Database-Integrated Analytics

Submitted By:

Shaolin Xie

May 05, 2021

Advisor: Mohamed Eltabakh

Abstract

The coordination between data analytics and database systems becomes exceedingly important in order for data scientists to efficiently analyze data that is stored inside the database. Currently, there are three approaches to use data analysis tools with databases: client-server connection, in-database processing, and embedded database. This project focuses on comparing the client-server connection to the in-database processing. Two machine learning models - Support Vector Machine and Random Forest - are implemented using each of the approaches and then tested on datasets of different scales. In this project, the in-database processing approach is achieved using Apache MADlib, and the client-server connection approach is implemented using python codes. After comparing the run-time efficiency and the testing accuracy of the two approaches, conclusions are drawn regarding the performance of each approach.

Table of Contents

1. Introduction

Big data is ingrained in every industry. Companies try to gather data to uncover the insights and trends of their business, and Relational Database Management System (RDBMS) is the most popular and widely used method for data storage and management. While the data is stored in the database systems, machine learning tools that analyze the models are usually built to work in a stand-alone fashion.

The gap between the analytical tools and the database creates inconvenience and inefficiency for data scientists to analyze data. Therefore, the coordination between machine learning and database systems becomes exceedingly important in order for data scientists to efficiently analyze data that is stored inside the database.

The goal of this project is to investigate and evaluate database-integrated analytics tools, specifically by comparing the client-server connection approach to the in-database processing approach. Two machine learning models - Support Vector Machine and Random Forest - are arbitrarily chosen and implemented in Apache MADlib and Python. The run time efficiency and testing accuracy are compared in order to determine whether MADlib functions or Python implementations have better performance in terms of conducting database-integrated analysis.

2. Background

The relational database management system is the most popular and widely used method for data scientists to store and manage data, however, there is a gap between the analytical tools and data storage. There will be lots of trouble if the data scientist just manages their data by storing them as text files, especially when the data scientist has to deal with multiple sources at the same time. Currently, there are three different approaches to use data analysis tools with the database: client-server connection, in-database processing, and embedded database.

2.1 Client-Server Connection

The client-server connection is where the database and tools are completely separated. The relationship between analytical tools and databases can be demonstrated by Figure 2.1.

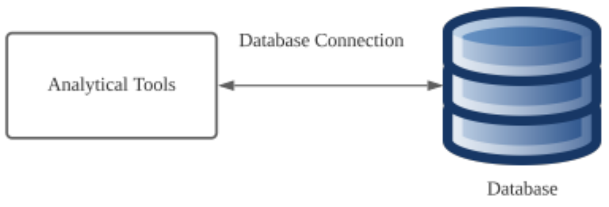


Figure 2.1 Client-server connection model

To analyze the data that is stored inside the database, the analytical tool needs to give commands to export data from the database to the analytical tool. After the tools process the data and compute the result, it will send the output back to the database. The process is shown in Figure 2.2 [1].

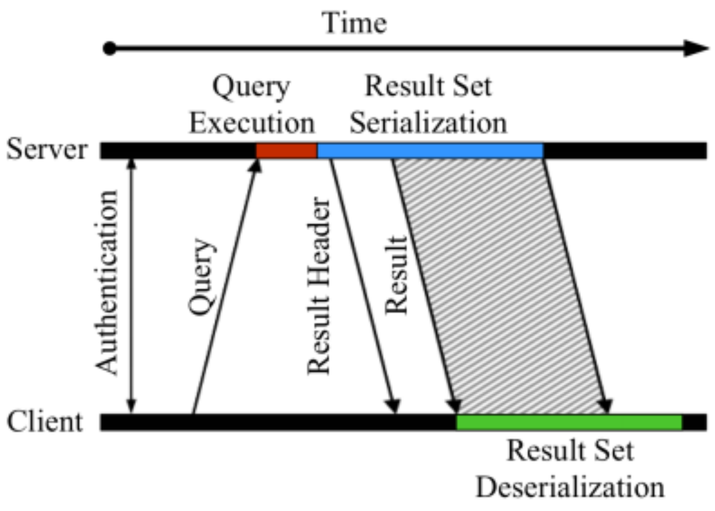


Figure 2.2 Client-server connection process

The client-server connection has the advantage that this approach is database-agnostic, where the ODBC or JDBC connectors can be used to connect to almost any database. It is also easy to load files without changing the algorithm pipelines in such a case. However, exporting tables could be time-consuming, especially when tables are large. It also requires the client to have the memory that is big enough to fit the full dataset. Moreover, the bottleneck heavily depends on the clients' protocols.

2.2 In-database Processing

For the in-database processing, the analytical tools are implemented inside the database. The data stays inside the database. The relationship of such an approach can be viewed as the graph Figure 2.3.

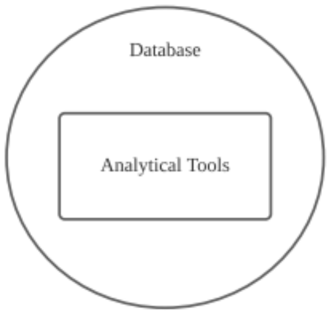


Figure 2.3 In-database processing model

First of all, this method solves the problem of time being spent on exporting data. It allows data scientists to perform some analysis inside the database. This approach can be easily achieved by loading data to the pipeline using standard-compliant SQL functions. However, it becomes difficult if we need more than simple analysis tools, such as building models and create user-defined functions. For the in-database approach, user-defined functions are usually written in C/C++ languages, which requires a significant rewrite of the existing pipelines as well as in-depth knowledge of the database internals.

2.3 Embedded Database

The embedded database approach lets users install and run databases within the client programs.

Figure 2.4 shows the structure of the embedded database.

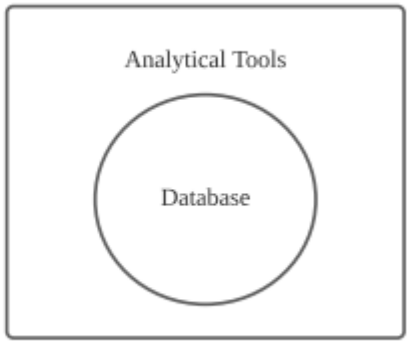


Figure 2.4 Embedded database model

This approach requires significant effort from the users to keep the database maintained and tuned. Its benefits don’t usually outweigh the efforts when the user is working on small-scale data analysis.

2.4 Apache MADlib Library

For the purpose of studying in-database processing, Apache MADlib is chosen for this project. The MADlib library provides helpful machine learning functions that operate with PostgreSQL and Greenplum databases.

2.4.1 MADlib Structure

As an in-database processing approach, there are three major components of MADlib source code: Python driver functions, C++ implementations functions, and the C++ database abstraction layer [2].

The driver functions are the main entry point from user input and are responsible for the flow control of the algorithms, such as validating input parameters, executing SQL statements, evaluating the results, and looping to execute more SQL statements.

The C++ functions are the C++ definitions of the core functions. They are implemented in C++ rather than Python as needed.

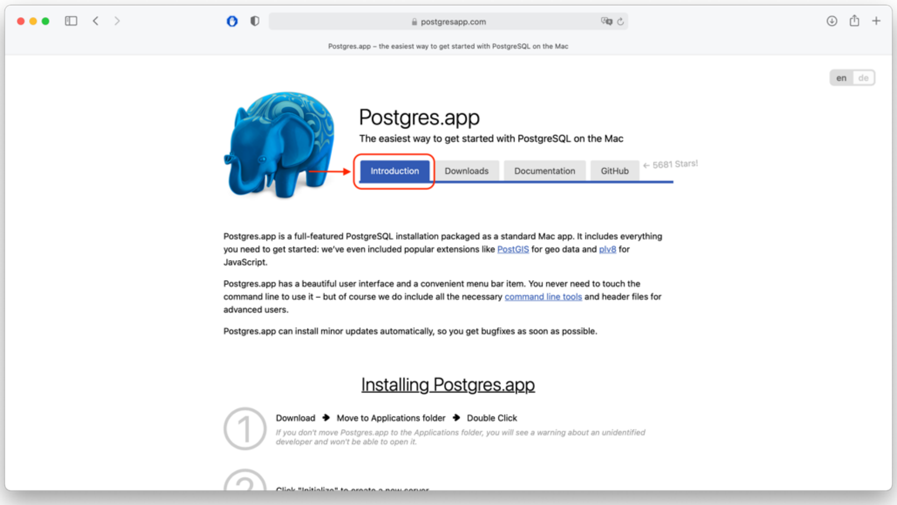
The C++ database abstraction functions provide a programming interface that abstracts all the Postgres internal details away such that MADlib can focus on the internal functionality rather than the platform integration logic.

For each MADlib module, there is a required .sql_in file that creates database objects for this method. There are also optional .py_in, and .c files, which are Python and C/C++ codes that help in pre/post-processing data and support method algorithms.

2.4.2 MADlib Installation

For this project, MADlib and Postgres are installed on OSX. Since MADlib only works with PostgreSQL 11 and 12, I download the dmg file for Postgres.app with version PorstgreSQL12.

I then installed the Postgres.app following the instructions on Postgresql.org and initialize databases in the app to port 5432, shown in Figures 2.5 and 2.6.


postgresapp.com
Postgres.app – the easiest way to get started with PostgreSQL on the Mac
en
de
Introduction
Downloads
Documentation
GitHub
← 5681 Stars!

Postgres.app

The easiest way to get started with PostgreSQL on the Mac

Postgres.app is a full-featured PostgreSQL installation packaged as a standard Mac app. It includes everything you need to get started: we've even included popular extensions like PostGIS for geo data and plv8 for JavaScript.

Postgres.app has a beautiful user interface and a convenient menu bar item. You never need to touch the command line to use it – but of course we do include all the necessary command line tools and header files for advanced users.

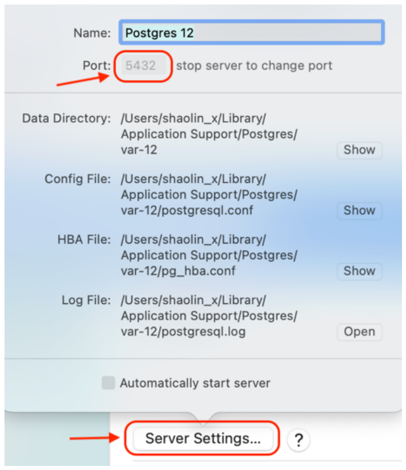
Postgres.app can install minor updates automatically, so you get bugfixes as soon as possible.

Installing Postgres.app

1 Download → Move to Applications folder → Double Click

If you don't move Postgres.app to the Applications folder, you will see a warning about an unidentified developer and won't be able to open it.

Figure 2.5 Postgre.app instructions



Name: Postgres 12

Port: 5432 stop server to change port

Data Directory: /Users/shaolin_x/Library/Application Support/Postgres/var-12

Config File: /Users/shaolin_x/Library/Application Support/Postgres/var-12/postgresql.conf

HBA File: /Users/shaolin_x/Library/Application Support/Postgres/var-12/pg_hba.conf

Log File: /Users/shaolin_x/Library/Application Support/Postgres/var-12/postgresql.log

Automatically start server

Figure 2.6 Port initialization

After that, I can use SQL commands from the terminal. In the terminal, I create a new database "mad" and installed MADlib using the following commands:

```
psql -d postgres -c "CREATE DATABASE mad"
/usr/local/madlib/bin/madpack -p postgres -c $USER@$HOST/mad install
```

Now "mad" database is created inside the Postgres app, shown in Figure 2.7, where I can execute all the MADlib commands [3].

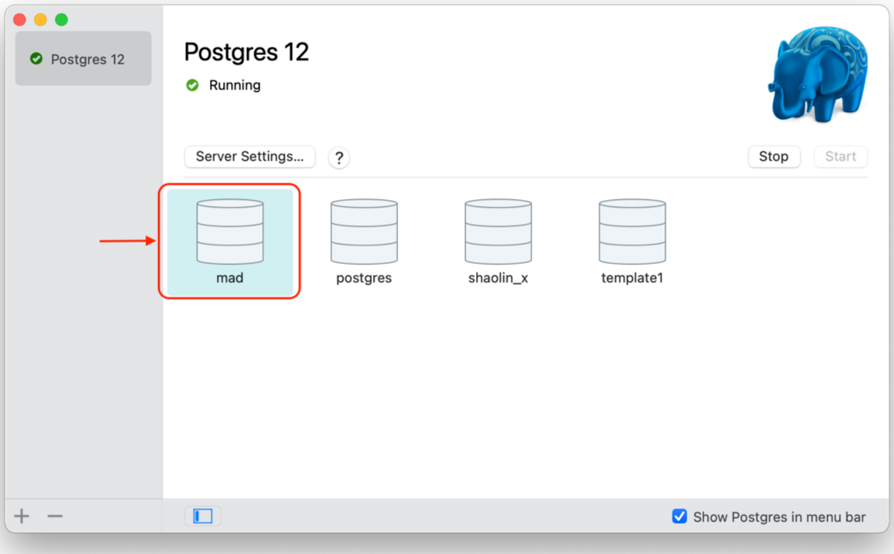


Figure 2.7 “mad” database in Postgres.app

2.5 Python Implementations

To achieve the client-server connection, I use psycopg2 to connect to the PostgreSQL database from Python. After importing the datasets from the database, I saved the data as data frames and can initiate analysis algorithms on them. Python is chosen in comparison to the MADlib functions since MADlib’s implementations of SVM and random forest are both partially composed by Python codes. Out of personal interests to comprehend the algorithms as well as understanding the code complexity of implementing the machine learning models, SVM and random forest functions are then built from scratch to support the tests, which I call the client-server standalone (CSS) method.

While my CSS implementations use the same algorithms as the MADlib functions, I also want to adopt a library that has the same level of maturity as the MADlib library such that the comparison of the efficiency and accuracy between the in-database and client-server methods is more accurate. Algorithms from scratch don’t usually have the ideal scaling when datasets become large, and the results could be off, and efficiency could become low. Therefore, I also choose to use the SVM and random forest classification functions in the Scikit-learn library as the second implementation and the reference group for the client-server applications.

3. Analytical Tools

Support Vector Machine and random forest models are chosen from the MADlib library due to their potential to provide useful results. They are both supervised learning models, which makes the testing results consistent and better monitored.

3.1 Supervised vs Unsupervised Learning

Machine learning is the technique to let computers learn and act like humans, and it can improve their behaviors by feeding them more data and observations. Supervised and unsupervised learning are two categories of machine learning.

Supervised machine learning, just like its name suggests, is the type of learning that has existing results or expected results when we train the models. Regression and classification are the two supervised machine learning categories, and general linear models and tree-based models are the most commonly used examples. Supervised learning usually has the following working pipeline:

- Split the dataset into a training set and a testing set.
- Build the model and train the model with a training set to find the relationship between inputs and outputs.
- Use the model to predict the result in the testing set.
- Compare the result and the real data in the training set.
- Adjust the model as needed.

Unsupervised machine learning, on the other hand, doesn't require a training and testing set. Data scientists simply need to input all the data and the model will then find the pattern itself. Clustering is one of the commonly used unsupervised learning models, where data is classified into groups or clusters based on the pattern [4].

3.2 Support Vector Machine (SVM)

SVM is a machine learning model that is used for both regression and classification, although it is more widely used for classification. The two-class linear SVM method classifies data points into two groups with a straight line on a plane, and the goal of such an algorithm is to find the line that best classifies two groups.

The following Figure 3.1 shows the possible hyperplanes that classify data points into two classes. But we need an algorithm to find the best hyperplane that classifies the data most accurately.

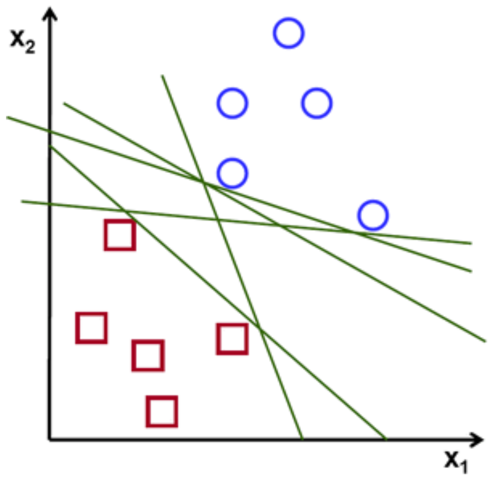


Figure 3.1 Possible hyperplanes for classification

SVM is used to find a hyperplane on an n-dimensional space that separates data points into different classes. Our objective is to find the hyperplane that maximizes margin, which is the distance from the hyperplane to the nearest training data point of each group. Such hyperplane is the optimal hyperplane, which is shown in the figure below. It is also called the decision boundary. Vectors that lie on the margin are called support vectors. Support vectors determine the shape of the decision boundary and changing them will affect the position of the hyperplane See Figure 3.2 for the SVM hyperplane[8].

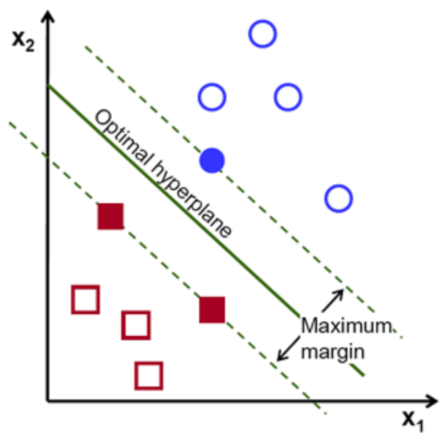


Figure 3.2 SVM classification hyperplane

The linear SVM model has the following expression:

$$f(x) = sign\left(\mathbf{w}^* \cdot x + \mathbf{b}^*\right)$$

To maximize the margin, SVM uses a cost function that has the following expression:

$$min_w \lambda \parallel w \parallel^2 + \sum_{i=1}^{n} (1 - y_i \langle x_i, w \rangle)_+$$

In the expression, there is the regularization parameter λ, which is used to balance the margin and the loss. The larger λ gives a wider margin. If the predicted value and the actual value are different, we calculate the loss value and add it to the loss. We want to find a weight *w* that minimizes the function.

Now to find gradients, we take partial derivative regarding *w* in the loss function, which looks like the following:

$$\frac{\delta}{\delta w_k} \lambda \parallel w \parallel^2 \quad = 2\lambda w_k$$

$$\frac{\delta}{\delta w_k} \left(1 - y_i \langle x_i, w \rangle\right)_+ \quad = \begin{cases} 0, & \text{if } y_i \langle x_i, w \rangle \geq 1 \\ -y_i x_{ik}, & \text{else} \end{cases}$$

The gradients are used to update the weights. When our model correctly predicts the class of a data point, we update the gradient using the following function:

$$w = w - \alpha \cdot (2\lambda w)$$

Otherwise, we update the gradient along with the loss:

$$w = w + \alpha \cdot (y_i \cdot x_i - 2\lambda w)$$

3.3 Random Forest

Random forest is a supervised machine learning model for classification. It is composed of many decision trees. How the decision tree works are that if we start from the root node, we determine whether a data point is classified to the left node or the right based on a true or false answer to the criteria. If the answer is true, the node is classified to the left node, otherwise to the right node. The process is repeated until the bottom nodes are reached [9].

Building on top of the decision tree algorithm, the random forest algorithm uses bootstrapping method to create many uncorrelated decision trees.

To construct the binary decision trees, a measurement of uncertainty called entropy is used. Entropy has the following expression:

$$H(X) = -\sum_j p_j \log p_j$$

where we want to minimize H(X) andp_j is the probability of class j. When we split a training set, we want to find a set that maximizes the entropy.

By finding the difference between fore and after a split, we calculate information gain (IG). The information gain has the following formula:

$$IG(D) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

where D_p, D_{left}, and D_{right} are the datasets from the parent, left child, and right child node, and N_p, N_{left}, and N_{right} are the number of observations of these nodes.

To build a random forest that contains n trees, we draw n bootstrap samples from the data and grow a decision tree from each of those samples. The predictions generated by random forest are the voting majority of all the terminal nodes.

4. Methodology

To compare the performances of in-database processing and client-server connection, SVM and random forest models are implemented on datasets of different scales. For each model, the Python functions are developed to be comparable to the corresponding MADlib functions. The three datasets were chosen each have 533 entries [5], 32419 entries [6], and 146956 entries [7]. After testing Python and MADlib functions on each of the datasets, the runtime and accuracy of each run are recorded and analyzed to conclude. All the datasets are two-class classification predictions. Datasets are cleaned and all independent variables have an integer value. All duplicates are removed from the datasets. In each implementation, the model is trained on the first 80% of the dataset and tested on the rest 20%.

To test different approaches, two databases are created in Postgres.app. "mad" database is used to initialize data tables and execute MADlib commands for the in-database approach, and the "mqp" database is used to initialize data tables for the client-server approach.

Because of the designs of MADlib functions, the attribute "id" of type SERIAL is created for each table. In addition, each dataset is split into training and testing set before they are separately loaded into the database as .csv files using the COPY command. The run time of each query is recorded using the "\timing on" command. Accuracy is calculated by finding the ratio of the correct predictions to the total number of test entries. To ensure the training and testing sets are consistent for each implementation, the shuffle is set to false for train_test_split() functions.

For "mqp" database, data tables are simply loaded from .csv files using the COPY command.

4.1 MADlib Function Algorithm – SVM

The MADlib SVM classification function has the following format:

```
svm_classification(
source_table,
model_table,
dependent_varname,
independent_varname,
kernel_func,
kernel_params,
grouping_col,
params,
verbose
)
```

In this project, the source_table is the training dataset, which is 80% of the whole dataset. Model_table is an output table that is generated from the SVM classifier. Depedent_varname is always called "Y" in the input table and has exactly two distinct values. Independent_varname is always all the columns that are other than "Y". Kernel_func is set to "linear" for all tests, and verbose is default to FALSE. All the other parameters default to NULL.

The SVM prediction function has the following format:

```
svm_predict (model_table,
          new_data_table,
          id_col_name,
          output_table)
```

The new_data_table contains prediction data, and the output_table contains the prediction results.

After training the training set with svm_classification(), a prediction table is generated using svm_predict(). The number of misclassifications is found and the accuracy rate is then calculated [11].

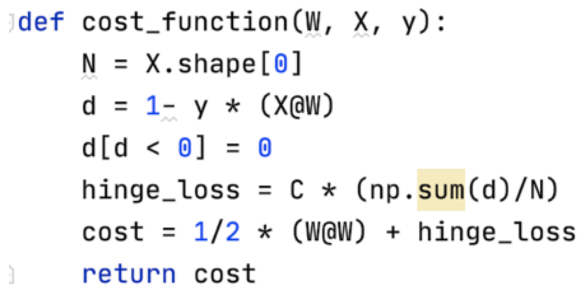
4.2 Python Function Algorithm – SVM

The Python code for SVM is mainly composed of three functions.

The cost_function() takes weights, input features and target output. Mathematically, the cost function is given by the given expression:

$$J(\mathbf{w}) = \frac{1}{2}||\mathbf{w}||^2 + C\left[\frac{1}{N}\sum_{i}^{n} \max\left(0,\ 1 - y_i * (\mathbf{w}\cdot x_i + b)\right)\right] \qquad (1)$$

The second half of the function is the hinge loss, which is calculated using the regularization parameter C times the average distance from each input data point to the output. The Python code looks like the following:



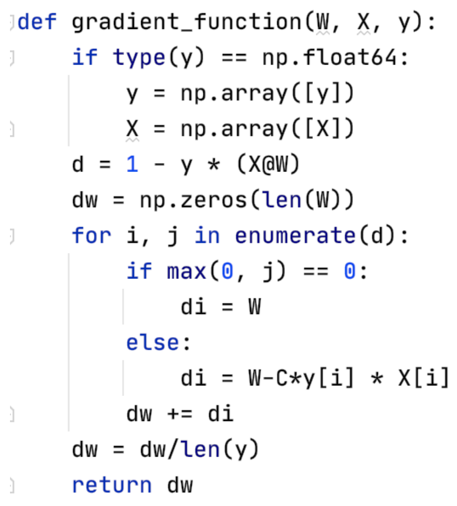
```python
def cost_function(W, X, y):
    N = X.shape[0]
    d = 1- y * (X@W)
    d[d < 0] = 0
    hinge_loss = C * (np.sum(d)/N)
    cost = 1/2 * (W@W) + hinge_loss
    return cost
```

The second function is the gradient_function(). It takes in the weights and the features and the output cost gradient. Its function looks like the following:

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i}^{n}\left[\frac{1}{2}||\mathbf{w}||^2 + C\max\left(0,\ 1 - y_i * (\mathbf{w}\cdot x_i)\right)\right] \qquad (4)$$

$$\nabla_w J(\mathbf{w}) = \frac{1}{N}\sum_{i}^{n}\begin{cases}\mathbf{w} & \text{if } \max\left(0,\ 1 - y_i * (\mathbf{w}\cdot x_i)\right) = 0\\ \mathbf{w} - Cy_i\, x_i & \text{otherwise}\end{cases} \qquad (5)$$

If the prediction is correct, we keep the weight; otherwise, we update the weight with the loss function. The Python codes look like the following:



```python
def gradient_function(W, X, y):
    if type(y) == np.float64:
        y = np.array([y])
        X = np.array([X])
    d = 1 - y * (X@W)
    dw = np.zeros(len(W))
    for i, j in enumerate(d):
        if max(0, j) == 0:
            di = W
        else:
            di = W-C*y[i] * X[i]
        dw += di
    dw = dw/len(y)
    return dw
```

The third method is the sgd() function. This function takes the features and the predictions as inputs and returns the calculated weights to the optimal hyperplane. The function iterates the implementations of the loss functions and gradient function until the stoppage criteria are signaled. In this method, the function will stop looping until there is no significant decrease in the cost function [10].

4.3 MADlib Function Algorithm - Random Forest

The MADlib random forest classification function has the following format:

```
forest_train(training_table_name,
        output_table_name,
        id_col_name,
        dependent_variable,
        list_of_features,
        list_of_features_to_exclude,
        grouping_cols,
        num_trees,
        num_random_features,
        importance,
        num_permutations,
        max_tree_depth,
        min_split,
        min_bucket,
        num_splits,
        surrogate_params,
        verbose,
        sample_ratio
        )
```

The training_table_name is the input training set. Output_table_name is the generated table containing the model. List_of_features is the list of predictors. list_of_features_to_exclude and grouping_cols are set to NULL in the tests. num_trees is the maximum number of trees to grow, which is set to 10. num_random_features is set to 1. The variable importance is set to false. nu_permutations is default to 1, max_depth of the tree is set to 5, min_split is the minimum number of splits that must exist in a node for a split to be attempted, which is set to 20; min_bucket is 3, which means a minimum of 3 observations needs to be in any terminal node. The num_splits is the number of splits per continuous variable; it is set to 5.

From the forest_train() function, a model table, a group table, and a summary table are produced.

The prediction function for random forest has the following syntax:

```
forest_predict(random_forest_model,
            new_data_table,
            output_table,
            type)
```

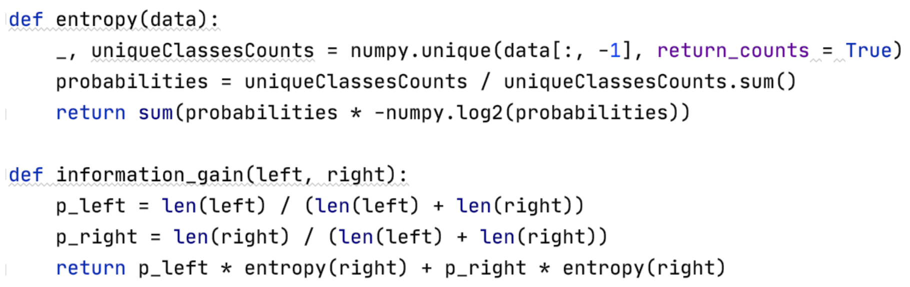
The random_forest_model table contains the random forest model from training. The new_data_table is the table containing the prediction data. The output_table contains the prediction results [12].

In the source code, the random forest module is implemented by using a py_in file, a sql_in file, and a cpp file that helps to process data, as long as inheriting some functions from the decision tree algorithm.

4.4 Python Function Algorithm – Random Forest

The random forest algorithm is mainly composed of two part: building a decision tree and bootstrapping datasets to make a random forest.

To build a decision tree, we use functions entropy() and information_gain() to determine the best split point. The entropy() function inputs the probability of a class within a node and calculates the entropy; the information_gain() function takes in the left and right child of a node and calculates the information gain of this particular split. The Python code looks like the following:

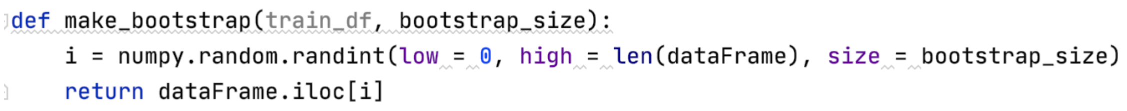


```
def entropy(data):
    _, uniqueClassesCounts = numpy.unique(data[:, -1], return_counts = True)
    probabilities = uniqueClassesCounts / uniqueClassesCounts.sum()
    return sum(probabilities * -numpy.log2(probabilities))

def information_gain(left, right):
    p_left = len(left) / (len(left) + len(right))
    p_right = len(right) / (len(left) + len(right))
    return p_left * entropy(right) + p_right * entropy(right)
```

Using entropy() and information_gain functions, the function spliting_point() is created to find the splitting point. It takes in the data frame and all the potential splits. The function iterates through each potential split and return the best splitting column and value that produces the highest information_gain [13].

Then I use the make_bootstrap() function to draw bootstrap samples. It takes in a training set as data frame and the bootstrap size, then returns the bootstrapped data frames of the given size. The code is as follows:



```
def make_bootstrap(train_df, bootstrap_size):
    i = numpy.random.randint(low = 0, high = len(dataFrame), size = bootstrap_size)
    return dataFrame.iloc[i]
```

Then I implemented the decision_tree() function, which takes the data frame as input, with current depth, minimum sample size, max tree depth, random attributes, and random split attributes. Then it outputs a decision tree.

With the implementation of decision_tree() function, the final random_forest() function is easily created, which takes the input values training data frame, the bootstrap size, random attributes, random splits attributes, forest size, and the tree max depth. The function then outputs the random forest.

5. Results and Analysis

The result table (Table 5.1) below is generated from testing different models in MADlib and with Python codes:

		In-Database Processing		**Client-Server Connection**			
		MADlib Functions		**CSS Functions**		**Scikit-learn Functions**	
	Dataset	Run Time (ms)	Accuracy	Run Time (ms)	Accuracy	Run Time (ms)	Accuracy
SVM	Small	525.48	70.09%	2143.68	85.04 %	556.607	85.04%

	Medium	9612.341	60.29%	12509.702	75.22%	13294.266	75.22%
	Large	48671.174	45.38%	53623.046	47.97%	53441.558	51.68%
Random Forest	Small	1062.238	85.04%	639.664	77.57%	728.246	82.24%
	Medium	7983.027	73.23%	30603.045	78.20%	5329.16	75.37%
	Large	171203.971	73.11%	170412.256	67.47%	20543.422	77.51%

Table 5.1 Test results

To see the results graphically, I created Figure 5.1 for run time comparisons, and Figure 5.2 for accuracy comparisons between three implementations.

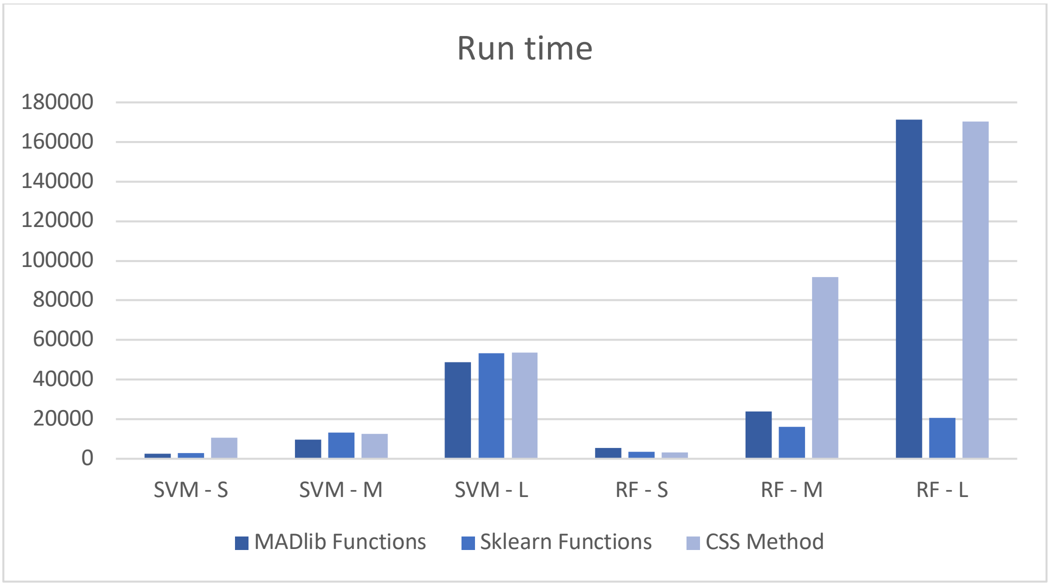


Figure 5.1 Run time comparisons

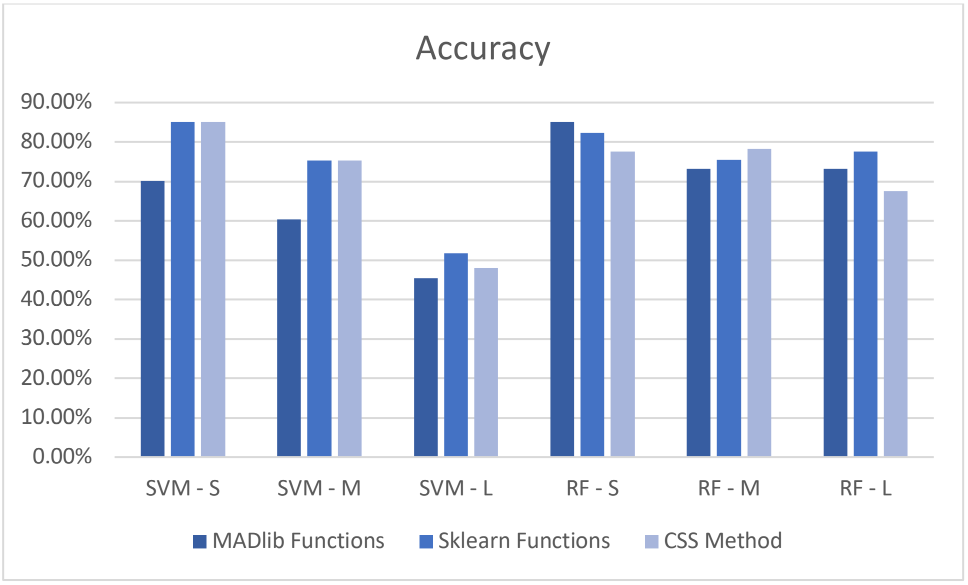


Figure 5.2 Accuracy comparisons

The graphs show that the MADlib SVM function has the shortest runtime among three implementations but doesn't have the same advantage for its random forest function. The MADlib random forest method never has shorter runtime comparing to the scikit-learn. CSS function is very inconsistent, but its runtime is usually large among the three tests.

The MADlib's random forest function has a run time that is significantly larger than the sklearn application, especially when working with the large dataset.

In terms of testing accuracy, both CSS implementation and the scikit-learn implementation have higher accuracy than the MADlib when fitting the SVM model. For the random forest model, MADlib has higher accuracy for the small dataset but lower accuracy for medium and large set when comparing to the scikit-learn approach.

I also want to make a note that although my own code is around six times the length of the MADlib codes, the length of codes to implement scikit-learn functions is almost the same as the MADlib code, which means that the simplicity to implement the in-database and client-server algorithms are almost the same if the algorithm is already defined in some library.

6. Conclusion

Based on the results from the experiments, I generated the conclusion that among the methods used in the project, the scikit-learn library has the steadiest performance and scales the best. The client-server approaches generally has slightly higher accuracy than the in-database approach and better efficiency when working with more complex algorithms.

The MADlib library is a powerful tool with mature algorithms and acquires big advantage in run time efficiency by operating analysis directly inside the database. However, it has its limitations in terms of test accuracy and algorithm efficiency. Although connecting to the database from Python takes some amount of time, the accuracy and efficient algorithm designs of the sklearn library could outweigh the cost to export large tables.

My inference for the big difference between the random forest runtime of the MADlib functions and the sklearn functions is that MADlib has multiple files that are integrated indifferent parts of its modules. The training complexity of the algorithm itself as well as the various coordination between different files slowed down the computation inside the database, and even led it to crash for quite a few times. The program also caused issue with the database itself that the database was unable to load the data tables, and I had to restart the laptop for multiple times. The program also encountered a few crashes when implementing random forest algorithms using the Python code, but I was always able to terminate the code and go back to fix the problem without affecting the database at all.

Although the CSS implementation also has quite inefficient run times, it is implemented in a single python file, whereas the MADlib functions requires coordination of several Python, SQL and C++ files as well as database objects to compose a machine learning mode. The simplicity to design code that is used outside the database creates convenience for the client-server approach. The inconvenience of the in-database processing also occurs when I needed to calculate the accuracy rate or split the dataset in to training and testing set, which are processes that I needed to execute outside the database during the tests.

Bibliography

1. Raasveldt, Mark. “Integrating Analytics with Relational Databases.” PhD@VLDB (2018).

2. “Apache Software Foundation.” Architecture - Apache MADlib - Apache Software Foundation, cwiki.apache.org/confluence/display/MADLIB/Architecture.

3. Worms, David. “Installing and Using MADlib with PostgreSQL on OSX.” Adaltas RSS, www.adaltas.com/en/2012/07/07/postgres-madlib-installation-example/.

4. Devin Soni. “Supervised vs. Unsupervised Learning.” *Medium*, Towards Data Science, 21 July 2020, towardsdatascience.com/supervised-vs-unsupervised-learning-14f68e32ea8d.

5. *UCI Machine Learning Repository: Blood Transfusion Service Center Data Set*, archive.ics.uci.edu/ml/datasets/Blood+Transfusion+Service+Center.

6. UCI Machine Learning Repository: Census Income Data Set, archive.ics.uci.edu/ml/datasets/Census+Income.

7. *UCI Machine Learning Repository: Clickstream Data for Online Shopping Data Set*, archive.ics.uci.edu/ml/datasets/clickstream+data+for+online+shopping.

8. Gandhi, Rohith. “Support Vector Machine - Introduction to Machine Learning Algorithms.” Medium, Towards Data Science, 5 July 2018, towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

9. *Random Forests From Scratch*, carbonati.github.io/posts/random-forests-from-scratch/.

10. Abbassi, Qandeel. “SVM From Scratch-Python.” Medium, Towards Data Science, 1 Apr. 2020, towardsdatascience.com/svm-implementation-from-scratch-python-2db2fc52e5c2#d7d8.

11. “Support Vector Machines.” *MADlib*, madlib.apache.org/docs/latest/group__grp__svm.html.

12. “Random Forest.” *MADlib*, madlib.apache.org/docs/latest/group__grp__random__forest.html.

13. Carbonati. “Carbonati/Machine-Learning.” GitHub, 23 Dec. 2017, github.com/carbonati/machine-learning/blob/master/random-forests/random%20forests.ipynb.

Appendix

The original goals of this project were to integrate a few machine learning functions by myself inside the MADlib library, which required the installation of the MADlib source code. I had made many attempts to install MADlib from source code, but an error kept prompting that "'cstddef' file not found". I first thought the error was caused by the incompatible version of boost that was installed, but the real reason was that a library called libstdc++, which has been removed from MAC OS around five years ago, is extensively used in the source code. Many attempts were made to integrate this function with my laptop, but they all failed. There were a lot of pre-installation steps done when trying to install the MADlib library. They are not used for the final paper, but the guide that I wrote could be helpful for other people or future references. The guide is shown below.

Current Mac version:
macOS Big Sur Version 11.2.3

1. Install GNU M4

https://www.gnu.org/software/m4/m4.html

Run command:

```
git clone git://git.sv.gnu.org/m4
```

Then cd to the directory of m4, which for me is /Users/shaolin_x/m4. Then run command:

```
git checkout -b branch-1.4 origin/branch-1.4
```

2. Install Xcode command line tools

Run command:

```
xcode-select –install
```

*If the xcode doesn't work properly after this installation, download Xcode from Apple App Store. The file is around 11GB.

3. Install Homebrew

https://brew.sh

Run command:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

4. Install cmake 3.19.7

Run command:

```
brew install cmake
```

5. Install Clang 12.0.0

https://clang.llvm.org/get_started.html

To install, follow the instructions from the website for "On Unix-like Systems"

*This installation took around 5 hours

The installed Clang version is as following:

```
shaolin_x@ShaolindeMacBook-Pro ~ % clang -v
Apple clang version 12.0.0 (clang-1200.0.32.29)
Target: x86_64-apple-darwin20.3.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

6. Install Bison3.7.6

Run command:

```
brew install bison
```

Make sure bison version is 3.7.6. Check use command:

```
bison --version
```

Mac has a built-in Bison 2.3 that is detected under path /usr/bin/bison.

To make sure the newer version of Bison is used, open .bash_profile file with command:

```
touch ~/.bash_profile; open ~/.bash_profile
```

add this line to the end of .bash_profile file:

```
export PATH="/usr/local/bin:/usr/local/sbin:~/bin:$PATH"
```

Then run command:

```
source ~/.bash_profile
```

To check the step is successful, run command:

```
sudo vi /etc/paths
```

Make sure `/usr/local/bin` is on top of `usr/bin`

7. Install Flex

Run command:

```
brew install flex
```

Make sure flex version is 2.6.4. Check use command:

```
flex --version
```

8. Install Doxygen

Download from https://www.doxygen.nl/download.html#srcbin.

Follow instructions under "GIT repository".

9. Install dot

Run command:

```
brew install graphviz
```

10. Install poppler

Run command:

```
brew install poppler
```

11. Install PostgreSQL12

Download PostgreSQL12 from https://postgresapp.com/downloads.html.

Follow the 3 steps under tab "Introduction" on https://postgresapp.com to finish installing

12. Install cmake 3.5.2

https://cmake.org/install/

Download source code cmake-3.5.2.tar.gz from https://cmake.org/files/v3.5/

cd to the source code directory, which for me is /Users/shaolin_x/cmake-3.5.2

Run commands:

```
./bootstrap

make

sudo make install
```

13. Install LaTeX

Follow the exact instructions on this website:
https://www.wellesley.edu/lts/techsupport/latex/latexmac

14. Install boost1.75.0

Run commands:

```
brew install boost
export BOOST_INCLUDEDIR=/usr/local/opt/boost/include/
```

15. Install Apache MADlib

Download apache-madlib-1.17.0-src.tar.gz from
https://dist.apache.org/repos/dist/release/madlib/1.17.0/.

Run commands:

```
cd /Users/shaolin_x/apache-madlib-1.17.0-src
./configure
mkdir build
cd build/
make
```

Or run commands:

```
cd /Users/shaolin_x/apache-madlib-1.17.0-src
```

```
mkdir build
cd build
cmake .. -DCXX11=1
make
```

Refer to installation guide:

https://cwiki.apache.org/confluence/display/MADLIB/Installation+Guide#InstallationGuide-CompileFromSourceCompilingFromSource

After running `cmake .. -DCXX11=1` command, it should report that:

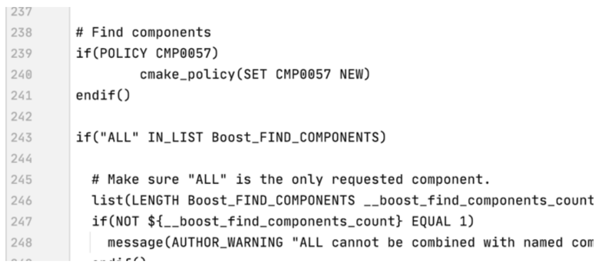
```
Boost 1.47 found.
-- No sufficiently recent version (>= 1.47) of Boost was found. Will download.
>> Adding PostgreSQL 12.0 (x86_64) to target list...
-- Could NOT find Greenplum (missing:  GREENPLUM_EXECUTABLE)
-- Using default web-based MathJax
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/shaolin_x/apache-madlib-1.17.0-src/build
```

After running `make` command, it might encounter error `CMP0057`.

To fix the error, go to `/usr/local/lib/cmake/Boost-1.75.0/BoostConfig.cmake` and add the following code to line 240:

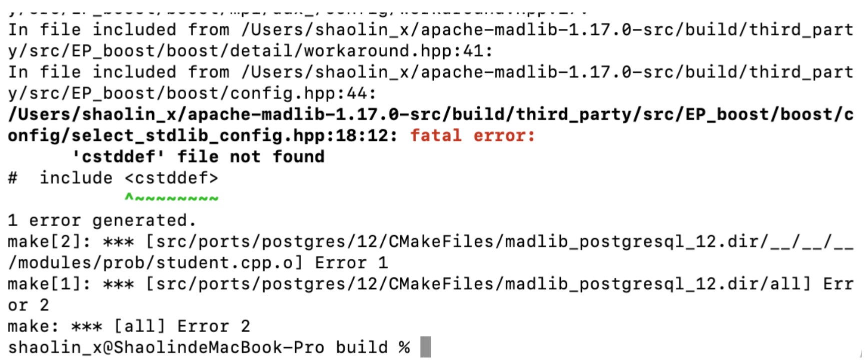
```
if(POLICY CMP0057)
cmake_policy(SET CMP0057 NEW)
endif()
```

It will look like this:



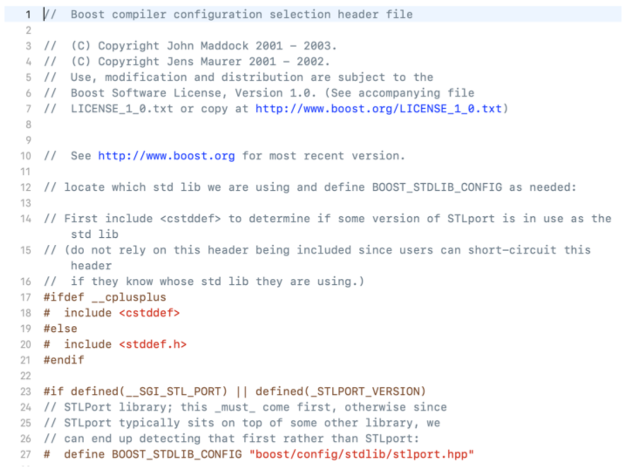
```
237
238  # Find components
239  if(POLICY CMP0057)
240          cmake_policy(SET CMP0057 NEW)
241  endif()
242
243  if("ALL" IN_LIST Boost_FIND_COMPONENTS)
244
245    # Make sure "ALL" is the only requested component.
246    list(LENGTH Boost_FIND_COMPONENTS __boost_find_components_count
247    if(NOT ${__boost_find_components_count} EQUAL 1)
248      message(AUTHOR_WARNING "ALL cannot be combined with named com
```

Then if I attempt to run `make` command again, the process will be stopped by error:



```
In file included from /Users/shaolin_x/apache-madlib-1.17.0-src/build/third_part
y/src/EP_boost/boost/detail/workaround.hpp:41:
In file included from /Users/shaolin_x/apache-madlib-1.17.0-src/build/third_part
y/src/EP_boost/boost/config.hpp:44:
/Users/shaolin_x/apache-madlib-1.17.0-src/build/third_party/src/EP_boost/boost/c
onfig/select_stdlib_config.hpp:18:12: fatal error:
      'cstddef' file not found
#  include <cstddef>
           ^~~~~~~~~
1 error generated.
make[2]: *** [src/ports/postgres/12/CMakeFiles/madlib_postgresql_12.dir/__/__/__
/modules/prob/student.cpp.o] Error 1
make[1]: *** [src/ports/postgres/12/CMakeFiles/madlib_postgresql_12.dir/all] Err
or 2
make: *** [all] Error 2
shaolin_x@ShaolindeMacBook-Pro build %
```

Looking at the file that generates error:



```
//  Boost compiler configuration selection header file

//  (C) Copyright John Maddock 2001 - 2003.
//  (C) Copyright Jens Maurer 2001 - 2002.
//  Use, modification and distribution are subject to the
//  Boost Software License, Version 1.0. (See accompanying file
//  LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)


//  See http://www.boost.org for most recent version.

// locate which std lib we are using and define BOOST_STDLIB_CONFIG as needed:

// First include <cstddef> to determine if some version of STLport is in use as the
    std lib
// (do not rely on this header being included since users can short-circuit this
    header
//  if they know whose std lib they are using.)
#ifdef __cplusplus
#  include <cstddef>
#else
#  include <stddef.h>
#endif

#if defined(__SGI_STL_PORT) || defined(_STLPORT_VERSION)
// STLPort library; this _must_ come first, otherwise since
// STLport typically sits on top of some other library, we
// can end up detecting that first rather than STLport:
#  define BOOST_STDLIB_CONFIG "boost/config/stdlib/stlport.hpp"
```

My guess is that the error is caused by boost.

Now the only problem I need to fix is to install a Boost that has version later than 1.47 but no later than 1.65 (required by MADlib).

16. Install boost again

*This is the part where I encounter problem.

Need to install Boost version >=1.47 and <=1.65.

Download boost_1_64_0.tar.bz2 from https://dl.bintray.com/boostorg/release/1.64.0/source/

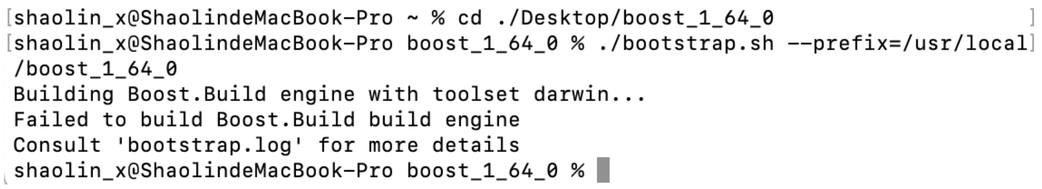
cd to the directory, which for me is ./Desktop/boost_1_64_0. Run commands:

```
cd ./Desktop/boost_1_64_0

./bootstrap.sh --prefix=/usr/local/boost_1_64_0

./b2 cxxflags="-std=c++14" install
```

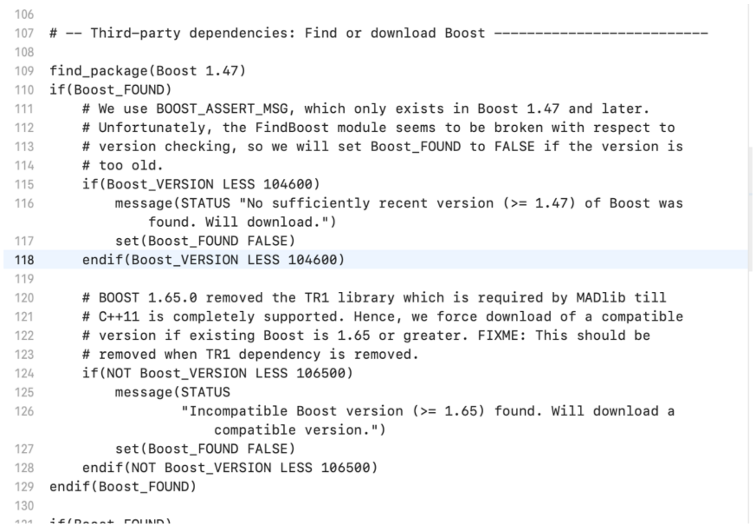
However, I encountered error message when running ./bootstrap.sh command:



```
shaolin_x@ShaolindeMacBook-Pro ~ % cd ./Desktop/boost_1_64_0
shaolin_x@ShaolindeMacBook-Pro boost_1_64_0 % ./bootstrap.sh --prefix=/usr/local
/boost_1_64_0
Building Boost.Build engine with toolset darwin...
Failed to build Boost.Build build engine
Consult 'bootstrap.log' for more details
shaolin_x@ShaolindeMacBook-Pro boost_1_64_0 %
```

Installation guide is on this website:
https://www.boost.org/doc/libs/1_69_0/more/getting_started/unix-variants.html#install-boost-build
Note that I tried to run the exact same commands for boost version 1.75 source file, and it compiled and built with no problem.
TODO: I don't know how to build any boost of version 1.60~1.65.

I also located the source code of how MADlib detects boost (in file apache-madlib-1.1.70-src/src/CMakeLists.txt).



```

# -- Third-party dependencies: Find or download Boost --------------------------

find_package(Boost 1.47)
if(Boost_FOUND)
    # We use BOOST_ASSERT_MSG, which only exists in Boost 1.47 and later.
    # Unfortunately, the FindBoost module seems to be broken with respect to
    # version checking, so we will set Boost_FOUND to FALSE if the version is
    # too old.
    if(Boost_VERSION LESS 104600)
        message(STATUS "No sufficiently recent version (>= 1.47) of Boost was
            found. Will download.")
        set(Boost_FOUND FALSE)
    endif(Boost_VERSION LESS 104600)

    # BOOST 1.65.0 removed the TR1 library which is required by MADlib till
    # C++11 is completely supported. Hence, we force download of a compatible
    # version if existing Boost is 1.65 or greater. FIXME: This should be
    # removed when TR1 dependency is removed.
    if(NOT Boost_VERSION LESS 106500)
        message(STATUS
                "Incompatible Boost version (>= 1.65) found. Will download a
                    compatible version.")
        set(Boost_FOUND FALSE)
    endif(NOT Boost_VERSION LESS 106500)
endif(Boost_FOUND)

```