MlLib Model Training Library

Xavier Mendez UTD ML Fall 2025

Professor Rishabh Iver

Table of Contents

Architectural Overview

The MlLib Library is intended for modular use with scripts defining specific project implementation classes and executing the classes needed functions. The focus of the library is on readability to better allow developers to understand the reasoning and use of different ml and mathematical function without exhausting cognitive load. The layout should be traceable and readable especially with an IDE allowing for quick navigation between classes. The Architecture was also built with reusability in mind to allow consistent results, and ease in comparing different models. To further the readability for a BE developer audience new code added uses camel case instead of python's common snake case.

A project specific script is the entry point of implementation. Within the project script a project class is defined and is meant to provide data source, a function buildModels() to declare, fit(), and evaluate() the models used in the project, and an object to store the results of the model training evaluations. Currently setup is required within the DataOrchestrator to provide model specific implementations. The DataOrchestractor handles the data source loading and transformations in preparation for model training. Once a model is fit the evaluator object belonging to the model will contain metrics derived during training and can be displayed through print functions belonging to the evaluator parent class.

The MlLib implementation is built with python 3.9 and primarily uses pandas.dataframes and np.ndarray data types for data manipulation and computation. Architectural patterns used in the design include Domain Driven Design, Visitor Behavior Design Pattern, Strategy Design Pattern, and Pipe and Filter Architecture. Libraries used within the MlLib implementation are os, sys, numpy, sklearn, typing, json, and pandas.

Main Implementation

This is intended to be implemented with scripts allowing Developers to define configurations that alter the model type, hyperparameters, seeds, and transformations applied to the data. This implementation is sorted into three core domains math, machine learning, and data. Developers use the dataDomain packages utilities to load datasets then build out the transformation pipeline for their dataset, and when the DataOrchestrator class is initialized these transformations and generated and ready to be used. Developers then use implementations of abstract/parent model classes within the mlDomain to create project specific subclasses that declare the configuration of parameter, and hyper parameters for the model to use.

An example use case for Developer using this library can be seen with linearRegression.py and logisticRegression.py files. A developer has the ability to declare a child class of either of these and implement a new approach to calculating the weight updates through overwriting the fit function and only changing the line which calls the function to produce the new weights and bias (assuming this new function has already been implemented). In logisticRegression.py this extensibility is furthered via the gridFit function which allows the loss function to be a parameter for training multiple models at once by leveraging dependency injection to pass in the loss function as an argument for use by calculateGradientDescent().

Once Models are trained genericEvaluator.py offers a modelEvaluator to derive custom model evacuators to meet the specific needs of the model being evaluated. Currently, the base model evaluator class is set up to provide functionality for binary classification cases, but is mainly intended to enforce the declaration of updateTestingPredictionData() and persistEvaluationRecord() which update model metrics on each training run (or whatever time the developer chooses) and then persists the data on to a record held within the object for display at the end of model training. Also available are some print function to be used with evaluation record.

Hyperparameter Grid

The hyperparameterGridOptions array contains model meta data, and options for model iterations. This parameter is declared within the project specific model implementations along with the evaluator object. This does NOT contain different transformations on the model, currently different transformations would have to be declared outside, and their data passed to a fit function belonging to the project specific model class implementation.

Example from AdClickPredictionLogReg.py

```
self.hyperparameterGridOptions = np.array([
    {
      'modelName': ['LogisticRegressionModelWithAgeBinning'],
      'learningRate': [0.005], #[0.0005, 0.001, 0.004, 0.008, 0.01, 0.015, 0.1],
      'epoch': [4], #[4, 5, 6, 7, 8, 10, 15],
      'lossFunction': [MSE(), MAE()],
      'weightRandSeed': [1], #[1,2,27],
      'initialBias': [1], #[-100, 1, 100]
    }
  ])
```

Current available options to declare for training multiple models with gridFit() are, learningRate, epoch, lossFunction, weightRandSeed, and initialBias. New options can be implemented through further implementation of the strategy pattern commonly by having the bass class using self.thingToMadeGridFittable or self.thing.gridDynamicThing = gridFittableThing if the new field is a part of a lower level implementation i.e. the loss function within the gradient descent calculation.

dataDomain Package

This Package hold classes responsible for handling and further delegating data related responsibilities. This Package focuses on separation of concerns between loading data, cleaning data, and transforming data. This package is prepped for rewriting into a data pipelining architecture that will allow the transformation configuration to live along side the hyperparameterGridOptions configuration further consolidating the project specific cases to the project specific class implementations. This Package currently handles CSV loading, classification and continuous data transformations, and uses pandas dataframes as main datatype.

Along with the other modules this module is intended for reuse. When the dataOrchestrator class is initialized the transformations for the entire project are completed allowing multiple project models to reuse data with the same transformation without having to recompute. To fetch a transformed dataframe the developer would pass the model name defined in the project class meta data as well as the dataOrchestrator in get_transformed_data(), and when ready to use the data get_transformed_data() would be called passing in the model name and returning the values and targets.

mathDomain Package

The Mathematical backbone of MlLib is built using the classes within the mathDomain package. The classes handle the calculations and mathematical foundation needed for model building. This package is split into algebra, graph, and probability folder with some classes needed to be refactored into their respective folder currently. This is planned to be expanded to allow for easy expansion of the hypothesis space, more options for splitting decision trees, and addition of classes to support random variable calculations and other probability-based classes.

AdclickPrediction MlLib Model Creation with Real Data

Implementing three model types using the MlLib functions so far has produced models with roughly %65 accuracy using various techniques. With these results, the current capabilities of this project are not high enough to be useful in a production capacity. Though the accuracy is not very high the extensibility of the library allows for easy adjustments and implementation of more advanced techniques resulting in a promising future outcome.

The LogisticRegression models had high precision and recall causing concern for underfitting the true positives are high and false positives almost nonexistent. The weights are likely not penalizing enough for incorrect false predictions during fitting. The second LogisticRegression model used binning on the scaled ages bucketing them into groups based of standard deviations then treating the age groups as categorical. This second model seemed to perform better with Iteration 94 having resulting metrics of accuracy = 0.6576, precision = 0.9839009287925696, and recall = 0.6569. These models are underfit likely due to the weights being constrained to the number of features currently and could benefit from testing an expanded hypothesis space.

LogisticRegression Model ParameterGrid:

```
self.hyperparameterGridOptions = np.array([
  {
    'modelName': ['LogisticRegressionModelWithAgeBinning'],
    'learningRate': [0.001, 0.004, 0.008, 0.01, 0.015, 0.1],
    'epoch': [6, 8, 10, 15, 20],
    'lossFunction': [MSE(), MAE()],
    'weightRandSeed': [1,2,27],
    'initialBias': [.5, 1, 2, 5]
  }
])
```

The DecisionTree implementation had a more balanced and promising model where accuracy = 0.6512, precision = 0.8068, and recall = 0.7014. This model also using binning for the age category, but unlike the logistic regression model did not one hot encode the

categorical variables. The model provides a good foundation and is ready to be used to build ensemble and boosted models that are based on decision tree classifiers.

AdclickPrediction Model Output

/Users/xaviermendez/Desktop/BotMaker/BotMaker/.venv/bin/python /Users/xaviermendez/Desktop/BotMaker/botMaker/MlLib/projectScripts/AdClickModelProjectBuildScript.py

Record Preview:

```
   id full_name   age  ... browsing_history time_of_day click
0  670  User670  22.0  ...       Shopping   Afternoon     1
1  3044 User3044   nan  ...            nan       nan     1
2  5912 User5912  41.0  ...      Education      Night     1
3  5418 User5418  34.0  ...  Entertainment    Evening     1
4  9452 User9452  39.0  ...   Social Media    Morning     0

[5 rows x 9 columns]
```

Shape: (10000, 9)

Column Types:

```
id                int64
full_name         object
age               object
gender            object
device_type       object
ad_position       object
browsing_history  object
```

time_of_day object

click int64

dtype: object

Numerical Summary:

	id	click
count	10000.000000	10000.000000
mean	5060.211400	0.650000
std	2861.758265	0.476993
min	5.000000	0.000000
25%	2529.000000	0.000000
50%	5218.000000	1.000000
75%	7466.000000	1.000000
max	10000.000000	1.000000

Memory Usage:

4.181785583496094 MB

Logistic Model Transformed Data Record Preview:

	age	click	...	time_of_day_Night	time_of_day_nan
0	-1.386448e+00	1	...	0.0	0.0
1	2.253536e-16	1	...	0.0	1.0
2	6.115249e-02	1	...	1.0	0.0
3	-4.721741e-01	1	...	0.0	0.0
4	-9.122653e-02	0	...	0.0	0.0

[5 rows x 20 columns]

Column Types:

age float64

click int64

gender_Male float64

gender_Non-Binary float64

gender_nan float64

device_type_Mobile float64

device_type_Tablet float64

device_type_nan float64

ad_position_Side float64

ad_position_Top float64

ad_position_nan float64

browsing_history_Entertainment float64

browsing_history_News float64

browsing_history_Shopping float64

browsing_history_Social Media float64

browsing_history_nan float64

time_of_day_Evening float64

time_of_day_Morning float64

time_of_day_Night float64

time_of_day_nan float64

dtype: object

Numerical Summary:

	age	click	...	time_of_day_Night	time_of_day_nan
count	1.000000e+04	10000.000000	...	10000.000000	10000.00000
mean	2.398082e-16	0.650000	...	0.190000	0.20000
std	7.235001e-01	0.476993	...	0.392321	0.40002
min	-1.691206e+00	0.000000	...	0.000000	0.00000
25%	-1.674160e-01	0.000000	...	0.000000	0.00000
50%	2.253536e-16	1.000000	...	0.000000	0.00000
75%	2.253536e-16	1.000000	...	0.000000	0.00000
max	1.813511e+00	1.000000	...	1.000000	1.00000

[8 rows x 20 columns]

Memory Usage:

1.5260009765625 MB

Duplicate Rows: 4589

Building Models...

Evaluation Summary : LogisticRegressionModelWithAgeBinning

Best Accuracy : 0.6548 (Iteration 349)

Best Precision: 1.0000 (Iteration 4)

Best Recall : 1.0000 (Iteration 40)

(Iteration 40)

{

```
  "modelData": {
    "epoch": 6,
    "initialBias": 1,
    "learningRate": 0.001,
    "lossFunction": "MAE",
    "modelName": "LogisticRegressionModelWithAgeBinning",
    "weightRandSeed": 1
  },
  "correctPredictions": 881,
  "truePositives": 13,
  "falsePositives": 1619,
  "trueNegatives": 868,
  "falseNegatives": 0,
  "accuracy": 0.3524,
  "precision": 0.007965686274509803,
  "recall": 1.0
}
```


(Iteration 4)


```
{
  "modelData": {
    "epoch": 6,
    "initialBias": 0.5,
    "learningRate": 0.001,
    "lossFunction": "MAE",
    "modelName": "LogisticRegressionModelWithAgeBinning",
```

```
      "weightRandSeed": 1
    },
    "correctPredictions": 1632,
    "truePositives": 1632,
    "falsePositives": 0,
    "trueNegatives": 0,
    "falseNegatives": 868,
    "accuracy": 0.6528,
    "precision": 1.0,
    "recall": 0.6528
}
```


(Iteration 349)


```
{
    "modelData": {
      "epoch": 10,
      "initialBias": 1,
      "learningRate": 0.015,
      "lossFunction": "MSE",
      "modelName": "LogisticRegressionModelWithAgeBinning",
      "weightRandSeed": 1
    },
    "correctPredictions": 1637,
    "truePositives": 1629,
    "falsePositives": 3,
    "trueNegatives": 8,
```

```
  "falseNegatives": 860,
  "accuracy": 0.6548,
  "precision": 0.9981617647058824,
  "recall": 0.6544797107271997
}
```


Evaluation Summary : LogisticRegressionModelWithAgeBinning

Best Accuracy : 0.6576 (Iteration 94)

Best Precision: 1.0000 (Iteration 1)

Best Recall : 1.0000 (Iteration 292)

(Iteration 1)


```
{
  "modelData": {
    "epoch": 6,
    "initialBias": 0.5,
    "learningRate": 0.001,
    "lossFunction": "MSE",
    "modelName": "LogisticRegressionModelWithAgeBinning",
    "weightRandSeed": 1
  },
  "correctPredictions": 1615,
  "truePositives": 1615,
  "falsePositives": 0,
  "trueNegatives": 0,
  "falseNegatives": 885,
```

```
  "accuracy": 0.646,
  "precision": 1.0,
  "recall": 0.646
}

(Iteration 292)
{
  "modelData": {
    "epoch": 10,
    "initialBias": 0.5,
    "learningRate": 0.001,
    "lossFunction": "MAE",
    "modelName": "LogisticRegressionModelWithAgeBinning",
    "weightRandSeed": 1
  },
  "correctPredictions": 889,
  "truePositives": 4,
  "falsePositives": 1611,
  "trueNegatives": 885,
  "falseNegatives": 0,
  "accuracy": 0.3556,
  "precision": 0.002476780185758514,
  "recall": 1.0
}

(Iteration 94)
```

```
{
  "modelData": {
    "epoch": 6,
    "initialBias": 2,
    "learningRate": 0.01,
    "lossFunction": "MAE",
    "modelName": "LogisticRegressionModelWithAgeBinning",
    "weightRandSeed": 1
  },
  "correctPredictions": 1644,
  "truePositives": 1589,
  "falsePositives": 26,
  "trueNegatives": 55,
  "falseNegatives": 830,
  "accuracy": 0.6576,
  "precision": 0.9839009287925696,
  "recall": 0.6568830095080612
}
```


Evaluation Summary : DecisionTreeModel

Best Accuracy : 0.6512 (Iteration 1)

Best Precision: 0.8068 (Iteration 1)

Best Recall : 0.7014 (Iteration 1)

(Iteration 1)


```
{
```

```
  "modelData": {
    "modelName": "DecisionTreeModel",
    "maxDepth": 15
  },
  "correctPredictions": 1628,
  "truePositives": 1311,
  "falsePositives": 314,
  "trueNegatives": 317,
  "falseNegatives": 558,
  "accuracy": 0.6512,
  "precision": 0.8067692307692308,
  "recall": 0.7014446227929374
}

Process finished with exit code 0
```

Difficulties

The core change in creating this library was how to separate and isolate the different portions of the whole ML domain. This was made difficult by the many different methods of mathematically representing the same idea or deriving the same variable. To further this issue ensuring the data types being consistent as they flowed down from the ml layer to the math calculations at a lower level was a challenge especially with trying to make the function as reusable and cross compatible as possible.

Another challenge in building this library was debugging and scaffolding different portions for development. The library is intended to be used end to end with many pieces coming together and if a data value or meta data value like number of columns was of or not made dynamic finding the cause in the stack became difficult as I had not defined which layer had which responsibilities from the beginning.

The last major challenge is similar to the second's lack of definition but pertains to the contract of how hyper parameters, parameters, and different data transformations in configurations were meant to be defined and which object(s) held responsibility of what. This is currently handled through the implementation of multiple design patterns and separation of concerns now, but during development determining the correct combinations of strategies for future extensibility took a good deal of consideration and effort.

Next Steps

MlLib 2.0 will feature extensions to the ml models it can create, additional model building control, and introduction of functionality to create static assets that allow a frontend skin to visually represent model training metrics. Standardization of the library and coding standards will also be included in future updates.

Feature model building extensions will focus on Neural Network and Probability Based Math functions. Different splitting functions for decision trees will also be added. The Datatransformer pipeline will be implemented, and data transformations migrated to project files. Additional options for regularization will be added along with other methods to reduce overfitting for the various models. Finally for the Regression models functionality to expand the hypothesis space will be included.