MIC Final Project: Distributed Machine Learning with Spark

Geoffrey Pettet and Saroj Sahoo

Background

Machine learning encompasses a large class of techniques that attempt to derive patterns from data that are difficult for humans to detect. The data that is being learned from can take many different forms: images, website user information, and stock prices to name a few. Machine learning techniques take this data, and try to determine patterns such as how the inputs are related to the desired outputs and how inputs are related to each other. Once learned, the patterns can help determine characteristics for new data samples.

There are many different kinds of machine learning techniques that have different applications. Artificial Neural networks are constructed of many small ‘neurons’ that chain together and activate each other much like our brains, and are used in image classification tasks. Clustering is an unsupervised (i.e. input data has no labels) technique that finds groups of similar samples, and is used in areas such as Anomaly detection. Genetic Algorithms copy evolution by having many different models that are each randomly mutated, with the worst performers being ‘killed off’, and is (fittingly) used in DNA sequencing. This is just a small look into the many machine learning techniques and their applications.

As machine learning has increased in popularity, many of its challenges have become more apparent. The most practically limiting challenges are *computation time* and *storage requirements*. Machine learning algorithms require large amounts of training data to be effective, sometimes needing millions of data points or more (this is known as big data). This data must be stored somewhere, as well as processed. A confounding factor is the complexity of Machine learning algorithms: which each technique has varying complexity, many are NP-hard. These two issues culmenate in very large training times that require very powerful and expensive hardware.

One way to try to overcome this issue is using distributed computing: instead of running the algorithm on one machine, split it into parts and run it on several smaller computers. This means that the data can be split across many different machines that can all compute in parallel. The biggest issue with this technique is that it is very complex: machine learning algorithms are generally complex to begin with, and parallelizing them correctly is even more difficult. Luckily there are many frameworks that have implemented distributed machine learning algorithms for us. These frameworks include Spark, Storm, and MPI variants, each with different pros and cons.

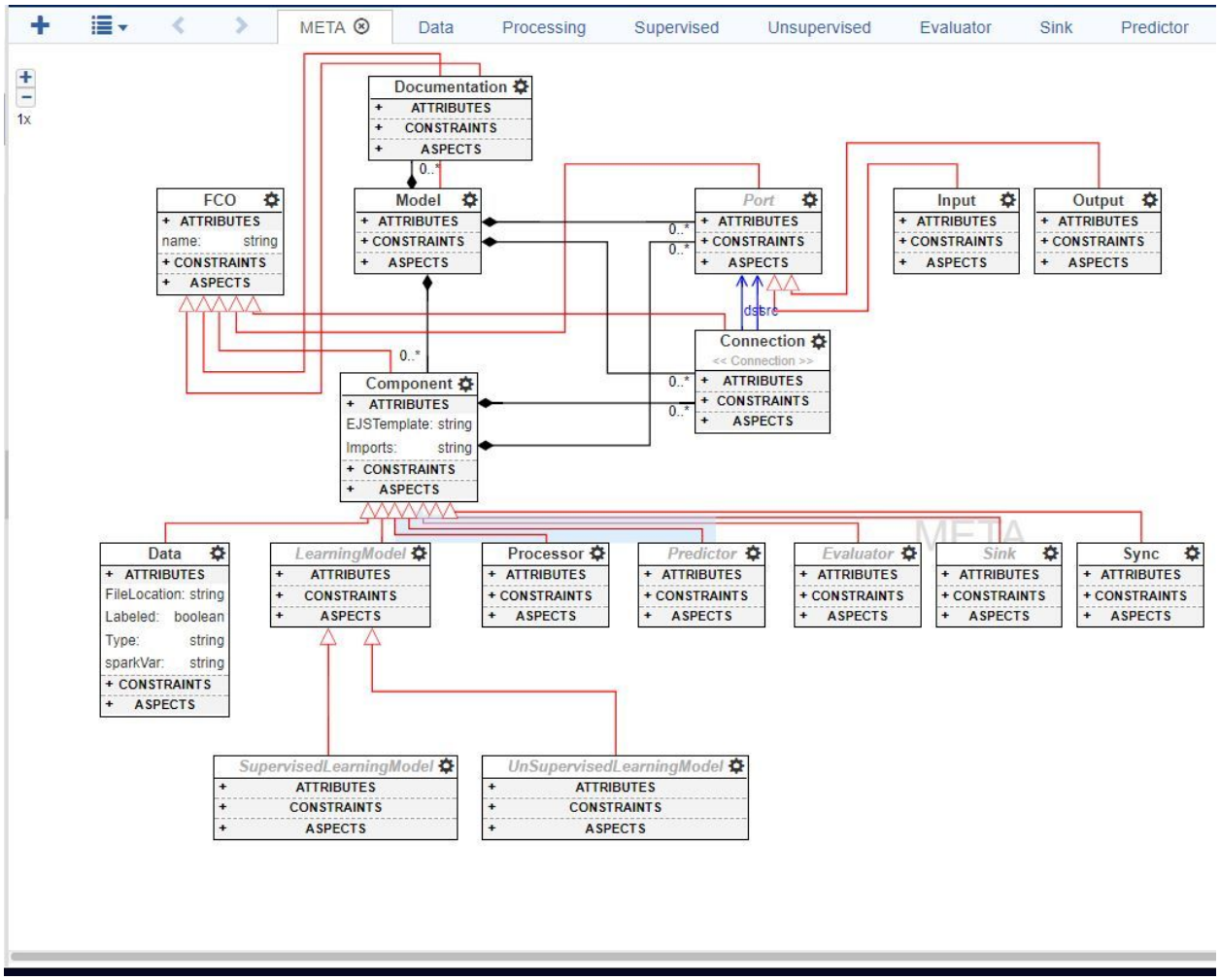
Project Description

The goal of our project is to allow a user to implement a complete distributed machine learning pipeline without having to worry about lower level implementation. These pipelines can include preprocessing, various machine learning techniques (including supervised and unsupervised models), as well as prediction and evaluation methods.

Once a user has created their visual pipeline in WebGME, we have created a plugin that converts that model into distributed code that can be run in the appropriate framework. As of now our plugin only supports Spark code generation, but we have built our metamodel in such a way so that it can easily be extended to other frameworks. This way users don't have to worry about the implementation language or details, and models can be reused on any supported framework.

META-Model

Our base meta model page has all the major nodes required in machine learning pipeline.



The various Nodes are:

-Model : This node is designed to contain the whole machine learning pipeline.

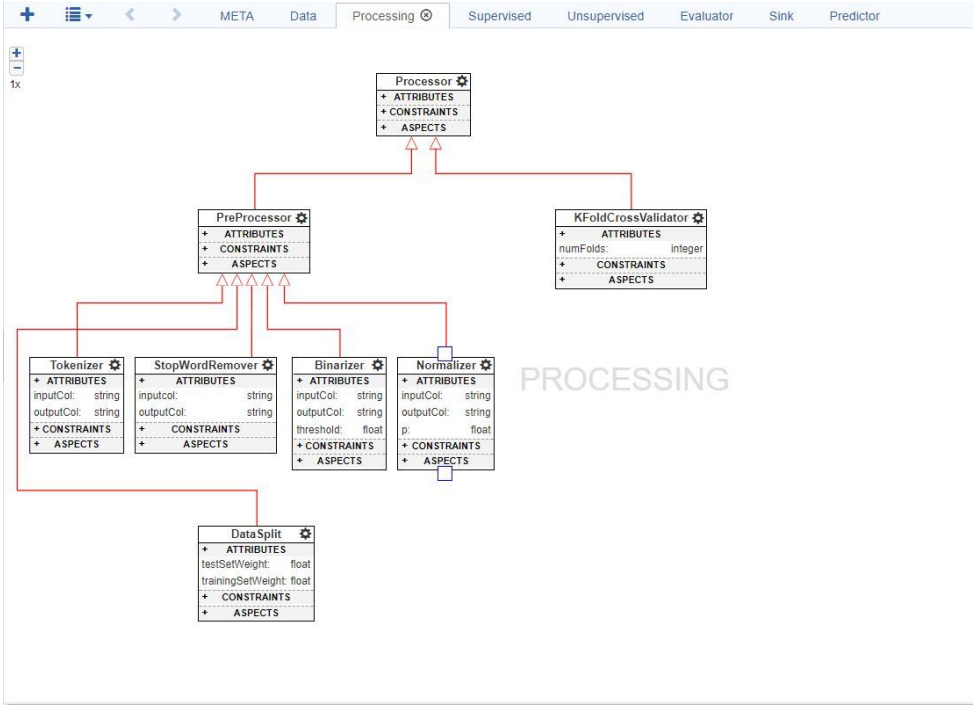
-Component : This node contains the major components of a machine learning pipeline. Various nodes contained inside the component are :

-Data : This node in itself contains the training, validation and datatobepredicted nodes. The data node has attributes like fileLocation which is user-defined i.e. the fileLocation provided by the uset will be used as the absolute file path in the code. Moreover it also has a boolean attribute called labeled hich describes wether the provided data is labeled or not.

- Training Data : The name itself is self explanatory - FileLocation attribute of the node will give the location of the training data that will be used to train the model.
- validationdata: this node will contain the location of test data in its attribute which will be used to test the created model.
- datatobepredicted: the attribute of this node will give the filepath of an unlabelled data that can used to generate labels using the created model.

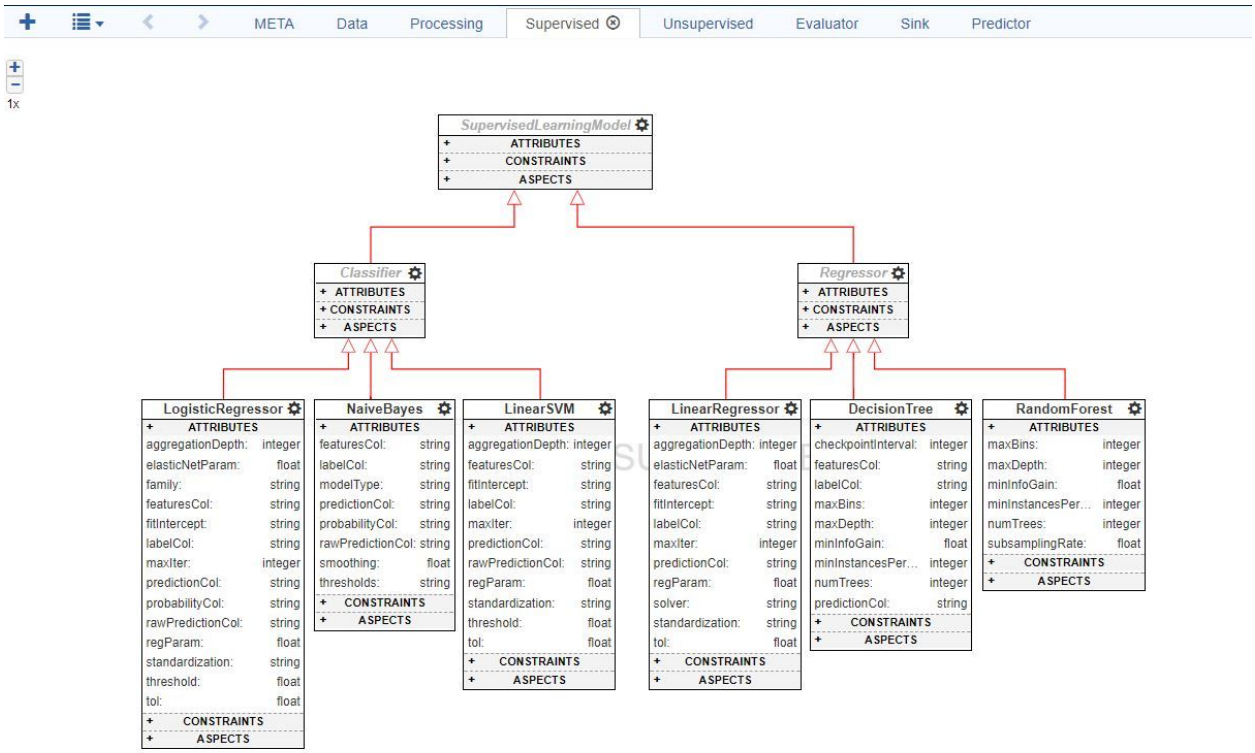
-Processor : This node contains various nodes that will allow the user to preprocess the raw data allowing them to create better performing ML pipelines. Below is the screenshot of the Processor node.

Various type of preprocessing allowed are : Normaliser,Tokenizer, stop word remover, Binarizer and Split.



-Learning Model : This is the core part of the ML pipeline. This node contains the various Supervised and unsupervised learning algorithms.

- SuperVisedLearningModel : This node contains few of the models that required labeled data as input and will give a prediction label as out when the model is run on an unlabeled data. This node is divided into further sub-categories:
    - Classifier : This node contains few of the Spark-ML classification models which will allow the user to classify data various classes as per need.
    - Regressor : This node contains various regressor models allowing user to get a prediction on unlabeled data using the features extracted from the training data set.



- UnsupervisedLearningModel : This node is similar to SupervisedLearningModel node and contains various unsupervised learning models like Clustering (k-means, GMM's)

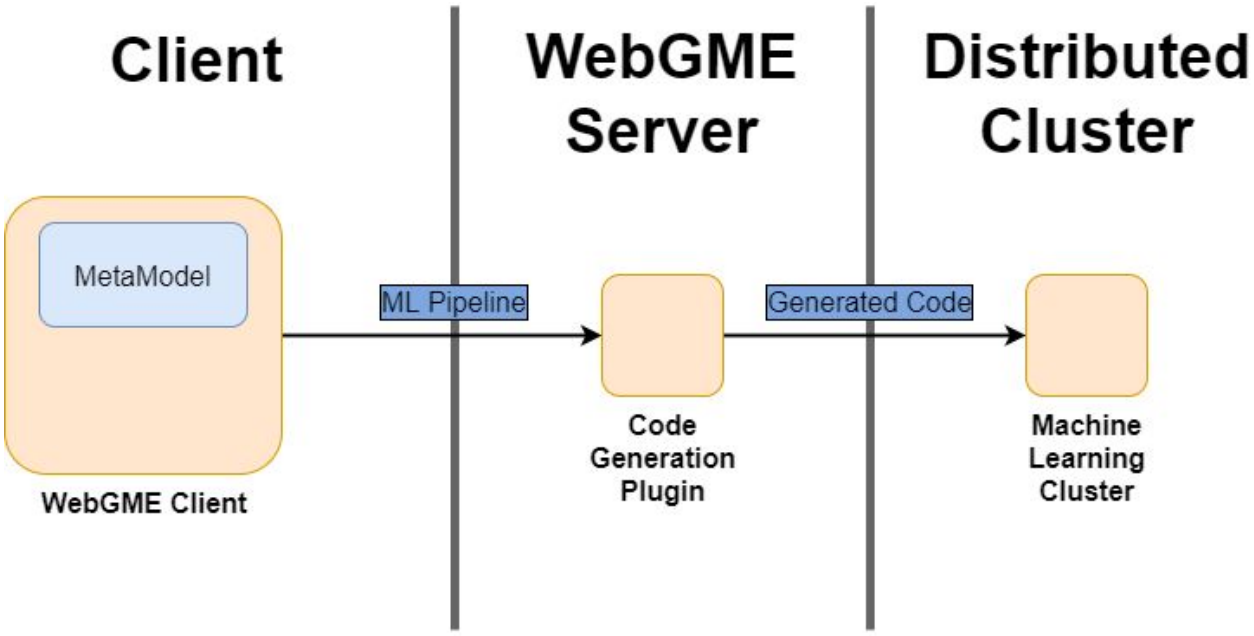
-Evaluators and Predictors : These contains various Evaluators and Predictors of the models allowing the user to select appropriate evaluation metric they would like to use to evaluate their model.

-Sink : This node allows the user to print to console any evaluation metric they would like to see.

Code Generation Plugin

Once the user has finalized their model, they then run our plugin on it to generate its implementation for the appropriate framework. Right now our plugin only generates pyspark scripts, but can easily be extended to allow for other frameworks. The plugin first iterates through the model to find all of the nodes the user has created. It then orders those nodes into a tree structure according to the sequence they can be executed in. This ensures that no lines of code are written before their prerequisite actions are completed. It then iterates through these tree, writing the appropriate code for each node. It does this using the EJS templating engine. The templates are stored on the meta nodes, and have keys corresponding to the inputs and attributes of each node. The plugin only has to supply a json object containing the node's attributes and connections to generate the appropriately formatted code. The use of these templates is key to the project's flexibility: to implement a new framework, we only need to add a new template to each meta node for that framework. The plugin itself only needs to change slightly to accommodate these new templates.

Implementation Diagram

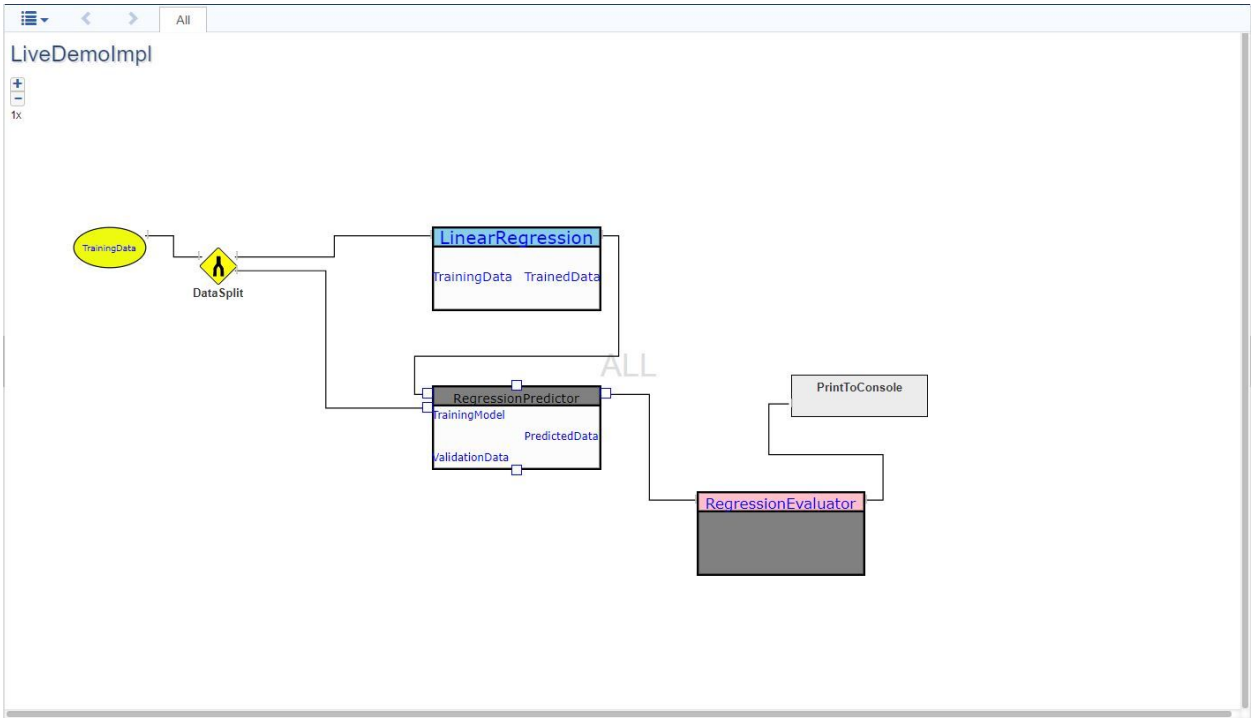


The diagram above shows the major components in our project's pipeline. First users create a machine learning pipeline using the WebGME client. They then run the plugin on the server to generate the appropriate code implementation of that model. This code can then be run on a cluster with the proper framework installed.

Example :

Below is a Linear Regression example on a Housing Training Data. Basically the training data contains features like the area of house, number of bedrooms, number of bathrooms etc. First the training data is split into training set and validation set at 7:3 ratio. After the split the training data is used as input to the LinearRegression Model. The LinearRegression model's hyperparameters have been changed accordingly in its attributes for better results. This model is then used as an input to RegressionPredictor also the Validation set obtained from split goes as input. The RegressionPredictor predicts the results which is then passed as input to the RegressionEvaluator which has rmse as its default evaluation metric, then the PrintToConsole node allows the user to the evaluated results of the model in console.

Once run through the plugin, it generates the appropriate spark implementation similar to the code shown below on the next page. All the variable names are based off the GUIDs of the components to ensure they are unique.



Result.py:

```
from pyspark.sql import SparkSession
from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.regression import LinearRegression


spark = SparkSession.builder.appName("generatedApplicaton").getOrCreate()
aaf90be7b_6e16_95ce_4ac4_03f32382b74b =
spark.read.format("libsvm").load('D:\Spark\spark-2.2.0-bin-hadoop2.7\spark-2.2.0-bin-hadoop2.
7\data\mllib\sample_linear_regression_data.txt')
(a3fe96886_2c9c_ccdf_6761_738f2e33f551, a6b3e1bd9_cc5f_ebf5_b80b_69cf582cfd98) =
aaf90be7b_6e16_95ce_4ac4_03f32382b74b.randomSplit([0.8, 0.2])

LinearRegressor = LinearRegression(featuresCol = "features", labelCol = "label", predictionCol
= "prediction", maxIter= 100, regParam= 0, elasticNetParam = 0 ,standardization = True, tol =
0.000001, fitIntercept = True, solver = "auto", aggregationDepth = 2)

acf14b2c1_bcec_c133_d388_b0a0d75c94f5 =
LinearRegressor.fit(a3fe96886_2c9c_ccdf_6761_738f2e33f551)

print("Coefficients: %s" % str(acf14b2c1_bcec_c133_d388_b0a0d75c94f5.coefficients))
print("Intercept: %s" % str(acf14b2c1_bcec_c133_d388_b0a0d75c94f5.intercept))
a3606a6f2_c32e_a469_e642_78cd23403450 =
acf14b2c1_bcec_c133_d388_b0a0d75c94f5.transform(a6b3e1bd9_cc5f_ebf5_b80b_69cf582cf
d98)
evaluator = RegressionEvaluator(labelCol="label", predictionCol="prediction",
metricName="rmse")
a9f47c719_ac97_2ad0_4007_d2c7c1f2954f =
evaluator.evaluate(a3606a6f2_c32e_a469_e642_78cd23403450)
print("Root Mean Squared Error : " + str(a9f47c719_ac97_2ad0_4007_d2c7c1f2954f))
```