DTI Guide: Machine Learning on C3 AI Suite

The C3 AI Suite has Types specifically designed to facilitate certain machine learning pipelines. Taking their inspiration from the Scikit learn machine learning pipeline. At the most general, we have the `MLPipe` Type defining the most basic behavior of the pipeline with the `train`, `process`, and `score` methods, and various specializations which define a whole pipeline of operations. With this structure, you can call 'train' on the top level pipeline Type and the whole pipeline is trained. Same with 'process' and 'score' to both process inputs and score results. We'll start by discussing the basic Types which are available, beginning with the abstract Types forming the basis of this Machine Learning system, then some concrete implementations of those Types and finally some examples. We'll then discuss how a new machine learning model may be ported onto the C3 AI platform.

Abstract Types

MLPipe: An abstract Type which defines general behaviors like train, process, and score. This Type is mixed in by nearly all Types in the C3 AI Machine Learning space.

MLPipeline: An abstract Type which mixes MLPipe and connects multiple steps into a full pipeline. The provided concrete implementation of this Type is `MLSerialPipeline`. The MLPipeline contains a list of MLSteps to perform.

MLStep: A helper Type to allow MLSerialPipeline and other implementations of MLPipeline store their step pipelines and metadata.

MLLeafPipe: This is an abstract Type mixing MLPipe which is meant for steps in a machine learning pipeline which have no child steps, i.e. this step performs a specific action of the pipeline. In the terminology of the C3 AI Suite, this is a 'leaf' of the pipeline. There are several concrete versions of this Type and they are usually different implementations within various machine learning systems.

CustomPythonMLPipe: A helper Type to act as a 'base' for defining new python based machine learning Pipes.

Concrete Types

MLSerialPipeline: This is the concrete implementation of the MLPipeline Type. Since MLSerialPipeline is so general, you won't have to subclass it.

SklearnPipe: This concrete implementation of MLLeafPipe provides a straightforward way to use sklearn machine learning pre-processing and modelling functions. See the TutorialIntroMLPipeline.ipynb notebook on your C3 Cluster for more information.

TensorflowClassifier: This Type allows the user to store a tensorflow estimator-based model, specifically a classifier. This Type currently works with tensorflow 1.9.0. See the TutorialTensorflowPipe.ipynb notebook on your C3 Cluster for more information.

TensorflowRegressor: This Type allows the user to store a tensorflow estimator-based model, specifically a regressor. This Type currently works with tensorflow 1.9.0. See the TutorialTensorflowPipe.ipynb notebook on your C3 Cluster for more information.

KerasPipe: This Type allows the user to store a keras tensorflow model. This Type currently uses tensorflow 2.1.0. See the TutorialKerasPipe.ipynb notebook on your C3 Cluster for more information.

XgBoostPipe: This Type implements the sklearn-compatible part of the Xgboost library. See the TutorialXgBoostMLPipeline.ipynb notebook on your C3 Cluster for more information.

Usage Examples

Let's take a look at the C3 AI developed TutoraiIIntroMLPipeline.ipynb notebook.

Prepare Data

The first step in any machine learning task is to prepare the data. In this case, we're going to use the popular iris dataset. We first use some sklearn functions to load the data, and split it into a training set and testing set.

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

iris = load_iris()
datasets_np = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)
```

Now, datasets_np consists of four numpy arrays: training data, training targets, testing data, and testing targets. If we inspect the array dimensions we expect the training data and testing data to have shape '(120,4)' and the training and testing targets to have shape '(120,)'.

We must now convert this numpy array into something usable by C3 AI Suite, a **Dataset**. C3 AI provides the helper function `c3.Dataset.fromPython()` to convert numpy arrays to datasets:

```
XTrain, XTest, yTrain, yTest = [c3.Dataset.fromPython(pythonData=ds_np) for ds_np in datasets_np]
```

Defining the C3 MLPipeline

C3 AI Machine Learning Pipelines can be thought of as a series of steps. These steps can be nested, so we can define a 'preprocessing' step (perhaps containing multiple steps itself) which can normalize and transform data into a better form for ML algorithms, and a regression step which runs the ML model on the transformed data. So, we'll build the MLPipeline step by step.

Preprocessing Pipeline

Let's build a preprocessing pipeline which will first scale the data within the interval [0,1], then we'll do a principal component analysis and extract the first two principal components. These components will be easy for a machine learning algorithm to use.

First, let's build sklearn StandardScaler, and PCA decomposition MLLeafPipes to contain those transformation steps:

```
# Define the MLLeafPipe which holds the StandardScaler step.
standardScaler = c3.SklearnPipe(
        name = "standardScaler",
        technique=c3.SklearnTechnique(
                # This tells ML pipeline to import sklearn.preprocessing.StandardScaler.
                name="preprocessing.StandardScaler",
                # This tells ML pipeline to call "transform" method on sklearn.preprocessing.StandardScaler
when we invoke the C3 action process() later.
                processingFunctionName="transform"
        )
)

# Define the MLLeafPipe which holds the PCA decomposition step.
pca = c3.SklearnPipe(
        name = "pca",
        technique=c3.SklearnTechnique(
                name="decomposition.PCA",
                processingFunctionName="transform",
                # hyperParameters are passed to sklearn.decomposition.PCA as kwargs
                hyperParameters={"n_components": 2}
        )
)
```

Now we can combine these two steps into a preprocessing MLSerialPipeline:

```
# Define the preprocessing pipeline which chains the scaler and pca steps together.
preprocessPipeline = c3.MLSerialPipeline(
        name="preprocessPipeline",
        steps=[c3.MLStep(name="standardScaler",
                    pipe=standardScaler),
                  c3.MLStep(name="pca",
                                      pipe=pca)]
)
```

Note that we store the individual steps as `MLStep` Types in the 'steps' array.

Now we have `preprocessPipeline` containing our preprocessing pipeline! We can use this pipeline as part of a larger pipeline now.

Regression Leaf Pipe

We now need to define our logistic regression model. We'll create an SklearnPipe (a concrete MLLeafPipe) which contains this example:

```
# Leaf-level Logistic Regression classifier.
logisticRegression = c3.SklearnPipe(
        name="logisticRegression",
        technique=c3.SklearnTechnique(
                name="linear_model.LogisticRegression",
                processingFunctionName="predict",
                hyperParameters={"random_state": 42}
        )
)
```

This SklearnPipe now contains the sklearn model.

Compose full Pipeline

Now, we build the final pipeline:

```
# Define complete MLPipeline
lrPipeline = c3.MLSerialPipeline(
        name="lrPipeline",
        steps=[c3.MLStep(name="preprocess",
                                    pipe=preprocessPipeline),
                  c3.MLStep(name="classifier",
                                    pipe=logisticRegression)],
    scoringMetrics=c3.MLScoringMetric.toScoringMetricMap(scoringMetricList=[c3.MLAccuracyMetric()])
)
```

This definition looks identical to the `preprocessPipeline` from before but for one addition, the '`scoringMetrics`' parameter. This parameter is for Pipelines that are meant to be trained. Here, we define the metric on which the ML algorithm should be trained to minimize. In this case, we see the `MLAccuracyMetric` has been chosen. Other metrics are available as well.

Execute '`c3ShowType(MLScoringMetric)`' in the JavaScript console and look at "Used By" to see other Types which mix in the MLScoringMetric Type.

Training the Pipeline

Once our MLPipeline has been defined we can use the training data from before to train it:

```
trainedLr = lrPipeline.train(input=XTrain, targetOutput=yTrain)
```

This produces a **new** Type `` `trainedLr` `` which contains all of the parameters and model definitions of the trained MLPipeline. We can now inspect this model's parameters and evaluate it on new data:

```
param_map = trainedLr.params()
param_map['preprocess__standardScaler__mean_']
```

Might return something like this:

```
c3.Arry<double>([5.809166666666665,
 3.0616666666666674,
 3.726666666666667,
 1.1833333333333333])
```

And to evaluate on new data:

```
prediction = trainedLr.process(input=XTest)
```

We can also score this MLPipeline based on the metric we chose with the '`score`' function:

```
score = trainedLr.score(input=XTest, targetOutput=yTest)
```

Storing and Retrieving the Trained Pipeline

Once a model is trained, you can store it as a persisted MLSerialPipeline Type. You can then retrieve this model later in a different script or different component of the C3 AI Suite. Let's look at storing:

```
upsertedPipeline = trainedLr.upsert()
```

Now `` `upsertedPipeline` `` contains an 'id' value of the upserted MLSerialPipeline object. We can retrieve this object one of two ways:

```
# Using the get function of the upsertedPipeline object
fetchedPipeline = upsertedPipeline.get()

pipeline_id = upsertedPipeline.id

# Using the MLSerialPipeline get function with the id
fetchedPipeline = c3.MLSerialPipeline.get(pipeline_id)
```

Now you can use `process` on new data with the fetched Pipeline!

Quick Example Using KerasPipe

Using snippets from the DTI created mnistExample, we'll show quickly how you can use the KerasPipe MLLeafPipe.

Suppose we have already prepared data XTrain and YTrain. We can build a tensorflow keras model and use a KerasPipe. First, define your model as you usually would:

```
import tensorflow as tf

# Build tensorflow model.
one_layer_simple_model = tf.keras.Sequential([
    tf.keras.layers.Reshape((28*28,), input_shape=(28,28)),
    tf.keras.layers.Dense(1000, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
one_layer_simple_model.compile(optimizer='Adam', loss='categorical_crossentropy')
```

Then, we create a KerasPipe Type using this model definition using the '`upsertNativeModel`' function. This function creates a new KerasPipe Type using the model definition object passed with the '`model`' parameter:

```
# Create the KerasPipe using the upsertNativeModel method
keras_pipe = c3.KerasPipe().upsertNativeModel(model=model)
# Set the epoch number to an appropriate amount for your model and data.
keras_pipe.technique.numEpochs = 10
```

Then we can train the model as we did with the sklearn model:

```
trained_pipe = keras_pipe.train(input=train_X, targetOutput=train_Y)
```

And finally, we can use the trained pipe with the '`process`' function:

```
result = trained_pipe.process(input=test_X)
```

Example Notebooks

Several C3 AI developed Jupyter notebooks exist which demonstrate the usage of these Pipeline Types:

- https://<vanity_url>/jupyter/notebooks/tutorials/TutorialIntroMLPipeline.ipynb
- https://<vanity_url>/jupyter/notebooks/tutorials/TutorialKerasPipe.ipynb
- https://<vanity_url>/jupyter/notebooks/tutorials/TutorialMLDataStreams.ipynb
- https://<vanity_url>/jupyter/notebooks/tutorials/TutorialProphetPipe.ipynb
- https://<vanity_url>/jupyter/notebooks/tutorials/TutorialStatsModelsTsaPipe.ipynb
- https://<vanity_url>/jupyter/notebooks/tutorials/TutorialTensorflowPipe.ipynb
- https://<vanity_url>/jupyter/notebooks/tutorials/TutorialXgBoostMLPipeline.ipynb

DTI developed notebooks:

- MNIST example: https://github.com/c3aidti/mnistExample

Additional Resources

- Developer Documentation
  - https://developer.c3.ai/docs/7.12.25/guide/guide-c3aisuite-basic/machine-learning-pipeline