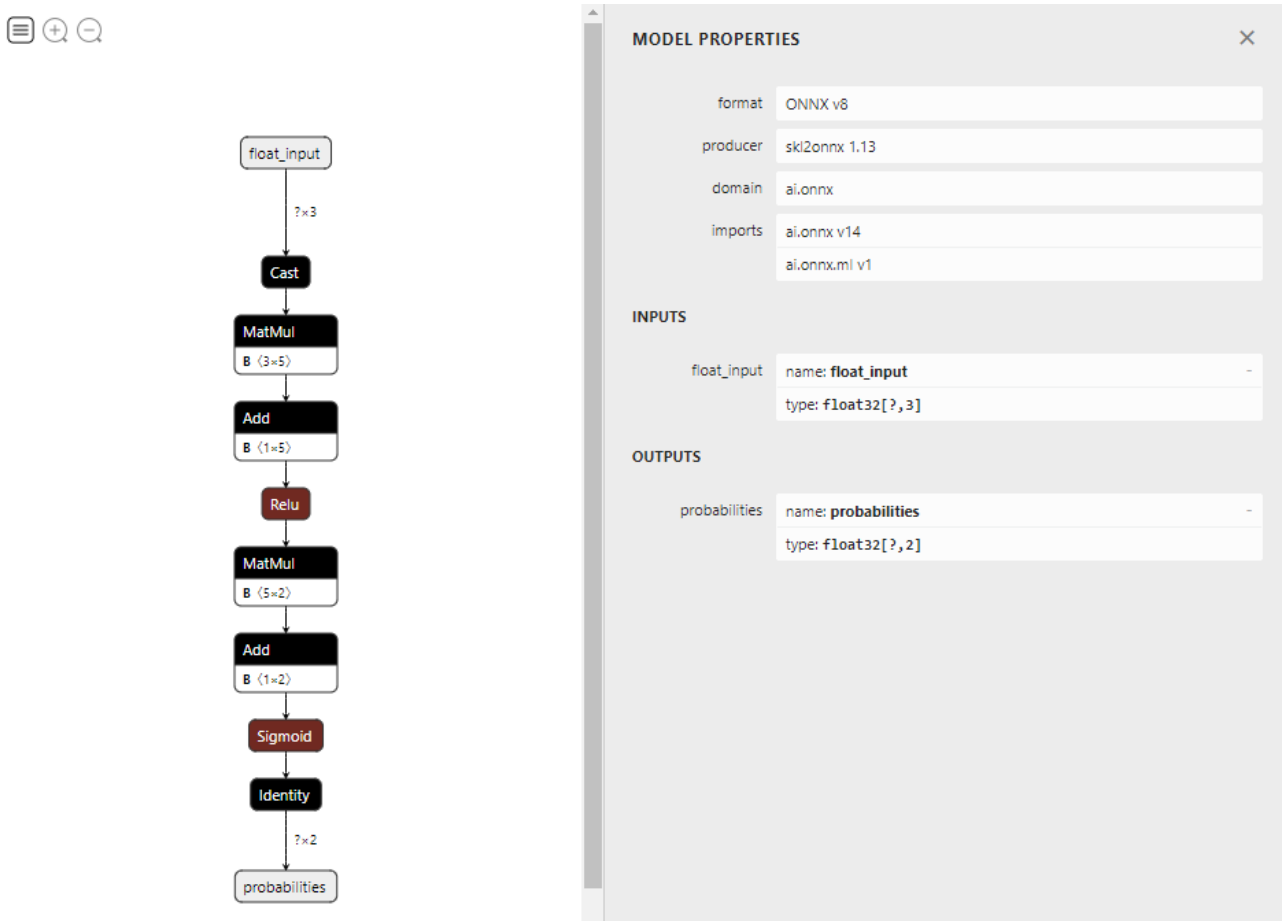
```
import sklearn.neural_network as skl
import numpy as np

input_dim = 3
output_dim = 5
n_samples = 100
dummy_input = np.random.random((n_samples,input_dim))
dummy_output = np.random.random((n_samples,output_dim))

model = skl.MLPRegressor(hidden_layer_sizes =(5), activation ='relu')
model.fit(dummy_input,dummy_output)

filename = 'skl_relu_reg'
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType
initial_type = [('float_input',FloatTensorType([None,input_dim]))]

onx = convert_sklearn(model,initial_types=initial_type)
with open(filename+'.onnx','wb') as f:
    f.write(onx.SerializeToString())
```



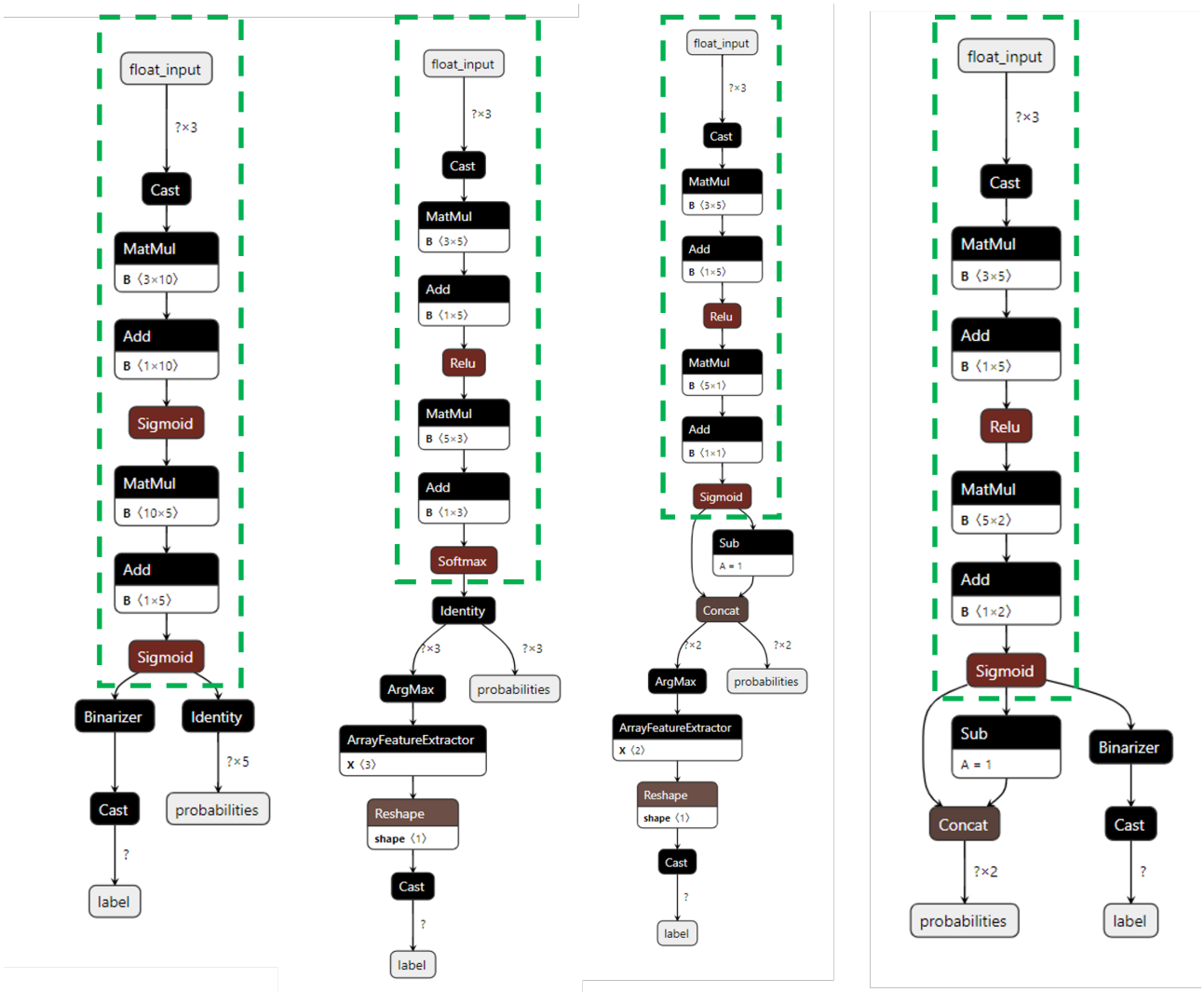

### MLP Classifier with Scikit-learn

```
import sklearn.neural_network as skl
import numpy as np
import onnx
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

def modify_onnx_MLPClassifier(onnx_MLPCLassifier):
    """ Function to modify onnx model from MLPClassifier to make it suitable for TwinCAT Machine Lea
rning
    The function removes unsupported nodes and uses the probability estimates of the classes as outp
ut for
    the model.
    The output of the modified onnx model is the same as the output of the predict_proba() method fr
om
    sklearn.neural_network.MLPClassifier.
    To get the class as an integer output either a binarization or an argmax function must be applie
d to the
    model output.
    """
    # Delete nodes after last sigmoid/softmax layer
    output_node_types = {'Sigmoid', 'Softmax'}
    len_onx_init = len(onnx_MLPCLassifier.graph.node)
    erased_node_inputs = []
    for idx, node in enumerate(reversed(onnx_MLPCLassifier.graph.node)):
        if node.op_type in output_node_types:
            idx_last_node = len_onx_init - idx -1
            break
        else:
            for idx, input in enumerate(node.input):
```

```
                erased_node_inputs.append(input)
            onnx_MLPCLassifier.graph.node.remove(node)
    # Get output dimension and clean initializers
    second_last_node = onnx_MLPCLassifier.graph.node[idx_last_node-1]
    for initializer in onnx_MLPCLassifier.graph.initializer:
        if initializer.name in second_last_node.input:
            output_dim = initializer.dims[1]
        if initializer.name in erased_node_inputs:
            onnx_MLPCLassifier.graph.initializer.remove(initializer)
    # Erase original outputs and create new output
    for output in reversed(onnx_MLPCLassifier.graph.output):
        onnx_MLPCLassifier.graph.output.remove(output)
    name_new_output = "probabilities"
    newOutput = onnx.helper.make_tensor_value_info(name_new_output, onnx.TensorProto.FLOAT, shape=(N
one, output_dim))
    onnx_MLPCLassifier.graph.output.append(newOutput)
    # Create new node and connect to new output
    last_node_output = onnx_MLPCLassifier.graph.node[idx_last_node].output
    new_node = onnx.helper.make_node("Identity", last_node_output, [name_new_output], "Identity_Out"
)
    onnx_MLPCLassifier.graph.node.append(new_node)
    return onnx_MLPCLassifier

input_dim = 3
output_dim = 2
n_samples = 100
dummy_input = np.random.random((n_samples,input_dim))
dummy_output = np.random.randint(2, size=(n_samples, output_dim))
model = skl.MLPClassifier(activation ='relu', hidden_layer_sizes=(5))
model.fit(dummy_input,dummy_output)
filename = 'skl_mlp_clf'
initial_type = [('float_input',FloatTensorType([None,input_dim]))]
onx = convert_sklearn(model,initial_types=initial_type, options={type(model): {'zipmap': False}})
onx = modify_onnx_MLPClassifier(onx)
with open(filename+'.onnx','wb') as f:
    f.write(onx.SerializeToString())
```



Observe the function modify_onnx_MLPClassifier. This modifies the lower part of the ONNX graph so that only operators supported by the TwinCAT Neural Network Inference Engine are used. Without this modification, the operators shown here will be generated (depending on the dimensionality of the problem). Only the area with the green border is supported.




## 5.1.2 Support vector machine

A support vector machine (SVM) can be used both for classification and for regression [▶ 60]. The SVM is a frequently used tool in particular with regard to classification tasks.

The fundamental goal of an SVM is to find a hyperplane in an N-dimensional space, wherein the distance between the closest data point and the plane is maximized. A hyperplane can only separate the space linearly (also called linear SVM). A non-linear separation is also possible by means of a so-called kernel trick (also called kernel SVM). The N-dimensional space is transformed into a higher-dimensional space here. A linear separation with a hyperplane is possible in an accordingly higher-dimensional space.

If a distinction needs to be made between several classes, several support vector machines are generated internally and classification takes place by means of comparisons. A one-class SVM can also be trained and used for anomaly detection.

### Supported properties

### ONNX support

The following ONNX operators are supported:

- SVMRegressor
- SVMClassifier

Supported kernel functions are listed in the following table:

| Kernel function | Description |
|---|---|
| Linear | $K(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{x}_1^{\mathrm{T}}\mathbf{x}_2$ |
| Radial Basis Function (RBF) | $K(\mathbf{x}_1\mathbf{x}_2) = \exp(-\gamma \left\|\mathbf{x}_1 - \mathbf{x}_2\right\|^2)$ |
| Sigmoid | $K(\mathbf{x}_1, \mathbf{x}_2) = \tanh(a\mathbf{x}_1^{\mathrm{T}}\mathbf{x}_2 + b)$ |
| Polynomial | $K(\mathbf{x}_1, \mathbf{x}_2) = (\mathbf{x}_1^{\mathrm{T}}\mathbf{x}_2 + 1)^d$ |

For samples of the export of SVMs as ONNX, see ONNX export of an SVM [▶ 28].

**Classification limitation**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

**Supported data types**

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

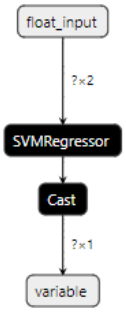
### 5.1.2.1 ONNX export of a SVM

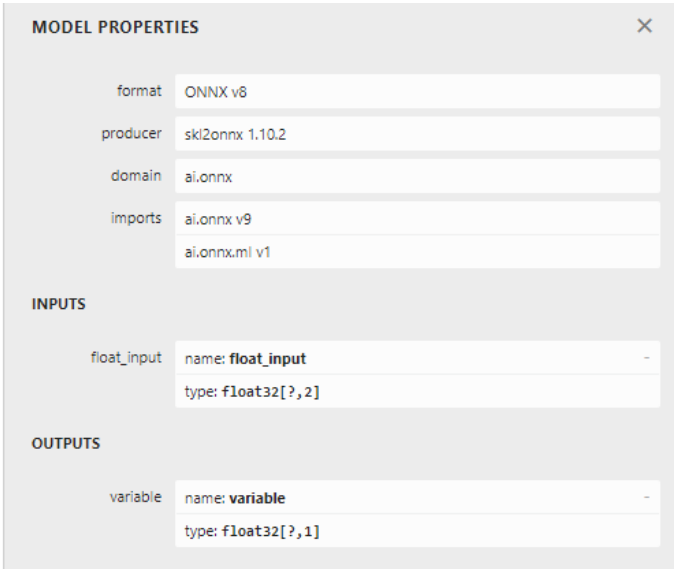
**Download Python samples**

A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

**SVM Regressor with Scikit-learn**

```
from sklearn.svm import SVR
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType


X = [[13,3],[1,16],[1,2]]
Y = [1.0,2.0,3.0]

model = SVR(kernel='rbf',gamma=10)
model.fit(X,Y)

out = model.predict(X)

input_type = [('float_input', FloatTensorType([None,len(X[0])]))]

onnx_filename = 'svr-rbf.onnx'
onx = convert_sklearn(model,initial_types=input_type)
with open(onnx_filename,"wb") as f:
    f.write(onx.SerializeToString())
```

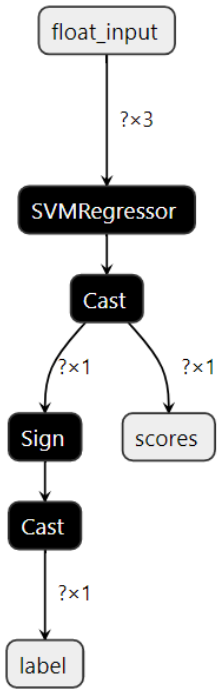


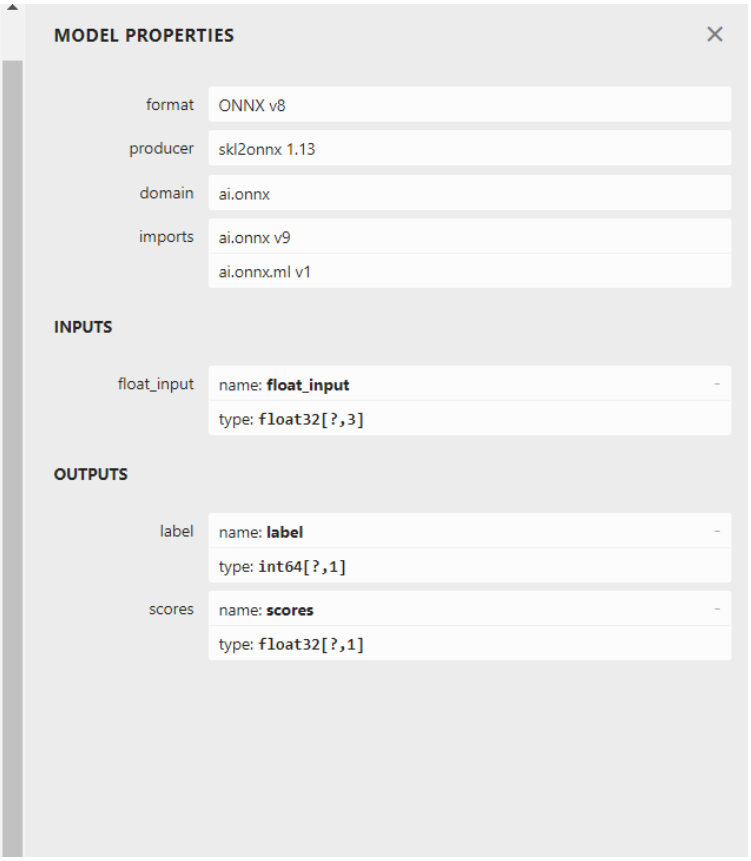




### SVM Classifier with Scikit-learn

*One-class SVM*

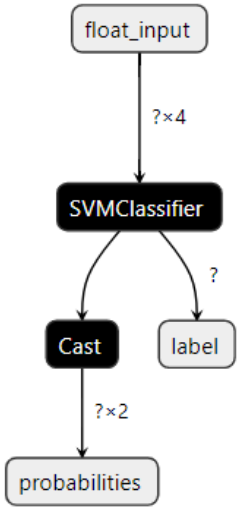
```
from sklearn.datasets import make_blobs
from sklearn.svm import OneClassSVM
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

n_samples = 150
input_dim = 3

X, _ = make_blobs(n_samples=n_samples, n_features=input_dim, centers=1, cluster_std=0.3, shuffle=Tru
e, random_state=42, )
svm = OneClassSVM(kernel='rbf', nu=0.3)
svm.fit(X)
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_sklearn(svm, initial_types=initial_type)
filename = 'skl_oneclass_svm'
with open(filename + '.onnx', "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
```

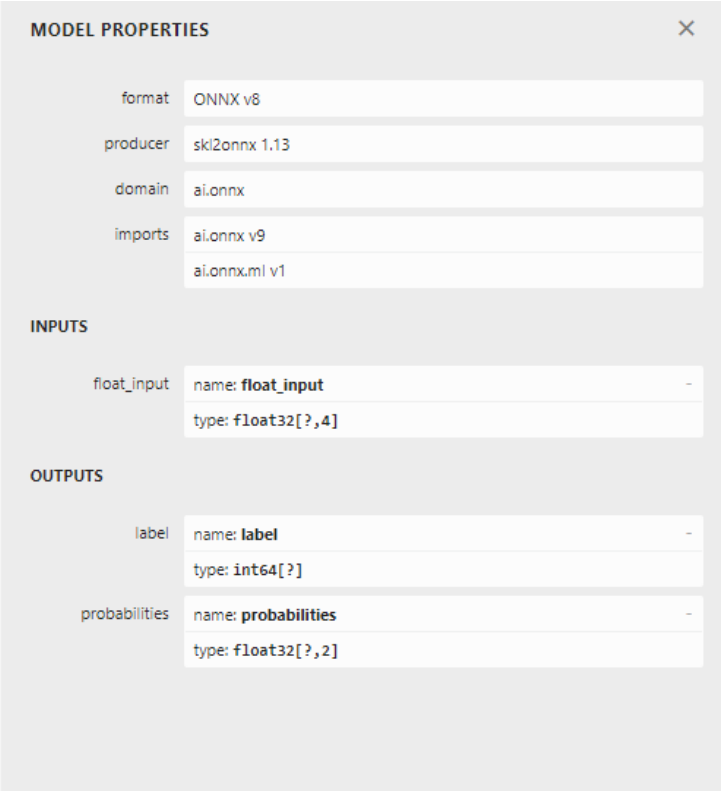



*Binary classification*

```
from sklearn.svm import SVC
import numpy as np

# random dataset
n_samples = 150
input_dim = 4
n_classes = 2 # binary classification

rand_in = np.random.random((n_samples, input_dim,))
rand_singleout_multiclass = np.random.randint(n_classes, size=(n_samples, 1))

X = rand_in
y = rand_singleout_multiclass

# train SVC
clr = SVC(kernel='linear', gamma=10) # decision_function_shape option is ignored for binary classifi
cation
clr.fit(X, y)

# Convert into ONNX format
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]

# # Zipmap should be always turned off as it's not implemented in TF3800
onx = convert_sklearn(clr, initial_types=initial_type, options={type(clr): {'zipmap':False}})
with open("svc_random.onnx", "wb") as f:
    f.write(onx.SerializeToString())
```




*Multi-class classification*

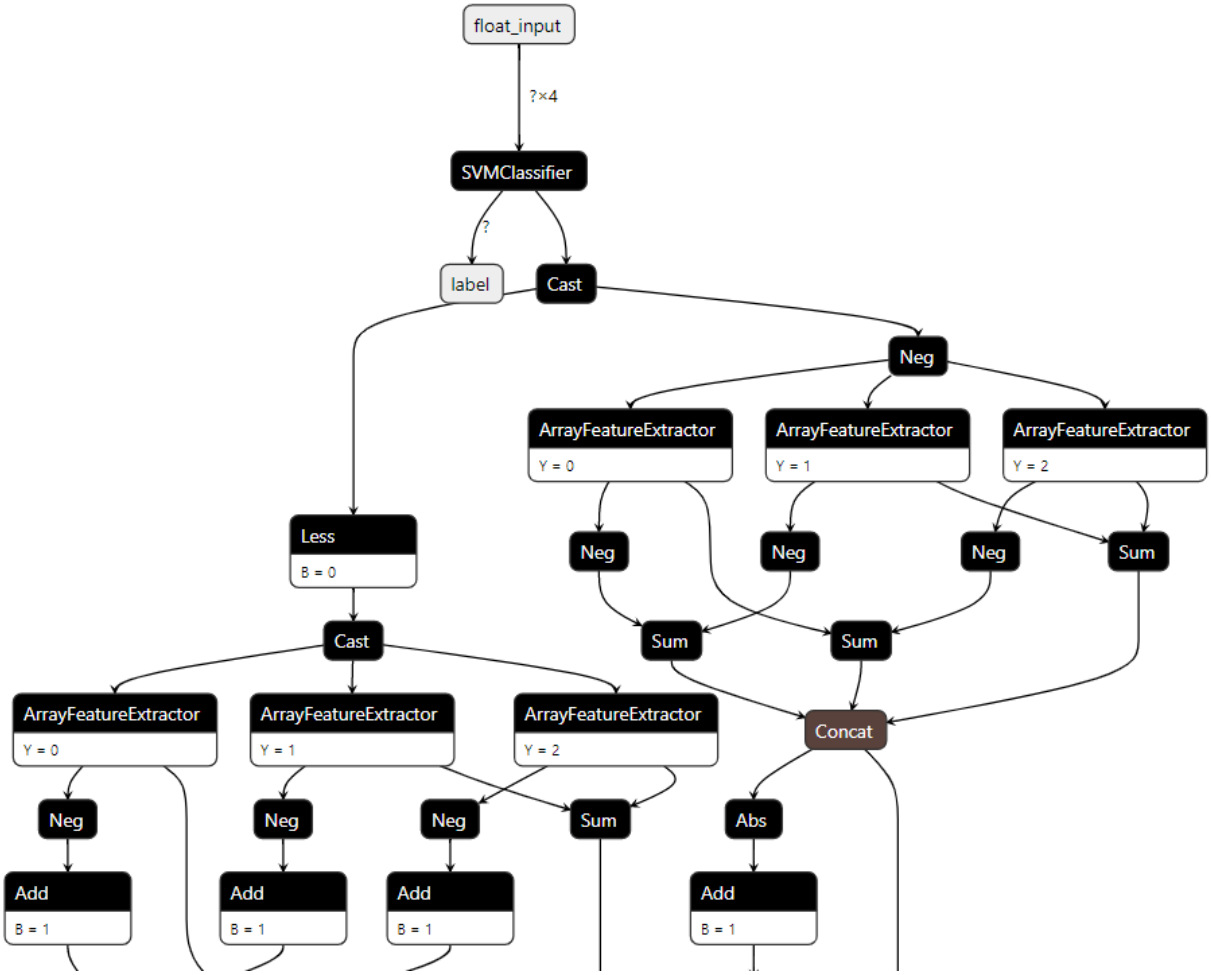
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

# data set
iris = load_iris()
X, y = iris.data, iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y)

# train SVC, note that decision_function_shape ovo is mandatory
clr = SVC(kernel='linear', gamma=10,decision_function_shape='ovo')
clr.fit(X_train, y_train)

# Convert into ONNX format
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

initial_type = [('float_input', FloatTensorType([None, 4]))]

onx = convert_sklearn(clr, initial_types=initial_type)
with open("svc_iris.onnx", "wb") as f:
    f.write(onx.SerializeToString())
```

**Observe the shape of the ONNX graph!**

The decision_function_shape “ovo” option must be used with multi-class SVC models so that an ONNX graph that is compatible with TF3800 is generated.

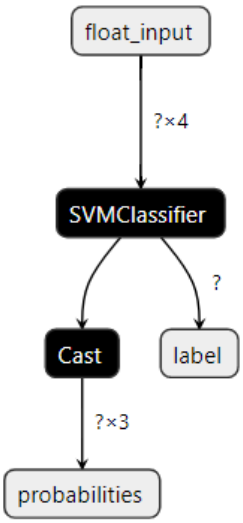
### Invalid ONNX graph

The following example shows the **invalid** ONNX graph: decision_function_shape “ovr”:




**Valid ONNX graph**

The following example shows the **valid** ONNX graph: decision_function_shape “ovo”:




### 5.1.3 k-Means

The k-Means algorithm is one of the *unsupervised* learning methods and is used for cluster analysis [▶ 60]. k-Means attempts to divide a random sample into k-clusters of the same variance; however, the number of clusters k must be known in advance. The algorithm scales well to a large number of samples and is one of the most widely used clustering methods.

Unsupervised means that the k-Means does not need to be trained with annotated (labeled) data. This property makes the algorithm very popular. As soon as the training has been executed and the clusters have been defined, new data can be assigned to the already known clusters in the inference.

#### Supported properties

#### ONNX support

So far, only export from Scikit-learn is supported. The specification of the ONNX Custom Attributes Key: “sklearn_model” value: “KMeans” is necessary for k-Means models so that the conversion step works in XML and BML.

**Restriction**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

An example of the export of an ONNX file from Scikit-learn for use in TwinCAT can be found here: ONNX export of a k-Means [▶ 33].

#### Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMllDataType [▶ 80].

#### 5.1.3.1 ONNX export of a k-Means

**Download Python samples**

A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

#### k-Means with Scikit-learn

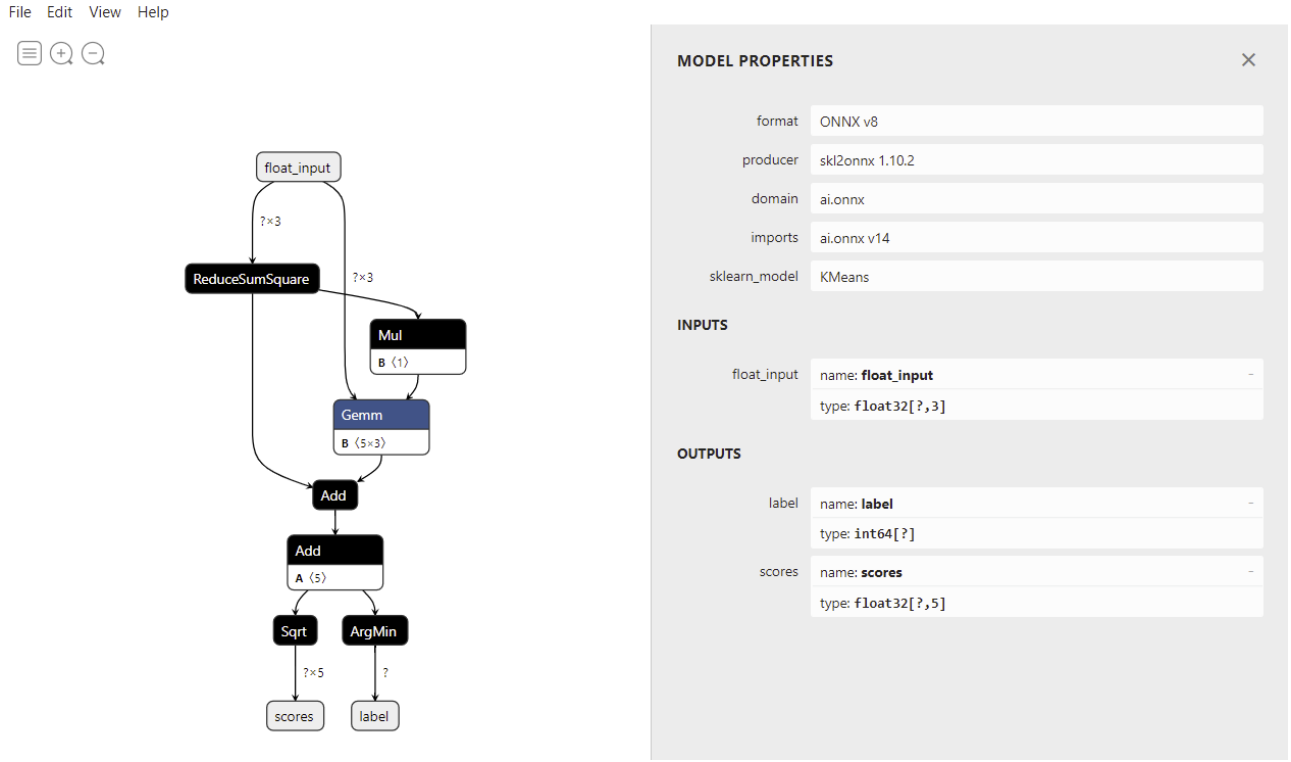
```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

# Generate data for clustering: num of samples, dimensions, num of clusters
X, y, centers = make_blobs(n_samples=50, n_features=3, centers=5, cluster_std=0.5, shuffle=True, ran
dom_state=42,  return_centers=True)
num_features = X.shape[1]
num_clusters = centers.shape[0]

# Define and train K-Means model
km = KMeans(n_clusters=num_clusters, init='k-means+
+', n_init=10, max_iter=500, tol=1e-04, random_state=42)
km.fit(X)

y_km = km.predict(X)

# Export model as ONNX
out_onnx = "kmeans.onnx"

initial_type = [('float_input', FloatTensorType([None, num_features]))]
```

```
onnx_model = convert_sklearn(km, initial_types=initial_type)

# for k-means models this meta info is mandatory. Otherwise convert process to TwinCAT-
specific format will fail!
meta =  onnx_model.metadata_props.add()
meta.key = "sklearn_model"
meta.value = "KMeans"

with open(out_onnx, "wb") as f:
    f.write( onnx_model.SerializeToString())
```

According to our level of knowledge, only Scikit-learn with skl2onnx is currently capable of converting a k-Means to ONNX. For this reason, the description is limited to that.

**ONNX Custom Attributes necessary**

The specification of the Custom Attribute "sklearn_model" and "KMeans" is necessary for k-means models, so that the conversion step in Beckhoff XML and Beckhoff BML works.




## 5.1.4 Principal Component Analysis

Principal Component Analysis (PCA) calculates so-called principal components with the help of which one can rotate the coordinate system of a given data set in such a way that the variance of the data, i.e. its information content, is maximized along the new principal axes. The covariance matrix of the transformed data is diagonalized and the order of principal components is sorted so that the first principal component carries the largest information portion of the data set, the second then carries the second largest information portion, and so on.

The latter principal components often contribute little information to the data set, which means that they can be ignored. As a result, the parameter space is reduced with the least possible loss of information (Dimension reduction [▶ 60]). The dimension reduction with PCA is often used for preprocessing prior to the cluster analysis or a classification. The PCA can also be used for anomaly detection [▶ 60] by reducing a large space to a few principal components and determining limit values for the important principal components “by hand”.

### Supported properties

### ONNX support

Supported ONNX operators:

- Sub
- MatMul

A sample showing how a PCA for dimension reduction can be exported from Scikit-learn and used in TwinCAT can be found here: ONNX export of a PCA [▶ 35].

**Supported data types**

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

### 5.1.4.1 ONNX export of a PCA

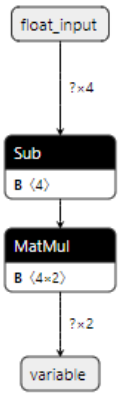
**Download Python samples**

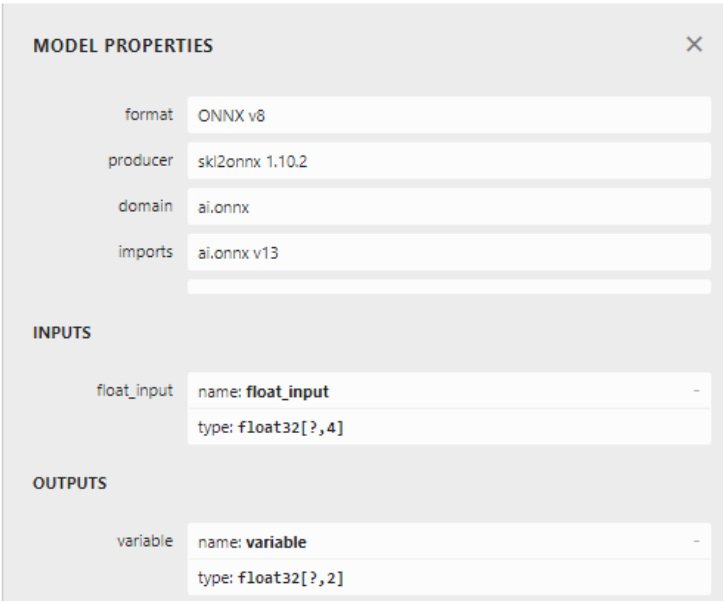
A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

**PCA with Scikit-learn**

```
import numpy as np
from sklearn.decomposition import PCA
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType, Int64TensorType

#Generate imput data_types
rng = np.random.RandomState(1)
X = np.dot(rng.rand(4, 3), rng.randn(3, 300)).T

# Dimensionality reduction with PCA
pca = PCA(n_components=2)
pca.fit(X)
X_pca = pca.transform(X)
print("original shape:   ", X.shape)
print("transformed shape:", X_pca.shape)

#Convert model to ONNX
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
model_onnx = convert_sklearn(pca, initial_types=initial_type)
meta =  model_onnx.metadata_props.add()
with open("pca.onnx", "wb") as f:
    f.write( model_onnx.SerializeToString())
```

According to our level of knowledge, only Scikit-learn with skl2onnx is currently capable of converting a PCA to ONNX. For this reason, the description is limited to that.




## 5.1.5 Decision Tree

A Decision Tree is an ML model that uses a tree-like structure to make predictions. It is a simple, but powerful tool for the prediction of values (regression) or classes (classification) [▶ 60] on the basis of several inputs, which works by dividing the data into smaller and smaller subsets until a final decision is made. The structure of the tree enables a simple interpretation and visualization of the model.

### Supported properties

### ONNX support

- TreeEnsambleClassifier
- TreeEnsambleRegressor

Samples of the export of Decision Tree models can be found here: ONNX export of a Decision Tree [▶ 36].

**Classification limitation**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

### Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

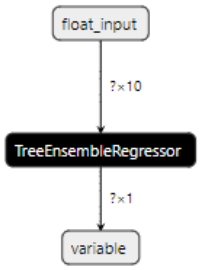
### 5.1.5.1 ONNX export of a Decision Tree

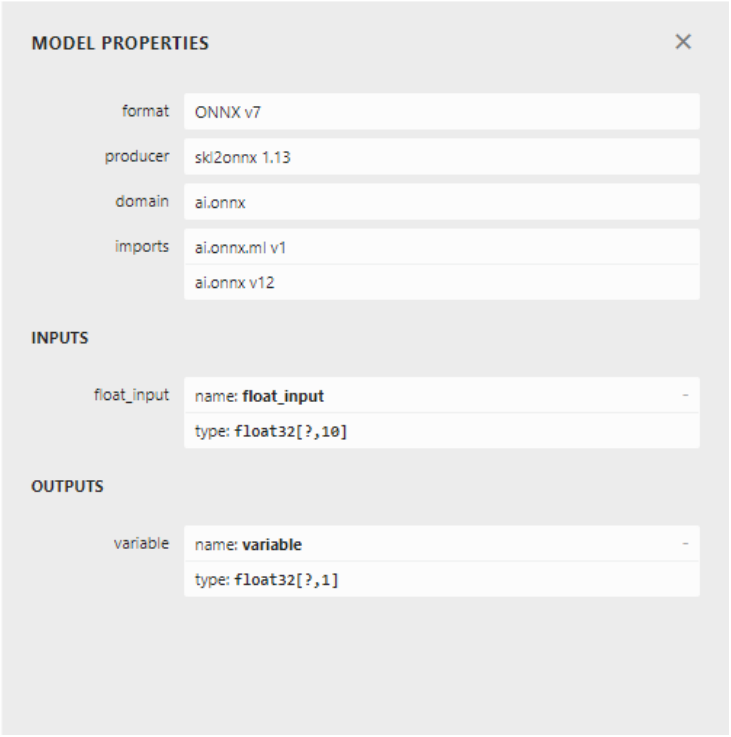
**Download Python samples**

A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

### Decision Tree Regressor with Scikit-learn

```
from sklearn.datasets import make_regression
from sklearn.tree import DecisionTreeRegressor
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

# # Generate data for regression
X, y =  make_regression(n_samples=300, n_features=10, n_informative=10, n_targets=1)

# # Construct Decision Tree-Model
model = DecisionTreeRegressor(criterion='squared_error', splitter='best', max_depth=None, min_sample
s_split=2, min_samples_leaf=1)
model.fit(X,y)
# # Convert model to ONNX
onnxfile = 'decisiontree-regressor.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_sklearn(model, initial_types=initial_type, target_opset=12)
# # Export to ONNX file
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```

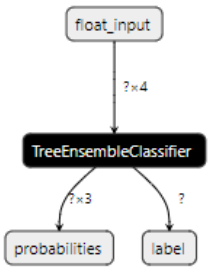


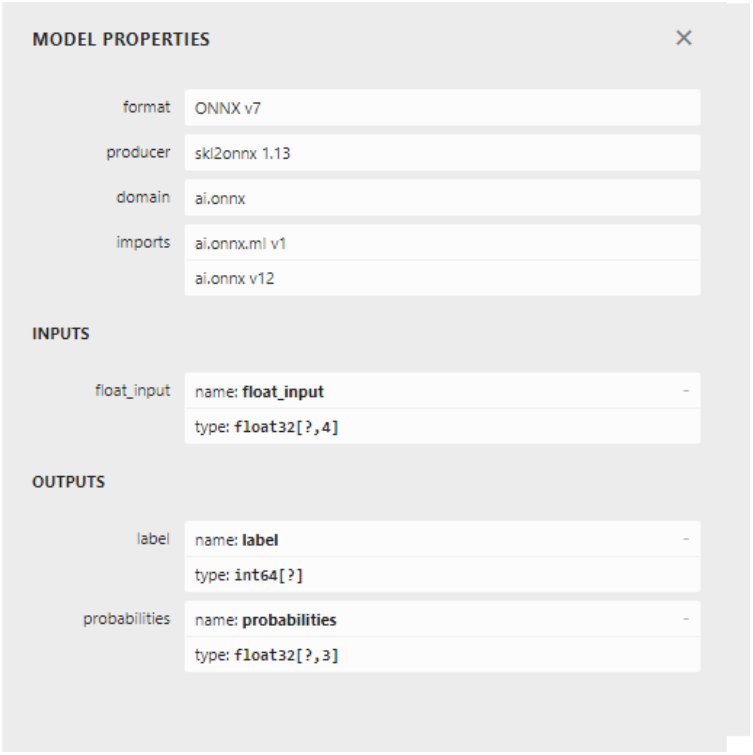




### Decision Tree Classifier with Scikit-learn

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

# # Load dataset
X, y = load_iris(return_X_y = True)
# # Construct Decision Tree-Model
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
model = DecisionTreeClassifier(criterion='gini', splitter='best', max_depth=None, min_samples_split=
2, min_samples_leaf=1)
model.fit(X_train,y_train)
# # Convert model to ONNX
onnxfile = 'decisiontree-classifier.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
# # Zipmap should be always turned off as it's not implemented in TF3800
onnx_model = convert_sklearn(model, initial_types=initial_type, options={type(model): {'zipmap':Fals
e}}, target_opset=12)
# # Export to ONNX file
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
```

```
f.close()
exit()
```




## 5.1.6 ExtraTree

An Extra Tree is the randomized variant of a Decision Tree [▶ 36]. It can also be used for the prediction of values (regression) or classes (classification) [▶ 60].

### Supported properties

### ONNX support

- TreeEnsambleClassifier
- TreeEnsambleRegressor

Examples of the export of Extra Tree models can be found here: ONNX export of an Extra Tree [▶ 39]

**Classification limitation**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

### Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

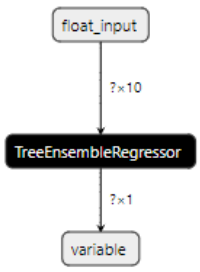
### 5.1.6.1 ONNX export of an Extra Tree

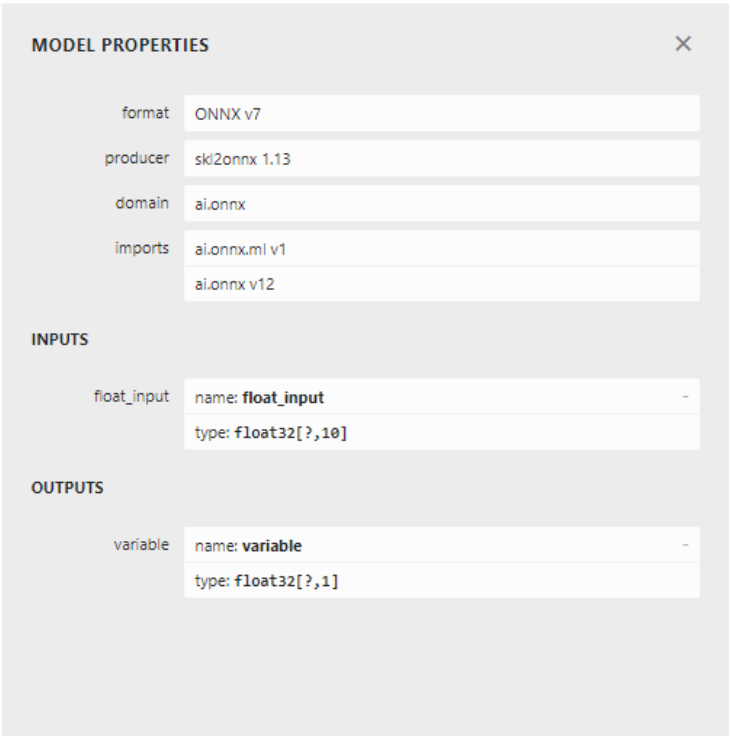
**Download Python samples**

A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

#### Extra Tree Regressor with Scikit-learn

```
from sklearn.datasets import make_regression
from sklearn.tree import ExtraTreeRegressor
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

# # Generate data for regression
X, y =  make_regression(n_samples=300, n_features=10, n_informative=10, n_targets=1)

# # Construct Extra Tree-Model
model = ExtraTreeRegressor(criterion='squared_error', splitter='random', max_depth=None, min_samples
_split=2, min_samples_leaf=1)
model.fit(X,y)
# # Convert model to ONNX
onnxfile = 'extratree-regressor.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_sklearn(model, initial_types=initial_type, target_opset=12)
# # Export to ONNX file
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```

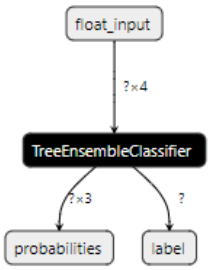


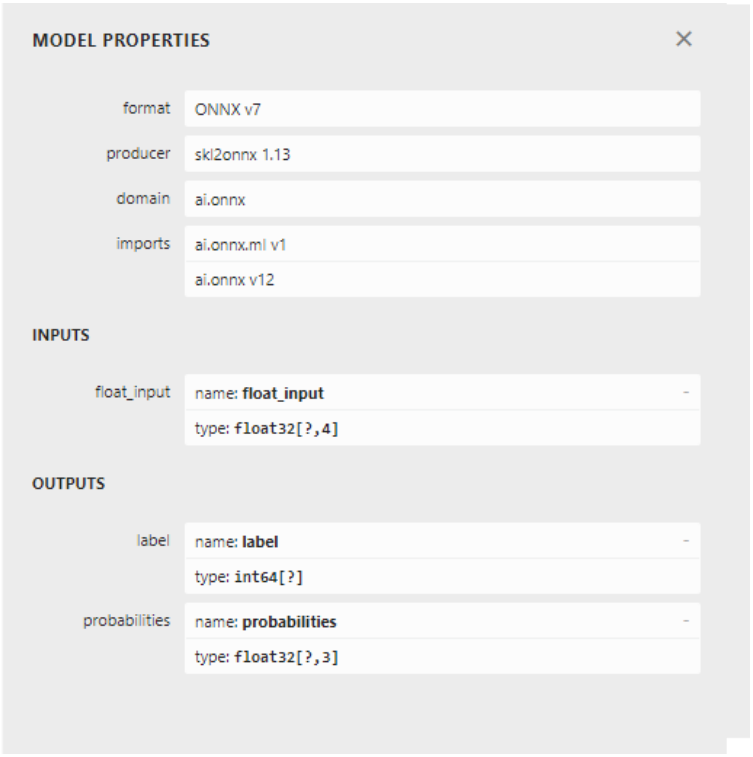




#### Extra Tree Classifier with Scikit-learn

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import ExtraTreeClassifier
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

# # Load dataset
X, y = load_iris(return_X_y = True)
# # Construct Decision Tree-Model
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
model = ExtraTreeClassifier(criterion='gini', splitter='random', max_depth=None, min_samples_split=2
, min_samples_leaf=1)
model.fit(X_train,y_train)
```

```
# # Convert model to ONNX
onnxfile = 'extratree-classifier.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
# # Zipmap should be always turned off as it's not implemented in TF3800
onnx_model = convert_sklearn(model, initial_types=initial_type, options={type(model): {'zipmap':Fals
e}}, target_opset=12)
# # Export to ONNX file
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```




## 5.1.7 Ensemble Tree methods

Ensemble methods combine several Decision Trees in order to achieve a better prediction performance. The basic principle is to train not just one model (one tree), but several trees – a forest of trees – and to combine the individual results into a common result.

There are basically two technologies with which an ensemble of trees can be created.

### Bagging

The bagging methods include:

- Random Forest [▶ 43]
- Extra Trees [▶ 41]

### Boosting

The boosting methods include:

- Gradient Boosting [▶ 46]
- Histogram Gradient Boosting [▶ 48]
- XGBoost [▶ 50]
- LightGBM [▶ 54]

**Unsupported additional Ensemble Tree methods**

The models BaggingClassifier, BaggingRegressor, AdaBoostClassifier and AdaBoostRegressor are also available in Scikit-learn. During an ONNX export they currently generate a graph that is incompatible with TwinCAT libraries, which means they cannot be supported.

#### 5.1.7.1 ExtraTrees

Extra Trees creates an ensemble of randomized Decision Trees [▶ 38]. Each tree is trained to a subset of the data set and the partial results are averaged. This increases the accuracy of the prediction in comparison with the Decision Tree [▶ 36] and the tendency toward overfitting is reduced.

**Supported properties**

**ONNX support**

- TreeEnsambleClassifier
- TreeEnsambleRegressor

Samples of the export of Extra Trees can be found here: ONNX export of Extra Trees [▶ 41].

**Classification limitation**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

**Supported data types**

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

##### 5.1.7.1.1 ONNX export of Extra Trees

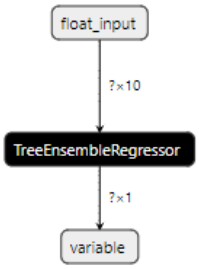
**Download Python samples**

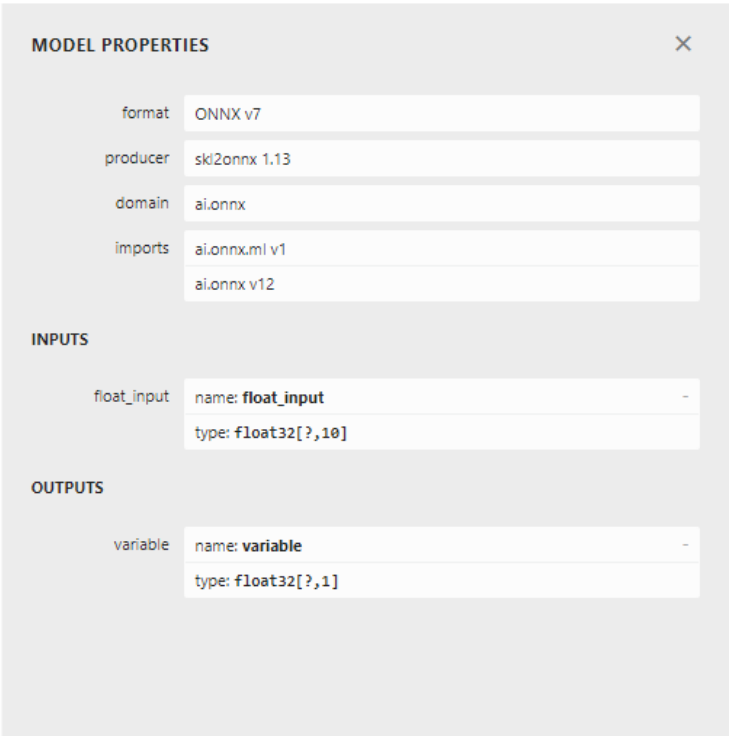
A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

**Extra Trees Regressor with Scikit-learn**

```
from sklearn.ensemble import ExtraTreesRegressor
from sklearn.datasets import make_regression
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType
# import onnx

# # Generate data for regression
X, y =  make_regression(n_samples=300, n_features=10, n_informative=10, n_targets=1)
# # Construct the model
model = ExtraTreesRegressor(max_depth=None, n_estimators=100)
model.fit(X,y)

# # Convert model to ONNX
onnxfile = 'extratrees-regressor.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_sklearn(model, initial_types=initial_type, target_opset=12)
# # Export to ONNX file
# onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
```

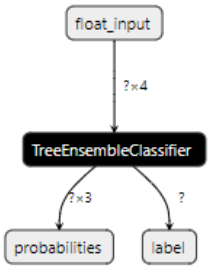
```
f.close()
exit()
```

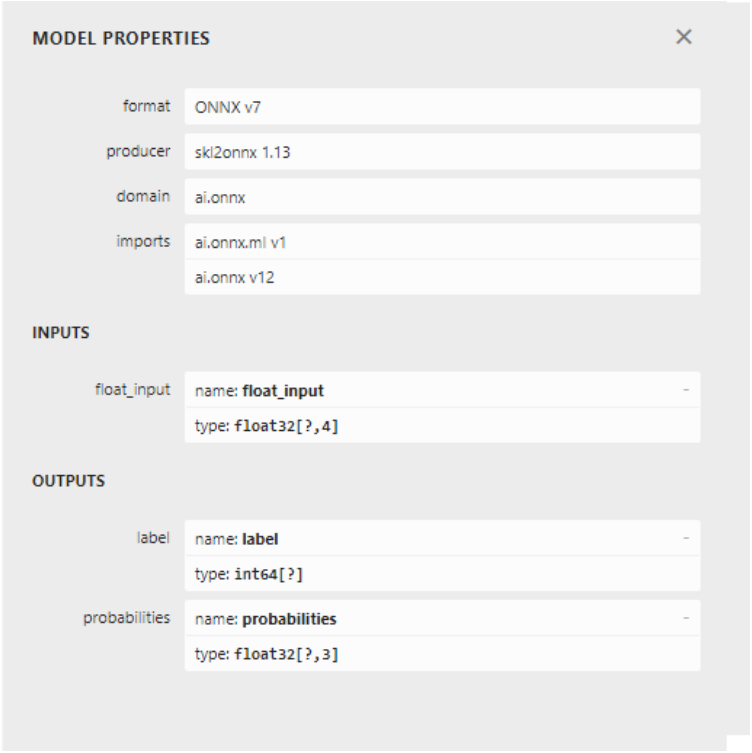



### Extra Trees Classifier with Scikit-learn

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import ExtraTreesClassifier
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

# # Load data for classification
X, y = load_iris(return_X_y = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
# # Construct ExtraTrees-Model
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
model = ExtraTreesClassifier(criterion ='entropy', n_estimators=100, max_features=None)
model.fit(X_train,y_train)
# # Export model into ONNX format
onnxfile = 'extratrees-classifier.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
# # Zipmap should be always turned off as it's not implemented in TF3800
onnx_model = convert_sklearn(model, initial_types=initial_type, options={type(model): {'zipmap':Fals
e}})
# # Export to ONNX file
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```




### 5.1.7.2 Random Forest

A Random Forest can be used both for classification and for regression [▶ 60]. The algorithm belongs to the ensemble methods, since a user-defined number of uncorrelated decision trees is built and trained. In the Random Forest, the prediction of the ensemble results from the averaged prediction of the individual trees.

Compared to individual Decision Trees [▶ 36], a Random Forest often has a better *accuracy* at the cost of the Random Forest is not being transparent with regard to the predictions made.

Compared to an SVM, a Random Forest is more efficient in terms of computing time, especially for high-dimensional data.

#### Supported properties

#### ONNX support

- TreeEnsambleClassifier
- TreeEnsambleRegressor

Samples of the export of Random Forest models can be found here: ONNX export of a Random Forest [▶ 44]

**Classification limitation**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

#### Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

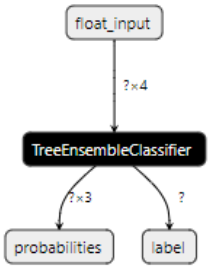
### 5.1.7.2.1 ONNX export of a Random Forest

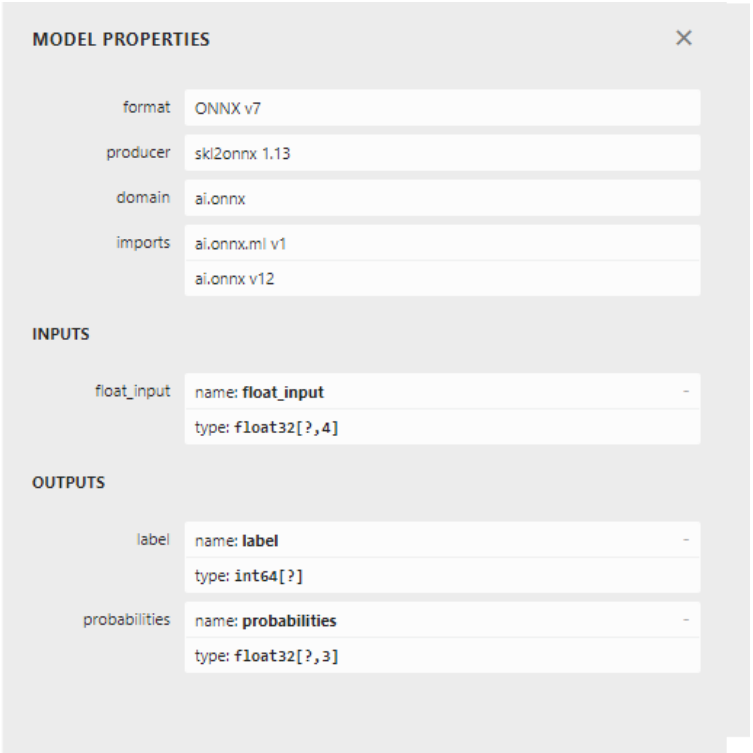
**Download Python samples**

A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

#### Scikit-learn: Random Forest Classifier

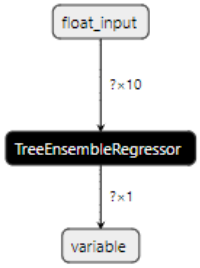
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

X, y = make_classification(n_samples=1000, n_features=3, n_informative=3, n_redundant=0, random_stat
e=1)

clf = RandomForestClassifier(n_estimators=100)
clf.fit(X, y)

initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
# # Zipmap should be always turned off as it's not implemented in TF3800
onnx_model = convert_sklearn(clf, initial_types=initial_type, options={type(clf): {'zipmap':False}})
with open("rf_classifier.onnx", "wb") as f:
    f.write( onnx_model.SerializeToString())
exit()
```

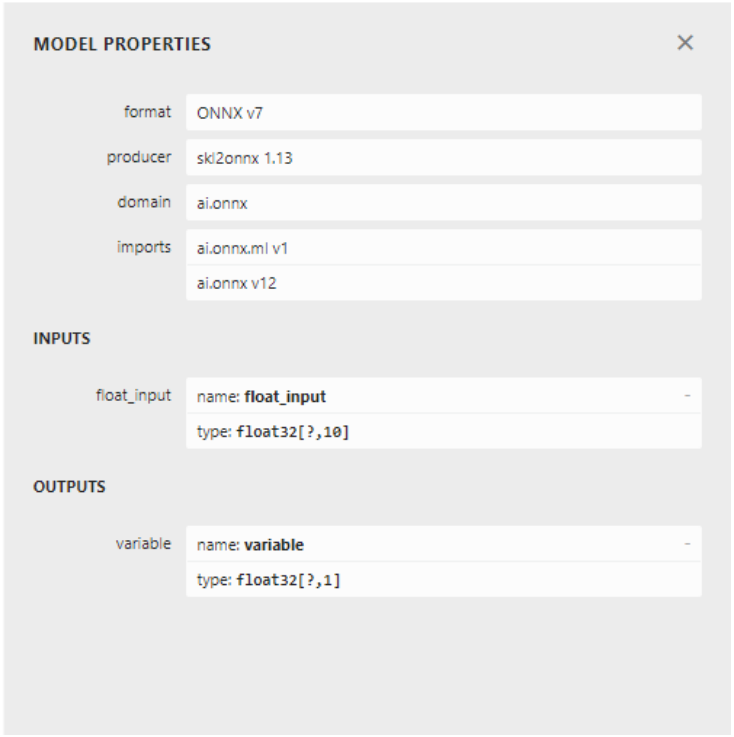



#### Scikit-learn: Random Forest Regressor

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import make_regression
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

X, y = make_regression(n_samples=1000, n_features=5, n_informative=5, random_state=2)

clf = RandomForestRegressor(n_estimators=100)
clf.fit(X, y)

initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_sklearn(clf, initial_types=initial_type)
with open("rf_regressor.onnx", "wb") as f:
    f.write( onnx_model.SerializeToString())
```

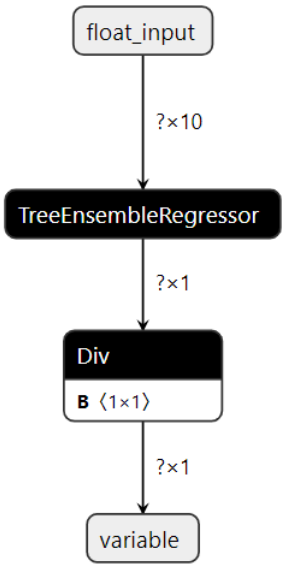


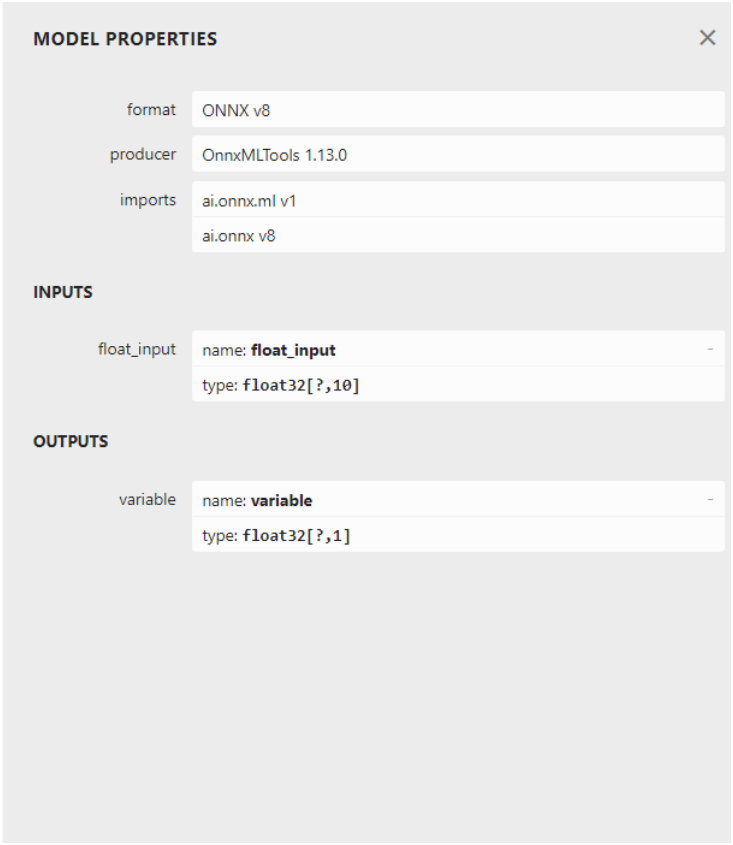




### LightGBM: Random Forest Regressor

```
import numpy as np
import onnx_graphsurgeon as gs
import lightgbm as lgb
from lightgbm import LGBMRegressor
from skl2onnx.common.data_types import FloatTensorType
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from onnxmltools.convert import convert_lightgbm
import onnxmltools.convert.common.data_types
import onnx
# # Generate data for regression
X, y =  make_regression(n_samples=300, n_features=10, n_informative=10, n_targets=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
# # Construct LightGBM-RandomForest-Model
model = LGBMRegressor(boosting_type='rf', class_weight=None, colsample_bynode=0.3, colsample_bytree=
1.0, importance_type='split', learning_rate=0.05, max_depth=-1, min_child_samples=2, min_child_weigh
t=0.001, min_split_gain=0.0, n_estimators=150, n_jobs=-1, num_class=1, num_leaves=500, objective='re
gression', random_state=None, reg_alpha=0.0, reg_lambda=0.0, silent=True, subsample=0.632, subsample
_for_bin=200000, subsample_freq=1)
model.fit(X_train, y_train,eval_set=[(X_test,y_test),
(X_train,y_train)],eval_metric='rmse', verbose=20)

# # Convert model to ONNX
onnxfile = 'lgbm-regressor-randomforest.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_lightgbm(model, initial_types=initial_type, target_opset=12)

# Manipulate ONNX graph

# # Import model to graph object
graph = gs.import_onnx(onnx_model)
graph.name = "LGBM-RandomForest"
# # Modify TreeEnsemble output shape (necessary to meet TwinCAT requirement, working on an update to
 make this step obsolete)
tree_node = [node for node in graph.nodes if node.op == "TreeEnsembleRegressor"][0]
tree_node.outputs[0].shape = [None, 1]
tree_node.outputs[0].dtype = np.float32
# # Modify DIV Node inputs to provide correct averaging (necessary to correct a bug in onnxmltools v
ersion 1.11.1)
div_node = [node for node in graph.nodes if node.op == "Div"][0]
div_node.inputs[1].to_constant(values=np.asarray([[model.n_estimators]], dtype=np.float32))
# # Export graph object to ONNX ProtoModel
graph.cleanup().toposort()
onnx_model = gs.export_onnx(graph)
# # Add ONNX domain tag to TreeEnsemble Node for proper node recognition (only a reset of the tag as
 it gets lost during onnx manipulation)
```

```
tree_node = [node for node in onnx_model.graph.node if node.op_type == "TreeEnsembleRegressor"][0]
tree_node.domain = "ai.onnx.ml"
tree_node.doc_string = "Converted from LGBMRegressor() model with explicit shaping"
# # Export ONNX model to file
onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```




### 5.1.7.3 Gradient Boosting

A Gradient Boosting model can be used both for classification and for regression [▶ 60]. Like the Random Forest [▶ 43], for example, the model is one of the Ensemble Tree [▶ 40] methods. Compared to an individual Decision Tree, the accuracy of the prediction can be improved with the Gradient Boosting model at the cost of the model no longer being simple to explain. Random Forest and Gradient Boosting differ from each other in the way the individual trees are generated.

#### Supported properties

#### ONNX support

- TreeEnsambleClassifier
- TreeEnsambleRegressor

Samples of the export of Gradient Boosting models can be found here: ONNX export of Gradient Boosting [▶ 47].

**Classification limitation**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

#### Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

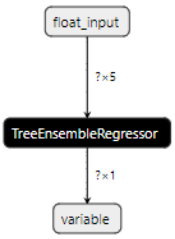
#### 5.1.7.3.1 ONNX export of Gradient Boosting

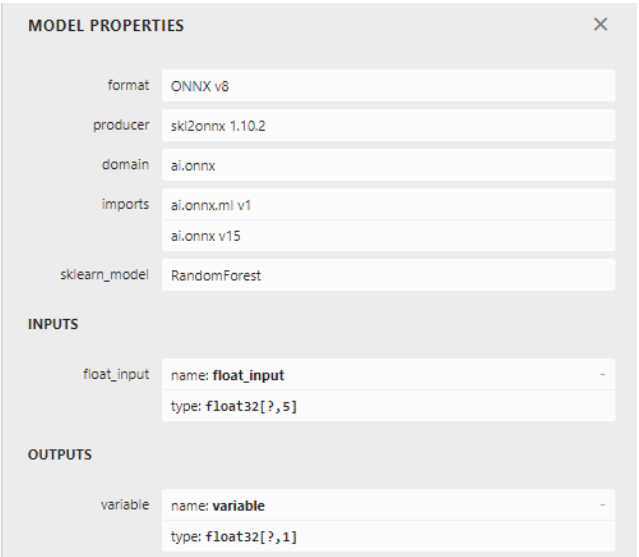
**Download Python samples**

A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

**Gradient Boosting Regressor with Scikit-learn**

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.datasets import make_regression
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType
# import onnx

# # Generate data for regression
X, y =  make_regression(n_samples=300, n_features=10, n_informative=10, n_targets=1)
# # Construct the model
model = GradientBoostingRegressor(learning_rate=0.08, max_depth=3, n_estimators=300)
model.fit(X,y)

# # Convert model to ONNX
onnxfile = 'gdb-regressor.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_sklearn(model, initial_types=initial_type, target_opset=12)
# # Export to ONNX file
# onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```

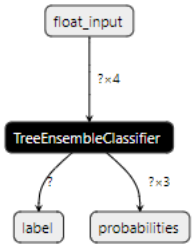


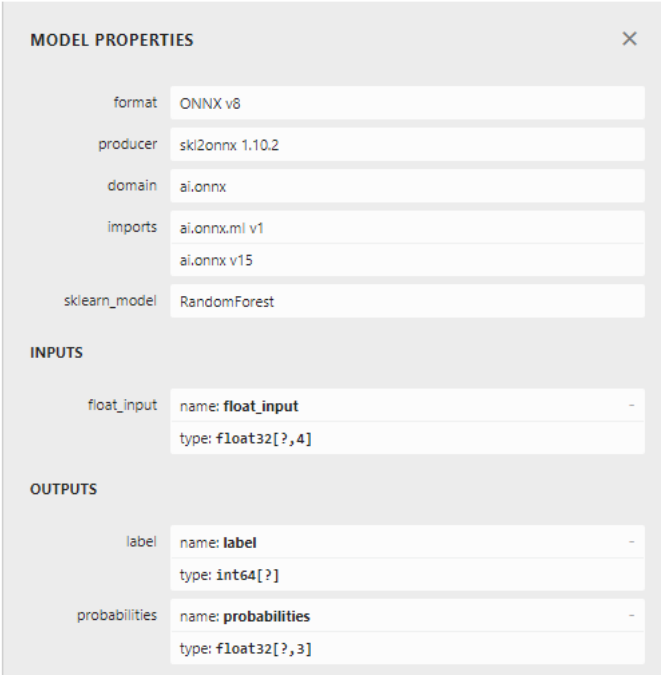




**Gradient Boosting Classifier with Scikit-learn**

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType
# import onnx

# # Load data for classification
X, y = load_iris(return_X_y = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
model = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1, max_depth=3)
```

```
model.fit(X_train,y_train)

# # Convert model to ONNX
onnxfile = 'gdb-classifier.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
# # Zipmap should be always turned off as it's not implemented in TF3800
onnx_model = convert_sklearn(model, initial_types=initial_type, options={type(model): {'zipmap':Fals
e}}, target_opset=12)
# # Export to ONNX file
# onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```




### 5.1.7.4 Histogram-based Gradient Boosting

A histogram-based Gradient Boosting model can be used both for classification and for regression [▶ 60].

The model is based on the Gradient Boosting [▶ 46]; here, however, the continual inputs are discretized in bins with the help of a histogram. This hugely accelerates the training of the model, in particular with very large data sets.

#### Supported properties

#### ONNX support

- TreeEnsambleClassifier
- TreeEnsambleRegressor

Samples of the export of Hist Gradient Boosting models can be found here: ONNX export of Hist Gradient Boosting [▶ 49].

**Classification limitation**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

#### Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

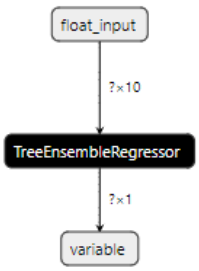
#### 5.1.7.4.1 ONNX export of Hist Gradient Boosting

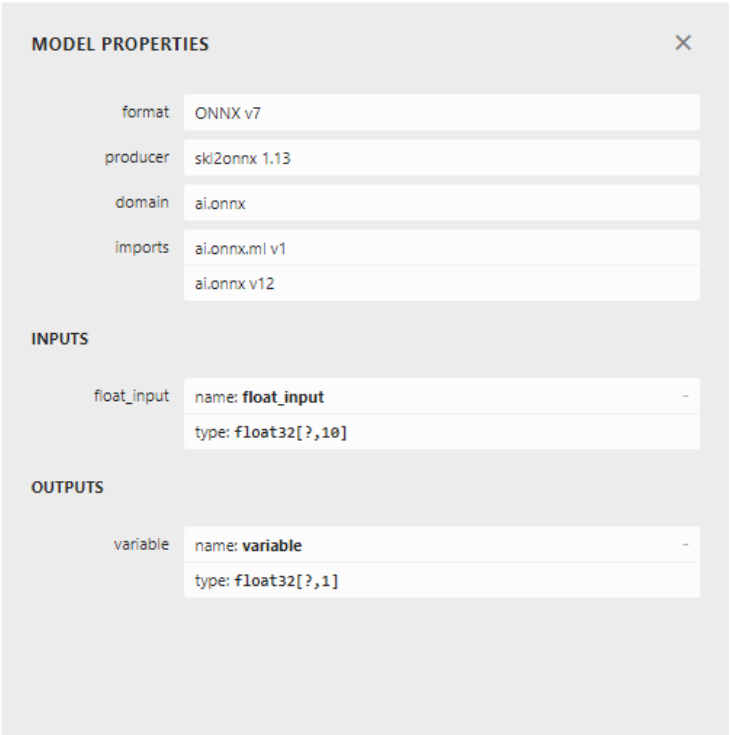
**Download Python samples**

A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

**Hist Gradient Boosting Regressor with Scikit-learn**

```
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.datasets import make_regression
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType
# import onnx

# # Generate data for regression
X, y =  make_regression(n_samples=300, n_features=10, n_informative=10, n_targets=1)
# # Construct the model
model = HistGradientBoostingRegressor(learning_rate=0.08, max_depth=3, max_iter=300)
model.fit(X,y)

# # Convert model to ONNX
onnxfile = 'histgdb-regressor.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_sklearn(model, initial_types=initial_type, target_opset=12)
# # Export to ONNX file
# onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```

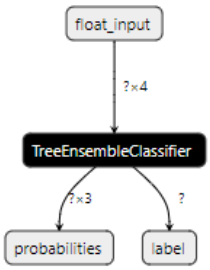


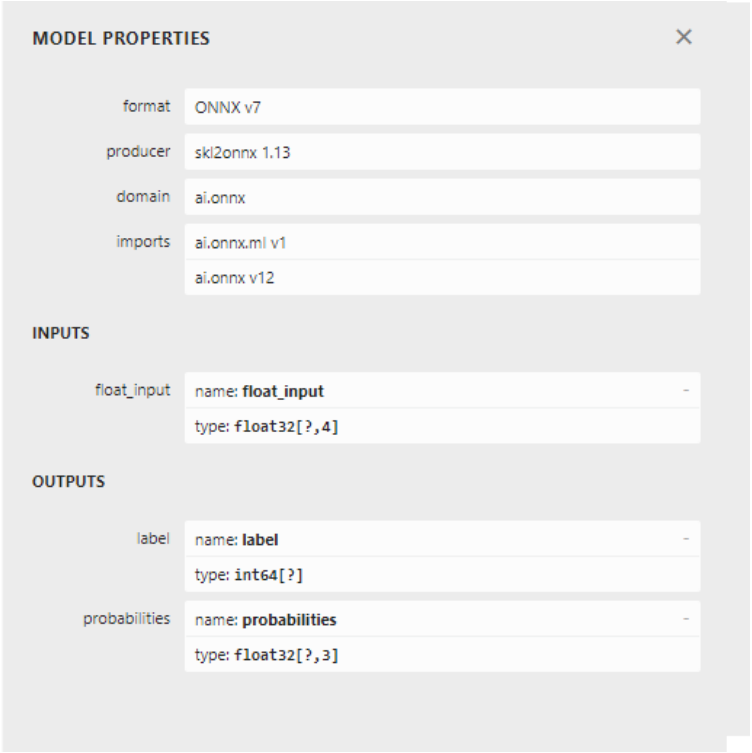




**Hist Gradient Boosting Classifier with Scikit-learn**

```
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
```

```
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType
# import onnx

# # Load data for classification
X, y = load_iris(return_X_y = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
# # Construct the model
model = HistGradientBoostingClassifier(learning_rate=0.08, max_depth=3, max_iter=300)
model.fit(X_train,y_train)

# # Convert model to ONNX
onnxfile = 'histgdb-iris.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
# # Zipmap should be always turned off as it's not implemented in TF3800
onnx_model = convert_sklearn(model, initial_types=initial_type, options={type(model): {'zipmap':Fals
e}}, target_opset=12)
# # Export to ONNX file
# onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```




### 5.1.7.5 XGBoost

An XGBoost model can be used both for classification and for regression [▶ 60].

Compared to Gradient Boosting [▶ 46], the XGBoost has advantages with regard to the generalization of the model. The training data set should be large – considerably more samples compared to the number of features used.

#### Supported properties

#### ONNX support

- TreeEnsambleClassifier
- TreeEnsambleRegressor

Samples of the export of XGBoost models can be found here: ONNX export of XGBoost [▶ 51]

**Classification limitation**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

**Supported data types**

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

### 5.1.7.5.1 ONNX export of XGBoost

**Download Python samples**

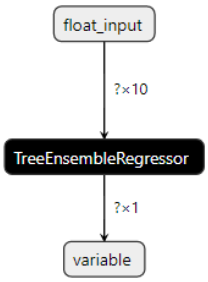
A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

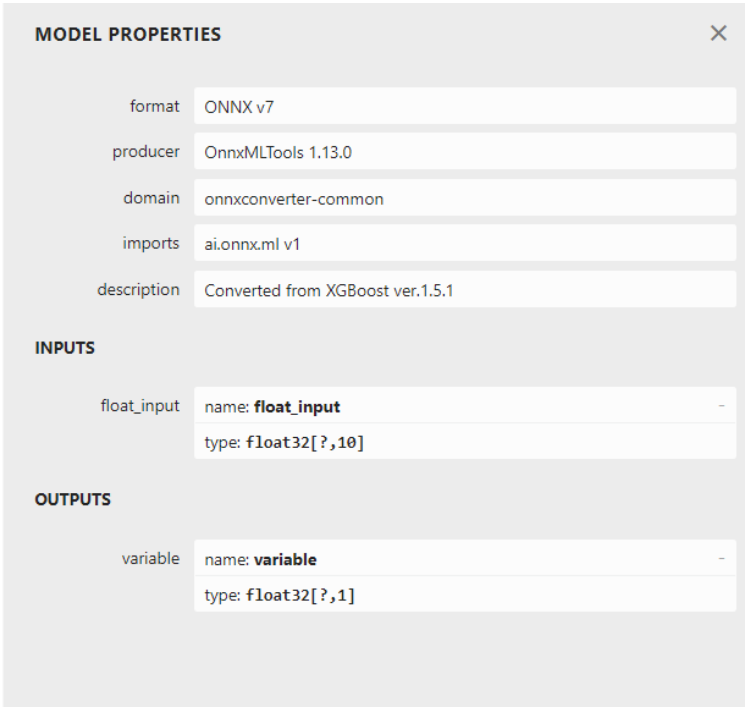
**Limitation of the version of XGBoost**

The supported versions of XGBoost are limited due to the ONNX export behavior: version <= 1.5.2 or >= 1.7.4.

**XGB Regressor**

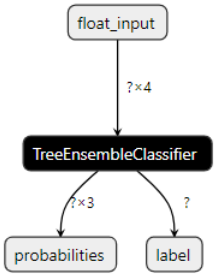
```
# # Important Requirement: XGBoost version must be 1.7.4 or higher
(otherwise onnxmltools 1.11.1 does not match)
import xgboost as xgb
from xgboost import XGBRegressor
from sklearn.datasets import make_regression
from skl2onnx.common.data_types import FloatTensorType
from onnxmltools.convert import convert_xgboost
import onnx

# # Generate data for regression
X, y =  make_regression(n_samples=300, n_features=10, n_informative=10, n_targets=1)
# # Construct XGB-Model
model = XGBRegressor(objective='reg:squarederror',booster='gbtree', max_depth=3, learning_rate=0.08,
 n_estimators=300)
model.fit(X,y)
# # Convert model to ONNX
onnxfile = 'xgb-regressor.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_xgboost(model, initial_types=initial_type, target_opset=12)
onnx_model.graph.doc_string = "Converted from XGBoost ver."+xgb.__version__
# # Export to ONNX file
onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```

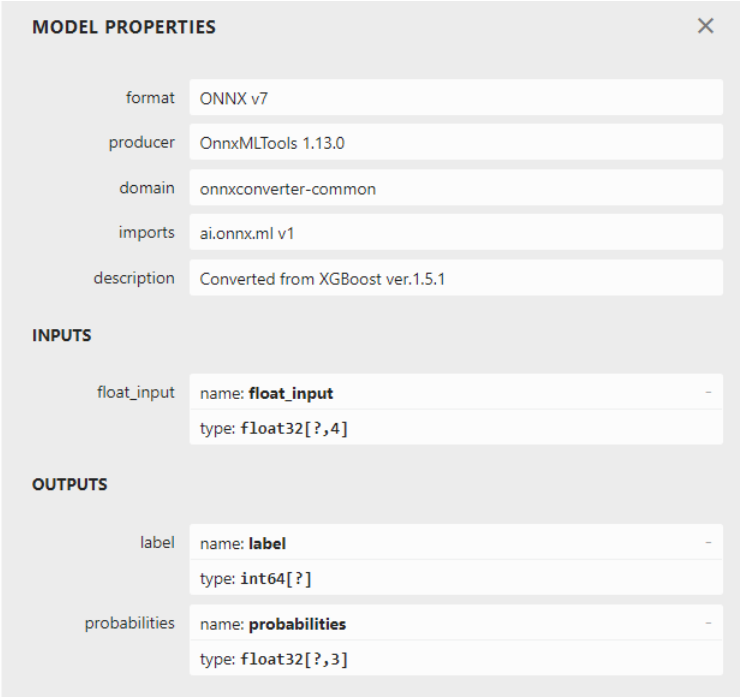



## XGB Classifier

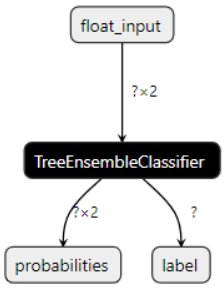
```
import onnx
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import xgboost as xgb # # Important Requirement: XGBoost version must be 1.7.4 or higher (otherwise
onnxmltools 1.11.1 does not match)
from xgboost import XGBClassifier
from skl2onnx.common.data_types import FloatTensorType
from onnxmltools.convert import convert_xgboost

X, y = load_iris(return_X_y = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
model = XGBClassifier(objective= 'multi:softmax', learning_rate=0.03, max_depth=3, n_estimators=300,
 eval_metric='mlogloss', early_stopping_rounds=20, verbosity=1, use_label_encoder=False)
model.fit(X_train,y_train, eval_set=[(X_train, y_train), (X_test, y_test)], verbose=True)
onnxfile = 'xgb-iris.onnx'
# # Convert model to ONNX
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_xgboost(model, initial_types=initial_type, target_opset=12)
onnx_model.graph.doc_string = "Converted from XGBoost ver."+xgb.__version__
# # Export to ONNX file
onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```




### XGB binary classifier

```
from skl2onnx.common.data_types import FloatTensorType
from sklearn.model_selection import train_test_split
from sklearn.datasets import  make_moons, make_circles, make_classification
import onnx
# # Important Requirement: XGBoost version must be 1.7.4 or higher
(otherwise onnxmltools 1.11.1 does not match)
import xgboost as xgb
from xgboost import XGBClassifier
from onnxmltools.convert import convert_xgboost

# # Generate data for binary classification
X, y =  make_moons(n_samples=300, noise=0.3, random_state=1)
# X, y = make_circles(n_samples=300, shuffle=True, noise=0.3, random_state=1, factor=0.8)
# # Construct XGB-Model
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
model = XGBClassifier(objective= 'binary:logistic', learning_rate=0.03, max_depth=3, n_estimators=30
0, eval_metric='auc', early_stopping_rounds=20, verbosity=1, use_label_encoder=False)
model.fit(X_train,y_train, eval_set=[(X_train, y_train), (X_test, y_test)], verbose=True)
# # Convert model to ONNX
onnxfile = 'xgb-binary.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_xgboost(model, initial_types=initial_type, target_opset=12)
onnx_model.graph.doc_string = "Converted from XGBoost ver."+xgb.__version__
# # Export to ONNX file
onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```




| | |
|---|---|
| format | ONNX v7 |
| producer | OnnxMLTools 1.13.0 |
| domain | onnxconverter-common |
| imports | ai.onnx.ml v1 |
| description | Converted from XGBoost ver.1.5.1 |

**INPUTS**

| | |
|---|---|
| float_input | name: **float_input**<br>type: `float32[?,2]` |

**OUTPUTS**

| | |
|---|---|
| label | name: **label**<br>type: `int64[?]` |
| probabilities | name: **probabilities**<br>type: `float32[?,2]` |

### 5.1.7.6 LightGBM

A LightGBM model can be used both for classification and for regression [▶ 60].

LightGBM is one of the histogram-based Gradient Boosting [▶ 48] methods. This makes training efficient, in particular with large data sets.

#### Supported properties

#### ONNX support

- TreeEnsambleClassifier
- TreeEnsambleRegressor

Samples of the export of LightGBM models can be found here: ONNX export of LightGBM [▶ 54].

**Classification limitation**

With classification models, only the output of the labels is mapped in the PLC. The scores/ probabilities are not available in the PLC.

#### Supported data types

A distinction must be made between "supported datatype" and "preferred datatype". The preferred datatype corresponds to the precision of the execution engine.

The preferred datatype is floating point 64 (E_MLLDT_FP64-LREAL).

When using a supported datatype, an efficient type conversion automatically takes place in the library. Slight losses of performance can occur due to the type conversion.

A list of the supported datatypes can be found in ETcMlIDataType [▶ 80].

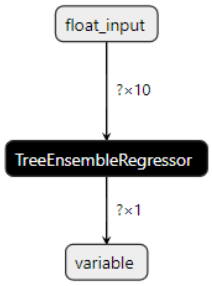
#### 5.1.7.6.1 ONNX export of LightGBM

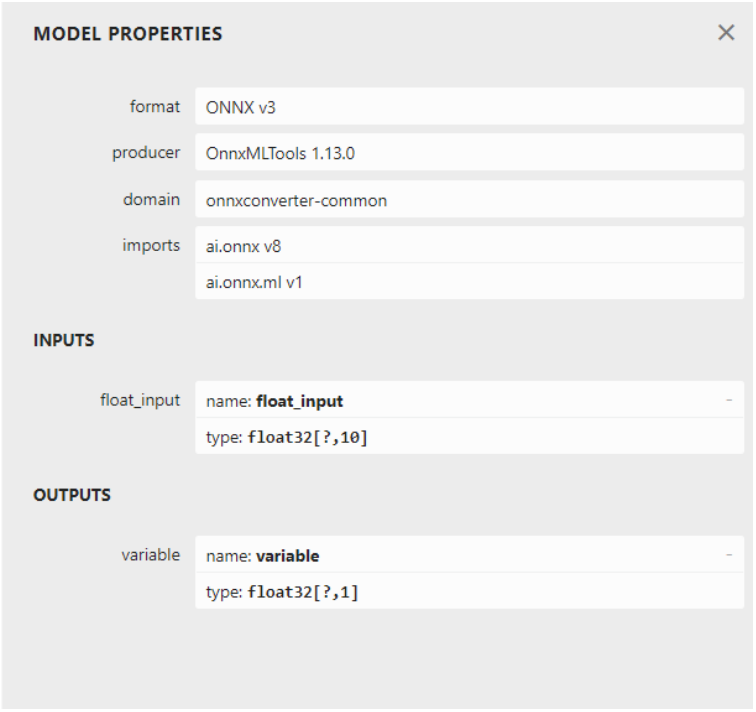
**Download Python samples**

A Zip archive containing all samples can be found here: Samples of ONNX export [▶ 61]

### LGBM Regressor

```
from lightgbm import LGBMRegressor
from skl2onnx.common.data_types import FloatTensorType
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from onnxmltools.convert import convert_lightgbm
import onnx

# # Generate data for regression
X, y =  make_regression(n_samples=300, n_features=10, n_informative=10, n_targets=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
# # Construct LightGBM-Model
model = LGBMRegressor(objective='regression', max_depth=31, learning_rate=0.05, n_estimators=300)
model.fit(X_train, y_train,eval_set=[(X_test,y_test),
(X_train,y_train)],eval_metric='rmse', verbose=20)

# # Convert model to ONNX
onnxfile = 'lgbm-regressor.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_lightgbm(model, initial_types=initial_type, target_opset=12)

onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```

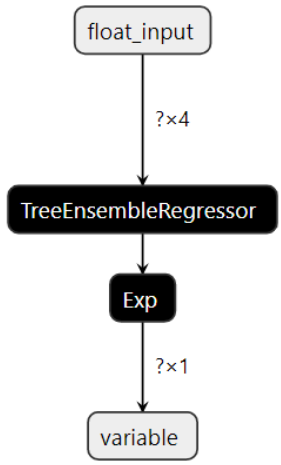


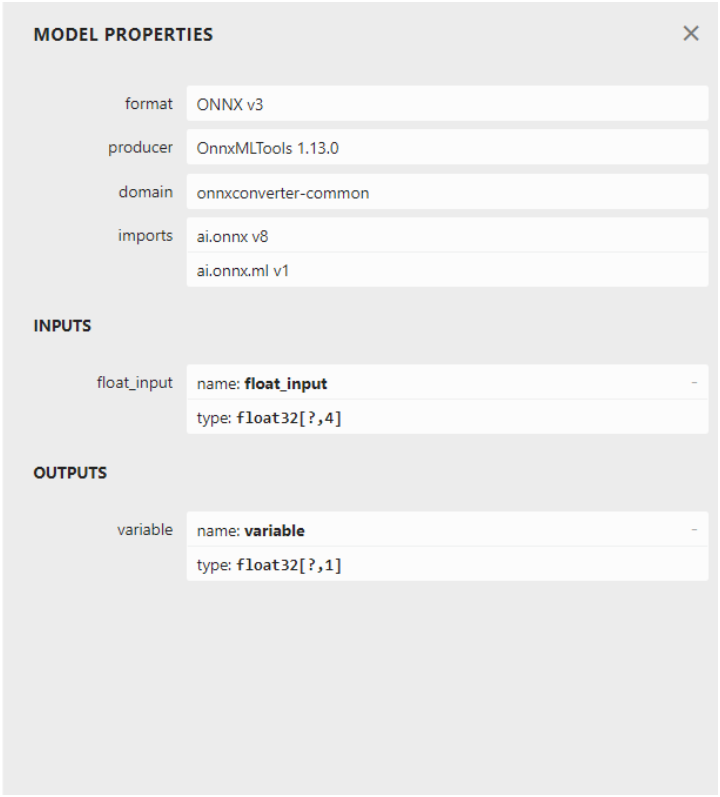




### LGBM Regressor (gamma objective)

```
import numpy as np
from lightgbm import LGBMRegressor
from skl2onnx.common.data_types import FloatTensorType
from sklearn.model_selection import train_test_split
from onnxmltools.convert import convert_lightgbm
import onnx

# # Generate data for regression
N_ROWS = 1000
N_COLS = 4
X = np.random.randn(N_ROWS, N_COLS)
# # For 'poisson' and 'gamma' objective, all target values need to be non-negative
y = abs(np.random.randn(N_ROWS))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
# # Construct LightGBM-Model
model = LGBMRegressor(objective='gamma', max_depth=-1, learning_rate=0.05, n_estimators=300)
model.fit(X_train, y_train,eval_set=[(X_test,y_test),
(X_train,y_train)],eval_metric='rmse', verbose=20)
```

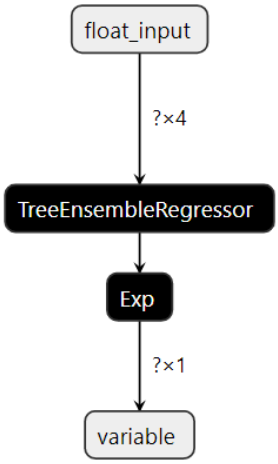
```
# # Convert model to ONNX
onnxfile = 'lgbm-regressor-gamma.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_lightgbm(model, initial_types=initial_type, target_opset=12)
onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```

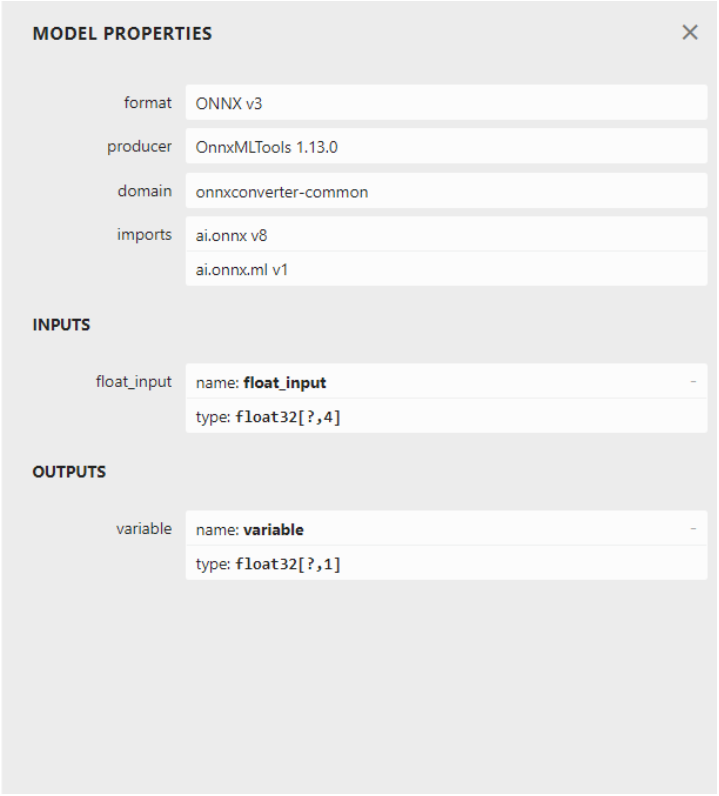



### LGBM Regressor (poisson objective)

```
import numpy as np
from lightgbm import LGBMRegressor
from skl2onnx.common.data_types import FloatTensorType
from sklearn.model_selection import train_test_split
from onnxmltools.convert import convert_lightgbm
import onnx

# # Generate data for regression
N_ROWS = 1000
N_COLS = 4
X = np.random.randn(N_ROWS, N_COLS)
# # For 'poisson' and 'gamma' objective, all target values need to be non-negative
y = abs(np.random.randn(N_ROWS))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
# # Construct LightGBM-Model
model = LGBMRegressor(objective='poisson', max_depth=-1, learning_rate=0.05, n_estimators=300)
model.fit(X_train, y_train,eval_set=[(X_test,y_test),
(X_train,y_train)],eval_metric='rmse', verbose=20)

# # Convert model to ONNX
onnxfile = 'lgbm-regressor-poisson.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
onnx_model = convert_lightgbm(model, initial_types=initial_type, target_opset=12)
onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```

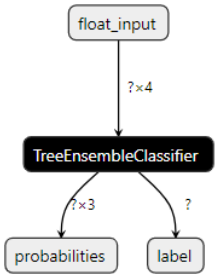


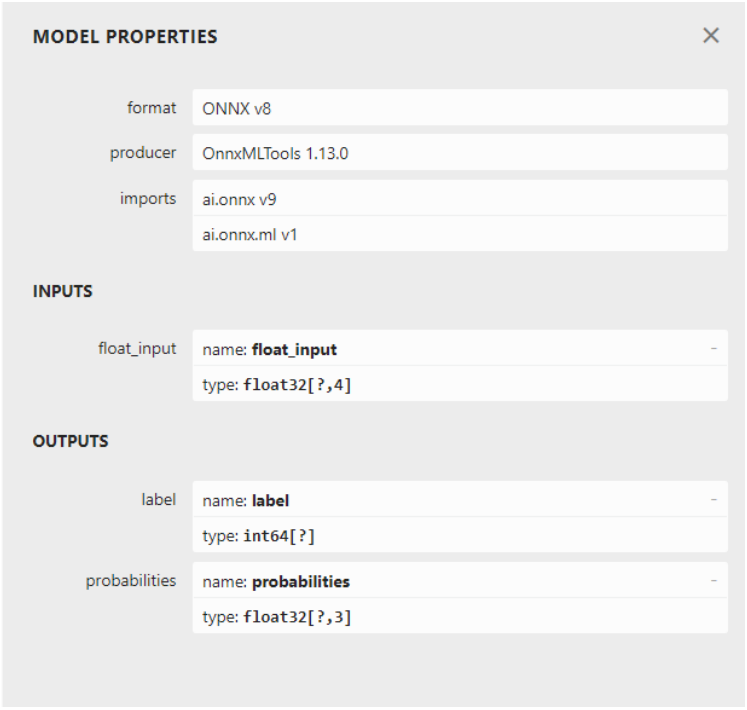




### LGBM Classifier

```
import onnx
import onnx_graphsurgeon as gs
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from lightgbm import LGBMClassifier
from skl2onnx.common.data_types import FloatTensorType
from onnxmltools.convert import convert_lightgbm

# # Load data for classification
X, y = load_iris(return_X_y = True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1)
# # Construct LightGBM Model
model = LGBMClassifier(objective='multiclass', learning_rate=0.05, max_depth=-1, n_estimators=100, r
andom_state=42)
model.fit(X_train,y_train,eval_set=[(X_test,y_test),
(X_train,y_train)],verbose=20,eval_metric='logloss')

# # Convert model to ONNX
onnxfile = 'lgbm-iris.onnx'
initial_type = [('float_input', FloatTensorType([None, X.shape[1]]))]
# # Zipmap should be always turned off as it's not implemented in TF3800
onnx_model = convert_lightgbm(model, initial_types=initial_type, zipmap=False)

# # Manipulate graph to force its ONNX conformity (necessary to correct a bug in onnxmltools version
 1.11.1)
graph = gs.import_onnx(onnx_model)
graph.name = "LGBMClassifier"
tree_node = [node for node in graph.nodes if node.op == "TreeEnsembleClassifier"][0]
tree_node.name = "TreeEnsembleClassifier"
tree_node.outputs = graph.outputs
tree_node.outputs[0].shape = [None]
tree_node.outputs[1].shape = [None, model.n_classes_]
# # Collect 2 artifacts of the converter
cast_node = [node for node in graph.nodes if node.op == "Cast"][0]
mul_node = [node for node in graph.nodes if node.op == "Mul"][0]
# # Clear outputs of these two nodes
mul_node.outputs.clear()
cast_node.outputs.clear()
# # Remove these nodes from the graph
graph.cleanup().toposort()
onnx_model = gs.export_onnx(graph)
nodes = onnx_model.graph.node
for node in nodes:
    # # Modify node attributes.
```

```
    if node.op_type == "TreeEnsembleClassifier":
        node.domain = "ai.onnx.ml" # # Domain info is required for ONNX conformity
        node.doc_string = "Converted from LGBMClassifier() model"
# # Export to ONNX file
onnx.checker.check_model(onnx_model)
with open(onnxfile, "wb") as f:
    f.write( onnx_model.SerializeToString())
f.close()
exit()
```




## 5.2 Machine Learning Cheat Sheet: selection of models

Which of the supported models is suitable for my problem? This question is frequently asked. The following information is intended to assist you in selecting suitable algorithms.

### Type of input data of the model

The first essential question concerns the type of input data of the model: image data, time series or *tabular data*?

The supported models are mainly suitable for *tabular data*. This means that the input of the model forms an array of values.

### Image data

**C**onvolutional **N**eural **N**etworks (CNNs) are usually used for the direct processing of image data. These are expected to be supported starting from Q3/2023. MLPs can also deliver adequate results for a restricted application area. For this purpose, an image pixel is input into the model as a vector.

In addition, it is expedient to extract features from the image data first and to use these features as the input data of an AI model. TwinCAT Vision provides a powerful library for image capture, preprocessing and feature generation. The features can then be combined as an array and the problem interpreted as a tabular data problem.

### Time series

Recurrent neural networks such as the LSTM are usually used for the direct processing of time series, i.e. series of data points in which the temporal sequence of the samples carries essential information. These are expected to be supported starting from Q3/2023. MLPs can also deliver adequate results for a restricted application area. N-samples are input into the model as vectors.

In addition, it is expedient to extract signal features from the time series and to use these features as the input data of an AI model. PLC libraries such as the Condition Monitoring library or the Analytics library are suitable for the feature generation of time series. In practice it has proven to be very efficient, for example, to form static variables over a defined time segment, such as mean value, standard deviation, maximum and minimum value, etc. The time segment might be the length of a process step, for example from the start to the end of a cut, or from the start to the end of a bending process. In addition to static variables, frequency-based features such as the signal power in defined frequency bands have proven to be useful, especially with rotary processes. The generated features are then combined in an array and used as the input for the AI model. Accordingly, the problem can be interpreted as a tabular data problem.

**Tabular Data**

Tabular Data can be used directly as the input for most AI models. Situations where an array of input data are directly available could be: the length, width and mass as well as its optical components in R, G and B values are measured with different measuring instruments. The values can be directly combined as an array of 6 elements and used as the input for an AI model – for example, for classification as OK or NOK.

**Description of the goal**

Once the type of input data has been defined, the question arises as to what exactly the AI should do, or what it can do under given conditions. Do annotated (labeled) data exist and what type are the labels?

**Clustering**

In clustering, the inner structure of the input data are analyzed. No annotated data are necessary for the cluster analysis; however, the number k of the expected clusters must be known.

**Anomaly detection**

A popular application, likewise for the case that no annotated data exist. In the training phase, only data that can be described as “normal” are presented to the model. In the inference phase, the model can distinguish between a known input data structure and an unknown input data structure. In the latter case an anomaly is assumed. The challenge in anomaly detection is the preprocessing of the training data, so that if possible only the normal case is used in the training, as well as the limited meaningfulness of the result.

**Dimensionality reduction**

Human beings are good at visualizing the 2- and 3-dimensional space. Point clouds in a 3D plot are easy to handle and can improve the understanding of processes. However, it quickly becomes confusing if several dimensions are involved. The purpose of dimension reduction is to map an N-dimensional input vector to a smaller vector while losing the least possible amount of information: for example, a 10-dimensional input is reduced to 3 dimensions while retaining 95% of the information. Redundant information of the input data is exploited. The dimension reduction is well suited for use as a feature generation step, e.g. before a classifier.

**Regression**

A regression problem requires the existence of an annotated data set. As a rule, a problem is described with N REAL or LREAL as the input of the model and M REAL or LREAL as the output.
Example: N features are created during a forming process (e.g.: maximum, standard deviation, skew of the servo motor current). For each of these features, the resulting diameter of the formed product in the longitudinal and transverse direction is known. From the 3 features, 2 values are estimated accordingly.

If the curve of a time series is to be modeled, the N past time values can be used as the input vector and the N+1 value as the label. The manual labeling of the data is thus unnecessary.

**Classification**

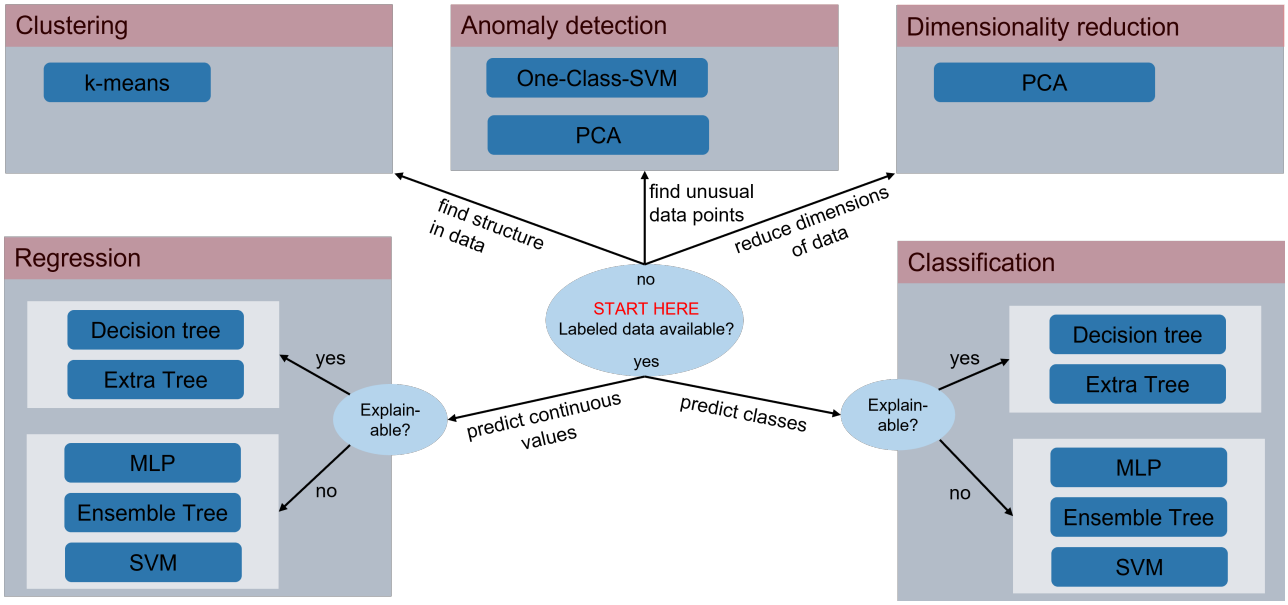
A classification problem requires an annotated data set. As a rule, N REAL or LREAL values are mapped here to a category that is usually represented as INT in TwinCAT. For example, whether a finished product corresponds to quality class A, B or C is calculated from N features.

### Explainability of an AI model

In some situations it is of great importance to be able to explain the results of an AI model, i.e. to answer the question, for example, as to why a model has classified a product as defective. Unfortunately, most algorithms work like black boxes, and the results can only be explained with difficulty if at all – even if they are very precise. Decision Trees are models that can be explained very well, because the path through the tree can be retraced with the individual limit values of the branches. However, the accuracy of these models is often not as convincing as with other models that cannot be explained.

### AI model Cheat Sheet

The following figure provides a simple guide to the selection of a suitable AI model. It illustrates the classification of AI models for different application purposes, provided the model input is tabular data.




## 5.3 Creation and conversion of ONNX

The learning process of a Machine Learning model takes place outside of the real-time, usually in a script language such as Python or R. The learned model is to be exported from the selected ML framework as an ONNX file. In order to load the description of the model in the TwinCAT runtime, the ONNX file must first be converted into a proprietary, TwinCAT 3-specific format.

An overview of the description formats for machine learning models:

- Open Neural Network Exchange Format (ONNX) [▶ 60]
- Beckhoff ML XML [▶ 73]
- Beckhoff ML BML [▶ 75]

Beckhoff's proprietary formats in XML and BML are directly readable from the Machine Learning Runtime. The ONNX data format must be converted to a Beckhoff proprietary format using provided converters [▶ 63].

Whereas ONNX and XML are openly visible formats, BML is a binary format and thus characterized above all by a small file size and an efficient loading behavior (execution time of the Configure method) in the XAR.

### 5.3.1 Open Neural Network Exchange (ONNX)

#### What is ONNX?

ONNX is an open file format for the representation of Machine Learning Models and is managed as a community project. Homepage of the ONNX community: onnx.ai

The ONNX format defines groups of operators in a standardized format, allowing learned models to be used interoperably with various frameworks, runtimes and further tools.

ONNX supports descriptions of neural networks as well as classic machine learning algorithms and is therefore the suitable format for both the TwinCAT Machine Learning Inference Engine and the TwinCAT Neural Network Inference Engine.

**Why ONNX?**

Through support for ONNX, Beckhoff integrates the TwinCAT Machine Learning products in an open manner and **thus guarantees flexible workflows**. While the automation specialist can work in TwinCAT 3, the data scientist can work with his usual tools (PyTorch, Scikit-Learn, ...).

The use of ONNX facilitates **cross-workgroup working**, both internally and cross-company with partners. The automation specialist provides the data scientist with recorded data. The data scientist creates an ML model and hands over his work as an ONNX file to the automation specialist. This file already contains all information to execute the created model in TwinCAT.

The **offline testing** of models is simplified, because all common AI frameworks can load and also execute the ONNX file.

**Which software supports ONNX?**

Supported tools of the ONNX community can be viewed here: onnx.ai/supported-tools.

Including, for example, the **frameworks:**

- PyTorch
- Keras/TensorFlow
- MXNet
- Scikit-learn
- …

Graph **Optimizer**

- ONNX Optimizer (https://github.com/onnx/onnx/blob/enable_noexpception_build/docs/Optimizer.md)

Graph **Visualizer**

- Netron (https://github.com/lutzroeder/Netron)
- …

## 5.3.2 Samples of ONNX export

**How do I create ONNX files?**

Below, several ways of exporting certain models as ONNX from different frameworks are shown **using examples**. The samples do not claim to be complete and only serve to provide a primary overview. For more detailed documentation, refer to the documentation for the respective framework.

The listed examples are limited to the creation of an ONNX file. Examples for conversion to make the file available in TwinCAT can be found here: Converting ONNX to XML and BML [▶ 63] as well as in the ZIP archive for the linked samples (see below) in the PythonAPI_mllib folder.

### Overview of available samples

| Python package | Model type | Option | Comment | Sample link |
|---|---|---|---|---|
| PyTorch | MLP Regressor | | | GoToPage [▶ 22] |
| Keras | MLP Regressor | | | GoToPage [▶ 23] |
| Scikit-learn | MLP Regressor | | | GoToPage [▶ 24] |
| Scikit-learn | MLP Classifier | | ONNX graph must be adapted | GoToPage [▶ 25] |
| Scikit-learn | SVR | | | GoToPage [▶ 28] |
| Scikit-learn | SVC | decision_function_shape='ovo' | | GoToPage [▶ 29] |
| Scikit-learn | k-means | | Meta Key must be entered in ONNX. | GoToPage [▶ 33] |
| Scikit-learn | PCA | | | GoToPage [▶ 35] |
| Scikit-learn | Decision Tree Classifier | | | GoToPage [▶ 37] |
| Scikit-learn | Decision Tree Regressor | | | GoToPage [▶ 37] |
| Scikit-learn | Extra Tree Classifier | | | GoToPage [▶ 39] |
| Scikit-learn | Extra Tree Regressor | | | GoToPage [▶ 39] |
| Scikit-learn | Extra Trees Classifier | | | GoToPage [▶ 42] |
| Scikit-learn | Extra Trees Regressor | | | GoToPage [▶ 41] |
| Scikit-learn | Random Forest Classifier | | | GoToPage [▶ 44] |
| Scikit-learn | Random Forest Regressor | | | GoToPage [▶ 44] |
| LightGBM | Random Forest Regressor | | ONNX graph must be adapted | GoToPage [▶ 45] |
| Scikit-learn | Gradient Boosting Classifier | | | GoToPage [▶ 47] |
| Scikit-learn | Gradient Boosting Regressor | | | GoToPage [▶ 47] |
| Scikit-learn | Hist Gradient Boosting Classifier | | | GoToPage [▶ 49] |
| Scikit-learn | Hist Gradient Boosting Regressor | | | GoToPage [▶ 49] |
| XGBoost | XGBClassifier | Not all configurations allow an ONNX export | Package version <= 1.5.2 or >= 1.7.4 required | GoToPage [▶ 52] |
| XGBoost | XCBRegressor | Not all configurations allow an ONNX export | Package version <= 1.5.2 or >= 1.7.4 required | GoToPage [▶ 51] |
| LightGBM | LGBMRegressor | Not all configurations allow an ONNX export | | GoToPage [▶ 55] |
| LightGBM | LGBMClassifier | | ONNX graph must be adapted | GoToPage [▶ 57] |

All samples can be downloaded here as a ZIP archive: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/13668699915/.zip

### 5.3.3 Conversion from ONNX to XML and BML

The method for converting one or more ONNX files to the Beckhoff specific XML or BML format is described below.

**Why convert ONNX to XML or BML?**

The ONNX format cannot be directly loaded to the target platform by the ML Runtime. Therefore, a conversion step is necessary in the engineering before the ML description file can be transferred to the target system. The creation of an XML file has the advantage that it is openly readable. This allows all information in the XML to be viewed during the engineering phase using a simple XML editor. The BML file is binary and the contents are therefore no longer simply viewable. In addition, a BML file is loaded considerably faster by the ML Runtime, therefore the *BML file is recommended for the delivery*.

**Which TwinCAT-specific information can be entered in the XML or BML file?**

*Creation of multi-engines*

It is possible to load more than one ML model into an instance of the FB_MllPrediction [▶ 81]. Switching between the models, referred to in the following as engines in this context, is possible without latency. This way, for example, a set of models can be trained for different work areas of the machine and the correct engine can be addressed without latency during the inference phase.

The condition for merging several models is that the model structure – the <Configuration> [▶ 74] section in Beckhoff XML – is identical for all models. For example, this means that MLPs can only be merged if their structure (number of layers, number of neurons per layer and activation functions) is the same. Only the model parameters, i.e. the weights in the example of the MLPs, may be different.

Cf. the Beckhoff ML XML description: XML Tag Parameters [▶ 75].

During merging, therefore, several AI models with an identical structure but different parameters are combined in a description file. The individual models (= engines) are loaded via a single description file in the ML Runtime. The engine ID that is to be addressed is to be transferred in each case to the Predict method [▶ 89] when calling in the PLC. The engine can be addressed via a string using the PredictRef method [▶ 90]. A GetEngineIdFromRef method [▶ 87] is also available for finding the associated ID from the reference.

Multi-engines should be regarded as an organization unit. Of course, it is also possible to instantiate several instances of a FB_MllPrediction [▶ 81] in the PLC and to load a dedicated description file into each FB.

*Minimum version of the ML Runtime driver*

When converting the description file, two entries are automatically set in the XML or BML file. One is the version of the converting component and the other is a “required version” of the ML Runtime driver. On loading the model file into TwinCAT, the “required version” is checked and a warning is output if the result of the query is false.

Cf. the Beckhoff ML XML description: XML Tag Auxiliary Specifications [▶ 74].

**Which application-specific information can be entered in the XML or BML file?**

*Input and output scalings*

The inputs of the AI model are often scaled for the training process. This scaling must then also be performed for the inference. This can either be implemented by hand in the PLC or entered directly as information in the XML or BML file. If the scaling entries are set, the scaling is performed automatically. A scaling and an offset must be specified for the scaling. The following applies:

```
y = x * Scaling + Offset
```

If scaled inputs are used for a model, back-scaling of the model output is usually also necessary. Therefore, an output scaling is available in addition to the input scaling.

**Output transformation for selected models only**

Whereas an input transformation is possible for all AI models, output transformations can only be used for models of the regression type.

Cf. the Beckhoff ML XML description: XML Tag Auxiliary Specifications [▶ 74].

*Model name, model version, model description, etc.*

For the unambiguous identification of a model description file, it is possible to add various descriptions of a model. These can be used as a free string:

- A model version
- A model name
- A model description
- A model author
- Further optional tags

Cf. the Beckhoff ML XML description: XML Tag Auxiliary Specifications [▶ 74].

*Free Custom Attributes section*

Custom Attributes are optional and may be freely used by the user. The number of attributes and the number of XML tags are not limited. The attributes are typed in the XML/BML so that the entries can be read again in the PLC. The methods GetCustomAttribute_array, GetCustomAttribute_fp64, GetCustomAttribute_int64 and GetCustomAttribute_str are available for this. See also Detailed sample [▶ 93].

**Examples of the use of Custom Attributes could be:**

- Specification of an internal version or identification number
- Specification of the input range of the individual features
- Description of the inputs and outputs
- …

Cf. the Beckhoff ML XML description: XML Tag CustomAttributes [▶ 74].

**How can I convert files and add information?**

Different interfaces are offered for the simple conversion and information modeling step in your work process:

- A GUI in the TwinCAT XAE “Machine Learning Model Manager [▶ 64]”
- A CLI “mllib_toolbox.exe [▶ 69]”
- An API as a Python package “beckhoff.toolbox [▶ 71]”

Not every interface offers the actions described above. The CLI is limited to basic applications – mainly conversion. The Python API and the GUI offer the largest functional scope.

#### 5.3.3.1 GUI

The TwinCAT 3 Machine Learning Model Manager is the central UI for the editing of ML model description files. The tool is integrated in Visual Studio and can be opened via the menu bar under **TwinCAT > Machine Learning**.

**Required Visual Studio version**

The graphic interface of the TwinCAT 3 Machine Learning Model Manager is compatible with Visual Studio 2017 and 2019 as well as the TcXaeShell. If you use a different version, you can run the interface as a standalone executable. This is located in *<TwinCATInstallDir>\3.1\Components\TcMachineLearning\ML_VS_Extension\ModelManagerStandalone.exe*.

As an alternative to the editing of ML model description files via the interface of the TwinCAT 3 Machine Learning Model Manager, you can also use a command line tool, see CLI [▶ 69], or a Python library, see Python API [▶ 71].

### Conversion of ML model files

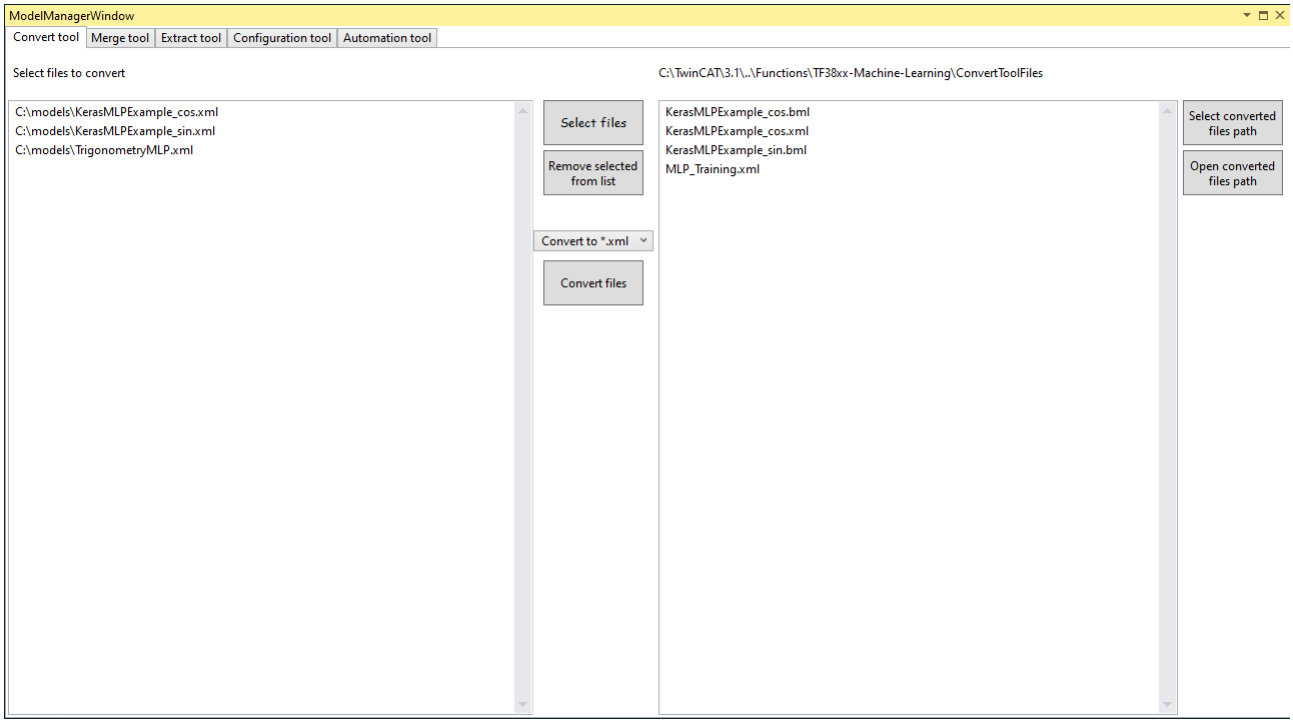
A conversion tool for ML model description files is located on the **Convert tool** tab. XML [▶ 73] and ONNX [▶ 60] files can be selected and converted to XML or BML [▶ 75] format.

**NOTICE**

**Conversion of Beckhoff BML back to XML is not provided for**

The objective of Beckhoff BML is to represent the content as a not freely readable binary file. Therefore, the conversion process from Beckhoff BML to Beckhoff XML is not provided for.

The File Browser is opened via **Select files** and ML model description files can be selected (multi-selection is possible by Ctrl + click). Selected ML model files are listed on the left-hand side with their path and file name. Files can be removed from the list again with **Remove selected from list**.




Listed ML model description files can be selected in the left-hand list (multi-selection is possible with Ctrl + click here, too) and converted with **Convert files** into the format selected in the drop-down menu. The converted files are saved in the *converted file path*. The default path is *<TwinCATPath>\Functions\TF38xx-Machine-Learning\ConvertToolFiles*. The *converted file path* can be opened in the File Browser by clicking **Open converted file path**.

The path can be changed with **Select converted files path**. The change is retained even after restarting the PC.

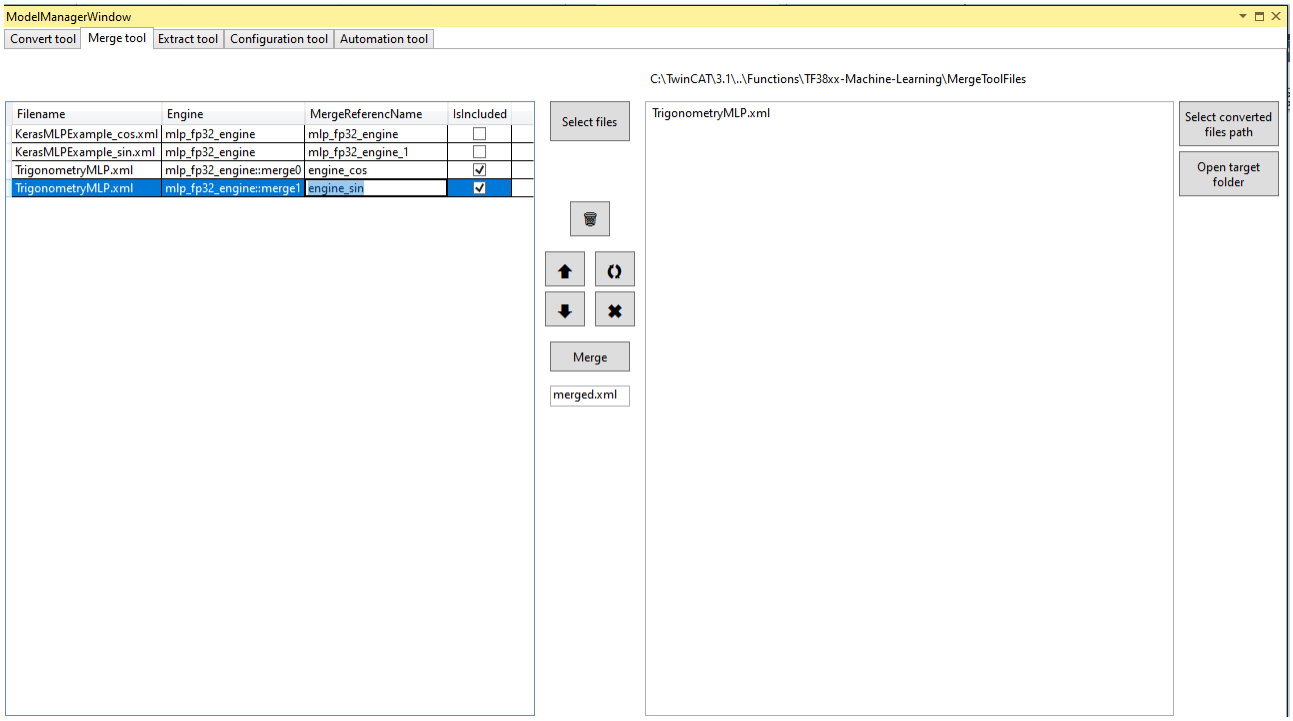
### Creating a multi-engine description

To create a multi-engine, multiple ML model description files must be loaded with **Select files**. The entries are then visible, engine-based, in the list on the left-hand side. If several engines already exist in a description file, they are listed individually in the list.

The **MergeReferenceName** field is freely editable. A reference name for the selected engine can be entered here so that this engine is addressable in the PLC via this reference, cf. PredictRef [▶ 90] and GetEngineIdFromRef [▶ 87]. If the EngineId is used in the PLC instead of the reference name, the rule is that the uppermost engine in the list bears the ID = 0 and those that follow it accordingly 1, 2, 3 and so on.

Using the **IsIncluded** checkbox you can specify whether or not a selected engine should be included in the merged description file.

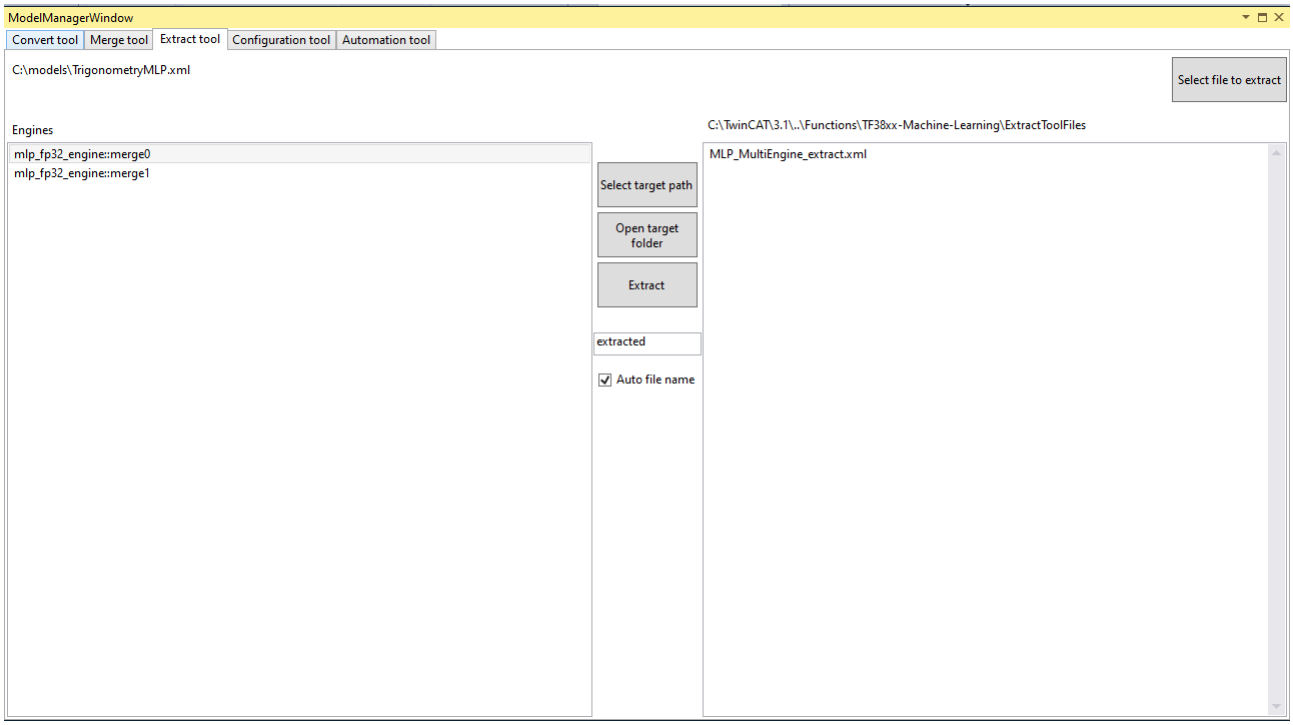
The position of the engines in the list can be manipulated in the central section of the user interface. Selected engines can be moved up or down in the list (up and down arrow). Any number of engines can be selected at the same time. Also, using the cross symbol, any number of engines can be manipulated at the same time with regard to the **IsIncluded** checkbox. If two engines are selected, their places can be swapped using the round double-arrow symbol. Engines can be removed from the list using the recycle bin symbol.




The name of the merged ML description file can be entered in the text box in the central section. With **Merge**, the file is generated and saved in the file path *<TwinCATPath>\Functions\TF38xx-Machine-Learning\MergeToolFiles*. The path can be changed with **Select target path**.

### Extracting multi-engine descriptions

Using the Extract tool it is possible to separate merged description files again. An ML model description file can be loaded using **Select file to extract**. All engines that it contains appear in the list on the left-hand side. If an engine is selected, it can be converted to a discrete ML description file using **Extract**. The name of the newly generated file is to be entered using the text box. If the **Auto file name** checkbox is active, the string in the text box is appended to the original file name. If the checkbox is inactive, only the string in the text box is used as the new file name. The newly generated file is saved in the file path *<TwinCATPath>\Functions\TF38xx-Machine-Learning\ExtractToolFiles*. The path can be changed with **Select target path**.

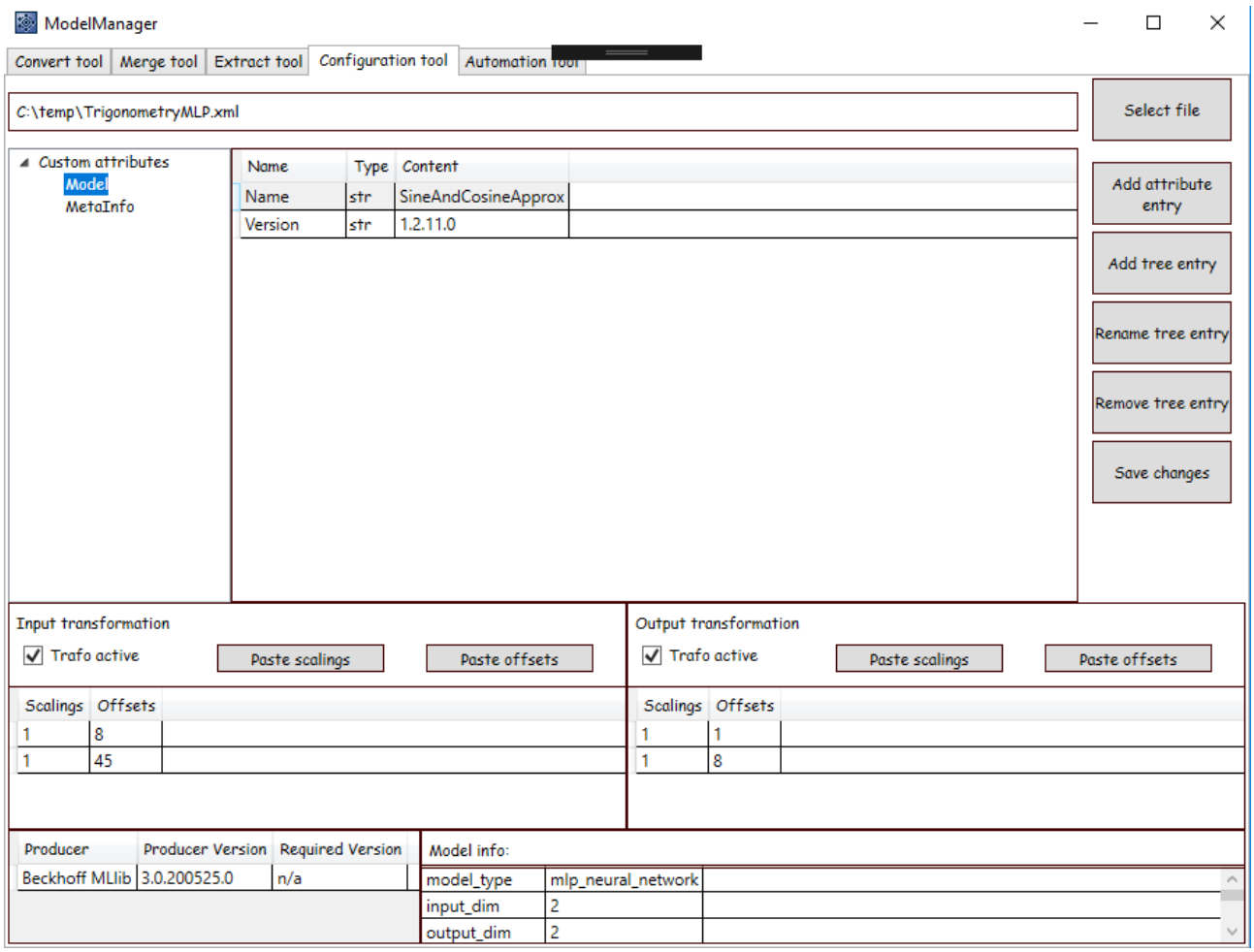



### Creating metadata for the model

The **Configuration tool** tab displays the configurator for:

- Custom attributes [▶ 73]
- Product Version and Target Version Information [▶ 74]
- Input and Output Scaling [▶ 74]

An ML description file can be selected using **Select file** and then edited. After editing, the original file is overwritten using **Save changes**.




The **Custom Attributes** are edited using the buttons:

- **Add attribute entry**: Adds an attribute to the selected tree item. The tree item must be selected in the left-hand list.
  - If an attribute is created, then the name, the data type and the value or values respectively must be specified.
  - Attributes are deleted by selecting the attribute and pressing the **Delete** button.
- **Add tree entry**: Adds a tree item under the selected tree item (as a subtree item).
- **Rename tree entry**: The selected tree item can be renamed.
- **Remove tree entry**: The selected tree item incl. the subtree items is deleted.

The editing area for **Input and Output Transformations** can be enabled by activating the **Trafo active** checkbox. Depending on the number of inputs and outputs, a corresponding number of rows is offered, in each of which scaling and offset are to be entered, cf. XML Tag Auxiliary Specifications [▶ 74]. A value from a list of numbers from the clipboard can be entered as an offset or scaling using the **Paste Scaling** and **Paste Offset** buttons. The number sequence can be separated by comma, semicolon or space. Only the number of numbers in the list must match the number of inputs or outputs respectively.

The **Producer and Target Version Information** is set automatically by the TwinCAT 3 ML Model Manager. The Target Version is determined automatically on the basis of the feature set of the model description file used. If an older ML Runtime version is used to load this model file, a warning message appears when executing the Configure method.

The number of **inputs and outputs** of the model as well as the **model type** are displayed in the lower right area of the window. This cannot be edited and is only for information.

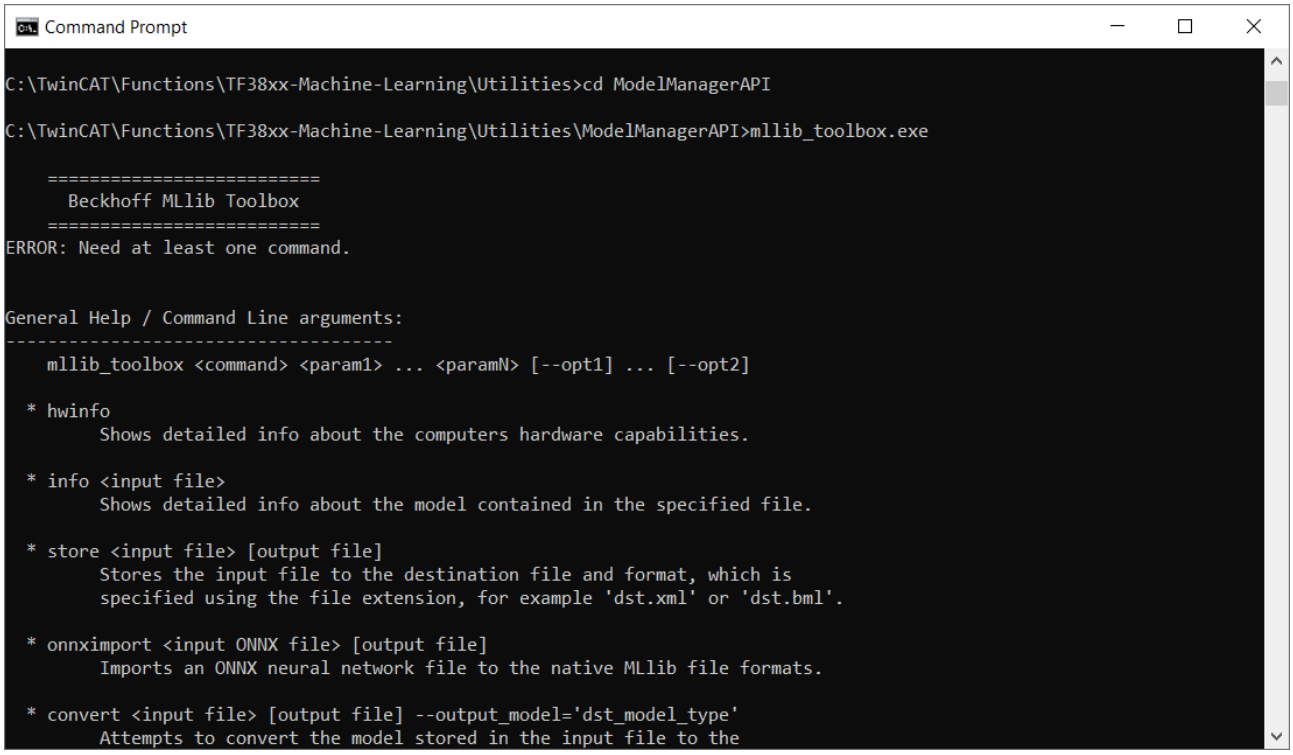
### 5.3.3.2 CLI

In addition to the GUI and the Python package, a command line tool is available for the programmatic processing of ML description files, e.g. conversion from ONNX to XML or BML: **mllib_toolbox.exe**

The executable is located in <TwinCatInstallDir>\Functions\TF38xx-Machine-Learning\Utilities\ModelManagerAPI.

#### Use of the executable

`mllib_toolbox.exe` can be used, for example, from the command prompt. The built-in help is displayed by running the exe without arguments.




You can move/copy the mllib_toolbox.exe to any other place you like on your PC. The exe uses the path environment variable, which points to the mllib_um.dll (default <TwinCatInstallDir>\3.1\Components\Base\Addins\TcMLExtension).

#### Conversion of ONNX files

ONNX files are converted using the method `onnximport`

```
mllib_toolbox.exe onnximport ".\decision tree\decisiontree-classifier.onnx" ".
\decision tree\decisiontree-classifier.bml"
```

Alternatively, an argument can also be used (--xml or –bml).

```
Mllib_toolbox onnximport myOnnxFile.onnx --xml
Mllib_toolbox onnximport myOnnxFile.onnx -bml
```

The conversion command automatically writes the “required version” (minimum version of the ML Runtime driver) in the generated XML or BML.

#### Merging and extracting engines

Use the `merge` command to create multi-engines. The following command merges the two named XML files to form a multi-engine with 2 engines. The last-named file is overwritten in the process.

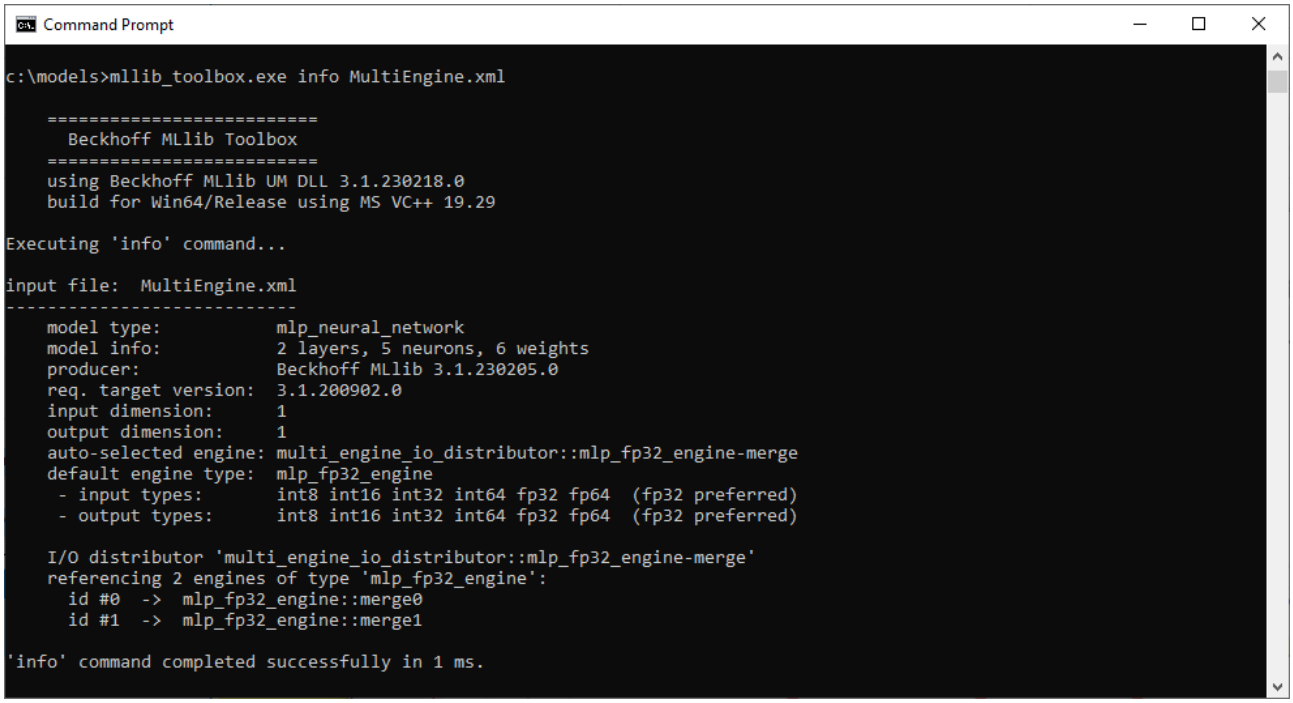
```
mllib_toolbox.exe merge KerasMLPExample_cos.xml KerasMLPExample_sin.xml
```

A new target file can also be specified. This may be both an XML and a BML file.

```
mllib_toolbox.exe merge KerasMLPExample_cos.xml KerasMLPExample_sin.xml MultiEngine.bml
```

The number of files to be combined with a command is not limited.

Use the Extract method to extract engines from a description file with multiple engines. Using the Info command, first check how many engines exist in the file – and what they are called.



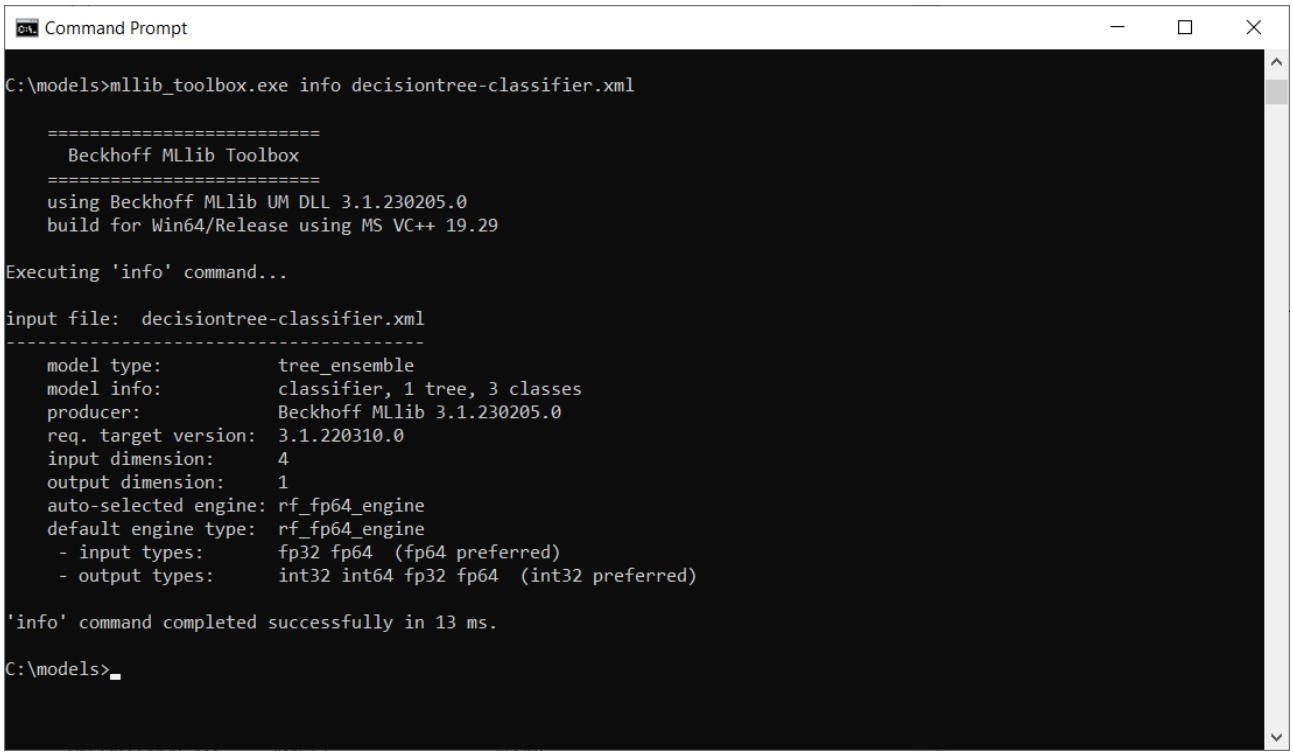

In the figure above you can see two engines with the designations mlp_fp32_engine::merge0 or merge1. To extract the first engine into the target file Extracted.xml, call:

```
mllib_toolbox.exe extract MultiEngine.xml?eng='mlp_fp32_engine::merge0' Extract.xml
```

### Displaying information from an ML model description file

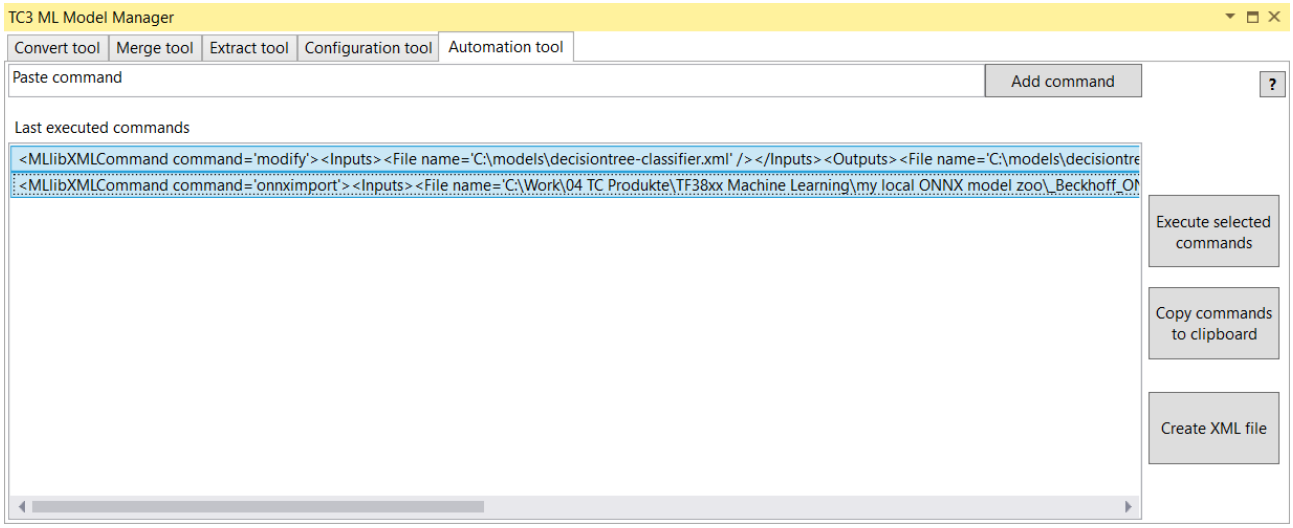
The `info` command can be used to quickly check what model is described in a description file. This can analyze ONNX files as well as XML and BML files.

```
mllib_toolbox.exe info decisiontree-classifier.xml
mllib_toolbox.exe info decisiontree-classifier.bml
mllib_toolbox.exe info decisiontree-classifier.onnx
```



### Execution of an instruction list

You can perform operations graphically in the Machine Learning Model Manager. Each operation that you perform in a session is written as an XML command on the “Automation Tool” tab.




In the figure above, for example, a conversion command and a command to enter Custom Attributes can be seen. Mark the commands that you wish to export and select **Create XML file**. Pay attention to the order in which you select the commands in the Machine Learning Model Manager. The order of selection determines the order of the commands in the exported file.

You can use the generated XML file to reproduce the command order via

```
mllib_toolbox.exe rawxml AutomationToolExport.xml
```

### Custom Attributes, Scalings and Model description

These properties are not available in the CLI. Use the Python package [▶ 71] or the TwinCAT Machine Learning Model Manager [▶ 64] for that. The CLI is limited to basic functions.

## 5.3.3.3 Python API

### Installation of the Python package

The Python package is stored as a whl file in the folder <TwinCatInstallDir>\Functions\TF38xx-Machine-Learning\Utilities\ModelManagerAPI\PythonPackage.

To install the package, use `pip install` <TwinCatInstallDir>\Functions\TF38xx-Machine-Learning\Utilities\ModelManagerAPI\PythonPackage\<whl-file-name>. The folder may contain different versions of the package (only if you have installed a new setup on top of an old TwinCAT Machine Learning Setup). **Make sure you always use the current version**.

```
pip  install  "C:\TwinCAT\Functions\TF38xx-Machine-
Learning\Utilities\ModelManagerAPI\PythonPackage\beckhoff_toolbox-3.1.230205-py3-none-any.whl"
```

### Use of the Beckhoff toolbox

For a description of the individual points such as Custom Attributes and Model Description, see: Conversion from ONNX to XML and BML [▶ 63].

The following source code is also available as a py file. See: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/13668699915/.zip in the PythonAPI_mllib directory.

The package is loaded with

```
import  beckhoff.toolbox as  tb
```

**Conversion** of an ONNX file to XML

```
tb.onnximport("../decision tree/decisiontree-regressor.onnx", "../decision tree/decisiontree-
regressor.xml")
```

```
tb.onnximport("../decision tree/decisiontree-regressor.onnx", "../decision tree/decisiontree-
regressor.bml")
```

Display of **model information**

```
tb.info("../decision  tree/decisiontree-regressor.onnx")
```

```
tb.info("../decision  tree/decisiontree-regressor.xml")
```

Add **Custom Attributes**

```
new_ca = { 'nID' : -34234, 'bTested' : True, 'fNum' : 324.3E-12, 'AnotherTreeItem' : { 'fPi' : 3.134
12, 'bFalseFlag' : False }}
tb.modify_ca("../decision tree/decisiontree-regressor.xml","../decision tree/decisiontree-regressor-
custom.xml", new_ca)
```

Add **Model Description** (name, version, author, etc. of a model)

```
model_description = {
    "new_version" : "2.3.1.0",
    "new_name" : "CurrentPreControlAxis42",
    "new_desc":"This is the most awesome model to control Axis42",
    "new_author":"Max",
    "new_tags":"awesome, ingenious, astounding",
}
tb.modify_md("../decision tree/decisiontree-regressor-custom.xml", "../decision tree/decisiontree-
regressor-md2.xml",
             **model_description)
```

Add **Input and Output Transformations**

> **Output Transformations only for selected models**
>
> Whereas an input transformation is possible for all AI models, output transformations can only be used for models of the regression type.

```
# add input / output scalings (input-output-transformations)
tb.modify_iot("../decision tree/decisiontree-regressor-md.xml", "../decision tree/decisiontree-
regressor-iot.xml",
              tb.iot_scaled_offset([1.2,3.4,1.0,1.1,1.0,1.0,1.0,1.0,1.0,1.0],
[1.2,3.4,1.0,1.1,1.0,1.0,1.0,1.0,1.0,1.0]),
              None)   # no output transformation
```

The function `iot_scaled_offset` expects a list of offsets and scalings. A value must be specified for each input or output.

```
def iot_scaled_offset(offsets : list, scalings : list):
```

The number of inputs in the example above is 10, therefore a list of 10 elements is transferred. No output transformation is available for decision trees, therefore a None is transferred here.

Generate **multi-engines**

Merge several models into one XML file.

```
# # Create a Multi-Engine
# merge two XML files - output file is Merged.xml
tb.merge(['KerasMLPExample_sin.xml','KerasMLPExample_cos.xml'],'Merged.xml')

# merge two specific engines from two XML files
tb.merge([tb.input_engine('Merged.xml','mlp_fp32_engine::merge1'),
          tb.input_engine('KerasMLPExample_cos.xml','mlp_fp32_engine')],
         'DoubleMerged.xml')

# merge two specific engines from two XML files and provide a reference name for both engines in the
 target file
tb.merge([tb.input_engine('KerasMLPExample_sin.xml','mlp_fp32_engine', 'sine'),
          tb.input_engine('KerasMLPExample_cos.xml','mlp_fp32_engine', 'cosine')],
         'MergedRef.xml')

# extract a specific engine from a Multi-Engine file
tb.extract(tb.input_reference('MergedRef.xml', 'sine'),'Extract.xml')
```

**Test-Predict** for the ML Runtime in Python

Generate an XML and subsequently predict-call the ML Runtime. The ML Runtime is compiled as a DLL and is called from Python as a user-mode process. No TwinCAT Runtime is required.

```
tb.onnximport("../decision tree/decisiontree- regressor.onnx", "../decision tree/decisiontree-
 regressor.xml")
inp_file_xml = "../decision tree/decisiontree-regressor.xml"
# define input and output format and input values (10 in, 1 out)
prediction = [{'input_type': 'fp64', 'output_type': 'fp64', 'input': [-1.0,-1.0,1.0,1.4,-1.0,-1.0,1.
0,1.4,4.2,4.2]}]
# call predict method of Python-ML-Runtime
out = tb.predict(inp_file_xml,prediction)
print(out)
```

Further sample of a Classifier.

```
tb.onnximport("../decision tree/decisiontree-classifier.onnx", "../decision tree/decisiontree-
classifier.xml")
inp_file_xml = "../decision tree/decisiontree-classifier.xml"
# define input and output format and input values (4 in, 1 out)
prediction = [{'input_type': 'fp64', 'output_type': 'int32', 'input': [-1.0,-1.0,1.0,1.4]}]
# call predict method of Python-ML-Runtime
out = tb.predict(inp_file_xml,prediction)
print(out)
```

### 5.3.3.4 Description of the Beckhoff-specific XML and BML format

#### 5.3.3.4.1 Beckhoff ML XML

**Introduction to Beckhoff ML XML**

The Beckhoff-specific XML format for the representation of trained Machine Learning Models forms a core component of the TwinCAT Machine Learning Inference Engine and TwinCAT Neural Network Inference Engine. The file is created from an ONNX file [▶ 60] using the TC3 Machine Learning Model Manager [▶ 64] or the Machine Learning Toolbox [▶ 69] or the provided Python package [▶ 71].

As opposed to ONNX, the XML-based description file can map TwinCAT-specific properties. The XML guarantees an extended functional scope of the TwinCAT Machine Learning product – see for example the concept of the Multi-engines [▶ 75]. On the other hand, it ensures seamless cooperation between the creator and user of the description file - compare Input and output transformations [▶ 74] and Custom Attributes [▶ 73].

Essential areas of the Beckhoff ML XML are described below. This helps you to understand the functions it provides.

**XML Tag <MachineLearningModel>**

Obligatory tag with 2 obligatory attributes. The tag is generated automatically and may not be manipulated.

Sample:

```
<MachineLearningModel modelName="Support_Vector_Machine" defaultEngine="svm_fp64_engine">
```

The attribute `modelName` can be read in the PLC via the method GetModelName [▶ 88]. The model type that is to be loaded is identified by the model name. For example, the attribute can take the values `support_vector_machine` or `mlp_neural_network` .

The attribute modelName in this tag should not be confused with the attribute str_modelName from <ModelDescription>.

**XML Tag <CustomAttributes>**

The tag CustomAttributes is optional and may be freely used by the user. The depth of the tree and the number of attributes are not limited. Creation can take place via the TC3 Machine Learning Model Manager. The XML can also be manually edited in this area.

Attributes can be read in the PLC via the methods GetCustomAttribute_array [▶ 84], GetCustomAttribute_fp64 [▶ 85], GetCustomAttribute_int64 [▶ 86] und GetCustomAttribute_str [▶ 86]. In the XML the typification is given by the prefixes str_, int64_, fp64_ and so on.

Sample:

```
<CustomAttributes>
   <Model str_Name="TempEstimator" str_Version="1.2.11.0" />
   <MetaInfo arrfp64_InputRange="0.10000000000000001,0.90000000000000002" int64_TheAnswer="42" />
</CustomAttributes>
```

Here, a model with the name "TempEstimator" is created in the version 1.2.11.0. Thus, an array and an integer value are provided as further information. Sample code for reading the CustomAttributes can be downloaded from the Samples [▶ 93] section.

**XML Tag <AuxilliarySpecifications>**

The AuxilliarySpecifications area is optional and is subdivided into the children <PTI> and <IOModification>.

Sample:

```
<AuxiliarySpecifications>
   <PTI str_producer="Beckhoff MLlib Keras Exporter" str_producerVersion="3.0.200525.0 str_requiredV
ersion="3.0.200517.0d"/>
   <ModelDescription str_modelVersion="2.3.1.0" str_modelName="CurrentPreControlAxis42" str_modelDes
c="This is the most awesome model to control Axis42" str_modelAuthor="Max" str_modelTags="awesome,in
genious,astounding" />
   <IOModification>
      <OutputTransformation str_type="SCALED_OFFSET" fp64_offsets="0.48288949404162623" fp64_scaling
s="1.4183887951105305"/>
   </IOModification>
</AuxiliarySpecifications>
```

**<PTI>**

PTI stands for "Product Version and Target Version Information". The tool with which the XML was created and version of the tool at the time of the XML generation are specified here.

A minimum version of the executive ML Runtime can also be specified via the attribute str_requiredVersion. The query is regarded as passed if the attribute is not set. If the attribute is set, the query is regarded as passed if the ML Runtime Version is higher than or equal to the required version. If the query is not passed, i.e. if the version of the ML Runtime used is lower than the required version, then a warning is displayed when executing the Configure method.

**<IOModification>**

If inputs or outputs of the learned model are scaled in the training environment, the scaling parameters used can be integrated directly in the XML file so that TwinCAT automatically performs the scaling in the ML Runtime.

The scaling takes place by means of y = x * Scaling + Offset.

**<ModelDescription>**

Write attributes to this optional tag

- the model version str_modelVersion
- the model name str_modelName
- the model description str_modelDesc
- the author of the model str_modelAuthor
- further optional tags str_modelTags

**XML Tag <Configuration>**

The obligatory area *Configuration* describes the structure of the loaded model.

Example - SVM

```
<Configuration str_operationType="SVM_TYPE_NU_REGRESSION" fp64_cost="0.1" fp64_nu="0.3" str_kernelFu
nction="KERNEL_FN_RBF" fp64_gamma="1.0" int64_numInputAttributes="1"/>
```

Example - MLP

```
<Configuration int_numInputNeurons="1" int_numLayers="2" bool_usesBias="true">
   <MlpLayer1 int_numNeurons="3" str_activationFunction="ACT_FN_TANH"/>
   <MlpLayer2 int_numNeurons="1" str_activationFunction="ACT_FN_IDENTITY"/>
</Configuration>
```

A configuration exists once only and is generated automatically.

**XML Tag <Paramaters>**

The obligatory area Parameters substantiates the loaded model with the described <Configuration>. The learned parameters of the model are stored here, e.g. the weights of the neurons.

In the standard case, i.e. a learned model is described in an XML, the <Parameters> tag exists only once in the XML.

```
<Parameters str_engine="mlp_fp32_engine" int_numLayers="2" bool_usesBias="true">
```

Several models with identical <Configuration> can be merged via the Machine Learning Model Manager so that both models are described in a single XML. Distinction can then be made between the parameter sets by *Engines*, which is specified as an attribute for each parameter tag.

Sample:

```
<Parameters str_engine="mlp_fp32_engine::merge0" int64_numLayers="2" bool_usesBias="true">
   …
</Parameters>
<Parameters str_engine="mlp_fp32_engine::merge1" int64_numLayers="2" bool_usesBias="true">
   …
</Parameters>
<IODistributor str_distributor="multi_engine_io_distributor::mlp_fp32_engine-
merge" str_engine_type="mlp_fp32_engine" int64_engine_count="2">
   <Engine0 str_engine_name="merge0" str_reference="sin_engine" />
   <Engine1 str_engine_name="merge1" str_reference="cos_engine" />
</IODistributor>
```

Two MLPs with an identical *Configuration* were merged here. The first engine bears the ID 0 and the internal name "mlp_fp32_engine::merge0" and can be addressed by the user via the reference "sin_engine". The second engine bears the ID 1 and the internal name "mlp_fp32_engine::merge1" and the reference "cos_engine".

The ID of the engine is sequentially incremented by the value one, starting from zero. The reference is a string that can be specified in the Model Manager during *Merge*.

If several engines are merged in an XML, all engines are loaded in the ML Runtime and are available for inference. The Predict method [▶ 89] is to be transferred when calling the engine ID that is to be used. The reference for the engine can be transferred via the PredictRef method [▶ 90]. A GetEngineIdFromRef method [▶ 87] is also available for finding the associated ID from the reference. Switching between the engines is possible without latency.

There is an example of the use of multi-engines in the PLC in the Samples area.

#### 5.3.3.4.2 Beckhoff ML BML

The BML format is a binary representation of the XML-based ML description file. As a result, the format is not openly visible and the file size is smaller in comparison with ONNX and XML. This also makes the charging process of the model much faster than charging an XML.

A BML file can be generated via the TC3 Machine Learning Model Manager [▶ 64] from an XML or an ONNX file. The way back from a BML file to an XML file is **not** provided for.

## 5.4 File management of the ML description files

**File management on the Engineering PC (XAE)**

*Conversion, editing of ONNX, XML and BML*

The Machine Learning Model Manager [▶ 64] serves as a central tool for the processing and conversion of Machine Learning models. If a file is loaded and edited, the resulting file is saved in the standard case in the following folders – depending on the action performed:

- \Functions\TF38xx-Machine-Learning\ConvertToolFiles
- \Functions\TF38xx-Machine-Learning\ExtractToolFiles
- \Functions\TF38xx-Machine-Learning\MergeToolFiles

Each default folder can be adapted in the Model Manager.

If the Machine Learning Toolbox [▶ 69] is used instead of the GUI-based Machine Learning Model Manager, the newly created file is stored in the active path as long as no other path has been specifically named.

*Integrating a model into a TwinCAT solution*

If a BML or XML description file is used in a TwinCAT solution, a distinction must be made between TcCOM API [▶ 78] and the PLC API [▶ 80] with regard to the file management.

**TcCOM API**

After the integration of a description file in the TcCOM TcMachineLearningModelCycal, the corresponding description file is copied into the Visual Studio project directory and is thus part of the project: <VS Projekt>_MLInstall.

On activating the configuration, the file is copied from the Visual Studio project directory into the boot folder on the target system: \TwinCAT\3.1\Boot\ML_Boot.

**PLC API**

When using the PLC API, the file name and path of the Machine Learning model file are specified in the PLC code as `T_MaxString` – accordingly, the user must ensure that a corresponding file exists coming from the target system. This means that the description file does not become part of the Visual Studio project directory, nor is it transferred automatically to the target system.

**Transfer of the ML description files to the target system on activating the configuration**

**TcCOM API**

When using the TcCOM object **TcMachineLearningModelCycal**, the ML description file is transferred automatically from the XAE system to the XAR system.

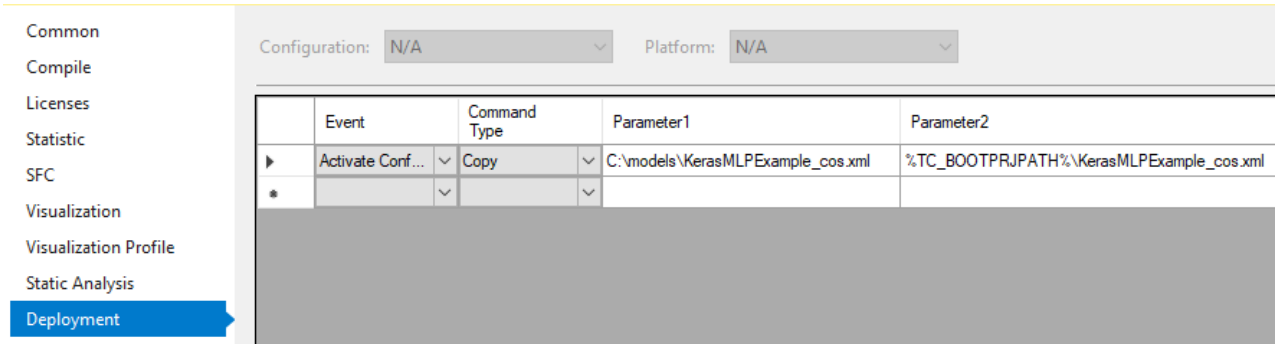
The file is transferred from the Visual Studio project folder <VS Project>_MLInstall to the Boot folder TwinCAT\3.1\Boot\ML_Boot on the XAR.

**PLC API**

If the **PLC API** is used, the user is responsible for the transfer of the ML description file. As a result, the flexibility of the application is increased on the one hand, while corresponding steps have to be implemented by the user on the other.

The ML description file can be transferred to the target system in many ways. One of them is named below by way of example.

- Via the properties of the PLC project under "Deployment" it is possible to specify which files are to be transferred to the target system in the case of a certain event, for example the activation of the configuration.




**NOTICE**

**Writing rights on the target system**

The writing rights on the operating system side and the Write Filter settings must be observed.

### Updating ML description files in the field

An important scenario in machine learning is the updating of data-based algorithms in the field during the running time of a machine. Here too, distinction must be made between use of the TcCOM [▶ 78] API and use of the PLC API [▶ 80].

### TcCOM API

In the case of the TcCOM API, the update behavior is the same as with other changes in the TcCOM area. An XAE system is necessary with an ADS route to the target system. In the XAE, a new ML description file can be integrated in the TwinCAT project. The new project is then transferred to the target system by activating the configuration. Therefore, it is necessary to restart the TwinCAT runtime here.

### PLC API

If the PLC API is used, it is possible to update the ML description file on the target system without restarting the TwinCAT runtime. To do this it is merely necessary to update the ML description file on the target system and to retrigger the Configure method.

In addition, for example using ADS, you can set the PLC variable containing the full path of the ML description file to the value of the new file that was transferred beforehand to the file system and then set the state of the state machine back to "load".

# 6 API

Two ways of programming are available to the user in TwinCAT 3. Static TcCOM objects can be created and triggered via a cyclic task, or an instance can be created from and configured via the PLC.

The programming interface of the PLC offers far greater flexibility than the use of a TcCOM instance; conversely, the latter is very simple and in many cases adequate.

## 6.1 TcCOM

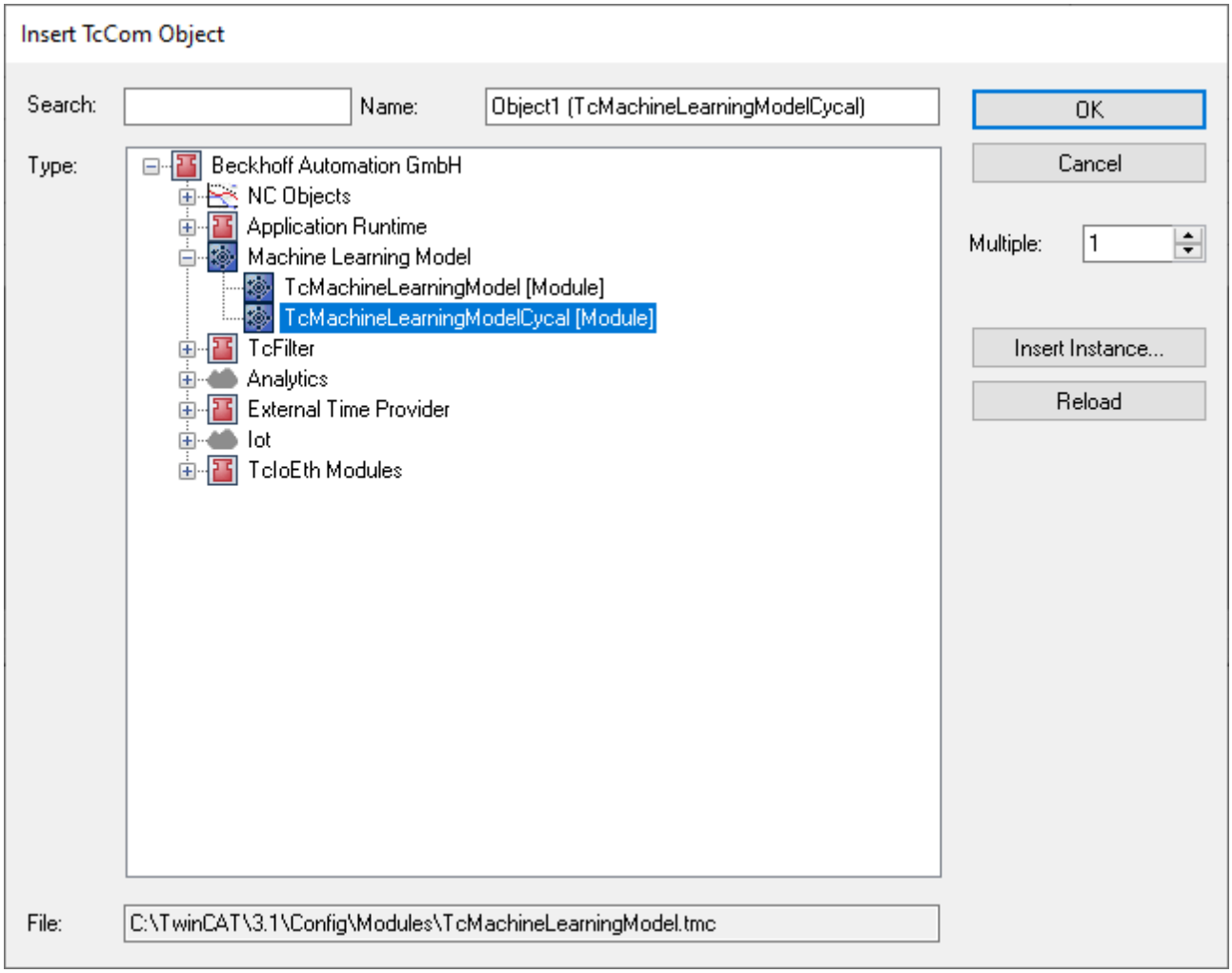
The use of a TcCOM for the inference of an ML model in TwinCAT provides a very simple possibility to execute trained models in the TwinCAT XAR. In principle, the entire procedure is documented in the quick start, so that the steps described there are initially repeated and a few further details are given below.

#### Incorporation of a model by means of TcCOM object

This section deals with the execution of machine learning models by means of a prepared TcCOM object. This interface offers a simple and clear way of loading models, executing them in real-time and generating appropriate links in your own application by means of the process image.

#### Generate a prepared TcCOM object TcMachineLearningModelCycal

1. To do this, select the node **TcCOM Objects** with the right mouse button and select **Add New Item…**

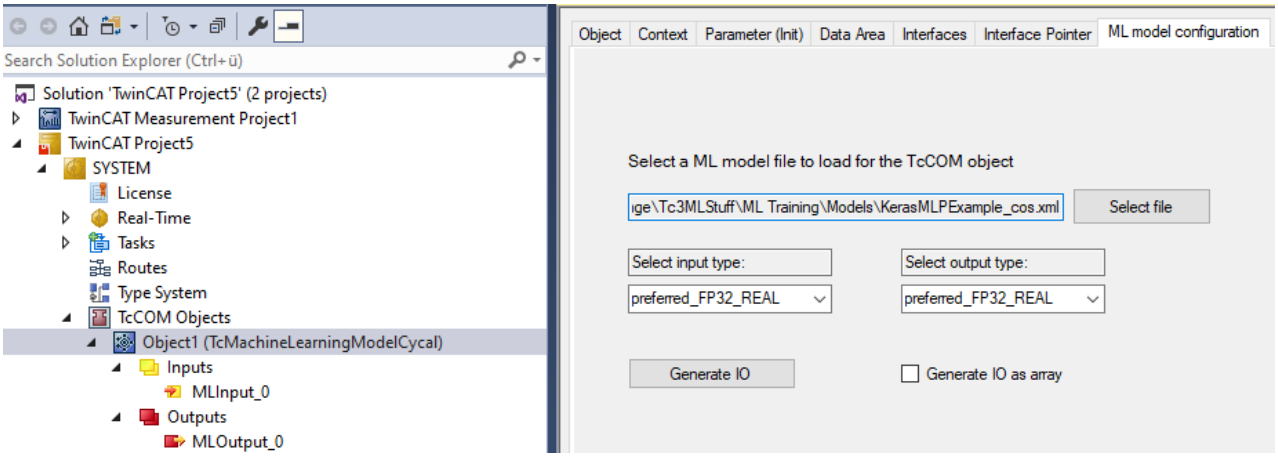



#### Under Tasks, generate a new TwinCAT task and assign this task context to the newly generated instance of TcMachineLearningModelCycal

2. To do this, open the **Context** tab of the generated object.
3. Select your generated task in the drop-down menu.

⇨ The instance of TcMachineLearningModelCycal has a tab called **ML model configuration** where you can load the description file of the ML algorithm (XML or BML) and the available data types for the inputs and outputs of the selected model are then displayed.

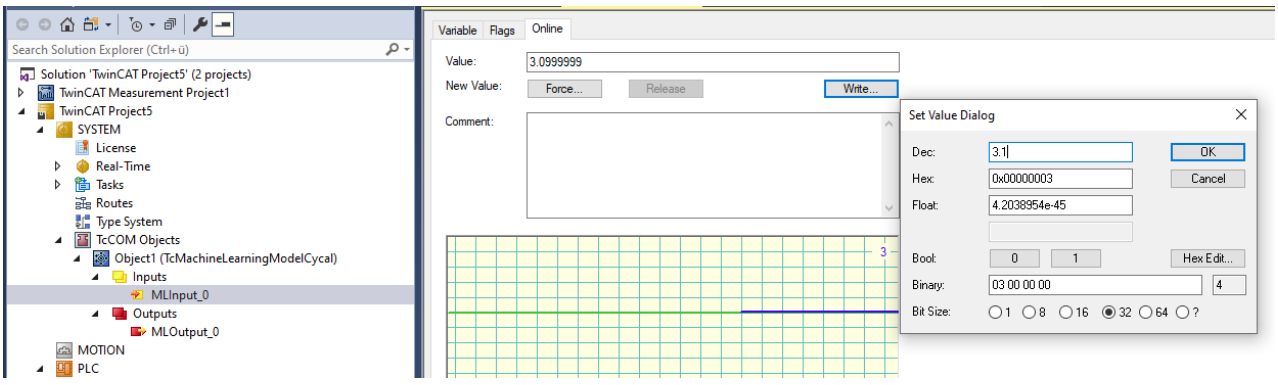
- The file does not have to be on the target system. It can be selected from the development system and is then loaded to the target system on activating the configuration.
  - A distinction is made between preferred and supported data types. The only difference is that a conversion of the data type takes place at runtime if a non-preferred type is selected. This may lead to slight losses in performance when using non-preferred data types.
- The data types for inputs and outputs are initially set automatically to the preferred data types. The process image of the selected model is created by clicking **Generate IO**. Accordingly, by loading *KerasMLPExample_cos.xml*, you get a process image with an input of the type REAL and an output of the type REAL.




**Activating the project on the target**

1. Before activating the project on a target, you must select the TF3810 license manually on the **Manage Licenses** tab under System>License in the project tree, as you wish to load a multi-layer perceptron (MLP).
2. Activate the configuration.

⇨ You can now test the model by manually writing at the input.




If the process image is larger, i.e. many inputs or outputs exist, it may be helpful not to generate each input individually as a PDO, but to define an input or output as an array type. To do this, check the checkbox **Generate IO as array** and click **Generate IO**.

Models with several engines, cf. XML Tag parameters [▶ 75], can be loaded, but only EngineId = 0 is used. Switching between the EngineIds with the TcCOM API is not provided for.

The ML description file used is automatically transferred from the Engineering system to the Runtime system on activating the configuration. File management details are described in the section File management of the ML description files [▶ 75].

# 6.2 PLC API

## 6.2.1 Datatypes

### 6.2.1.1 ETcMlIDataType

#### Syntax

Definition:

```
TYPE ETcMllDataType :
(
    E_MLLDT_UNDEFINED  := 0,
    E_MLLDT_INT8_SINT  := 10,
    E_MLLDT_INT16_INT  := 20,
    E_MLLDT_INT32_DINT := 30,
    E_MLLDT_INT64_LINT := 40,
    E_MLLDT_FP16       := 50,
    E_MLLDT_FP16B      := 55,
    E_MLLDT_FP32_REAL  := 60,
    E_MLLDT_FP64_LREAL := 70,
    E_MLLDT_SPECIAL    := 99
)BYTE;
END_TYPE
```

#### Values

| Name | Description |
|---|---|
| E_MLLDT_UNDEFINED | invalid / undefined data type |
| E_MLLDT_INT8_SINT | 8-bit signed integer number (SINT / char) |
| E_MLLDT_INT16_INT | 16-bit signed integer number (INT / short) |
| E_MLLDT_INT32_DINT | 32-bit signed integer number (DINT / long) |
| E_MLLDT_INT64_LINT | 64-bit signed integer number (LINT / long long) |
| E_MLLDT_FP16 | 16-bit IEEE floating point number (future usage) |
| E_MLLDT_FP16B | 16-bit "bfloat16" floating point number (future usage) |
| E_MLLDT_FP32_REAL | 32-bit IEEE floating point number (REAL / float) |
| E_MLLDT_FP64_LREAL | 64-bit IEEE floating point number (LREAL / double) |
| E_MLLDT_SPECIAL | Function-specific byte stream |

### 6.2.1.2 ST_MlIPredictionParameters

#### Syntax

Definition:

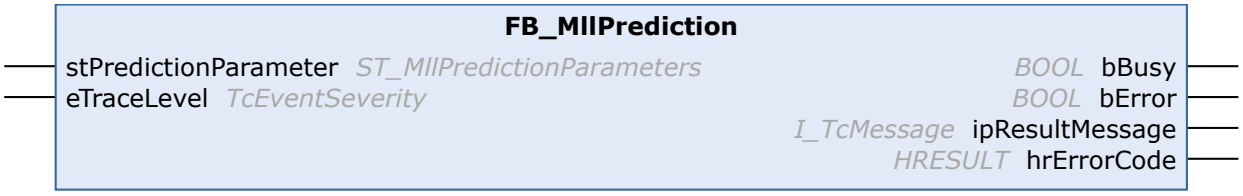
```
TYPE ST_MllPredictionParameters :
STRUCT
    MlModelFilepath : STRING(255);
    MaxConcurrency  : UINT;
END_STRUCT
END_TYPE
```

#### Parameters

| Name | Type | Default | Description |
|---|---|---|---|
| MlModelFilepath | STRING(255) | | File path of the loaded model. Either *.xml or *.bml file |
| MaxConcurrency | UINT | 1 | Maxium number of threads calling predict on the FB in the same time. |

## 6.2.2 Function Blocks

### 6.2.2.1 FB_MllPrediction




#### Syntax

Definition:

```
FUNCTION_BLOCK FB_MllPrediction
VAR_INPUT
    stPredictionParameter : ST_MllPredictionParameters;
    eTraceLevel           : TcEventSeverity;
END_VAR
VAR_OUTPUT
    bBusy                 : BOOL;
    bError                : BOOL;
    ipResultMessage       : I_TcMessage;
    hrErrorCode           : HRESULT;
END_VAR
```

#### Inputs

| Name | Type | Description |
|---|---|---|
| stPredictionParameter | ST_MllPredictionParameters [▶ 80] | General Prediction parameters |
| eTraceLevel | TcEventSeverity | Eventlogger trace level. Default = Critical. Must be set before Configure method is called |

#### Outputs

| Name | Type | Description |
|---|---|---|
| bBusy | BOOL | True if a asynchronous action is taking place |
| bError | BOOL | Indicates error in method |
| ipResultMessage | I_TcMessage | Contains the last invoked error |
| hrErrorCode | HRESULT | Unique error code |

### Methods

| Name | Description |
|---|---|
| CheckPreferredIODataTypes [▶ 83] | |
| CheckSupportedIODataTypes [▶ 83] | |
| Configure [▶ 84] | |
| GetCustomAttribute_array [▶ 84] | |
| GetCustomAttribute_fp64 [▶ 85] | |
| GetCustomAttribute_int64 [▶ 86] | |
| GetCustomAttribute_str [▶ 86] | |
| GetEngineIdFromRef [▶ 87] | |
| GetInputDim [▶ 87] | |
| GetMaxConcurrency [▶ 88] | |
| GetModelName [▶ 88] | |
| GetOutputDim [▶ 89] | |
| Predict [▶ 89] | |
| PredictRef [▶ 90] | |
| Reset [▶ 91] | |
| SetActiveEngineOptions [▶ 92] | |

#### General information

The `FB_MllPrediction` is a central Function Block for the usage of TC3 Machine Learning in the PLC. The Function Block offers a variety of Methods as described above. Basically, the `FB_MllPrediction` offers the functionality to load and to execute ML models. Hence, it is an interface to the TwinCAT 3 integrated inference engine (ML Runtime).

#### Error handling

Note that all methods of `FB_MllPrediction` return a `BOOL` which indicates if the execution on the method caused any error, e.g.

```
bFailed := fbprediction.GetInputDim(nInputDim);
```

The evaluation of the return value of the methods is equivalent to the evaluation of the Output `bError` of `FB_MllPrediction`.

Further, `FB_MllPrediction` references the TwinCAT 3 EventLogger and thus ensures that information (events) is provided via the standardized interface I_TcMessage. The trace level can be adjusted using TcEventSeverity.

#### Sample Code

Sample code for the usage of the Function Block is available here [▶ 93].

**System Requirements**

### 6.2.2.1.1 CheckPreferredIODataTypes


**Syntax**

Definition:

```
METHOD CheckPreferredIODataTypes : BOOL
VAR_INPUT
    fmtInputType  : ETcMllDataType;
    fmtOutputType : ETcMllDataType;
    IsPreferred   : Reference To BOOL;
END_VAR
```

**Inputs**

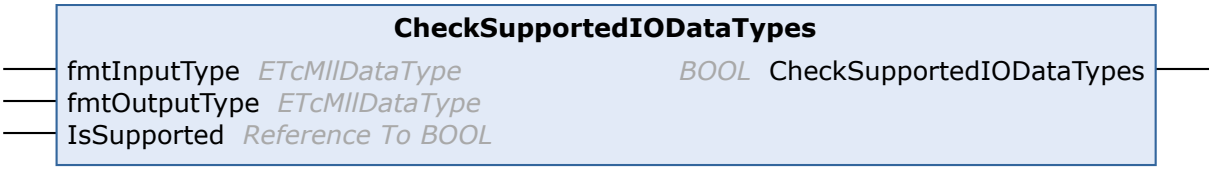
| Name | Type | Description |
|---|---|---|
| fmtInputType | ETcMllDataType [▶ 80] | Input type to check |
| fmtOutputType | ETcMllDataType [▶ 80] | Output type to check |
| IsPreferred | Reference To BOOL | Return true if preferred typo |

**Return value**

BOOL

A distinction is made between preferred and supported data types. The only difference is that a conversion of the data type takes place at runtime if a non-preferred type is selected. This may lead to slight losses in performance when using non-preferred data types.

### 6.2.2.1.2 CheckSupportedIODataTypes




**Syntax**

Definition:

```
METHOD CheckSupportedIODataTypes : BOOL
VAR_INPUT
    fmtInputType  : ETcMllDataType;
    fmtOutputType : ETcMllDataType;
    IsSupported   : Reference To BOOL;
END_VAR
```

**Inputs**

| Name | Type | Description |
|---|---|---|
| fmtInputType | ETcMllDataType [▶ 80] | Input type to check |
| fmtOutputType | ETcMllDataType [▶ 80] | Output type to check |
| IsSupported | Reference To BOOL | returns true if supported |

### Return value

BOOL

## 6.2.2.1.3 Configure


### Syntax

Definition:

```
METHOD Configure : BOOL
```

### Return value

BOOL

The method loads the specified ML model description file and configures the inference engine. Specify all settings using the `stPredictionParameter` before calling the configure method.

```
fbPredict.stPredictionParameter.MlModelFilepath := 'C:/myModel.xml';
fbPredict.stPredictionParameter.MaxConcurrency := 1;
bConfigured := fbPredict.Configure();
```

## 6.2.2.1.4 GetCustomAttribute_array


### Syntax

Definition:

```
METHOD GetCustomAttribute_array : BOOL
VAR_INPUT
    sCustomAttributeName : T_MaxString;
    fmtAttributeDataType : Reference To ETcMllDataType;
    pDataBuffer          : PVOID;
    nDataBufferLen       : UDINT;
    nArrayLength         : Reference To UDINT;
    pnBytesWritten       : Pointer To UDINT;
END_VAR
```

### Inputs

| Name | Type | Description |
|---|---|---|
| sCustomAttributeName | T_MaxString | Name of the custom attribute |
| fmtAttributeDataType | Reference To ETcMllDataType [▶ 80] | Data format of the custom attribute |
| pDataBuffer | PVOID | Destination data buffer, where the custom attribute is copied into |
| nDataBufferLen | UDINT | Length of the destination data buffer in bytes |
| nArrayLength | Reference To UDINT | Number of data type elements (i.e. number of fp32 values) found in the custom attribute |
| pnBytesWritten | Pointer To UDINT | Returns the number of bytes that have been written to the destination buffer |

### Return value

BOOL

Methods reads a custom attribute specified by `sCustomAttributeName` of type Array. Refer to this sample code [▶ 93] showing how to read an array of LREAL.

## 6.2.2.1.5 GetCustomAttribute_fp64


### Syntax

Definition:

```
METHOD GetCustomAttribute_fp64 : BOOL
VAR_INPUT
    sCustomAttributeName : T_MaxString;
    lrDataOut            : Reference To LREAL;
END_VAR
```

### Inputs

| Name | Type | Description |
|---|---|---|
| sCustomAttributeName | T_MaxString | Name of the custom fp64 attribute |
| lrDataOut | Reference To LREAL | Output value of the custom fp64 attribute |

### Return value

BOOL

Methods reads a custom attribute specified by `sCustomAttributeName` of type LREAL.

```
sCustomKey : T_MaxString := XmlTreeItem/XmlAttribute';
fValue : LREAL;
```

```
fbprediction.GetCustomAttribute_fp64(sCustomKey, fValue);
```

### 6.2.2.1.6 GetCustomAttribute_int64


#### Syntax

Definition:

```
METHOD GetCustomAttribute_int64 : BOOL
VAR_INPUT
    sCustomAttributeName : T_MaxString;
    nDataOut             : Reference To LINT;
END_VAR
```

#### Inputs

| Name | Type | Description |
|---|---|---|
| sCustomAttributeName | T_MaxString | Name of the custom int64 attribute |
| nDataOut | Reference To LINT | Output value of the custom int64 attribute |

#### Return value

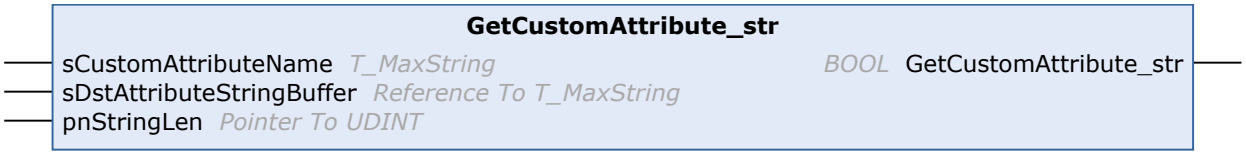
BOOL

Methods reads a custom attribute specified by `sCustomAttributeName` of type LINT.

```
sCustomKey : T_MaxString := XmlTreeItem/XmlAttribute';
iValue : LINT;
```

```
fbprediction.GetCustomAttribute_int64(sCustomKey, iValue);
```

### 6.2.2.1.7 GetCustomAttribute_str




#### Syntax

Definition:

```
METHOD GetCustomAttribute_str : BOOL
VAR_INPUT
    sCustomAttributeName      : T_MaxString;
    sDstAttributeStringBuffer : Reference To T_MaxString;
    pnStringLen               : Pointer To UDINT;
END_VAR
```

#### Inputs

| Name | Type | Description |
|---|---|---|
| sCustomAttributeName | T_MaxString | Name of the custom string attribute |
| sDstAttributeStringBuffer | Reference To T_MaxString | Pointer to a string buffer to write the custom string attribute into |
| pnStringLen | Pointer To UDINT | (Optional) Actual length of the string attribute |

### Return value

BOOL

Methods reads a custom attribute specified by `sCustomAttributeName` of type T_MaxString.

```
sCustomKey : T_MaxString := XmlTreeItem/XmlAttribute';
sValue : T_MaxString ;
```

```
fbprediction.GetCustomAttribute_str(sCustomKey, sValue);
```

## 6.2.2.1.8 GetEngineIdFromRef


### Syntax

Definition:

```
METHOD GetEngineIdFromRef : BOOL
VAR_INPUT
    sEngineRef : T_MaxString;
    nEngineId  : Reference To UDINT;
END_VAR
```

### Inputs

| Name | Type | Description |
|---|---|---|
| sEngineRef | T_MaxString | Reference string of model engine (or parameter set) used for prediction, use default value 0 if there are no multi-engines used |
| nEngineId | Reference To UDINT | Id of model engine (or parameter set) |

### Return value

BOOL

In case of multi engines in an ML model description file, engines can be selected via ID or reference name. While the Predict-method expects an ID as input, the PredictRef method expects a reference name as input.

`GetEngineIdFromRef` can convert a reference name into an engine ID. Reference names can be found in the Beckhoff ML XML file inside the <IODistributor> section, see XML attribute str_reference, and can be set via the TC3 Machine Learning Model Manager, see Merge Tool.

## 6.2.2.1.9 GetInputDim


### Syntax

Definition:

```
METHOD GetInputDim : BOOL
VAR_INPUT
    nInputDim : Reference To UDINT;
END_VAR
```

### Inputs

| Name | Type | Description |
|---|---|---|
| nInputDim | Reference To UDINT | Size of the input data array |

### Return value

BOOL

## 6.2.2.1.10 GetMaxConcurrency


### Syntax

Definition:

```
METHOD GetMaxConcurrency : BOOL
VAR_INPUT
    nConcurrency : Reference To UDINT;
END_VAR
```

### Inputs

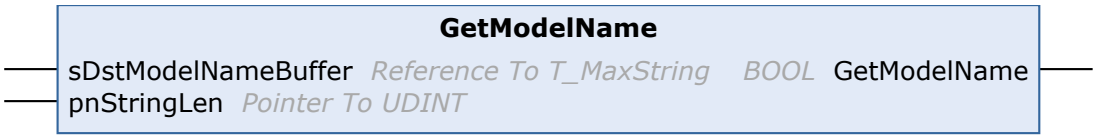
| Name | Type | Description |
|---|---|---|
| nConcurrency | Reference To UDINT | Maximum supported number of concurrently processing threads |

### Return value

BOOL

Methods reads out current configuration of inference engine regarding possible number of concurrently processing threads. This value is set by the user using `stPredictionParameter` before calling Configure method.

## 6.2.2.1.11 GetModelName




### Syntax

Definition:

```
METHOD GetModelName : BOOL
VAR_INPUT
    sDstModelNameBuffer : Reference To T_MaxString;
    pnStringLen         : Pointer To UDINT;
END_VAR
```

### Inputs

| Name | Type | Description |
|---|---|---|
| sDstModelNameBuffer | Reference To T_MaxString | Name of the loaded model |
| pnStringLen | Pointer To UDINT | (Optional) Actual length of the string attribute |

### Return value

BOOL

The method reads the model name of the loaded ML description file. The model name is an identifier of the machine learning model type. Returned strings can be for example ‘support_vector_machine’ or ‘mlp_neural_network’.

## 6.2.2.1.12 GetOutputDim


### Syntax

Definition:

```
METHOD GetOutputDim : BOOL
VAR_INPUT
    nOutputDim : Reference To UDINT;
END_VAR
```

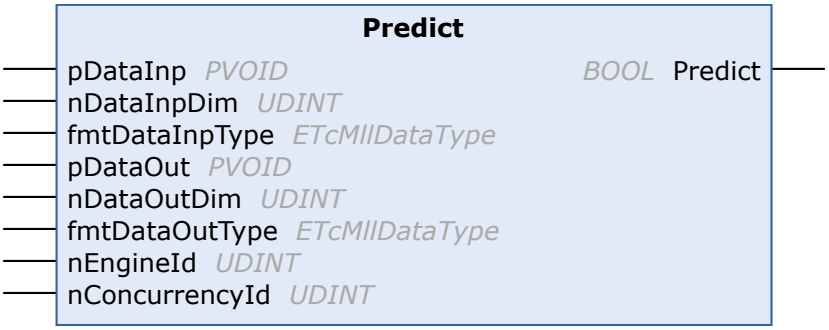
### Inputs

| Name | Type | Description |
|---|---|---|
| nOutputDim | Reference To UDINT | Size of the output data array |

### Return value

BOOL

## 6.2.2.1.13 Predict




### Syntax

Definition:

```
METHOD Predict : BOOL
VAR_INPUT
    pDataInp       : PVOID;
    nDataInpDim    : UDINT;
    fmtDataInpType : ETcMllDataType;
    pDataOut       : PVOID;
    nDataOutDim    : UDINT;
    fmtDataOutType : ETcMllDataType;
    nEngineId      : UDINT;
    nConcurrencyId : UDINT;
END_VAR
```

### Inputs

| Name | Type | Description |
|---|---|---|
| pDataInp | PVOID | Pointer to the input data array (e.g. ARRAY[0..10] OF REAL in PLC) |
| nDataInpDim | UDINT | Number of inputs in the current vector |
| fmtDataInpType | ETcMlIDataType [▶ 80] | ETcMlIDataType data type of input array |
| pDataOut | PVOID | Pointer to the output data array (e.g. ARRAY[0..10] OF REAL in PLC) |
| nDataOutDim | UDINT | Number of outputs in the current vector |
| fmtDataOutType | ETcMlIDataType [▶ 80] | ETcMlIDataType data type of output array |
| nEngineId | UDINT | Id of model engine (or parameter set) used for prediction, use default value 0 if there are no multi-engines used |
| nConcurrencyId | UDINT | Id of the processing thread. Important: Never have two concurrently processing threads use the same id. |

### Return value

BOOL

The method performs the inference of the loaded model with the given input data and stores the result in the output data. Use configure method to load a ML model description file before calling Predict method.

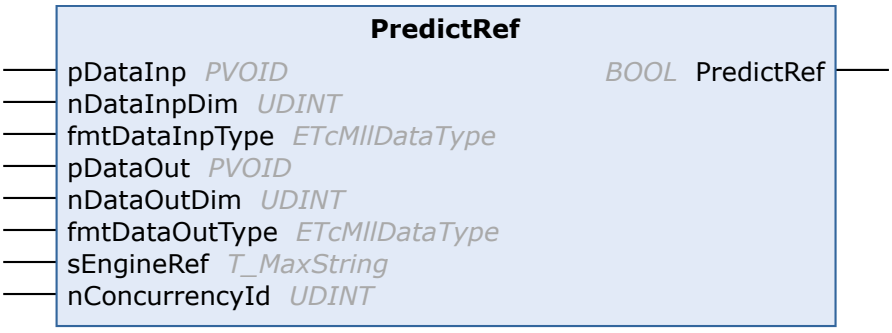
For the inputs and outputs a pointer, the number of inputs/outputs and the data type are needed.

Sample call:

```
dtype      : ETcMlIDataType.E_MLLDT_FP32_REAL;
nInputDim  : UDINT := 3;
nOutputDim : UDINT := 2;
nInput     : ARRAY[1..3] OF REAL;
nOutput    : ARRAY[1..2] OF REAL;
nCurrentEngineID : UDINT := 0;
nConcurrencyId : UDINT := 0;
```

```
fbprediction.Predict(
        pDataInp:=ADR(nInput) ,
        nDataInpDim:= nInputDim,
        fmtDataInpType:= dtype,
        pDataOut:=ADR(nOutput) ,
        nDataOutDim:= nOutputDim,
        fmtDataOutType:= dtype,
        nEngineId:= nCurrentEngineID,
        nConcurrencyId:= nConcurrencyId );
```

## 6.2.2.1.14 PredictRef




### Syntax

Definition:

```
METHOD PredictRef : BOOL
VAR_INPUT
    pDataInp       : PVOID;
    nDataInpDim    : UDINT;
    fmtDataInpType : ETcMllDataType;
    pDataOut       : PVOID;
    nDataOutDim    : UDINT;
    fmtDataOutType : ETcMllDataType;
    sEngineRef     : T_MaxString;
    nConcurrencyId : UDINT;
END_VAR
```

### Inputs

| Name | Type | Description |
|---|---|---|
| pDataInp | PVOID | Pointer to the input data array (e.g. ARRAY[0..10] OF REAL in PLC) |
| nDataInpDim | UDINT | Number of inputs in the current vector |
| fmtDataInpType | ETcMllDataType [▶ 80] | ETcMllDataType data type of input array |
| pDataOut | PVOID | Pointer to the output data array (e.g. ARRAY[0..10] OF REAL in PLC) |
| nDataOutDim | UDINT | Number of outputs in the current vector |
| fmtDataOutType | ETcMllDataType [▶ 80] | ETcMllDataType data type of output array |
| sEngineRef | T_MaxString | Reference string of model engine (or parameter set) used for prediction, use default value 0 if there are no multi-engines used |
| nConcurrencyId | UDINT | Id of the processing thread. Important: Never have two concurrently processing threads use the same id. |

### Return value

BOOL

The method performs the inference of the loaded model with the given input data and stores the result in the output data. Use configure method to load a ML model description file before calling PredictRef method.

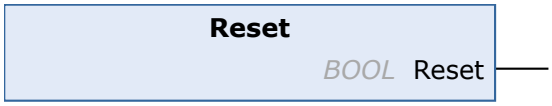
For the inputs and outputs a pointer, the number of inputs/outputs and the data type are needed.

Sample call:

```
dtype      : ETcMllDataType.E_MLLDT_FP32_REAL;
nInputDim  : UDINT := 3;
nOutputDim : UDINT := 2;
nInput     : ARRAY[1..3] OF REAL;
nOutput    : ARRAY[1..2] OF REAL;
sCurrentEngineRef: T_MaxString := 'EngineRef';
nConcurrencyId : UDINT := 0;
```

```
fbprediction.Predict(
        pDataInp:=ADR(nInput) ,
        nDataInpDim:= nInputDim,
        fmtDataInpType:= dtype,
        pDataOut:=ADR(nOutput) ,
        nDataOutDim:= nOutputDim,
        fmtDataOutType:= dtype,
        sEngineRef:= sCurrentEngineRef,
        nConcurrencyId:= nConcurrencyId );
```

## 6.2.2.1.15 Reset



### Syntax

Definition:

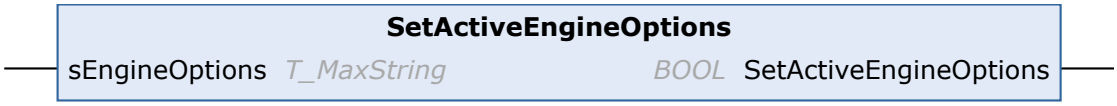
```
METHOD Reset : BOOL
```

### Return value

BOOL

This Methods resets the FB's error state.

## 6.2.2.1.16 SetActiveEngineOptions




### Syntax

Definition:

```
METHOD SetActiveEngineOptions : BOOL
VAR_INPUT
    sEngineOptions : T_MaxString;
END_VAR
```

### Inputs

| Name | Type | Description |
|---|---|---|
| sEngineOptions | T_MaxString | |

### Return value

BOOL

Input is a JSON-String with the following Key-Value-Pairs:

| Key | Value | Defaut-Value* |
|---|---|---|
| allow_FPU | TRUE / FALSE | TRUE |
| Allow_SSE3 | TRUE / FALSE | TRUE |
| Allow_AVX | TRUE / FALSE | TRUE |
| Allow_FMA | TRUE / FALSE | TRUE |
| Allow_AVX_512F | TRUE / FALSE | TRUE |

*Default-Values: The library uses as default the maximum performance. Hence, all available SIMD-extensions provided by the target PC's CPU are set to TRUE by default.

Sample Code to disable FMA and allow AVX (all others will be left unaltered):

```
fbPredict : FB_MllPrediction;
EngineOpts : T_MaxString := '{ "allow_AVX":"true", "allow_FMA":"false" }';
```

```
fbPredict.SetActiveEngineOptions(EngineOpts);
```

# 7 Samples

## 7.1 PLC API

### 7.1.1 Quick start

The sample from the section Quick start [▶ 16] can be downloaded here: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8746884875/.zip.

The ZIP contains a tszip archive (see PLC documentation, tszip) and a Beckhoff ML XML file (KerasMLPExample_cos.XML). Copy the XML file to the place defined in the PLC as the destination, or change the string to a different path.

### 7.1.2 Detailed example

The sample can be downloaded here: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8763219467/.zip.

The ZIP contains a tszip archive (see PLC documentation, tszip) and a Beckhoff ML XML file (TrigonometryMLP.XML). Copy the XML file to the place defined in the PLC as the destination, or change the string to a different path.

The ML model description file contains an MLP with an input and an output, cf. XML tag <Configuration> with `int64_numInputNeurons = 1` as well as the second (last) layer with `int64_numNeurons = 1`. Two parameter tags exist, i.e. the file contains two MLPs that are trained differently but are identical in structure (<Configuration>). One of them is an MLP that was trained to approximate a sine function, while the other is an MLP that is intended to approximate a cosine function. In the <IODistributor> area it can be seen that one engine is reachable with the reference "sin_engine" and the other with the reference "cos_engine". Some metadata are stored in the <CustomAttributes> area, e.g. the name of the model, the version and the validity range of the input variables.

As in the quick start sample, a simple state machine is run through in the PLC source code. It differs from the quick start sample in the executability of the "Configure" state and the use of several engines.

The "Configure" state shows by way of example how flexibly you can handle the number of inputs and outputs and how you can read as much information as possible from the description file and put it to use directly in the PLC.

You can switch between the two engines manually in the online view by setting the EngineId to 0 or 1.

### 7.1.3 Parallel, non-blocking access to an inference module

This sample shows how an instance of `FB_MllPrediction` can be accessed from two tasks running concurrently.

The sample can be downloaded here: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8775872011/.zip.

The instance `fbpredict` is declared in the GVL_ML. All programs in the PLC thus have access to the instance. The following are created as programs:

- P_InitML: The step sequence for initializing/loading an ML model is described here.
- P_Predict_Task1: The Predict method of the `fbpredict` is called, wherein PRG is executed on Core 1.
- P_Predict_Task2: The Predict method of the `fbpredict` is called, wherein PRG is executed on Core 2.

The essential components for the concurrent execution of 2 Predict calls are:

- The maximum number of concurrent accesses must be specified with the Configure method:
  `GVL_ML.fbpredict.stPredictionParameter.MaxConcurrency := nMaxConcurrency;`
  with nMaxConcurrency = 2.
  The instance then keeps this number of **independent** inference machines available.
- A unique ID of the calling context must be specified when calling the Predict method. These are declared as constants in P_Predict_Task1 and P_Predict_Task2, see `nConcurrencyId`.
  The user must ensure that each calling context transfers a unique ID with the Predict call.
- The remainder of the source code is largely identical to the Quick start sample [▶ 93].

# 8 Support and Service

Beckhoff and their partners around the world offer comprehensive support and service, making available fast and competent assistance with all questions related to Beckhoff products and system solutions.

### Download finder

Our download finder contains all the files that we offer you for downloading. You will find application reports, technical documentation, technical drawings, configuration files and much more.

The downloads are available in various formats.

### Beckhoff's branch offices and representatives

Please contact your Beckhoff branch office or representative for local support and service on Beckhoff products!

The addresses of Beckhoff's branch offices and representatives round the world can be found on our internet page: www.beckhoff.com

You will also find further documentation for Beckhoff components there.

### Beckhoff Support

Support offers you comprehensive technical assistance, helping you not only with the application of individual Beckhoff products, but also with other, wide-ranging services:

- support
- design, programming and commissioning of complex automation systems
- and extensive training program for Beckhoff system components

| | |
|---|---|
| Hotline: | +49 5246 963-157 |
| e-mail: | support@beckhoff.com |

### Beckhoff Service

The Beckhoff Service Center supports you in all matters of after-sales service:

- on-site service
- repair service
- spare parts service
- hotline service

| | |
|---|---|
| Hotline: | +49 5246 963-460 |
| e-mail: | service@beckhoff.com |

### Beckhoff Headquarters

Beckhoff Automation GmbH & Co. KG

Huelshorstweg 20
33415 Verl
Germany

| | |
|---|---|
| Phone: | +49 5246 963-0 |
| e-mail: | info@beckhoff.com |
| web: | www.beckhoff.com |

# 9 Appendix

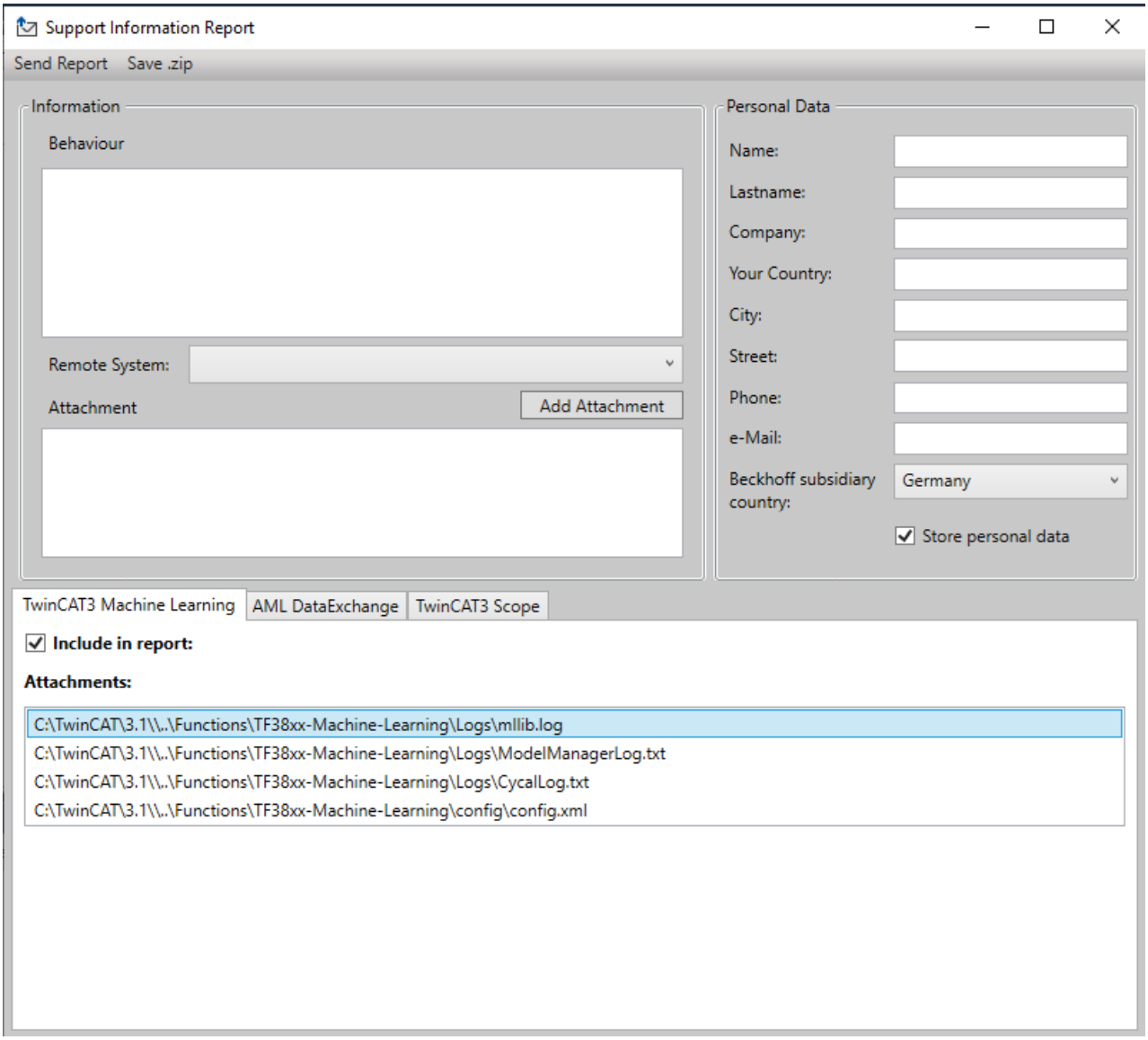
## 9.1 Log files

The log files are important for Support. They can be found under *<TwinCATInstallPath>\Functions\TF38xx-Machine-Learning\Logs*.

The files **CycalLog.txt** and **ModelManagerLog.txt** are created on the XAE system. These log the behavior during the engineering of components. CycalLog.txt contains logs regarding the configuration of TcMachineLearningModelCycal TcCOM. The TC3 Machine Learning Model Manager writes to the file ModelManagerLog.txt.

On the runtime system, the file **mllib.log** is created for logging the behavior during the machine runtime.

You can access Report an Issue via the Visual Studio menu bar under TwinCAT > Machine Learning. This dialog opens the **Support Information Report**, which assists you in sending a report to Beckhoff Support.




## 9.2 Third-party components

This software contains third-party components.

Please refer to the license file provided in the following folder for further information:
*<TwinCatInstallPath>\Functions\TF38xx-Machine-Learning\Legal*

## 9.3 XML Exporter

### Classification of the XML Exporters

The XML Exporters provided by Beckhoff can be freely used and changed. They are open source and under MIT license. This offers customers the option of adapting the XML Exporter according to their needs, for example by adding company-specific or project-specific CustomAttributes, cf.XML Tag CustomAttributes [▶ 73].

The XML exporters are provided after installation of the product in the folder *<TwinCATPath>\Functions\TF38xx-Machine-Learning\Utilities\exporter*.

**The XML Exporters are based on special versions of the libraries**

It is recommended to export created ML models via the ONNX format as well as to convert correspondingly to XML or BML. In the early phase of TwinCAT Machine Learning, the XML Exporters were intended as a transitional solution until all relevant libraries offered comprehensive ONNX support. This is the case today, so XML Exporters are no longer tested and updated with newer libraries.

### Direct XML export of an MLP

Only network architectures that have a sequential structure in the following sense are supported: each neuron of one layer is exclusively linked to each neuron of the following layer. It is possible to export layers without bias.

### Export from Keras / Tensor Flow

- File: KerasMlp2Xml.py
- Sample call: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8746685963/.zip
- Requirements for the Python environment:
  - Keras with TensorFlow backend (Tensor Flow Version 1.15.0)
  - Numpy (Version 1.17.4)
  - Matplotlib
- Supported activation functions: tanh, sigmoid, softmax, relu, linear/identity, exp, softplus, softsign
- Only **sequential models** can be exported with the XML Exporter, no **functional models**. The model is to be generated accordingly with:

```
from tensorflow.keras.models import Sequential
model = Sequential()
```

- Dropout layers are supported (only relevant for training, ignored when exporting)
- Dense layers are supported. The activation functions must be transferred to the layer as an argument and not as a discrete activation layer.

```
from keras.layers import Activation, Dense
# this will not work !!!
model.add(Dense(64))
model.add(Activation('tanh'))
# this will work
model.add(Dense(64,activation='tanh'))
```

- The API is described in detail in the header of the file KerasMlp2Xml.py.
  `net2xml(net,output_scaling_bias=None,output_scaling_scal=None)`
  - `net`, obligatory, class of the trained model
  - `output_scaling_bias`, optional, list (in case of several features), otherwise float or int
  - `output_scaling_scal`, optional, list (in case of several features), otherwise float or int
  - A string document is returned that can be saved as an XML.

### Export from MATLAB®

- File: MatlabMlp2Xml.m
- Sample call: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8746880267/.zip
- Requirements for the MATLAB® environment:
  - MATLAB®
  - Deep Learning Toolbox
- Supported activation functions: tanh, sigmoid, softmax, relu, linear/identity
- Supported models of the Deep Learning Toolbox: fitnet, patternnet
- ProcessFcns are supported by the types mapminmax and mapstd
- The use of ProcessFcns is optional
- If a ProcessFcn is used in the output layer, its activation function must be purelin
- The API is described in detail in the header of the file MatlabMlp2Xml.m
  `MatlabMlp2Xml(net, fnstr, varargin)`
- net, obligatory, class of the trained model
- fnstr, obligatory, string with path and file name
- output_scaling_bias, optional, vector
- output_scaling_scal, optional, vector

### Direct XML export of an SVM

### Export from Scikit-learn

- File: SciKitLearnSvm2Xml.py
- Sample call: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8746882571/.zip
- Requirements for the Python environment:
  - Python Interpreter: 3.6 or higher
  - Scikit-learn: Version 0.22.0 or higher
  - Matplotlib
  - Numpy
- Only numerical class labels can be exported.
- Supported classes or models: SVC, NuSVC, OneClassSVM, SVR, NuSVR
  - LinearSVR and LinearSVC are not supported by the Exporter, but can alternatively be implemented via the classes SVR and SVC, each with a linear kernel.
- Supported kernel functions: linear, rbf, sigmoid, polynomial
  - Neither individual kernel functions nor precomputed functions are supported
- Remarks about model parameters:
  - *Gamma* = scale is not supported
  - *Gamma* = auto_deprecated: it is exported gamma = 0.0
  - *Gamma* = auto: it is exported gamma = 1/n_features.
  - *C* = inf is not supported
  - *decision_function_shape* = ovr is not supported. *decision_function_shape* = ovo must be used. Default in Scikit-learn is ovr!
  - *break_ties* is ignored because *decision_function_shape* = ovr is not supported.
- The API is described in detail in the header of the file SciKitLearnSvm2Xml.py.
  `svm2xml(svm, input_scaling_bias=None, input_scaling_scal=None)`
  - `net`, obligatory, class of the trained model
  - `input_scaling_bias`, optional, list (in case of several features), otherwise float or int

- `input_scaling_scal`, optional, list (in case of several features), otherwise float or int
- A string document is returned that can be saved as an XML.

### 9.3.1 XML Exporter - samples

Small samples of the use of the XML Exporter [▶ 97] provided by Beckhoff can be downloaded here.

- Export of an MLP from Keras/TensorFlow: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8746685963/.zip
- Export of an MLP from MATLAB®: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8746880267/.zip
- Export of an SVM from Scikit-learn: https://infosys.beckhoff.com/content/1033/tf38x0_tc3_ML_NN_Inference_Engine/Resources/8746882571/.zip

More Information:
**www.beckhoff.com/tf3800**

Beckhoff Automation GmbH & Co. KG
Hülshorstweg 20
33415 Verl
Germany
Phone: +49 5246 9630
info@beckhoff.com
www.beckhoff.com