onnx_sklearn_custom

November 26, 2021

1 Convert custom transformer into ONNX

The notebook explains how to create a converter for a custom transformer following scikit-learn API.
Xavier Dupré - Senior Data Scientist at Microsoft - Computer Science Teacher at ENSAE, github/xadupre, github/sdpython.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu(last_level=2)
```

[1]:
```
<IPython.core.display.HTML object>
```

[2]:
```
import numpy as np
from pyquickhelper.helpgen import NbImage
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from jupytalk.talk_examples.sklearn2019 import (
    profile_fct_graph, onnx2str, onnx2dotnb)
from sklearn.base import TransformerMixin
from sklearn.preprocessing import MinMaxScaler
from skl2onnx import to_onnx
from onnxruntime import InferenceSession
%matplotlib inline
```

[3]:
```
from logging import getLogger
logger = getLogger('skl2onnx')
logger.disabled = True
```

Many functions are implemented in sklearn2019.py.

1.1 Custom converter

Let's implement a converter which applies of MinMaxScaler and then applies a logarithm.

[4]:
```
class MinMaxLogScaler(TransformerMixin):

    def __init__(self, feature_range=(1, 10), op_version=None):
        self.feature_range = feature_range
        self.op_version = op_version

    def fit(self, X, y=None):
        self.estimator_ = MinMaxScaler(feature_range=self.feature_range)
        self.estimator_.fit(X)
```

```
        return self

    def transform(self, X):
        X2 = self.estimator_.transform(X)
        return np.log(X2)

    def __repr__(self):
        return "{0}(feature_range={1})".format(self.__class__.__name__,
                                               self.feature_range)


X = np.array([[0, 1, 2],
              [-1, 0, 100],
              [1, 0, 3],
              [4, 4, 4]], dtype=np.float64)

tr = MinMaxLogScaler()
tr.fit(X)
tr.transform(X)
```

[4]:
```
array([[ 1.02961942e+00,  1.17865500e+00,  0.00000000e+00],
       [-2.22044605e-16,  0.00000000e+00,  2.30258509e+00],
       [ 1.52605630e+00,  0.00000000e+00,  8.78613558e-02],
       [ 2.30258509e+00,  2.30258509e+00,  1.68622712e-01]])
```

1.2 Custom conversion based on OnnxOperatorMixin

Let's rewrite the previous class by inheriting from *OnnxOperatorMixin*. We need two operators: * Scaler * Log

[5]:
```
from skl2onnx.algebra.onnx_operator_mixin import OnnxOperatorMixin
from skl2onnx.algebra.onnx_ops import OnnxScaler, OnnxLog

target_opset = 12


class MinMaxLogScalerOnnx(MinMaxLogScaler, OnnxOperatorMixin):

    def to_onnx_operator(self, inputs=None, outputs=('Y', )):
        if inputs is None:
            raise RuntimeError("inputs should contain one name")

        op = self.estimator_
        i0 = self.get_inputs(inputs, 0)
        return OnnxLog(OnnxScaler(i0, scale=op.scale_,
                                  offset = -op.min_ / (op.scale_ + 1e-8),
                                  op_version=self.op_version),
                       output_names=outputs,
                       op_version=self.op_version)

X = np.array([[0, 1, 2],
              [-1, 0, 100],
              [1, 0, 3],
```

```
              [4, 4, 4]], dtype=np.float64)

tr = MinMaxLogScalerOnnx(op_version=target_opset)
tr.fit(X)

try:
    tr.to_onnx(X.astype(np.float32))
except Exception as e:
    print(e)
```

```
Shape of output 'Y' cannot be infered. onnx_shape_calculator must be overriden
and return a shape calculator.
```

onnx cannot always infer the output shape so a new method must be added to return this information (design might evolve in the future).

[6]:
```
from skl2onnx.common.data_types import FloatTensorType


class MinMaxLogScalerOnnx(MinMaxLogScaler, OnnxOperatorMixin):

    def to_onnx_operator(self, inputs=None, outputs=('Y', )):
        if inputs is None:
            raise RuntimeError("inputs should contain one name")

        op = self.estimator_
        i0 = self.get_inputs(inputs, 0)
        return OnnxLog(
                OnnxScaler(
                    i0, scale=op.scale_.astype(np.float32),
                    offset=(-op.min_ / (op.scale_ + 1e-8)).astype(np.float32),
                    op_version=self.op_version),
                output_names=outputs,
                op_version=self.op_version)

    def onnx_shape_calculator(self):
        def shape_calculator(operator):
            operator.outputs[0].type = FloatTensorType(shape=operator.inputs[0].type.
 ↪shape)
        return shape_calculator


X = np.array([[0, 1, 2],
              [-1, 0, 100],
              [1, 0, 3],
              [4, 4, 4]], dtype=np.float64)

tr = MinMaxLogScalerOnnx(op_version=target_opset)
tr.fit(X)

model_onnx = tr.to_onnx(X.astype(np.float32))
onnx2dotnb(model_onnx)
```

[6]:
```
<jyquickhelper.jspy.render_nb_js_dot.RenderJsDot at 0x1288389c080>
```

1.2.1 Comparison with raw outputs

[7]:
```
sess = InferenceSession(model_onnx.SerializeToString())

inputs = {'X': X.astype(np.float32)}
sess.run(None, inputs)[0]
```

[7]:
```
array([[ 1.0296195e+00,  1.1786550e+00, -5.9604645e-08],
       [ 0.0000000e+00,  0.0000000e+00,  2.3025851e+00],
       [ 1.5260563e+00,  0.0000000e+00,  8.7861314e-02],
       [ 2.3025849e+00,  2.3025851e+00,  1.6862264e-01]], dtype=float32)
```

[8]:
```
from skl2onnx.helpers.investigate import compare_objects
compare_objects(tr.transform(X), sess.run(None, inputs)[0])
```

Everything is ok.

1.2.2 Custom transformer in a pipeline

[9]:
```
data = load_iris()
X, y = data.data, data.target
```

[10]:
```
pipe = Pipeline([('scaler', MinMaxLogScalerOnnx(op_version=target_opset)),
                 ('lr', LogisticRegression(multi_class="auto"))])
pipe.fit(X, y)
```

[10]:
```
Pipeline(steps=[('scaler', MinMaxLogScalerOnnx(feature_range=(1, 10))),
                ('lr', LogisticRegression())])
```

[11]:
```
pipe_onnx = to_onnx(pipe, X.astype(np.float32), target_opset=target_opset)
onnx2dotnb(pipe_onnx)
```

[11]:
```
<jyquickhelper.jspy.render_nb_js_dot.RenderJsDot at 0x12884684e10>
```

1.3 Separate converter

The previous design requires the operator to inherit from *OnnxOperatorMixin*. The conversion involves: * *scope*: every node in the graph must have a unique name, the scope ensures that it is. * *container*: internal container for all nodes added during the conversion
The converter uses an *operator*. This refers to ONNX operator, like a placeholder with named inputs and outputs which receives the ONNX nodes. When the converter is implemented, it needs to be registered so that *skl2onnx* can use it when needed in a pipeline.

[12]:
```
from skl2onnx.common.data_types import FloatTensorType


def convert_sklearn_minxmaxlog_scaler(scope, operator, container):
    # operator = placeholder for the converted scikit-learn operator
    # operator.inputs = defined inputs
    # operator.outputs : defined outputs
    # The conversion is independant from any other converted models in the pipeline.
    X = operator.inputs[0]
    out = operator.outputs[0]
    opv = container.target_opset
```

```
    # The raw operator is the scikitl-learn model to be converted.
    op = operator.raw_operator

    # The ONNX definition of the new operator which links
    # X to out.
    # X.onnx_name is the unique name of X.
    # out.onnx_name is the unique name of out.
    onnx_op = OnnxLog(OnnxScaler(X.onnx_name, scale=op.estimator_.scale_,
                                 offset = -op.estimator_.min_ / (op.estimator_.scale_␣
↪+ 1e-8),
                                 op_version=opv),
                      output_names=out.onnx_name,
                      op_version=opv)

    # Let's finally add this subgraph to the container
    # by adding the final node.
    onnx_op.add_to(scope, container)


def shape_sklearn_minxmaxlog_scaler(operator):
    # The shape calculator defines the dimension of
    # every output.
    op_input = operator.inputs[0]
    op = operator.raw_operator
    N = op_input.type.shape[0]
    C = op_input.type.shape[1]

    # This line tells the first output is a float matrix
    # which has the same dimension as the input.
    operator.outputs[0].type = FloatTensorType([N, C])


from skl2onnx import update_registered_converter

# registration of the converter.
update_registered_converter(
            MinMaxLogScaler,
            "MinMaxLogScaler",
            shape_sklearn_minxmaxlog_scaler,
            convert_sklearn_minxmaxlog_scaler)
```

[13]:
```
pipe = Pipeline([('scaler', MinMaxLogScaler()),
                 ('lr', LogisticRegression(multi_class="auto"))])
pipe.fit(X, y)
```

[13]:
```
Pipeline(steps=[('scaler', MinMaxLogScaler(feature_range=(1, 10))),
                ('lr', LogisticRegression())])
```

[14]:
```
pipe_onnx = to_onnx(pipe, X.astype(np.float32), target_opset=target_opset)
onnx2dotnb(pipe_onnx)
```

[14]:
```
<jyquickhelper.jspy.render_nb_js_dot.RenderJsDot at 0x12883e33cc0>
```

Node in ONNX may have multiple outputs. *skl2onnx* defines a default number of outputs for transformers, regressor, classifier but this default number can be changed by defining a default parser.

1.4 Appendix

[15]:
```
import onnx, skl2onnx, sklearn, onnxruntime
mods = [onnx, skl2onnx, onnxruntime, sklearn]
for m in mods:
    print(m.__name__, m.__version__)
```

```
onnx 1.7.105
skl2onnx 1.7.1076
onnxruntime 1.3.996
sklearn 0.24.dev0
```

[16]:
```

```

[17]:
```

```