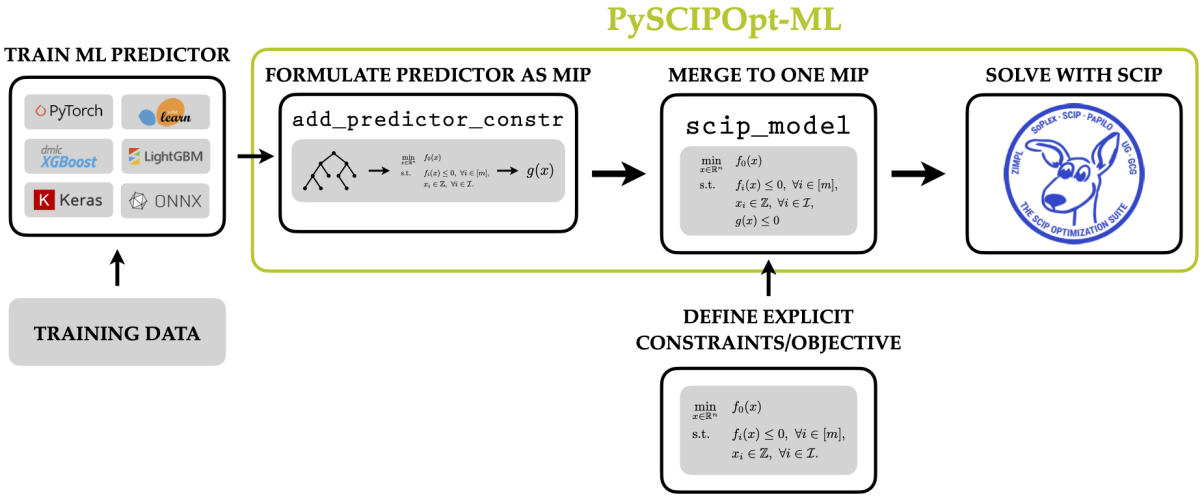
PySCIPOpt-ML

Mark Turner <turner@zib.de>

May 26, 2025

GETTING STARTED

PySCIPOpt-ML is a python interface to automatically formulate Machine Learning (ML) models into Mixed-Integer Programs (MIPs). PySCIPOPT-ML allows users to easily optimise MIPs with embedded ML constraints.



CHAPTER

ONE

INSTALLATION

`pyscipopt-ml` can be installed from PyPI using `pip`. Python 3.8 or higher is required.

```
pip install pyscipopt-ml
```

CHAPTER

TWO

HELPFUL INFORMATION

- Looking what ML models are supported? Try *Supported*
- Looking for the MIP formulations of each model? Try *Mixed Integer Formulations*
- Interested in installation options? Try the README at the GitHub Repo
- Having trouble or a feature is missing? Raise an issue at GitHub
- Want to see the API? Try *API Reference Manual*

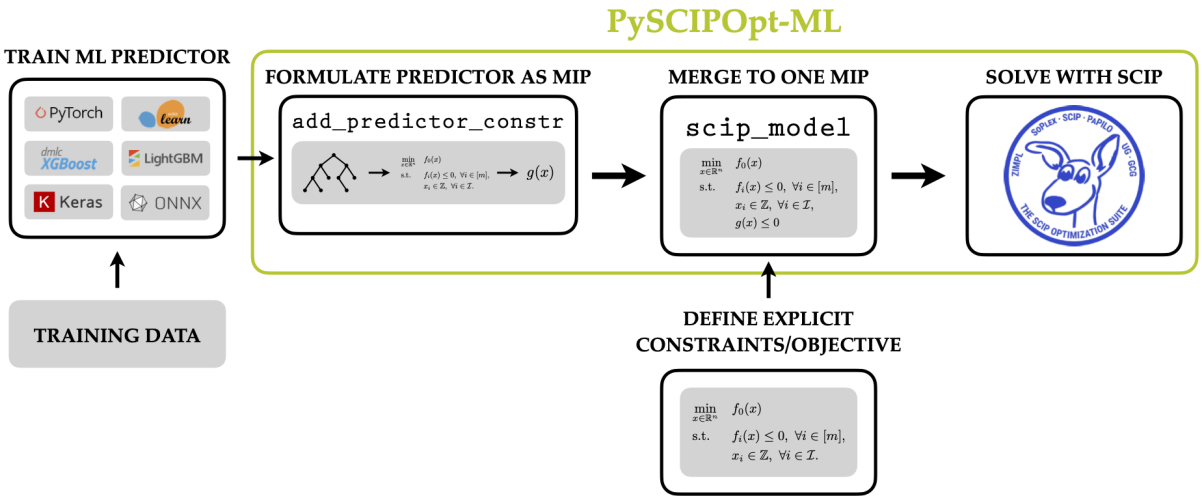
2.1 Basics

2.1.1 Introduction

PySCIPOpt-ML is a Python package for the automatic formulation of machine learning (ML) models into Mixed-Integer Programs (MIPs) using SCIP. This automatic formulation allows users to easily optimise mathematical optimisation problems with embedded ML constraints, without worrying about the exact formulation and how to extract the data from the ML interface.

The package currently supports various scikit-learn objects. It can also embed gradient boosting regression models from XGBoost, and LightGBM. Finally, it supports Sequential Neural Networks from PyTorch, Keras, and ONNX.

The package is actively developed and users are encouraged to raise an issue on GitHub if there are ML models that are currently available, or there are optimisation related problems with the created MIPs.



2.1.2 Install

We encourage to install the package via pip (or add it to your *requirements.txt* file):

```
(venv) pip install pyscipopt-ml
```

Note

If not already installed, this should install the pyscipopt and numpy packages.

Note

The package can also be installed from source. To do so first clone the package and then run:

```
python -m pip install .
```

Note

The following table lists the version of the relevant packages that are tested and supported.

Package
pyscipopt
numpy
torch
tensorflow
scikit-learn
lightgbm
xgboost
onnx

Installing any of the machine learning packages is only required if the predictor you want to insert uses them (i.e. to insert a Scikit-Learn based predictor you need to have scikit-learn installed).

2.1.3 Usage

The main function provided by the package is *pyscipopt_ml.add_predictor_constr()*. It takes as arguments: a PySCIPOpt Model, a *supported ML model*, input PySCIPOpt variables, and output PySCIPOpt variables.

By calling the function, the PySCIPOpt Model is augmented with variables and constraints so that, in a solution, the values of the output variables are predicted by the regression model from the values of the input variables. More formally, if we denote by g the prediction function of the embedded ML model, by x the input variables and by y the output variables, then $y = g(x)$ in any solution.

The function *add_predictor_constr* returns a modeling object derived from the class `AbstractPredictorConstr`. That object keeps track of all the variables and constraints that have been added to the PySCIPOpt to establish the relationship between input and output variables of the ML model.

The modeling object can perform a few tasks:

- It can print a summary of what it added with the `print_stats` method.
- Once SCIP computed a solution to the optimization problem, it can compute the difference between what the ML model predicts from the input values and the values of the output variables in SCIP's solution with the `get_error` method.

The function *add_predictor_constr* is a shorthand that should add the correct model for any supported ML model, but individual functions for each ML model are also available. For the list of frameworks and ML models supported, and the corresponding functions please refer to the *supported* section. We also briefly outline how the various ML models are formulated in SCIP in the *Mixed Integer Formulations* section.

For examples on how to use the package please refer to the the *example*.

2.2 Basic Example - Function Approximation

2.2.1 Explanation and Description

In this tutorial, we will see how to embed a trained ML predictor into a SCIP Model.

The scenario for this basic example is that we have two non-linear functions, which we will approximate by some ML technique (in this case a neural network). The aim of the optimisation problem is to maximise the approximation of the first function, while satisfying some equality constraint on the second. We can describe this scenario mathematically.

Let $\mathbf{x}$ be the input to both functions. Let $f(\mathbf{x})$ be the first function, and $g(\mathbf{x})$ be the second function. Their approximations via some ML technique are $f'(\mathbf{x})$ and $g'(\mathbf{x})$. The MIP is:

$$\begin{aligned} \min \quad & f'(\mathbf{x}) \\ \text{s.t.} \quad & g'(\mathbf{x}) = 10 \end{aligned}$$

2.2.2 Code Walkthrough

To begin with, we first need to be able to generate the non-linear functions:

```
def build_random_quadratic_functions(seed=42, n_inputs=5, n_samples=1000):
    # Set the random seed so the results don't change
    np.random.seed(seed)

    # Generate two random quadratic functions f(x) = x^{t}Qx + Ax + c
    quadratic_coefficients = np.round(
        np.random.uniform(0, 5, size=(2, n_inputs, n_inputs)), decimals=3
    )
    linear_coefficients = np.round(np.random.uniform(0, 1, size=(2, n_inputs)),␣
↪decimals=3)
    constant = np.round(np.random.uniform(0, 1, size=(2,)), decimals=3)

    # Generate data from the quadratic function
    X = np.random.uniform(-10, 10, size=(n_samples, n_inputs))
    y_1 = np.zeros((n_samples,))
    y_2 = np.zeros((n_samples,))
    for i in range(n_samples):
        y_1[i] = (
            X[i] @ quadratic_coefficients[0] @ X[i].T
            + linear_coefficients[0] @ X[i].T
```

(continues on next page)

(continued from previous page)

```
            + constant[0]
        )
        y_2[i] = (
            X[i] @ quadratic_coefficients[1] @ X[i].T
            + linear_coefficients[1] @ X[i].T
            + constant[1]
        )

    return X, y_1, y_2
```

Now that we have the ability to create training data for two non-linear functions, let us make a general function for creating the SCIP model that we will embed the trained ML predictors.

```
def build_basic_scip_model(n_inputs):
    # Initialise a SCIP Model
    scip = Model()

    # Create the input variables
    input_vars = np.zeros((1, n_inputs), dtype=object)
    for i in range(n_inputs):
        # Tight bounds are important for MIP formulations of neural networks. They often␣
↪drastically improve
        # performance. As our training data is in the range [-10, 10], we pass that as␣
↪bounds [-10, 10].
        # These bounds will then propagate to other variables.
        input_vars[0][i] = scip.addVar(name=f"x_{i}", vtype="C", lb=-10, ub=10)

    # Create the output variables. (Note that these variables will be automatically␣
↪constructed if not specified)
    output_vars = np.zeros((2, 1), dtype=object)
    for i in range(2):
        output_vars[i] = scip.addVar(name=f"y_{i}", vtype="C", lb=None, ub=None)

    # Now set additional constraints and set the objective
    scip.addCons(output_vars[1][0] == 10, name="fix_output_reg_2")
    scip.setObjective(output_vars[0][0])

    return scip, input_vars, output_vars
```

We now are only lacking the trained ML predictor to insert into the SCIP model. So we now train a ML predictor In the example below, we provide a function that has the ability to train either a Scikit-Learn MLPRegressor, or a PyTorch fully connected Sequential model with ReLU activation functions.

```
def build_and_optimise_function_approximation_model(
seed=42, n_inputs=5, n_samples=1000, sklearn_or_torch="sklearn", layers_sizes=(20, 20,␣
↪10)
):
    assert len(layers_sizes) == 3

    X, y_1, y_2 = build_random_quadratic_functions(
        seed=seed, n_inputs=n_inputs, n_samples=n_samples
    )
```

(continues on next page)

(continued from previous page)

```
if sklearn_or_torch == "sklearn":
    reg_1 = MLPRegressor(
        random_state=seed,
        hidden_layer_sizes=(layers_sizes[0], layers_sizes[1], layers_sizes[2]),
    ).fit(X, y_1.reshape(-1))
    reg_2 = MLPRegressor(
        random_state=seed,
        hidden_layer_sizes=(layers_sizes[0], layers_sizes[1], layers_sizes[2]),
    ).fit(X, y_2.reshape(-1))
else:
    torch.random.manual_seed(seed)
    reg_1 = nn.Sequential(
        nn.Linear(n_inputs, layers_sizes[0]),
        nn.ReLU(),
        nn.Linear(layers_sizes[0], layers_sizes[1]),
        nn.ReLU(),
        nn.Linear(layers_sizes[1], layers_sizes[2]),
        nn.ReLU(),
        nn.Linear(layers_sizes[2], 1),
    )
    reg_2 = nn.Sequential(
        nn.Linear(n_inputs, layers_sizes[0]),
        nn.ReLU(),
        nn.Linear(layers_sizes[0], layers_sizes[1]),
        nn.ReLU(),
        nn.Linear(layers_sizes[1], layers_sizes[2]),
        nn.ReLU(),
        nn.Linear(layers_sizes[2], 1),
    )

    # Convert data into PyTorch tensors
    X_tensor = torch.tensor(X, dtype=torch.float32)
    y_1_tensor = torch.tensor(y_1, dtype=torch.float32)
    y_2_tensor = torch.tensor(y_2, dtype=torch.float32)

    # Create a DataLoader for handling batches
    dataset_1 = TensorDataset(X_tensor, y_1_tensor)
    dataset_2 = TensorDataset(X_tensor, y_2_tensor)
    batch_size = 32
    dataloader_1 = DataLoader(dataset_1, batch_size=batch_size, shuffle=True)
    dataloader_2 = DataLoader(dataset_2, batch_size=batch_size, shuffle=True)

    # Initialise the loss function and optimizer
    criterion = nn.MSELoss()
    optimizer_1 = optim.Adam(reg_1.parameters(), lr=0.001, weight_decay=0.0001)
    optimizer_2 = optim.Adam(reg_2.parameters(), lr=0.001, weight_decay=0.0001)

    # Training loop
    for epoch in range(200):
        for batch_X, batch_y in dataloader_1:
            # Forward pass
```

(continues on next page)

(continued from previous page)

```
                outputs = reg_1(batch_X)

                # Calculate loss
                loss = criterion(outputs, batch_y.view(-1, 1))  # Assuming y is a 1D␣
↪array

                # Backward pass and optimization
                optimizer_1.zero_grad()
                loss.backward()
                optimizer_1.step()
            for batch_X, batch_y in dataloader_2:
                # Forward pass
                outputs = reg_2(batch_X)

                # Calculate loss
                loss = criterion(outputs, batch_y.view(-1, 1))  # Assuming y is a 1D␣
↪array

                # Backward pass and optimization
                optimizer_2.zero_grad()
                loss.backward()
                optimizer_2.step()

    # Now build the SCIP Model and embed the neural networks
    scip, input_vars, output_vars = build_basic_scip_model(n_inputs)
    mlp_cons_1 = add_predictor_constr(
        scip, reg_1, input_vars, output_vars[0], unique_naming_prefix="reg_1_"
    )
    mlp_cons_2 = add_predictor_constr(
        scip, reg_2, input_vars, output_vars[1], unique_naming_prefix="reg_2_"
    )

    return scip
```

To execute the above code we can now run:

```
# Get the SCIP Model with the embedded trained predictors
scip = build_and_optimise_function_approximation_model()

# Optimize the model
scip.optimize()

# We can check the "error" of the MIP embedding via the difference between SKLearn /␣
↪Torch and SCIP output
if np.max(mlp_cons_1.get_error()) > 10**-3:
    error = np.max(mlp_cons_1.get_error())
    raise AssertionError(f"Max error {error} exceeds threshold of {10 ** -3}")
if np.max(mlp_cons_2.get_error()) > 10**-3:
    error = np.max(mlp_cons_2.get_error())
    raise AssertionError(f"Max error {error} exceeds threshold of {10 ** -3}")
```

2.3 Advanced Example - Wine Manufacturer

2.3.1 Explanation and Description

In this example we take the point of view of a wine manufacturer.

Thanks to available open source data Cortez *et al.* [CCA+09], which was accessed here, we can build a ML predictor for wine quality given a set of attributes of the wine. This can either be modelled as a regression model, with the quality being in the range [0,1], or some quality threshold can be set, and the task made into a classification task. For this exercise we model it is a regression task.

The goal of the MIP in the following example is to create a diverse bouquet of wines with the highest average quality. To do this, we make an assumption that the features of the grapes and resulting wine are identical. In this example we can purchase grapes from a set of suppliers, all of whom have grapes that by themselves would result in low quality wine. However, we can blend the grapes such that the predicted quality of the wine increases. For additional constraints, each supplier has a limit on the amount of grapes that can be sold, and an associated cost per unit, where we have a total budget. Finally, we assume that each wine of the created bouquet will have the same volume.

We describe the MIP formulation mathematically:

- Let I be the index set of the wine bouquet that will be created.
- Let J be the index set of features for each grape / wine.
- Let K be the index set of vineyards where the grapes can be purchased
- Let $x_{i,j}$ be the value of feature j for wine i
- Let y_i be the quality of the wine i
- Let $m_{i,k}$ be the amount of grapes from vineyard k used in wine i
- let f be the ML predictor that determines the quality of a wine
- Let v[j][k] be the constant values of feature j from vineyard k
- Let c[k] be the constant cost of one unit of grapes from vineyard k
- Let a[k] be the amount of grapes available from vineyard k
- Let B be the total budget available

$$
\begin{aligned}
\text{max} \quad & \sum_i y_i \\
s.t. \quad & x_{i,j} = \sum_k m_{i,k} * v[j][k] \quad \forall i, j \\
& \sum_k m_{i,k} = 1 \quad \forall i \\
& \sum_i m_{i,k} \leq a[k] \quad \forall k \\
& y_i = f(x_{i,1}, ..., x_{i,|J|}) \quad \forall i \\
& \sum_{i,k} m_{i,k} * c[k] \leq B
\end{aligned}
$$

2.3.2 Code Walkthrough

Below we will introduce a function for creating this example with a variety of ML frameworks that can train gradient boosting decision trees and random forests.

This example is taken directly from one of the tests in the GitHub repository. See more examples by searching there.

```
def build_and_optimise_wine_manufacturer(
    seed=42,
    n_vineyards=35,
    n_wines_to_produce=5,
    min_wine_quality=4.25,
    sklearn_xgboost_lightgbm="sklearn",
    gbdt_or_rf="rf",
    n_estimators=3,
    max_depth=3,
    epsilon=0.0001,
):
    assert sklearn_xgboost_lightgbm in ("sklearn", "xgboost", "lightgbm")
    assert gbdt_or_rf in ("gbdt", "rf")

    # Path to red wine data
    data_dict = read_csv_to_dict("./tests/data/wineQualityReds.csv")

    features = [
        "fixed.acidity",
        "volatile.acidity",
        "citric.acid",
        "residual.sugar",
        "chlorides",
        "free.sulfur.dioxide",
        "total.sulfur.dioxide",
        "density",
        "pH",
        "sulphates",
        "alcohol",
    ]
    n_features = len(features)
    budget = 1.7 * n_wines_to_produce

    # Generate the actual input data arrays for the ML predictors
    X = []
    quality = np.array([float(x) for x in data_dict["quality"]]).reshape(
        -1,
    )
    for feature in features:
        X.append(np.array([float(x) for x in data_dict[feature]]))
    X = np.swapaxes(np.array(X), 0, 1)

    # Train the ML predictor
    if sklearn_xgboost_lightgbm == "sklearn":
        if gbdt_or_rf == "rf":
            reg = RandomForestRegressor(
                random_state=seed, n_estimators=n_estimators, max_depth=max_depth
```

(continues on next page)

(continued from previous page)

```
            ).fit(X, quality)
        else:
            reg = GradientBoostingRegressor(
                random_state=seed, n_estimators=n_estimators, max_depth=max_depth
            ).fit(X, quality)
    elif sklearn_xgboost_lightgbm == "xgboost":
        if gbdt_or_rf == "gbdt":
            reg = XGBRegressor(
                random_state=seed, n_estimators=n_estimators, max_depth=max_depth
            ).fit(X, quality)
        else:
            reg = XGBRFRegressor(
                random_state=seed, n_estimators=n_estimators, max_depth=max_depth
            ).fit(X, quality)
    else:
        if gbdt_or_rf == "gbdt":
            reg = LGBMRegressor(
                random_state=seed, n_estimators=n_estimators, max_depth=max_depth
            ).fit(X, quality)
        else:
            reg = LGBMRegressor(
                random_state=seed,
                n_estimators=n_estimators,
                max_depth=max_depth,
                boosting_type="rf",
                bagging_freq=1,
                bagging_fraction=0.5,
            ).fit(X, quality)

    # Create artificial data from some vineyards
    np.random.seed(seed)
    vineyard_order = np.arange(X.shape[0])
    np.random.shuffle(vineyard_order)
    vineyard_litre_limits = np.random.uniform(0.25, 0.35, n_vineyards)
    vineyard_costs = np.random.uniform(1, 2, n_vineyards)
    vineyard_features = []
    low_quality_vineyards_i = 0
    for i in vineyard_order:
        if low_quality_vineyards_i >= n_vineyards:
            break
        if quality[i] <= 5:
            low_quality_vineyards_i += 1
            vineyard_features.append(X[i])
    vineyard_features = np.array(vineyard_features)

    # Create the SCIP Model
    scip = Model()

    # Create variables deciding the features of each wine
    feature_vars = np.zeros((n_wines_to_produce, n_features), dtype=object)
    quality_vars = np.zeros((n_wines_to_produce, 1), dtype=object)
    wine_mixture_vars = np.zeros((n_wines_to_produce, n_vineyards), dtype=object)
```

(continues on next page)

(continued from previous page)

```
    for i in range(n_wines_to_produce):
        quality_vars[i][0] = scip.addVar(vtype="C", lb=0, ub=10, name=f"quality_{i}")
        for j in range(n_features):
            max_val = np.max(X[:, j])
            min_val = np.min(X[:, j])
            lb = max(0, min_val - 0.1 * max_val)
            ub = 1.1 * max_val
            feature_vars[i][j] = scip.addVar(vtype="C", lb=lb, ub=ub, name=f"feature_{i}_
↪{j}")
        for k in range(n_vineyards):
            wine_mixture_vars[i][k] = scip.addVar(
                vtype="C", lb=0, ub=vineyard_litre_limits[k], name=f"mixture_{i}_{k}"
            )

    # Now create constraints on the wine blending
    for i in range(n_wines_to_produce):
        for j in range(n_features):
            scip.addCons(
                feature_vars[i][j]
                == quicksum(
                    wine_mixture_vars[i][k] * vineyard_features[k][j] for k in range(n_
↪vineyards)
                ),
                name=f"mixture_cons_{i}_{j}",
            )
    for i in range(n_wines_to_produce):
        scip.addCons(
            quicksum(wine_mixture_vars[i][k] for k in range(n_vineyards)) == 1,
            name=f"wine_mix_{i}",
        )
    for k in range(n_vineyards):
        scip.addCons(
            quicksum(wine_mixture_vars[i][k] for i in range(n_wines_to_produce))
            <= vineyard_litre_limits[k],
            name=f"vineyard_limit_{k}",
        )

    # Add the budget constraint
    scip.addCons(
        quicksum(
            quicksum(wine_mixture_vars[i][k] * vineyard_costs[k] for k in range(n_
↪vineyards))
            for i in range(n_wines_to_produce)
        )
        <= budget,
        name=f"budget_cons",
    )

    # Add the ML constraint. Add in a single batch!
    pred_cons = add_predictor_constr(
        scip, reg, feature_vars, quality_vars, unique_naming_prefix="wine_",␣
```

(continues on next page)

(continued from previous page)

```
↪epsilon=epsilon
    )

    # Add a constraint ensuring minimum wine quality on those produced
    for i in range(n_wines_to_produce):
        scip.addCons(quality_vars[i][0] >= min_wine_quality, name=f"min_quality_{i}")

    # Set the SCIP objective
    scip.setObjective(
        quicksum(quality_vars[i][0] for i in range(n_wines_to_produce)) / n_wines_to_
↪produce,
        sense="maximize"
    )

    return scip
```

Two important things to note in the insertion of the ML predictors.

- While it was a single function call to insert the ML predictor, the ML predictor was actually added n many times. Specifically, the same ML predictor was inserted for each of the generated wines
- When using decision trees, or any ML predictors that are based on decision trees, it is important to be aware of the epsilon value. In the above example we set a default value of 0.0001. This ensures that the output of the ML predictor matches the ML framework, but it removes a small portion of the feasible region, risking getting an infeasible model for a feasible problem. This is by default 0, because while the error can be arbitrarily large for decision trees, the error stems only from a numerically insignificant perturbation of the input

Back to the example: We can then create various models with different characteristics and different ML frameworks by changing the parameters of the input. For example, to insert ML predictors from XGBoost using Random Forests, we would do the following:

```
# Build the SCIP model with embedded ML predictors
scip = build_and_optimise_wine_manufacturer(
    seed=42,
    n_vineyards=35,
    n_wines_to_produce=5,
    min_wine_quality=4.25,
    sklearn_xgboost_lightgbm="xgboost",
    gbdt_or_rf="rf",
    n_estimators=3,
    max_depth=3,
    epsilon=0.0001
    )

# Optimise the SCIP model
scip.optimize()

# We can check the "error" of the MIP embedding via the difference between SKLearn and␣
↪SCIP output
if np.max(pred_cons.get_error()) > 10**-3:
    error = np.max(pred_cons.get_error())
    raise AssertionError(f"Max error {error} exceeds threshold of {10 ** -3}")

return scip
```

2.4 Library of Examples - SurrogateLIB

For those interested in more applications, we refer to *SurrogateLIB*. SurrogateLIB is an accompanying library of instances that are generated using PySCIPOPt-ML. The generators for the instances also serve as the tests for the repository, and are located in the `tests` directory. A complete description of each instance and the instance `.mps` files themselves are located here

2.5 Supported ML Models

The package currently supports various Scikit-Learn objects. It can also embed gradient boosting regression models from XGBoost, and LightGBM. Finally, it supports Sequential Neural Networks from PyTorch and Keras. In *Mixed Integer Formulations*, we briefly outline the MIP formulations used for the various ML models.

2.5.1 Scikit-learn

The following table lists the name of the models supported, the name of the corresponding object in the Python framework, and the function that can be used to insert it in a SCIP model.

Table 1: Supported ML models of scikit-learn

Machine Learning Model	Scikit-learn object	Functions to insert
Ordinary Least Square	`LinearRegression` `Ridge` `ElasticNet` `Lasso`	*add_linear_regression_constr*
Partial Least Square	`PLSRegression` `PLSRegression`	*add_pls_regression_constr*
Logistic Regression	`LogisticRegression`	*add_logistic_regression_constr*
Neural-network	`MLPRegressor` `MLPClassifier`	*add_mlp_regressor_constr* *add_mlp_classifier_constr*
Decision tree	`DecisionTreeRegressor` `DecisionTreeClassifie`	*add_decision_tree_regressor_constr* *add_decision_tree_classifier_constr*
Gradient boosting	`GradientBoostingRegre` `GradientBoostingClass`	*add_gradient_boosting_regressor_constr* *add_gradient_boosting_classifier_constr*
Random Forest	`RandomForestRegressor` `RandomForestClassifie`	*add_random_forest_regressor_constr* *add_random_forest_classifier_constr*
Support Vector Machines	`SVR` `SVC` `LinearSVR` `LinearSVC`	*add_support_vector_regressor_constr* *add_support_vector_classifier_constr*
Centroid Clustering	`KMeans` `MiniBatchKMeans`	*add_centroid_cluster_constr*
Pipeline	`Pipeline`	*add_pipeline_constr*
MultiOutput	`MultiOutputClassifier` `MultiOutputRegressor`	*add_multi_output_classifier_constr* *add_multi_output_regressor_constr*

2.5.2 PyTorch

In PyTorch, only `torch.nn.Sequential` objects are supported.

They can be embedded in a SCIP model with the function *pyscipopt_ml.torch.add_sequential_constr()*.

Currently, only five types of layers are supported:

- `Linear layers`,
- `ReLU layers`,
- `Sigmoid layers`,
- `Tanh layers`
- `Softmax layers`
- `Softplus layers`

In the case of the final layer being an activation function used for classification, e.g. `Softmax`, simply set *output_type=="classification"* when inserting the predictor constraint. The result is that the class with highest value is assigned value 1 and all other classes are assigned value 0. Essentially, explicitly modelling the final activation function for classification purposes is unnecessary from a MIP perspective as the maximum value is preserved after the function is applied.

2.5.3 Keras

For Keras, only keras.Model and keras.Sequential are supported.

They can be embedded in a SCIP model with the function *pyscipopt_ml.keras.add_keras_constr()*.

The supported layer types and activation functions are the same as in torch (see above). This support holds for the classification case when the final layer is an unsupported activation function, e.g. softmax. Please read the above explanation in the PyTorch section, and in such use cases set *output_type="classification"* when inserting the predictor constraint.

2.5.4 ONNX

For ONNX we also support standard feed forward neural networks. These must be provided in the ModelProto format.

They can be embedded in a SCIP model with the function *pyscipopt_ml.onnx.add_onnx_constr()*.

The supported layer types and activation functions are the same as in torch and keras (see above). The classification trick for the final layer is not done for ONNX models, so be warned that there will be a performance difference for imported classification models.

2.5.5 XGBoost

Models for XGBoost's Scikit-Learn interface can be embedded in a SCIP model. The following table lists the name of the models supported, the name of the corresponding object in the Python framework, and the function that can be used to insert it in a SCIP model.

Table 2: Supported ML models of xgboost

XGBoost object	Function to insert
xgboost.XGBRegressor	*add_xgbregressor_constr*
xgboost.XGBClassifier	*add_xgbclassifier_constr*
xgboost.XGBRFRegressor	*add_xgbregressor_rf_constr*
xgboost.XGBRFClassifier	*add_xgbclassifier_rf_constr*

Currently only "gbtree" boosters are supported.

2.5.6 LightGBM

Models for LightGBM's Scikit-Learn interface can be embedded in a SCIP model. The following table lists the name of the models supported, the name of the corresponding object in the Python framework, and the function that can be used to insert it in a SCIP model.

Table 3: Supported ML models of lightgbm

LightGBM object	Function to insert
lightgbm.LGBMRegressor	*add_lgbregressor_constr*
lightgbm.LGBMClassifier	*add_lgbclassifier_constr*

Currently "gbdt" and "rf" boosters are supported.

2.6 Mixed Integer Formulations

In this page, we give a quick overview of the mixed-integer formulations used to represent the various machine learning (ML) models supported by the package.

Throughout, we denote by x the input to the ML model (i.e. the independent variables) and by y the output of the ML model (i.e. the dependent variables).

For all examples we will consider only a single sample as input. That sample depending on the model will have multiple features and multiple outputs, being either continuous (regression) or discrete (classification). Inputting more samples, which can be done in all cases by this package, creates the same set of constraints per sample.

2.6.1 Argmax Formulation

These formulations are used in most cases of classification, i.e., when the output needs to be categorical. Different formulations are used depending on the dimension of the output y. For all formulations of argmax there are potential issues when multiple classes have approximately the largest value. For these cases we will use ϵ to explain when issues can arise, where by default in SCIP $\epsilon = 10^{-6}$. Unlike standard argmax, instead of returning the index of the largest argument, we want the index of the largest argument to take value 1 and all other indices to take value 0. We denote y' as the output of the ML model before argmax has been applied.

In the scenario where y is single dimensional (which normally makes an argmax redundant), we interpret argmax as checking whether the value is greater than *0.5* or not.

$$y \leq y' + 0.5$$
$$y \geq y' - 0.5$$

This formulation has an issue of tolerances around 0.5. So values in the range $[0.5 - \epsilon, 0.5 + \epsilon]$ can all be either 0 or 1.

In the scenario where y is two-dimensional, we formulate argmax with indicator constraints

$$\begin{aligned} y_0 = 1 \Rightarrow y'_0 \leq y'_1 \\ y_1 = 1 \Rightarrow y'_1 \leq y'_0 \\ y_0 + y_1 = 1 \end{aligned}$$

This formulation also has an issue with tolerances. If both y'_0 and y'_1 take values differing by less than ϵ, then either y_0 or y_1 can be selected by SCIP.

In the scenario where y is three-dimensional or more, we formulate argmax with SOS constraints. Let $m \in \mathbb{R}$ be an additional variable used to find the maximum value, and let $s \in \mathbb{R}^n_{\geq 0}$ be additional slack variables.

$$\begin{aligned} & y'_i + s_i + m = 0 \quad \forall i \in \{0, ..., n-1\} \\ & SOS1(s_i, y_i) \quad \forall i \in \{0, ..., n-1\} \\ & \sum_{i \in \{0,...,n-1\}} y_i = 1 \end{aligned}$$

This formulation has an issue with tolerances when there are classes with output values less than ϵ smaller than the largest output value. In such a case SCIP can select either the largest value label or any of those ϵ close class labels as argmax.

For ease of notation, this constraint will simply be referred to as argmax(y')

2.6.2 Linear Regression

Denoting by $\beta \in \mathbb{R}^{n+1}$ the computed weights of linear regression, its model takes the form

$$y = \sum_{i=1}^{n} \beta_i x_i + \beta_0.$$

Since this is linear, it can be represented directly in SCIP using linear constraints. Note that the model fits other techniques such as Ridge, Lasso, and ElasticNet.

2.6.3 Logistic Regression

The standard logistic function, also referred to as the sigmoid function in some communities, is $f(x) = \frac{1}{1+e^{-x}}$.

In the case of regression, and a single dimensional output, the logistic regression formulation is

$$y = f(\sum_{i=1}^{n} \beta_i x_i + \beta_0) = \frac{1}{1 + e^{-\sum_{i=1}^{n} \beta_i x_i - \beta_0}}$$

In the case of regression with multi-dimensional output, the regression formulation depends on scikit-learn. This can change depending on user defined parameters within the framework. The two potential formulations are

$$y_j = \frac{f(\sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j})}{\sum_k f(\sum_{i=1}^{n} \beta_{i,k} x_i + \beta_{0,k}))} \quad \forall j$$

$$y_j = \frac{e^{\sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j}}}{\sum_k e^{\sum_{i=1}^{n} \beta_{i,k} x_i + \beta_{0,k}}} \quad \forall j$$

In the case of classification we avoid modelling the non-linearities. For y being single dimension we formulate logistic regression for classification with

$$\begin{aligned} y &\leq 1 \sum_{i=1}^{n} \beta_i x_i + \beta_0 \\ y &\geq \sum_{i=1}^{n} \beta_i x_i + \beta_0 \\ y &\in \{0, 1\} \end{aligned}$$

In the case of multi-class classification we formulate logistic regression using the argmax formulation

$$\begin{aligned} y'_j =& \sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j} \quad \forall j \\ y =& argmax(y') \end{aligned}$$

2.6.4 Neural Networks

The package currently models dense neural network with ReLU, Sigmoid, and Tanh activations. For all formulations we let i be the node index of the input layer and j be node index of the out layer.

For dense layers with a ReLU activation function, we introduce slack variables $s \in \mathbb{R}^n_{\geq 0}$, with the formulation of the layer given by:

$$\begin{aligned} y_j = \sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j} + s_j \quad \forall j \\ SOS1(y_j, s_j) \quad \forall j \end{aligned}$$

Note that this formulation is non-standard in the literature. The standard formulation uses big-M constraints, for which bounds are found through feasibility and optimality based bound tightening procedures. Empirically such formulations have been shown to be the current state-of-the-art. Such a formulation fails completely, however, when the big-M values becomes sufficiently large, and is less friendly numerically overall w.r.t. the difference between the true output of the predictor and that which SCIP returns. Therefore we have decided to use SOS1 constraints, which will likely be slower on well-scaled neural networks.

An option to use the traditional big-M formulation is provided if one embeds the neural network predictor with the argument *formulation=bigm*. Warning: When using this formulation, the user must provide appropriate input bounds. The formulation for the big-M model introduces binary variables $z \in \{0, 1\}^n$, and performs a propagation routine to obtain lower and upper bound of the output neurons denoted by $L, U \in \mathbb{R}^n$. The formulation of the layer is given by:

$$\begin{aligned} y_j &\geq 0 \\ y_j &\geq \sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j} \quad \forall j \\ y_j &\leq \sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j} - (1 - z_j) L_j \quad \forall j \\ y_j &\leq z_j U_j \quad \forall j \end{aligned}$$

For dense layers with a Sigmoid activation function the formulation is:

$$y_j = \frac{1}{1 + e^{-(\sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j})}} \quad \forall j$$

For dense layers with a Tanh activation function the formulation is:

$$y_j = \frac{1 - e^{-2(\sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j})}}{1 + e^{-2(\sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j})}} \quad \forall j$$

For dense layers with a Softmax activation function the formulation is:

$$y_j = \frac{e^{\sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j}}}{\sum_{j'} e^{\sum_{i=1}^{n} \beta_{i,j'} x_i + \beta_{0,j'}}} \quad \forall j$$

For dense layers with a Softplus activation function the formulation is:

$$y_j = \log(1 + e^{\sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j}}) \quad \forall j$$

As the maximum is preserved over all these activation functions, and other activation functions such as Softmax, the inserted predictor constraint for classification purposes does not explicitly model the final activation layer. In such a case the formulation used is:

$$\begin{aligned} y'_j =& argmax(\sum_{i=1}^{n} \beta_{i,j} x_i + \beta_{0,j}) \quad \forall j \\ y =& argmax(y') \end{aligned}$$

2.6.5 Decision Tree

In a decision tree, each leaf l is defined by a number of constraints on the input features of the tree that correspond to the branches taken in the path leading to l. We formulate decision trees by introducing one binary decision variable δ_l for each leaf of the tree.

In the decision tree exactly one leaf is chosen. This constraint is formulated as:

$$\sum_l \delta_l = 1$$

To ensure that the input vector maps to the correct leaf, however, we need to introduce additional notation and constraints. For a node v, we denote by i_v the feature used for splitting and by θ_v the value at which the split is made. At a leaf l of the tree, we have a set $\mathcal{L}_l$ of inequalities of the form $x_{i_v} \leq \theta_v$ corresponding to the left branches leading to l and a set $\mathcal{R}_l$ of inequalities of the form $x_{i_v} > \theta_v$ corresponding to the right branches.

For each leaf, the inequalities describing $\mathcal{L}_l$ and $\mathcal{R}_l$ are imposed using indicator constraints:

$$\begin{aligned} \delta_l = 1 \rightarrow x_{i_v} \leq \theta_v - \frac{\epsilon}{2}, \quad \forall x_{i_v} \leq \theta_v \in \mathcal{L}_l, \\ \delta_l = 1 \rightarrow x_{i_v} \geq \theta_v + \frac{\epsilon}{2}, \quad \forall x_{i_v} > \theta_v \in \mathcal{R}_l. \end{aligned}$$

In our implementation, ϵ can be specified by a keyword parameter *epsilon* in functions that add a decision tree constraint. By default the value for ϵ is 0. When ϵ is smaller than the default tolerance in SCIP (as it is by default), and you have a solution where $x_{i_v} \approx \theta_v$, then SCIP can select an arbitrary child node of that decision in the tree.

Here is a concrete example. Let an internal node of the decision tree be for the feature x_4 and value 5. Then the decisions are:

$$\begin{aligned} x_4 \leq 5 \\ x_4 \geq 5 \end{aligned}$$

When $x_4 \approx 5$, both these conditions are true for SCIP, and therefore both child nodes can be reached. The result is that for a value of $x_4 = 4.999999999$, SCIP could say that $x_4 \geq 5$, and then the output of SCIP can be drastically different to that which is returned by *decision_tree.predict()*. The purpose of ϵ is to break these ties, and enforce that only one

of the decision can ever be true. The downside is that it introduces a small area of model infeasibility. For instance, if $\epsilon = 0.001$, and the only solution to the above example is $x_4 = 5$, then that solution is no longer valid according to the formulation. Therefore, we warn users to be careful when setting ϵ to be non-zero.

When using decision trees for classification, we create constraints that ensure the correct class is selected depending on the leaf node. Let y_j be the output for class j, and L_j be the set of leaf nodes that predict class j. The constraint ensuring the class is selected according to the leaf node is:

$$y_j = \sum_{l \in L_j} l$$

2.6.6 Random Forests

The formulation of Random Forests is a linear combination (aggregation) of decision trees. Each decision tree is represented using the model above. The same difficulties with the choice of ϵ apply to this case.

In the case of classification, after the linear combination (aggregation) is performed, the output is piped through the argmax formulation.

2.6.7 Gradient Boosting Trees

The formulation of Gradient Boosting Trees is a linear combination (aggregation) of decision trees. Each decision tree is represented using the model above. The same difficulties with the choice of ϵ apply to this case.

In the case of classification, after the linear combination (aggregation) is performed, the output is piped through the argmax formulation.

2.6.8 Support Vector Machines

For support vector machines, currently only linear and polynomial kernels are supported. In addition, currently only binary classification is supported, as modelling the sklearn one-vs-all relation is non-trivial.

The formulation for a linear kernel is simply a linear regression model. That is:

$$y = \sum_{i=1}^{n} \beta_i x_i + \beta_0.$$

The formulation for a polynomial kernel requires introducing some additional notation. Let S be the set of support vectors, d be the degree of the polynomial kernel, $\Lambda \in \mathbb{R}^S$ be the dual coefficients, and $v \in \mathbb{R}^{n \times S}$ be the support vectors. Then the result of the regression function is:

$$y = \sum_{s=1}^{|S|} \Lambda_s \gamma^d (\sum_{i=1}^{n} x_i v_{i,s})^d$$

As $|S|$ is not known until after training, it is possible that the embedded models for polynomial kernels can be much larger than the feature sizes would suggest. Hence, the MIP model may become quite large and difficult to optimise.

In the case of classification, the output of the regression function is piped into the argmax formulation centred around 0.

2.6.9 Centroid Clustering

The formulation for centroid clustering is as follows, where d_k are the distance variables between the input vector and cluster k and $C \in \mathbb{R}^{n \times K}$ are the cluster centers:

$$d_k = \sum_{i=1}^{n} (x_i - C_{i,k})^2 \quad \forall k \in K$$
$$y = argmin(d)$$

The argmin function in practice is accomplished by using argmax on the negative variables.

The above formulation is non-linear. A linearised version that uses the L1 norm instead of the L2 norm is also provided, although it should be noted that points can be misclassified when using this formulation as it is an approximation. Let $pos_{i,k} \geq 0$ and $neg_{i,k} \geq 0$ be variables whose sum is the L1 distance in a dimension to a given centroid.

$$pos_{i,k} - neg_{i,k} = x_i - C_{i,k} \quad \forall i, k \in [n] \times K$$
$$SOS1(pos_{i,k}, neg_{i,k}) \quad \forall i, k, \in [n] \times K$$
$$d_k = \sum_{i=1}^{n} (pos_{i,k} + neg_{i,k}) \quad \forall k \in K$$
$$y = argmin(d)$$

2.7 API Reference manual

High level function and modelling objects

2.7.1 Generic Predictor Constraint

pyscipopt_ml.**add_predictor_constr**(*scip_model*, *predictor*, *input_vars*, *output_vars=None*, *unique_naming_prefix='p_'*, ***kwargs*)

Formulate predictor in PySCIPOpt model.

The formulation predicts the values of output_vars using input_vars according to predictor.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The pyscipopt model where the predictor should be inserted.
- **predictor** – The predictor to insert.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for predictor in scip_model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for predictor in scip_model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip_model to insert the predictor in it

Return type

AbstractPredictorConstr

Note

The parameters *input_vars* and *output_vars* can be either

- Lists of variables (List of lists etc. for higher dimensional input)
- np.ndarray of variables

For internal use in the package they are cast into a np.ndarray of variables

They should have dimensions that conform with the input/output of the predictor. We denote by *n_samples* the number of samples (or objects) that we want to predict with our predictor. We denote by *n_features* the dimension of the input of the predictor. We denote by *n_output* the dimension of the output.

The *input_vars* are therefore of shape *(n_samples, n_features)* and the *output_vars* of shape *(n_samples, n_outputs)*. In the case of *output_vars* not being passed, appropriate variables will be automatically created. In the case of *n_samples == 1* the first dimension can simply be removed from the input.

class pyscipopt_ml.modelling.base_predictor_constraint.**AbstractPredictorConstr**(*scip_model*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *skip_validate=False*, ***kwargs*)

Base class to store all information of embedded ML model by :py:func`pyscipopt_ml.add_predictor_constr`.

This class is the base class to store everything that is added to a SCIP model when a trained predictor is inserted into it. Depending on the type of the predictor, a class derived from it will be returned by *pyscipopt_ml.add_predictor_constr()*.

Warning

Users should usually never construct objects of this class or one of its derived classes. They are returned by the *pyscipopt_ml.add_predictor_constr()* and other functions.

abstract get_error(*eps*)

Returns error in SCIP's solution with respect to prediction from input.

Returns

error – pyscipopt_ml.modelling.base_predictor_constr.AbstractPredictorConstr.output Assuming that we have a solution for the input and output variables *x, y*. Returns the absolute value of the differences between *predictor.predict(x)* and *y*. Where predictor is the regression / classification model represented by this object.

Return type

ndarray of same shape as

Raises

NoSolution – If the SCIP model has no solution (either was not optimized or is infeasible).

property input

Returns the input variables of embedded predictor.

Returns

output

Return type
np.ndarray

`property input_values`

Returns the values for the input variables if a solution is known.

Returns
input_vals

Return type
np.ndarray

Raises
`NoSolution` – If SCIP has no solution (either was not optimized or is infeasible).

`property output`

Output variables of embedded predictor.

Returns
output

Return type
np.ndarray

`property output_values`

Returns the values for the output variables if a solution is known.

Returns
output_value

Return type
np.ndarray

Raises
`NoSolution` – If SCIP has no solution (either was not optimized or is infeasible).

`print_stats`(*file=None*)

Print statistics on model additions stored by this class.

This function prints detailed statistics on the variables and constraints that were added to the model.

Parameters
`file` (*None, optional*) – Text stream to which output should be redirected. By default, this is sys.stdout.

Specific functions and modeling for each supported ML object:

2.7.2 Scikit-learn regression models

Functions and modeling object for Scikit-learn models

Linear Regression Constraint

Module for inserting ordinary Scikit-Learn regression models into a PySCIPOpt *Model*.

The following linear models are tested and should work:

- sklearn.linear_model.LinearRegression
- sklearn.linear_model.Ridge
- sklearn.linear_model.Lasso
- sklearn.linear_model.ElasticNet

pyscipopt_ml.sklearn.**add_linear_regression_constr**(*scip_model*, *linear_regression*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate linear_regression as a constraint in a PySCIPOpt Model.

The formulation predicts the values of output_vars using input_vars according to linear_regression.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The scip model where the predictor should be inserted.
- **linear_regression** (sklearn.linear_model.LinearRegression)
- **types** (*The linear regression to insert. It can be any of the following*) –
  - sklearn.linear_model.LinearRegression
  - sklearn.linear_model.Ridge
  - sklearn.linear_model.Lasso
  - sklearn.linear_model.ElasticNet
- **input_vars** (*list or dict*) – Decision variables used as input for the regression model
- **output_vars** (*list or dict*) – Decision variables used as output for the regression model
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip to formulate linear_regression.

Return type

LinearRegressionConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.sklearn.linear_regression.**LinearRegressionConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Class to model trained `sklearn.linear_model.LinearRegression` with SCIP

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

Partial Least Square Regression Constraint

Module for formulating simple Scikit-Learn Partial Least Squares models in a PySCIPOpt model.

`pyscipopt_ml.sklearn.`**`add_pls_regression_constr`**(*scip_model*, *pls_regression*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate pls_regression in scip_model.

The formulation predicts the values of output_vars using input_vars according to pls_regression.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The PySCIPOpt model where the predictor will be inserted.
- **pls_regression** (`sklearn.cross_decomposition.PLSRegression` or) – `sklearn.cross_decomposition.PLSCanonical` The partial least squares model to insert.
- **input_vars** (*:list or dict*) – Decision variables used as input for partial least squares regression in model.
- **output_vars** (*list or dict*) – Decision variables used as output for partial least squares regression in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip_model to formulate pls_regression.

Return type

PLSRegressionConstr

> **Note**
>
> See *add_predictor_constr* for acceptable values for input_vars and output_vars

class `pyscipopt_ml.sklearn.pls.`**`PLSRegressionConstr`**(*scip_model*, *predictor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Class to model trained `sklearn.cross_decomposition.PLSRegression` or `sklearn.cross_decomposition.PLSCanonical` with SCIP

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

`add_regression_constr`()

Add the prediction constraints to SCIP.

Logistic Regression Constraint

Module for formulating a `sklearn.linear_model.LogisticRegression` in a PySCIPOpt Model.

`pyscipopt_ml.sklearn.`**`add_logistic_regression_constr`**(*scip_model*, *logistic_regression*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *output_type='classification'*, ***kwargs*)

Formulate logistic_regression in a SCIP Model.

The formulation predicts the values of output_vars using input_vars according to logistic_regression.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The PySCIPOpt model where the predictor will be inserted.
- **logistic_regression** (`sklearn.linear_model.LogisticRegression`) – The logistic regression model to insert.
- **input_vars** (*:list or dict*) – Decision variables used as input for logistic regression in model.
- **output_vars** (*list or dict*) – Decision variables used as output for logistic regression in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **output_type** (*{'classification', 'regression'}, default='classification'*) – If the option chosen is 'classification' the output is 1 for exactly one class and 0 for all others. If the option 'regression' is chosen the output is the probability of each class.

Returns

Object containing information about what was added to scip_model to formulate logistic_regression.

Return type

LogisticRegressionConstr

Raises

- **NoModel** – If the logistic regression is not a binary label regression
- **ParameterError** – If the value of output_type is set to a non-conforming value (see above).

Warning

When there is a (near) tie for the most likely class, users should be aware that SCIP tolerances will allow either of the two classes to be selected. In a scenario where there are two classes, and the logistic regression model assigns probability of class 1 as 0.5000000001. Naturally this is larger than 0.4999999999 assigned to class 2, however from a numerics point of view in SCIP both can be selected.

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.sklearn.logistic_regression.**LogisticRegressionConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *output_type='classification'*, ***kwargs*)

Class to model trained `sklearn.linear_model.LogisticRegression` with SCIP

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

Centroid Based Clustering Constraint

Module for formulating a `sklearn.cluster.KMeans` into a PySCIPOpt Model.

pyscipopt_ml.sklearn.**add_centroid_cluster_constr**(*scip_model*, *centroid_clusteror*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *formulation='l2'*, ***kwargs*)

Formulate centroid_clusteror in scip_model.

The formulation predicts the values of output_vars using input_vars according to centroid_clusteror.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the predictor should be inserted.
- **centroid_clusteror** (`sklearn.cluster.KMeans`) – The centroid clusteror to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for centroid clustering in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for centroid clustering in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **formulation** (*str, optional*) – The formulation type used when embedding the centroid clustering predictor. Valid types are "l2" (standard norm, same as the predictor), and "l1" for a linearised version. Warning: The linearised version will incorrectly label some points.

Returns

Object containing information about what was added to scip_model to formulate centroid_clusteror.

Return type

CentroidClusterConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.sklearn.centroid_cluster.**CentroidClusterConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars*, *unique_naming_prefix*, *formulation*, ***kwargs*)

Class to model trained sklearn.cluster.KMeans
with SCIP

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

Decision Tree Regressor Constraint

Module for formulating a sklearn.tree.DecisionTreeRegressor or a sklearn.tree.DecisionTreeClassifier in a PySCIPOpt Model.

pyscipopt_ml.sklearn.**add_decision_tree_regressor_constr**(*scip_model*, *decision_tree_regressor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *epsilon=0.0*, ***kwargs*)

Formulate decision_tree_regressor into a SCIP Model.

The formulation predicts the values of output_vars using input_vars according to decision_tree_regressor.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP Model where the predictor should be inserted.
- **decision_tree_regressor** (sklearn.tree.DecisionTreeRegressor) – The decision tree regressor to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for decision tree in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for decision tree in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **epsilon** (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns
Object containing information about what was added to scip_model to formulate decision_tree_regressor

Return type
DecisionTreeRegressorConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

pyscipopt_ml.sklearn.**add_decision_tree_classifier_constr**(*scip_model*, *decision_tree_classifier*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *epsilon=0.0*, ***kwargs*)

Formulate decision_tree_classifier into a SCIP Model.

The formulation predicts the values of output_vars using input_vars according to decision_tree_classifier.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP Model where the predictor should be inserted.
- **decision_tree_classifier** (sklearn.tree.DecisionTreeClassifier) – The decision tree classifier to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for decision tree in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for decision tree in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **epsilon** (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns

Object containing information about what was added to scip_model to formulate decision_tree_classifier

Return type

DecisionTreeClassifierConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

Warning

Although decision trees with multiple outputs are tested they were never used in a non-trivial optimization model. It should be used with care at this point.

class pyscipopt_ml.sklearn.decision_tree.**DecisionTreeConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *epsilon=0.0*, *classification=False*, *formulation='leafs'*, ***kwargs*)

Class to model trained sklearn.tree.DecisionTreeRegressor or trained sklearn.tree.DecisionTreeClassifier with pyscipopt.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

Gradient Boosting Regressor Constraint

Module for formulating a sklearn.ensemble.GradientBoostingRegressor or a sklearn.ensemble.GradientBoostingClassifier into a PySCIPOpt Model.

pyscipopt_ml.sklearn.**add_gradient_boosting_regressor_constr**(*scip_model*, *gradient_boosting_regressor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate gradient_boosting_regressor into scip_model.

The formulation predicts the values of output_vars using input_vars according to gradient_boosting_regressor.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the predictor should be inserted.
- **gradient_boosting_regressor** (sklearn.ensemble.GradientBoostingRegressor) – The gradient boosting regressor to insert as predictor.
- **input_vars** (*np.ndarray or list*) – Decision variables used as input for gradient boosting regressor in model.
- **output_vars** (*np.ndarray or list, optional*) – Decision variables used as output for gradient boosting regressor in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip_model to formulate gradient_boosting_regressor.

Return type

GradientBoostingConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

pyscipopt_ml.sklearn.**add_gradient_boosting_classifier_constr**(*scip_model*, *gradient_boosting_classifier*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate gradient_boosting_classifier into scip_model.

The formulation predicts the values of output_vars using input_vars according to gradient_boosting_classifier.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the predictor should be inserted.
- **gradient_boosting_classifier** (sklearn.ensemble.GradientBoostingClassifier) – The gradient boosting classifier to insert as predictor.

- **input_vars** (*np.ndarray or list*) – Decision variables used as input for gradient boosting classifier in model.
- **output_vars** (*np.ndarray or list, optional*) – Decision variables used as output for gradient boosting classifier in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip_model to formulate gradient_boosting_classifier.

Return type

GradientBoostingConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.sklearn.gradient_boosting.**GradientBoostingConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars*, *unique_naming_prefix*, *classification*, ***kwargs*)

Class to model trained `sklearn.ensemble.GradientBoostingRegressor` with SCIP.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*.

Random Forest Constraint

Module for formulating a `sklearn.ensemble.RandomForestRegressor` or `sklearn.ensemble.RandomForestClassifier` into a PySCIPOpt Model.

pyscipopt_ml.sklearn.**add_random_forest_regressor_constr**(*scip_model*, *random_forest_regressor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate random_forest_regressor in scip_model.

The formulation predicts the values of output_vars using input_vars according to random_forest_regressor.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the predictor should be inserted.
- **random_forest_regressor** (`sklearn.ensemble.RandomForestRegressor`) – The random forest regressor to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for random forest in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for random forest in model.

- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns
Object containing information about what was added to scip_model to formulate random_forest_regressor.

Return type
RandomForestConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

pyscipopt_ml.sklearn.**add_random_forest_classifier_constr**(*scip_model*, *random_forest_classifier*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate random_forest_classifier in scip_model.

The formulation predicts the values of output_vars using input_vars according to random_forest_classifier.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the predictor should be inserted.
- **random_forest_classifier** (sklearn.ensemble.RandomForestClassifier) – The random forest classifier to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for random forest in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for random forest in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns
Object containing information about what was added to scip_model to formulate random_forest_classifier.

Return type
RandomForestConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.sklearn.random_forest.**RandomForestConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars*, *unique_naming_prefix*, *classification*, ***kwargs*)

Class to model trained sklearn.ensemble.RandomForestRegressor or sklearn.ensemble.RandomForestClassifier with SCIP

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

Pipeline Constraint

Module for formulating a `sklearn.pipeline.Pipeline` into a PySCIPOpt Model. The pipeline's transformers (or preprocessing steps) can be any of the following: - `sklearn.preprocessing.StandardScaler` - `sklearn.preprocessing.PolynomialFeatures` - `sklearn.preprocessing.Normalizer` - `sklearn.preprocessing.Binarizer`

The final step of the pipeline must be a valid predictor.

pyscipopt_ml.sklearn.**add_pipeline_constr**(*scip_model*, *pipeline*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate pipeline into scip_model.

The formulation predicts the values of output_vars using input_vars according to pipeline.

Parameters

- **scip_model** (*SCIP Model*) – The PySCIPOpt Model where the predictor should be inserted.
- **pipeline** (`sklearn.pipeline.Pipeline`) – The pipeline to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for support vector in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for support vector in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip_model to embed the predictor into it

Return type

PipelineConstr

Raises

NoModel – If the translation to SCIP of one of the elements in the pipeline is not implemented or recognized.

Notes

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.sklearn.pipeline.**PipelineConstr**(*scip_model*, *pipeline*, *input_vars*, *output_vars*, *unique_naming_prefix*, ***kwargs*)

Class to formulate a trained `sklearn.pipeline.Pipeline` into a PySCIPOpt model.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*.

property input

Returns input variables of pipeline, i.e. input of its first step.

property input_values

Returns input values of pipeline in solution, i.e. input of its first step.

property output

Returns output variables of pipeline, i.e. output of its last step.

property output_values

Returns output values of pipeline in solution, i.e. output of its last step.

Support Vector Constraint

Module for formulating a sklearn.svm.SVR, sklearn.svm.SVC, sklearn.svm.LinearSVR, or sklearn.svm.LinearSVC into a PySCIPOpt Model.

pyscipopt_ml.sklearn.**add_support_vector_regressor_constr**(*scip_model*, *support_vector_regressor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate support_vector_regressor in scip_model.

The formulation predicts the values of output_vars using input_vars according to support_vector_regressor.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the predictor should be inserted.
- **support_vector_regressor** (sklearn.svm.SVR) – The support vector regressor to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for support vector in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for support vector in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip_model to formulate support_vector_regressor.

Return type

SupportVectorConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

pyscipopt_ml.sklearn.**add_support_vector_classifier_constr**(*scip_model*, *support_vector_classifier*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate support_vector_classifier in scip_model.

The formulation predicts the values of output_vars using input_vars according to support_vector_classifier.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the predictor should be inserted.
- **support_vector_classifier** (sklearn.svm.SVC) – The support vector classifier to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for support vector in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for support vector in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip_model to formulate support_vector_classifier.

Return type

SupportVectorConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.sklearn.support_vector.**SupportVectorConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars*, *unique_naming_prefix*, *classification*, ***kwargs*)

Class to model trained sklearn.svm.SVR, sklearn.svm.SVC, sklearn.svm.LinearSVR, or sklearn.svm.LinearSVC with SCIP

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

MLP Constraint

Module for formulating a sklearn.neural_network.MLPRegressor or sklearn.neural_network.MLPClassifier in a PySCIPOpt Model.

pyscipopt_ml.sklearn.**add_mlp_regressor_constr**(*scip_model*, *mlp_regressor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate mlp_regressor into scip_model.

The formulation predicts the values of output_vars using input_vars according to mlp_regressor.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the predictor should be inserted.
- **mlp_regressor** (sklearn.neural_network.MLPRegressor) – The multi-layer perceptron regressor to insert as predictor.

- **input_vars** (*np.ndarray or list*) – Decision variables used as input for regression in model.
- **output_vars** (*np.ndarray or list, optional*) – Decision variables used as output for regression in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns
Object containing information about what was added to scip_model to formulate mlp_regressor.

Return type
MLPRegressorConstr

Raises
NoModel – If the translation to SCIP of the activation function for the network is not implemented.

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

pyscipopt_ml.sklearn.**add_mlp_classifier_constr**(*scip_model*, *mlp_classifier*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate mlp_classifier into scip_model.

The formulation predicts the values of output_vars using input_vars according to mlp_classifier.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the predictor should be inserted.
- **mlp_classifier** (sklearn.neural_network.MLPClassifier) – The multi-layer perceptron classifier to insert as predictor.
- **input_vars** (*np.ndarray or list*) – Decision variables used as input for regression in model.
- **output_vars** (*np.ndarray or list, optional*) – Decision variables used as output for regression in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns
Object containing information about what was added to scip_model to formulate mlp_classifier.

Return type
MLPConstr

Raises
NoModel – If the translation to SCIP of the activation function for the network is not implemented.

> **Note**
>
> See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.sklearn.mlp.**MLPConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *classification=False*, ***kwargs*)

Class to model trained `sklearn.neural_network.MLPRegressor` or `sklearn.neural_network.MLPClassifier` with PySCIPOpt.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*.

MultiOutput Constraint

Module for formulating a `sklearn.multioutput.MultiOutputRegressor` or `sklearn.multioutput.MultiOutputClassifier` into a PySCIPOpt Model.

pyscipopt_ml.sklearn.**add_multi_output_regressor_constr**(*scip_model*, *multi_output_regressor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate a multi_output_regressor into scip_model.

The formulation predicts the values of output_vars using input_vars according to multi_output_regressor.

Parameters

- **scip_model** (*SCIP Model*) – The PySCIPOpt Model where the predictor should be inserted.
- **multi_output_regressor** (`sklearn.multioutput.MultiOutputRegressor`) – The multi_output_regressor to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for multi_output_regressor in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for multi_output_regressor in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip_model to embed the predictor into it

Return type

MultiOutputConstr

Raises

NoModel – If the translation to SCIP of one of the elements in the multi_output_regressor is not implemented or recognized.

Notes

See *add_predictor_constr* for acceptable values for input_vars and output_vars

pyscipopt_ml.sklearn.**add_multi_output_classifier_constr**(*scip_model*, *multi_output_classifier*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Formulate a multi_output_classifier into scip_model.

The formulation predicts the values of output_vars using input_vars according to multi_output_classifier.

Parameters

- **scip_model** (*SCIP Model*) – The PySCIPOpt Model where the predictor should be inserted.
- **multi_output_classifier** (sklearn.multioutput.MultiOutputClassifier) – The multi_output_classifier to insert as predictor.
- **input_vars** (*list or np.ndarray*) – Decision variables used as input for multi_output_classifier in model.
- **output_vars** (*list or np.ndarray, optional*) – Decision variables used as output for multi_output_classifier in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

Returns

Object containing information about what was added to scip_model to embed the predictor into it

Return type

MultiOutputConstr

Raises

NoModel – If the translation to SCIP of one of the elements in the multi_output_classifier is not implemented or recognized.

Notes

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.sklearn.multi_output.**MultiOutputConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars*, *unique_naming_prefix*, *classification*, ***kwargs*)

Class to formulate a trained sklearn.multioutput.MultiOutputRegressor or sklearn.multioutput.MultiOutputClassifier into a PySCIPOpt model.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*.

Utility

Scikit-Learn Helper

class pyscipopt_ml.sklearn.skgetter.**SKgetter**(*predictor*, ***kwargs*)

Utility class for sklearn models convertors.

Implement some common functionalities: check predictor is fitted, get error

predictor

Scikit-Learn predictor embedded into SCIP model.

get_error(*eps=None*)

Returns error in SCIP's solution with respect to the actual output of the trained predictor

Parameters

eps (*float or int or None, optional*) – The maximum allowed tolerance for a mismatch between the actual predictive model and SCIP. If the error is larger than eps an appropriate warning is printed

Returns

error – The absolute values of the difference between SCIP's solution and the trained ML model's output given the input as defined by SCIP. The matrix is the same dimension as the output of the trained predictor. Using sklearn / pyscipopt, the absolute difference between model.predict(input) and scip.getVal(output).

Return type

np.ndarray

Raises

NoSolution – If SCIP has no solution (either was not optimized or is infeasible).

class pyscipopt_ml.sklearn.skgetter.**SKtransformer**(*scip_model*, *transformer*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, ***kwargs*)

Utility class for sklearn preprocessing models convertors.

Implement some common functionalities.

transformer

Scikit-Learn transformer embedded into SCIP Model.

get_error(*eps=None*)

Returns error in SCIP's solution with respect to the actual output of the trained predictor

Parameters

eps (*float or int or None, optional*) – The maximum allowed tolerance for a mismatch between the actual predictive model and SCIP. If the error is larger than eps an appropriate warning is printed

Returns

error – The absolute values of the difference between SCIP's solution and the trained ML model's output given the input as defined by SCIP. The matrix is the same dimension as the output of the fitted transformer. Using sklearn / pyscipopt, the absolute difference between transformer.transform(input) and scip.getVal(output).

Return type

np.ndarray

Raises

NoSolution – If SCIP has no solution (either was not optimized or is infeasible).

2.7.3 Pytorch Sequential Network Constraint

pyscipopt_ml.torch.**add_sequential_constr**(*scip_model*, *sequential_model*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *output_type='regression'*, ***kwargs*)

Formulate sequential_model into scip_model.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the sequential model should be inserted.
- **sequential_model** (torch.nn.Sequential) – The sequential model to insert as predictor.
- **input_vars** (*np.ndarray*) – Decision variables used as input for the sequential neural network in PySCIPOpt Model.
- **output_vars** (*np.ndarray*) – Decision variables used as output for the sequential neural network in the PySCIPOpt Model
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **output_type** (*{"classification", "regression"}, default="regression"*) – If the option chosen is "classification" the output is 1 for exactly one class and 0 for all others. If the option chosen is "regression" then the output for each node of the final layer is the value from the sequential torch model.

Returns

Object containing information about what was added to model to insert the predictor into it

Return type

SequentialConstr

Raises

NoModel – If a section of the Pytorch model structure (layer or activation) is not implemented.

Warning

Only torch.nn.Linear, torch.nn.ReLU, torch.nn.Sigmoid, torch.nn.Tanh, torch.nn.Softmax, and torch.nn.Softplus layers are supported.

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.torch.sequential.**SequentialConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *output_type='regression'*, ***kwargs*)

Transform a pytorch Sequential Neural Network to SCIP constraints with input and output as matrices of variables.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

get_error(*eps=None*)

Returns error in SCIP's solution with respect to the actual output of the sequential neural network

Parameters

eps (*float or int or None, optional*) – The maximum allowed tolerance for a mismatch between the actual predictive model and SCIP. If the error is larger than eps an appropriate warning is printed

Returns

error – The absolute values of the difference between SCIP's solution and the trained ML model's output given the input as defined by SCIP. The matrix is the same dimension as the output of the trained predictor. Using torch / pyscipopt, the absolute difference between model.forward(input) and scip.getVal(output).

Return type

np.ndarray

Raises

NoSolution – If SCIP has no solution (either was not optimized or is infeasible).

2.7.4 Keras Model / Keras Sequential Network Constraint

pyscipopt_ml.keras.**add_keras_constr**(*scip_model*, *keras_model*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *output_type='regression'*, ***kwargs*)

Formulate keras_model into scip_model.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the sequential model should be inserted.
- **keras_model** (*keras.Model <https://keras.io/api/models/model/>*) – The keras model to insert as predictor.
- **input_vars** (*np.ndarray*) – Decision variables used as input for the sequential neural network in PySCIPOpt Model.
- **output_vars** (*np.ndarray*) – Decision variables used as output for the sequential neural network in the PySCIPOpt Model
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **output_type** (*{"classification", "regression"}, default="regression"*) – If the option chosen is "classification" the output is 1 for exactly one class and 0 for all others. If the option chosen is "regression" then the output for each node of the final layer is the value from the keras model.

Returns

Object containing information about what was added to scip_model to formulate keras_model into it

Return type

KerasNetworkConstr

Raises

NoModel – If the translation for some of the Keras model structure (layer or activation) is not implemented.

> **Warning**
>
> Only Dense (with *relu / tanh / sigmoid / softplus / softmax* activation) are supported.

Notes

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.keras.keras.**KerasNetworkConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars*, *unique_naming_prefix*, *output_type*, ***kwargs*)

Transform a keras dense Neural Network to SCIP constraints with input and output as matrices of variables.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

get_error(*eps=None*)

Returns error in SCIP's solution with respect to the actual output of the keras model

Parameters

eps (*float or int or None, optional*) – The maximum allowed tolerance for a mismatch between the actual predictive model and SCIP. If the error is larger than eps an appropriate warning is printed

Returns

error – The absolute values of the difference between SCIP's solution and the trained ML model's output given the input as defined by SCIP. The matrix is the same dimension as the output of the trained predictor. Using torch / pyscipopt, the absolute difference between model.forward(input) and scip.getVal(output).

Return type

np.ndarray

Raises

NoSolution – If SCIP has no solution (either was not optimized or is infeasible).

2.7.5 ONNX ModelProto Constraint

pyscipopt_ml.onnx.**add_onnx_constr**(*scip_model*, *onnx_model*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *output_type='regression'*, ***kwargs*)

Formulate an onnx ModelProto into scip_model.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The SCIP model where the ONNX ModelProto should be inserted.
- **onnx_model** (*onnx.ModelProto*) – The onnx model to insert as predictor.
- **input_vars** (*np.ndarray*) – Decision variables used as input for the feed forward neural network in PySCIPOpt Model.
- **output_vars** (*np.ndarray*) – Decision variables used as output for the feed forward neural network in the PySCIPOpt Model
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.

- **output_type** (*{"classification", "regression"}, default="regression"*) – If the option chosen is "classification" the output is 1 for exactly one class and 0 for all others. If the option chosen is "regression" then the output for each node of the final layer is the value from the ONNX model.

Returns
Object containing information about what was added to model to insert the predictor into it

Return type
ONNXConstr

Raises
NoModel – If a section of the ONNX model structure (layer or activation) is not implemented.

Warning

Only MatMul, Gemm, Relu, Sigmoid, Tanh, Softmax, and Softplus operator types are supported. Add, Cast, and Reshape are also often possible, although not in arbitrary contexts.

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.onnx.neural_net_onnx.**ONNXConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *output_type='regression'*, ***kwargs*)

Transform an ONNX ModelProto that represents a fully connected feed forward neural network to SCIP constraints with input and output as matrices of variables.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*..

get_error(*eps=None*)

Returns error in SCIP's solution with respect to the actual output of the ONNX neural network

Parameters
eps (*float or int or None, optional*) – The maximum allowed tolerance for a mismatch between the actual predictive model and SCIP. If the error is larger than eps an appropriate warning is printed

Returns
error – The absolute values of the difference between SCIP's solution and the trained ML model's output given the input as defined by SCIP. The matrix is the same dimension as the output of the trained predictor. Using ONNX / pyscipopt, the absolute difference between onnxruntime.InferenceSession(input) and scip.getVal(output).

Return type
np.ndarray

Raises
NoSolution – If SCIP has no solution (either was not optimized or is infeasible).

2.7.6 XGBoost Constraint

pyscipopt_ml.xgboost.**add_xgbregressor_constr**(*scip_model*, *xgboost_regressor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *epsilon=0.0*, ***kwargs*)

Formulate xgboost_regressor (gradient boosting decision tree) as constraints into a pyscipopt Model.

The formulation predicts the values of output_vars using input_vars according to xgboost_regressor.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The pyscipopt Model where the predictor should be inserted.
- **xgboost_regressor** (xgboost.XGBRegressor) – The gradient boosting regressor to insert as predictor.
- **input_vars** (*list or dict*) – Decision variables used as input for gradient boosting regressor in model.
- **output_vars** (*list or dict, optional*) – Decision variables used as output for gradient boosting regressor in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **epsilon** (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns

Object containing information about what was added to scip_model to formulate gradient_boosting_regressor.

Return type

XGBoostRegressorConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

Raises

NoModel – If the booster is not of type "gbtree".

pyscipopt_ml.xgboost.**add_xgbclassifier_constr**(*scip_model*, *xgboost_classifier*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *epsilon=0.0*, ***kwargs*)

Formulate xgboost_classifier (gradient boosting decision tree) as constraints into a pyscipopt Model.

The formulation predicts the values of output_vars using input_vars according to xgboost_classifier.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The pyscipopt Model where the predictor should be inserted.
- **xgboost_classifier** (xgboost.XGBClassifier) – The gradient boosting classifier to insert as predictor.

- **input_vars** (*list or dict*) – Decision variables used as input for gradient boosting regressor in model.
- **output_vars** (*list or dict, optional*) – Decision variables used as output for gradient boosting regressor in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **epsilon** (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns
Object containing information about what was added to scip_model to formulate gradient_boosting_classifier.

Return type
XGBoostClassifierConstr

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

Raises
NoModel – If the booster is not of type "gbtree".

pyscipopt_ml.xgboost.**add_xgbregressor_rf_constr**(*scip_model*, *xgboost_regressor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *epsilon=0.0*, ***kwargs*)

Formulate xgboost_regressor (random forest) as constraints into a pyscipopt Model.

The formulation predicts the values of output_vars using input_vars according to xgboost_regressor.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The pyscipopt Model where the predictor should be inserted.
- **xgboost_regressor** (xgboost.XGBRFRegressor) – The gradient boosting regressor to insert as predictor.
- **input_vars** (*list or dict*) – Decision variables used as input for gradient boosting regressor in model.
- **output_vars** (*list or dict, optional*) – Decision variables used as output for gradient boosting regressor in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **epsilon** (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns
Object containing information about what was added to scip_model to formulate gradient_boosting_regressor.

Return type
XGBoostRegressorConstr

> **Note**
>
> See *add_predictor_constr* for acceptable values for input_vars and output_vars

Raises
NoModel – If the booster is not of type "gbtree".

pyscipopt_ml.xgboost.**add_xgbclassifier_rf_constr**(*scip_model*, *xgboost_classifier*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *epsilon=0.0*, ***kwargs*)

Formulate xgboost_classifier (random forest) as constraints into a pyscipopt Model.

The formulation predicts the values of output_vars using input_vars according to xgboost_classifier.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The pyscipopt Model where the predictor should be inserted.
- **xgboost_classifier** (xgboost.XGBClassifier) – The gradient boosting classifier to insert as predictor.
- **input_vars** (*list or dict*) – Decision variables used as input for gradient boosting regressor in model.
- **output_vars** (*list or dict, optional*) – Decision variables used as output for gradient boosting regressor in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **epsilon** (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns
Object containing information about what was added to scip_model to formulate gradient_boosting_classifier.

Return type
XGBoostClassifierConstr

> **Note**
>
> See *add_predictor_constr* for acceptable values for input_vars and output_vars

Raises
NoModel – If the booster is not of type "gbtree".

class pyscipopt_ml.xgboost.xgboost_constr.**XGBoostConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars*, *unique_naming_prefix=''*, *epsilon=0.0*, *aggr='sum'*, *classification=False*, ***kwargs*)

Class to model trained `xgboost.XGBRegressor` or `xgboost.XGBClassifier` with constraints in a pyscipopt Model.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*.

class pyscipopt_ml.xgboost.xgbgetter.**XGBgetter**(*predictor*, *input_vars*, *output_type='regular'*, ***kwargs*)

Utility class for xgboost models convertors.

Implement some common functionalities: check predictor is fitted, output dimension, get error

predictor

Xgboost predictor embedded into SCIP model.

extract_raw_data_and_create_tree_vars(*epsilon=0.0*)

Function for extracting information from xgb.Booster and creating additional modelling variables.

Parameters

epsilon (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns

- **trees** (*list*) – A list of tree dictionaries similar in structure to that provided by SKlearn
- **constant** (*float*) – A constant value that is added to all regression output of the decision trees
- **tree_vars** (*np.ndarray*) – A numpy array filled with variables that represent output of trees from the trained XGBoost model

get_error(*eps=None*)

Returns error in SCIP's solution with respect to the actual output of the trained predictor

Parameters

eps (*float or int or None, optional*) – The maximum allowed tolerance for a mismatch between the actual predictive model and SCIP. If the error is larger than eps an appropriate warning is printed

Returns

error – The absolute values of the difference between SCIP's solution and the trained ML model's output given the input as defined by SCIP. The matrix is the same dimension as the output of the trained predictor. Using sklearn / pyscipopt, the absolute difference between model.predict(input) and scip.getVal(output).

Return type

np.ndarray

Raises

NoSolution – If SCIP has no solution (either was not optimized or is infeasible).

2.7.7 LightGBM Constraint

pyscipopt_ml.lightgbm.**add_lgbregressor_constr**(*scip_model*, *lightgbm_regressor*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *epsilon=0.0*, ***kwargs*)

Formulate lightgbm_regressor as constraints into a pyscipopt Model. Accommodates both Gradient Boosting Decision Trees and Random Forests as boosting types.

The formulation predicts the values of output_vars using input_vars according to lightgbm_regressor.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The pyscipopt Model where the predictor should be inserted.
- **lightgbm_regressor** (lightgbm.LGBMRegressor) – The gradient boosting regressor to insert as predictor.
- **input_vars** (*list or dict*) – Decision variables used as input for gradient boosting regressor in model.
- **output_vars** (*list or dict, optional*) – Decision variables used as output for gradient boosting regressor in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **epsilon** (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns

Object containing information about what was added to scip_model to formulate gradient_boosting_regressor.

Return type

LightGBMRegressorConstr

Raises

NoModel – If the boosting type is not "gbdt" or "rf".

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

pyscipopt_ml.lightgbm.**add_lgbclassifier_constr**(*scip_model*, *lightgbm_classifier*, *input_vars*, *output_vars=None*, *unique_naming_prefix=''*, *epsilon=0.0*, ***kwargs*)

Formulate lightgbm_classifier as constraints into a pyscipopt Model. Accommodates both Gradient Boosting Decision Trees and Random Forests as boosting types.

The formulation predicts the values of output_vars using input_vars according to lightgbm_classifier.

Parameters

- **scip_model** (*PySCIPOpt Model*) – The pyscipopt Model where the predictor should be inserted.
- **lightgbm_classifier** (lightgbm.LGBMClassifier) – The gradient boosting classifier to insert as predictor.

- **input_vars** (*list or dict*) – Decision variables used as input for gradient boosting classifier in model.
- **output_vars** (*list or dict, optional*) – Decision variables used as output for gradient boosting classifier in model.
- **unique_naming_prefix** (*str, optional*) – A unique naming prefix that is used before all variable and constraint names. This parameter is important if the SCIP model is later printed to file and many predictors are added to the same SCIP model.
- **epsilon** (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns
Object containing information about what was added to scip_model to formulate gradient_boosting_classifier.

Return type
LightGBMClassifierConstr

Raises
NoModel – If the boosting type is not "gbdt" or "rf".

Note

See *add_predictor_constr* for acceptable values for input_vars and output_vars

class pyscipopt_ml.lightgbm.lightgbm_constr.**LightGBMConstr**(*scip_model*, *predictor*, *input_vars*, *output_vars*, *unique_naming_prefix=''*, *epsilon=0.0*, *classification=False*, ***kwargs*)

Class to model trained lightgbm.LGBMRegressor or
lightgbm.LGBMClassifier with constraints in a pyscipopt Model.

Stores the changes to the SCIP Model for representing an instance into it. Inherits from *AbstractPredictorConstr*.

class pyscipopt_ml.lightgbm.lgbgetter.**LGBgetter**(*predictor*, *input_vars*, *output_type='regular'*, ***kwargs*)

Utility class for lightgbm models convertors.

Implement some common functionalities: check predictor is fitted, output dimension, get error

predictor

Lightgbm predictor embedded into SCIP model.

extract_raw_data_and_create_tree_vars(*epsilon=0.0*)

Function for extracting information from lgb._Booster and creating additional modelling variables.

Parameters
epsilon (*float, optional*) – Small value used to impose strict inequalities for splitting nodes in MIP formulations.

Returns

- **trees** (*list*) – A list of tree dictionaries similar in structure to that provided by SKlearn
- **tree_vars** (*np.ndarray*) – A numpy array filled with variables that represent output of trees from the trained LightGBM model

`get_error`(*eps=None*)

Returns error in SCIP's solution with respect to the actual output of the trained predictor

Parameters

eps (*float or int or None, optional*) – The maximum allowed tolerance for a mismatch between the actual predictive model and SCIP. If the error is larger than eps an appropriate warning is printed

Returns

error – The absolute values of the difference between SCIP's solution and the trained ML model's output given the input as defined by SCIP. The matrix is the same dimension as the output of the trained predictor. Using sklearn / pyscipopt, the absolute difference between model.predict(input) and scip.getVal(output).

Return type

np.ndarray

Raises

NoSolution – If SCIP has no solution (either was not optimized or is infeasible).

2.8 Similar Software

This code base was heavily inspired by Gurobi-MachineLearning. The API and general architecture was made to match, so that users could easily transfer between one to the other. If there is a feature missing here, or you are looking for alternatives, then give it a try!

Another similar piece of software is OMLT (Ceccon *et al.* [CJH+22]). As opposed to PySCIPOPT-ML, OMLT is much more general. It uses general modelling frameworks for both the MIP side and ML side as opposed to using a specific MIP solver (SCIP) and direct interfaces to ML frameworks. If interested particularly in non-standard neural network embeddings, please check it out.

2.9 Bibliography

BIBLIOGRAPHY

[CJH+22] Francesco Ceccon, Jordan Jalving, Joshua Haddad, Alexander Thebelt, Calvin Tsay, Carl D Laird, and Ruth Misener. Omlt: optimization & machine learning toolkit. *The Journal of Machine Learning Research*, 23(1):15829–15836, 2022.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: a scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 785–794. 2016.

[CCA+09] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision support systems*, 47(4):547–553, 2009.

[KMF+17] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: a highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 2017.

[Opt23] Gurobi Optimization. Gurobi-machinelearning. 2023. URL: https://github.com/Gurobi/gurobi-machinelearning/.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, and others. Pytorch: an imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 2019.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

PYTHON MODULE INDEX

p

INDEX

R

S

T

X