Lab Semantic Data Web Technologies

Topic: Distillation of DNN Networks into Gradient Boost tree model

Md. Emtazul Haque, email: s6mdhaqu@uni-bonn.de
Jelena Trajkovic, email: s6jetraj@uni-bonn.de
Anam Siddiqi, email: s6ansidd@uni-bonn.de

Supervisor: Firas Kassawat

Documentation File

WS 2020/21

Abstract:

The main idea of the project that we have worked on is based on the **Interpretable Deep Models for ICU Outcome Prediction (Zhengping Che,Sanjay Purushotham, Robinder Khemani and Yan Liu)**

EHR(Electronic Health Care) data is under exponential growth. New opportunities and needs for discovering meaningful data-driven representations and patterns of diseases.

Deep learning models have a superior performance when applied to healthcare prediction tasks, but clinicians have difficulties in understanding and applying these models. Deep learning models suffer from lack of interpretability.

Decision tree methods are widely employed in the healthcare domain, and they are easily interpretable, but they can easily overfit and perform poorly on large EHR datasets.

The question is - how to learn interpretable models from well trained deep network models?

In the reference paper, the authors propose a knowledge distillation approach called **interpretable mimic learning**. Main idea of knowledge distillation is to train a large, slow and accurate model and transfer its knowledge to a smaller, faster, but yet still accurate model, via utilizing the soft labels learned from the teacher model as the target labels for training the student model. Two pipelines are proposed. Our task in this project was to implement both of them and discuss the obtained results.

Pipeline 1:

1) In the first step, we train feed forward DNN, given the input X and the original target y. . Then, for each input sample X, we obtain the soft prediction score ynn ∈ [0, 1] from the prediction layer of the neural network.
2) In the second step, we train a mimic Gradient boosting model, given the raw input X and the soft label ynn as the model input and target, respectively.

Pipeline 2:

In the second pipeline, instead of taking soft labels from DNN, we are taking the learned features, extracted from the highest hidden layer. We then feed learned DNN features into the Helper Classifier, to predict the original label. We are taking soft predictions from the classifier. We train the GBT model with original input and obtained soft labels.

In both pipelines we train GBT models with soft labels. However, the accuracy results on the same test dataset are slightly different.

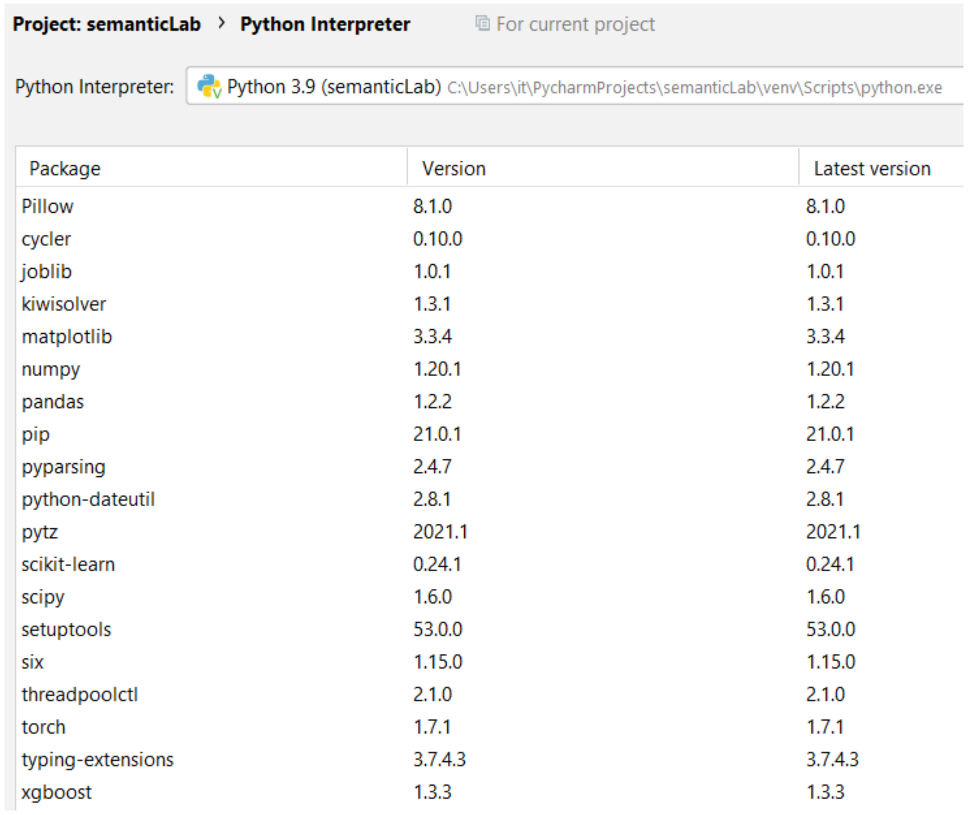
Steps to run the project:

1. Extract all files provided in the .zip folder and open it in an IDE like PyCharm.
2. Import the following packages into your project using pip:

-Numpy,
-Torch,
-Scikit learn,
-Xgboost,
-Matplotlib,
-Pandas
-Graphviz

3. Download the 64 bit exe file from the location https://gitlab.com/graphviz/graphviz/-/package_files/6164164/download and install it as per the instructions given here : https://forum.graphviz.org/t/new-simplified-installation-procedure-on-windows/224
   Also, add Graphviz to the system path in Environment variables. After this installation is complete,restart your PyCharm IDE for the changes to reflect.

   The project should have the following packages included(Settings->Python Interpreter):



Project: semanticLab › **Python Interpreter** For current project

Python Interpreter: Python 3.9 (semanticLab) C:\Users\it\PycharmProjects\semanticLab\venv\Scripts\python.exe

Package	Version	Latest version
Pillow	8.1.0	8.1.0
cycler	0.10.0	0.10.0
joblib	1.0.1	1.0.1
kiwisolver	1.3.1	1.3.1
matplotlib	3.3.4	3.3.4
numpy	1.20.1	1.20.1
pandas	1.2.2	1.2.2
pip	21.0.1	21.0.1
pyparsing	2.4.7	2.4.7
python-dateutil	2.8.1	2.8.1
pytz	2021.1	2021.1
scikit-learn	0.24.1	0.24.1
scipy	1.6.0	1.6.0
setuptools	53.0.0	53.0.0
six	1.15.0	1.15.0
threadpoolctl	2.1.0	2.1.0
torch	1.7.1	1.7.1
typing-extensions	3.7.4.3	3.7.4.3
xgboost	1.3.3	1.3.3

4. Run the **Pipelines.py** file in the project to see the following output:
   - ❏ Training 717 (Total number of training data used)
   - ❏ Test 308 (Total number of training data used)

- ❑ NN Accuracy: xx.xxx (Accuracy of the Neural Network where x={0,..9})
- ❑ GBT(only soft labels) Accuracy: xx.xxx (Accuracy of the Gradient Boosting Tree with only soft labels where x={0,..9})
- ❑ GBT(with helper classifier) Accuracy: xx.xxx (Accuracy of the Gradient Boosting Tree with Logistic Regression Classifier where x={0,..9})
- ❑ GBT(hard labels) Accuracy: xx.xxx (Accuracy of the Gradient Boosting Tree trained with Hard labels where x={0,..9})

Dataset:

We have used a dataset that is provided in the project folder - heart.csv. This data set dates from 1988 and consists of four databases: Cleveland, Hungary, Switzerland, and Long Beach V. It contains 76 attributes, but all published experiments refer to using a subset of 14 of them. The "target" field refers to the presence of heart disease in the patient. It is integer valued 0 = no disease and 1 = disease. The dataset contains following attributes:

age: age in years

sex: sex(1 = male, 0 = female)

cp: chest pain type (4 values)

trestbps: resting blood pressure

chol: serum cholesterol in mg/d**l**

fbs: fasting blood sugar > 120 mg/dl (1 = true; 0 = false)

restecg: resting electrocardiographic results (values 0,1,2)

thalach: maximum heart rate achieved

exang: exercise induced angina (1 = yes; 0 = no)

oldpeak: ST depression induced by exercise relative to rest

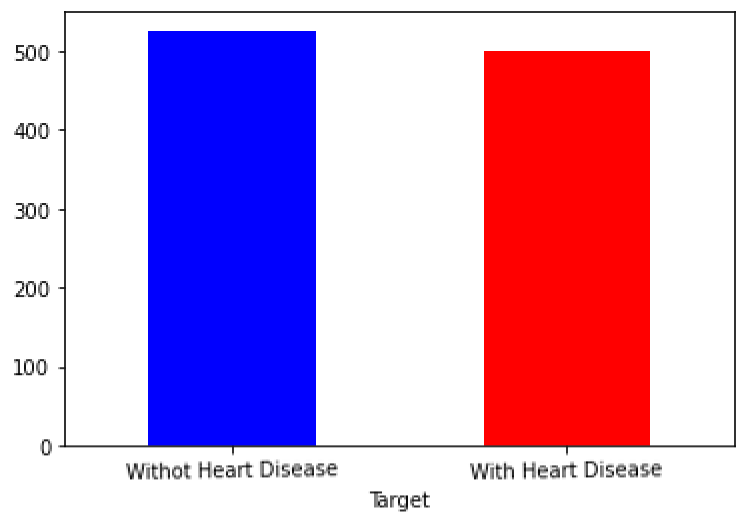
slope: the slope of the peak exercise ST segment

ca: number of major vessels (0-3) colored by flourosopy

thal: 0 = normal; 1 = fixed defect; 2 = reversable defect

Target: presence of heart disease (1 = yes; 0 = no)

According to the fact that the number of subjects with heart disease (526) is approximately the same as the number of subjects without heart disease (499), this dataset is well balanced.



File description:

GetTrainAndTestData.py

Classes	**Functions**
class **CSVDataset**(torch.utils.data.dataset.Dataset)	**prepare_data**(path)

Classes

1. class **CSVDataset**(torch.utils.data.dataset.Dataset)

   Methods defined here:

   __getitem__(self, idx)

   This function returns the sample at the index idx

   __init__(self, path)

   Pytorch provides class Dataset, which we are extending here and customizing, with respect to our dataset. We are loading the csv file as dataframe, we are storing the inputs to X(attributes according to which we are making predictions), and y(original targets). We are scaling our input variables using StandardScaler(). Original targets are not scaled, as the target value is 0 or 1.

Parameters

----

path : path to the heart.csv file

Returns

None.

__len__(self)

This function returns the number of rows in the dataset (number of samples we are working with).

get__inputs(self, idx)

This function returns the sample, without original target, at the index idx.

get_splits(self, n_test=0.3)

This function determines sizes for training and testing data, according to the n_test. Usually, 30%(0.3) of data is used for testing, 70% is used for training.

Parameters

----

n_test : percentage of data used for testing(real number between 0 and 1)

Returns

----

indexes for train and test rows

Functions

1. **prepare_data**(path)

   Pytorch provides DataLoader class. The aim of this class is to load Dataset instance during the model training and evaluation. Indexes for rows of data, which will be used for training and testing, returned by get_splits function, are passed to DataLoader, along with batch_size (we have opted to propagate one by one sample through the network) and shuffle parameter, which tells us whether the data should be shuffled every epoch. As that is better learning strategy, we have opted to shuffle the training dataset every epoch.

   Parameters

   ----

   path : path to the heart.csv file

   Returns

   ----

   train_dl : training dataset

   test_dl : test dataset

NN.py

Classes	**Functions**
class **MLP**(torch.nn.modules.module.Module)	**evaluate_model**(test_dl, model)
	get_last_layer(data, model)
	get_soft_labels(data, model)
	train_model(train_dl, test_dl, model)

Classes

1. class **MLP**(torch.nn.modules.module.Module)

   Methods defined here:

   __init__(self, n_inputs)

   The constructor of MLP class defines the layers. We define MLP
   with input layer(n_inputs = 13 neurons), first hidden layer(26 neurons),
   second hidden layer(13 neurons) and output layer. That gives us 689
   weights to be adjusted. We are using Kaiming for the weight initialisation
   strategy for hidden1 - > hidden2, hidden2 - > output layer, according to the
   fact that we are using ReLU() activation function for the both hidden layers.
   For the output layer, we are using Sigmoid() activation function, suitable
   for our binary classification task. We are using Xavier initialization for
   the weights from hidden2 -> output, because it can solve Sigmoid() vanishing
   gradient problem.

   forward(self, X)

   This function takes the input data (rows of heart.csv, without original
   target labels). The input data is fed in the forward direction through
   the network. Each hidden layer accepts the input data, processes it, as per
   the activation function and passes to the successive layer.

   Parameters

   X : input values from heart.csv dataset (without the last "target column")

Returns

X : calculated output(target) value for the given input

forwardLastHiddenLayer(self, X)

This function takes the input data (rows of heart.csv, without original target labels). The input data is fed in the forward direction through the network. Each hidden layer accepts the input data, processes it, as per the activation function and passes to the successive layer. This function returns the output of the second hidden layer. The goal is to extract the learned features from the last hidden layer (in our case - second hidden layer)

Parameters

X : input values from heart.csv dataset (without the last "target column")

Returns

X : learned features from the second hidden layer

Functions

1. **evaluate_model**(test_dl, model)

   After we have trained our model, we are calculating the accuracy of trained model on the test dataset - percentage of samples that are classified correctly.

Parameters

----

test_dl : test dataset

model : object of MLP class

Returns

----

acc : model accuracy on test dataset

2. **get_last_layer**(data, model)

   For implementing the second pipeline, we are using logistic regression as helper classifier. We are extracting activations from the last hidden layer and feed them into the helper classifier to predict the original task. This function returns the activations from the last hidden layer. We keep track on the inputs for which we extract activations, along with the target(original) labels.

   Parameters

   ----

   data : training dataset

   model : object of MLP class

   Returns

   ----

   xinputs : activations from the last hidden layer

   oinputs : inputs in the order in which we calculate activations true : original (true) target values

3. **get_soft_labels**(data, model)

   For implementing the first pipeline, we are using predicted soft labels for GBT training. This function returns predicted soft labels, along with inputs, we keep track on the inputs for which we make predictions, along with the target(original) labels.

   Parameters

   ----

   data : training dataset

   model : object of MLP class

   Returns

   ----

   xinputs : inputs in the order in which we calculate predictions

   predictions : soft labels, without rounding

   true : original (true) target values

4. **train_model**(train_dl, test_dl, model)

   For training of defined MLP model, we have to define loss function and optimization algorithm that will be used. Binary cross entropy loss is used as loss function. Stochastic gradient descent is used for optimization. SGD class provides standard algorithm. In the outer loop, we are defining the number of training epochs. In each epoch, the inner loop is required for enumerating the mini batches for SGD. Each update of the model consists of the following steps: clear the gradients, feed the inputs to the network, calculate loss, backpropagate the error through the network, update model weights.Additionaly, this

function plots training and validation learning curves.

Parameters

----

train_dl : training dataset

test_dl : test dataset

model : object of MLP class

Returns

None.

GBT.py

Classes	**Functions**
	showTree(model, blockSize, title)
	testGbt(model, X, y)
	trainXGbtClassification(X, y)

Functions

1. **showTree**(model, blockSize, title)
   Show GBT trees. Last tree from each block.
   To save computational power all the trees generated are not shown.
   If it is required to show all trees then provide blockSize=1

   Parameters
   ----
   model : trained GBT model

blockSize : size of block from which last tree will be shown
title : Title of the figure

Returns

None.

2. **testGbt**(model, X, y)
   After we have trained our GBT model, we are calculating the accuracy of
   trained model on the test dataset - percentage of samples that are
   classified correctly.

   Parameters
   ----
   model : trained GBT model
   X : Test inputs
   y : Desired(actual) output for the given test inputs

   Returns
   ----
   acc : model accuracy on test dataset

3. **trainXGbtClassification**(X, y)
   objective function = "multi:softprob" (Used for multiclass classification)
   Maximum number of trees = 100
   Learning rate = 0.1
   Maximum tree depth = 3

   For first pipeline:
   We pass the training inputs and generated soft labels from the NN to
   this function to train the Gradient Boosting Tree(GBT).
   For second pipeline:
   We pass the training inputs and generated soft labels from the Helper classifier to
   this function to train the Gradient Boosting Tree(GBT).

   Parameters

X : Training inputs
y : Soft labels

Returns
----
model : Trained GBT model

Pipelines.py

Classes	Functions
	firstPipeline(train_dl, model, xTest, yTest)
	gbtWithHardLabels(xTrain, yTrain, xTest, yTest)
	secondPipeline(train_dl, model, xTest, yTest)
	trainAndSaveNN(train_dl, test_dl, model)

Functions

1. **firstPipeline**(train_dl, model, xTest, yTest)
   Implements the first pipeline. Steps are
   1. Get soft labels from trained NN model
   2. Train GBT with the soft labels

   Finally it calculates the accuracy of trained GBT model
   with respect to test data and plot decision trees.

   Parameters
   ----
   train_dl : Training data
   model : Trained Neural Network model
   xTest : Test inputs
   yTest : Desired(actual) outputs for test inputs

Returns

None.

2. **gbtWithHardLabels**(xTrain, yTrain, xTest, yTest)
   Train GBT with Hard labels and calculate it's accuracy with test data and plot decision trees.

   Parameters
   ----
   xTrain : Training inputs
   yTrain : Training outputs (Teacher value)
   xTest : Test inputs
   yTest : Desired(actual) outputs for test inputs

   Returns

   None.

3. **secondPipeline**(train_dl, model, xTest, yTest)
   Implements the second pipeline. Steps are
   1. Get learned features from trained NN model
   2. Feed the learned features to the Helper classifier
   3. Train GBT with the soft labels obtained from helper classifier

   Finally it calculates the accuracy of trained GBT model
   with respect to test data and plot decision trees.

   Parameters
   ----
   train_dl : Training data
   model : Trained Neural Network model
   xTest : Test inputs
   yTest : Desired(actual) outputs for test inputs

Returns

None.

4. **trainAndSaveNN**(train_dl, test_dl, model)
   Train the Neural Network and save the trained model

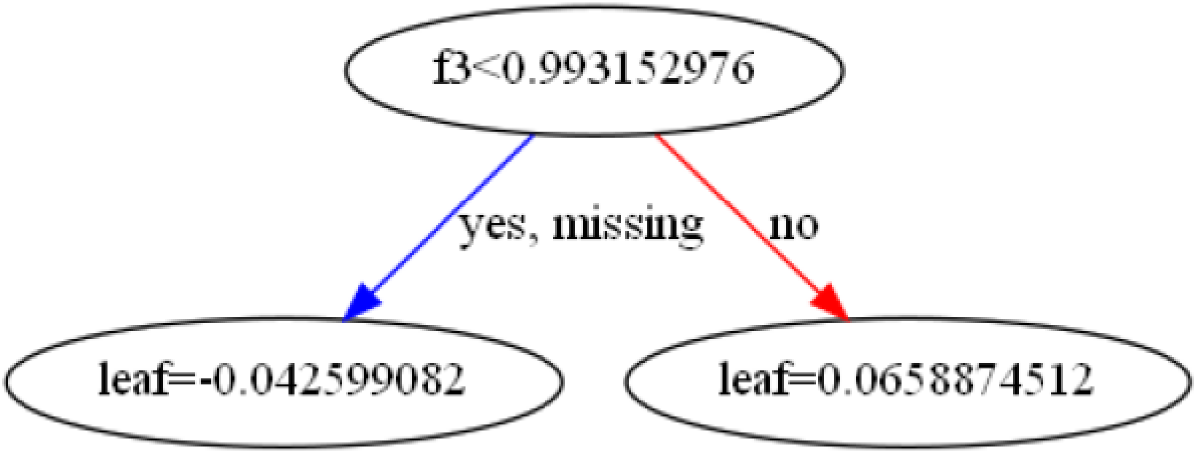
   Parameters
   ----
   train_dl : Training data
   test_dl : Test data
   model : Neural Network model

   Returns

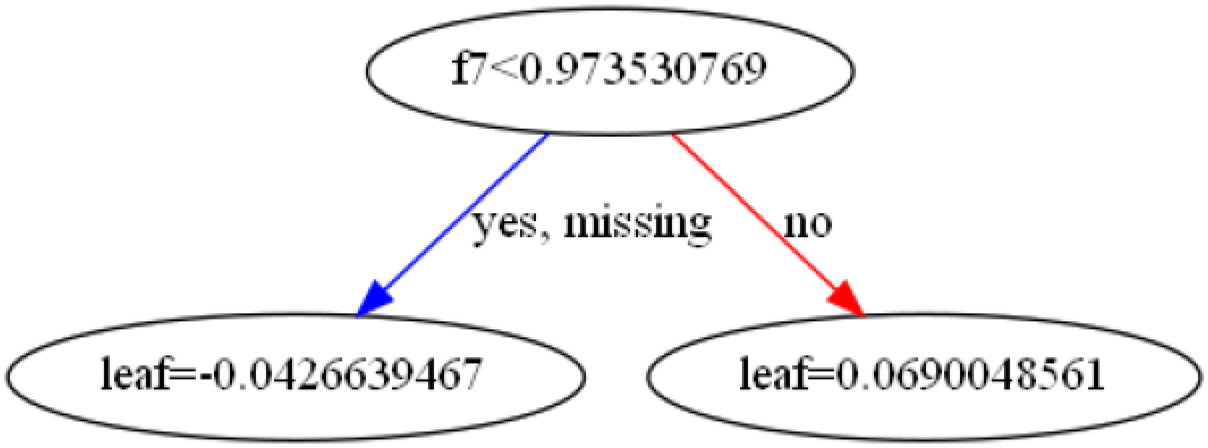
   None.

Some plotted decision trees:

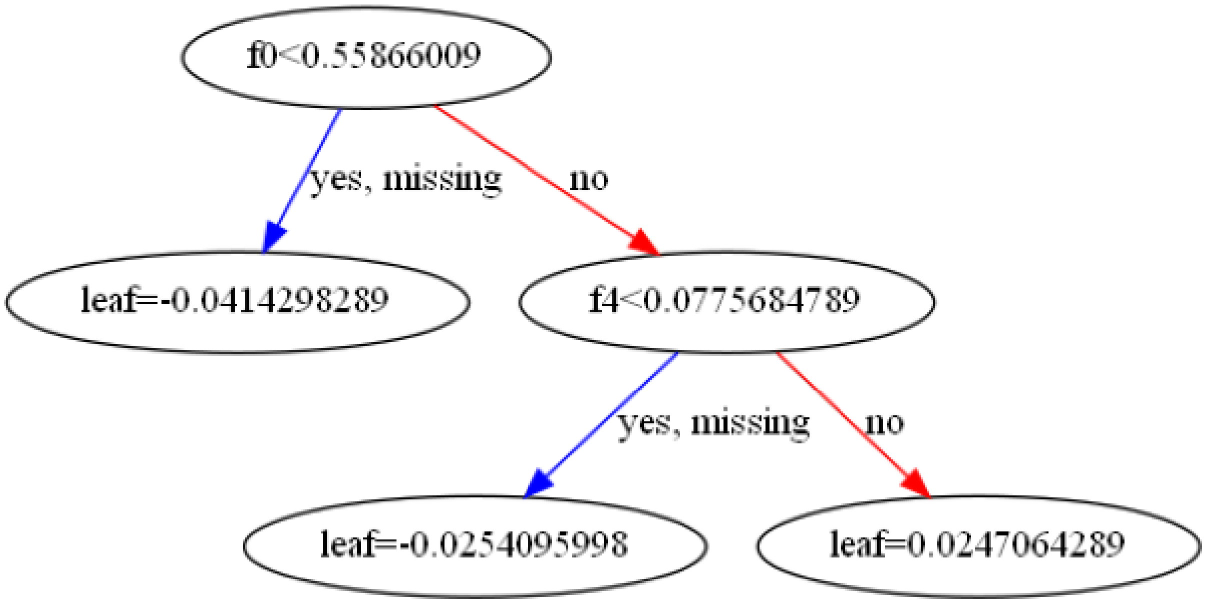
Pipeline 1, Tree number: 15



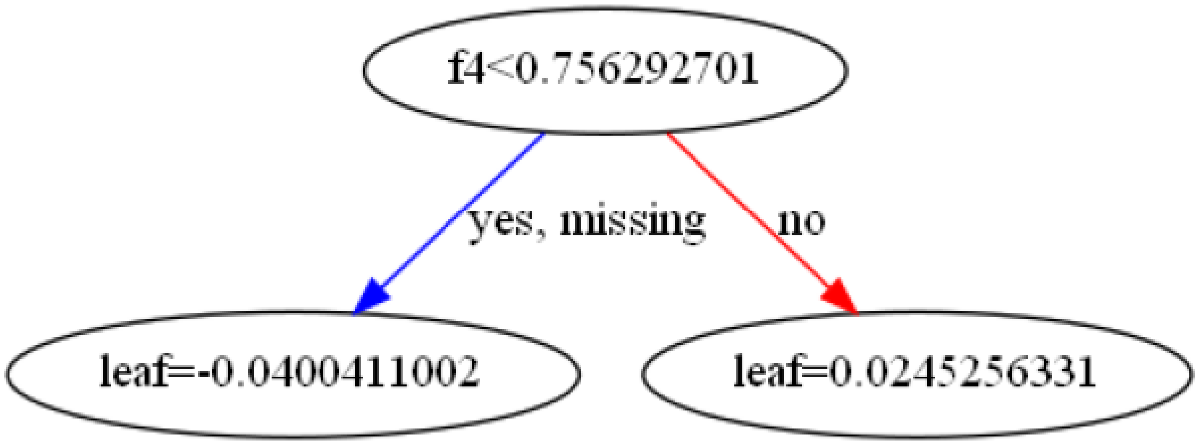
Pipeline 1, Tree number: 25



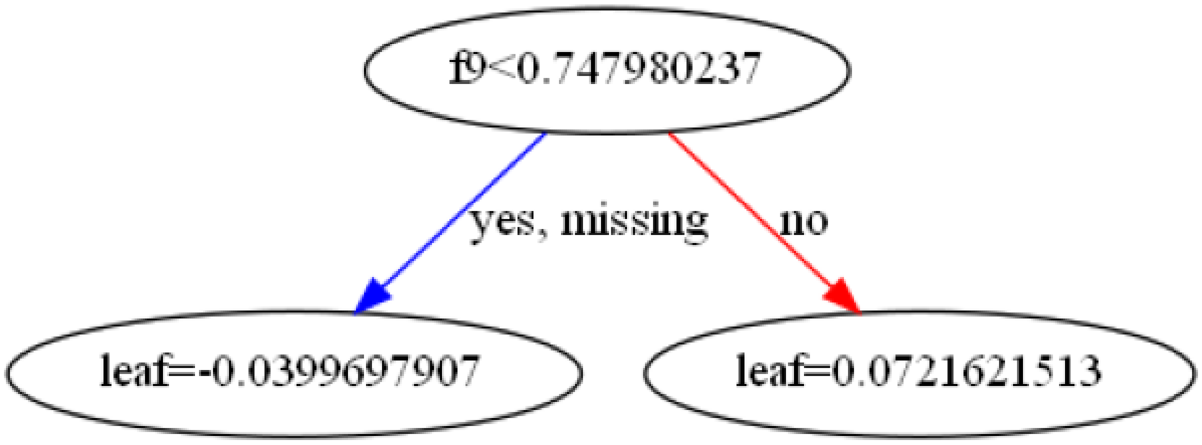
Pipeline 1, Tree number: 60



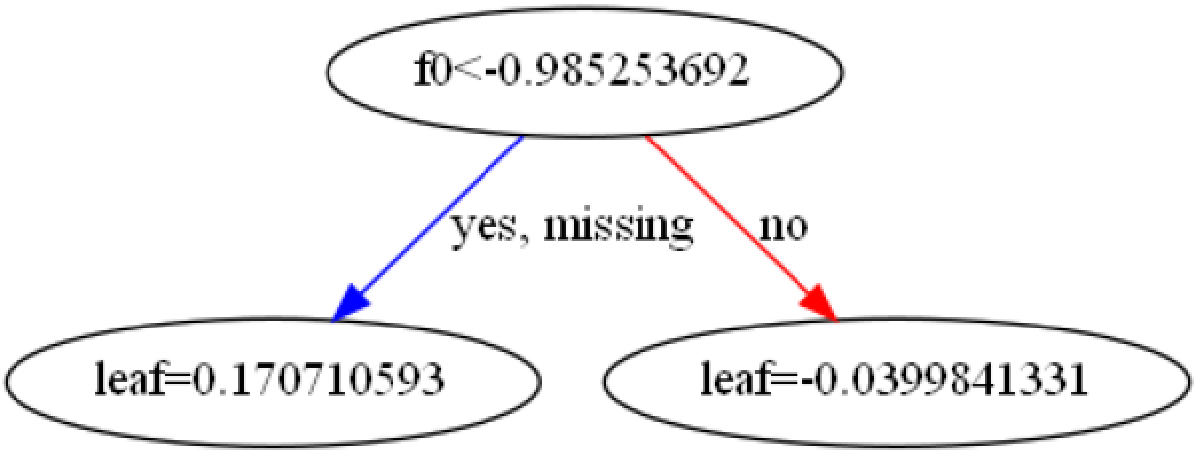
Pipeline 2, Tree number: 5



Pipeline 2, Tree number: 15



Pipeline 2, Tree number: 30


f0<-0.985253692
yes, missing
no
leaf=0.170710593
leaf=-0.0399841331