In-Database Embedded Analytics

A Major Qualifying Project report to be submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the Degree of Bachelor of Science

Submitted By:

Kepei Lei

Elene Kavtaradze

Advisor(s):

Mohamed Eltabakh

03/23/2021

Table of Figures **3**

Abstract **4**

Introduction **5**

Integrating Analytics with Relational Databases *5*

PostgreSQL *8*

MADlib *9*

Methodology **10**

K-Means *10*

Loop 12

MADlib 13

Naive Bayes *13*

Postgres 14

MADlib 15

Logistic Regression *15*

Postgres 16

MADlib 18

Random Forest *19*

Postgres 19

MADlib 20

Analysis and Results **22**

Installation, Usage, and Syntax *22*

Postgres 22

MADlib 22

K-Means *23*

Naïve Bayes *25*

Logistic Regression *25*

Random Forest *26*

Conclusions **28**

Acknowledgments **29**

Works Cited **30**

Appendices **32**

Appendix A *32*

Appendix B *33*

Appendix C *35*

Appendix D *37*

Appendix E *41*

Table of Figures

Figure 1: Different ways to implement analytical tools with the database. Adapted from (Raasveldt) 7
Figure 2: Squared Euclidean Distance function........ 11
Figure 3: Beginning of Recursive CTE for K-Means........ 12
Figure 4: The function call of K-Means in MADlib........ 13
Figure 5: Part of the Classifier function for Naïve Bayes........ 15
Figure 6: The function call of Naïve Bayes in MADlib 15
Figure 7: The function of logistic regression prediction we implemented in PL/pgSQL........ 17
Figure 8: The function of logistic regression we implemented in PL/pgSQL........ 18
Figure 9: The function call of logistic regression training from MADlib 18
Figure 10: Part of the split function we implemented in PL/pgSQL. 19
Figure 11: Part of the prediction function we implemented in PL/pgSQL........ 20
Figure 12: The function call of random forest from MADlib........ 20
Figure 13: Runtimes of K-Means implementations in PostgreSQL 23
Figure 14: Runtimes of K-Means implementations in PostgreSQL and MADlib........ 24
Figure 15: Implementations of Naïve Bayes in MADlib and PostgreSQL 25
Figure 16: The time it took to run the training on the given data size for logistic regression. 26
Figure 17: The time it took to run the training on the given data size for the random forest. 27

Abstract

This paper recognizes the disconnect between database systems and data analytics tools. To eliminate the need to export data from the database systems into analytical tools, we explore implementing analytics modules inside databases. We implemented K-Means, Naïve Bayes, Logistic Regression, and Random Forest algorithms in PostgreSQL and MADlib. We found that MADlib has a slight advantage over PostgreSQL implementations.

Introduction

Our world revolves around data. From individuals to large corporations, many entities handle large amounts of data. There is a demand for storing more data more efficiently, and there is a demand for analyzing the data faster and cheaper (Panoho). There is much work being done on databases and analytics tools. The database system developers have focused on minimizing redundancy and optimizing the database system's read and writing performances. Analytics tools are getting better and more efficient. However, there is currently a disconnect between these two systems (Raasveldt 1). Database systems and analytics tools are being built and optimized as standalone entities.

A database system is the go-to storage solution for many entities, but not for analytics. As data scientists have few tools to utilize database systems' analytics capabilities, they use external analytics tools. This process requires saving the data from the database to input into the analytics tools. As a result, the data scientists end up performing I/O action on the same data twice. It is not as efficient as working directly in the database system, eliminating the need to store and upload data.

Object-oriented databases allow developers to implement complex functions with less redundant efforts. Since the database system compiles, optimizes, and runs the code, a function built into the database is supposed to have the best performance: it fetches the least data needed and takes advantage of the I/O optimization developers already did for database systems. The goal of our project is to implement commonly used machine learning analytics models in database languages. We will be implementing algorithms in Postgres and comparing their performance to the already available tool MADlib.

Integrating Analytics with Relational Databases

Integrating Analytics with Relational Databases by Mark Raasveldt discusses the importance and challenges of integrating data-intensive analytical tools with RDBMS.

According to the paper, these tools manually manage their data through flat-file storage (structured text or binary files). Maintaining this type of data requires much manual force, especially if it is large and lacks a rigid schema. It is also hard to share data with multiple users. Modifying this data is prone to corruption because ACID properties cannot be upheld. Because of this disconnect, instead of using an RDBMS for operations, scientists have implemented common database operations inside libraries such as dplyr or Pandas. These libraries, however, are not efficient and lead to memory overloads and poor performance. MADlib, an Apache incubation project, aims to integrate machine learning modules into a database system (Hellerstein, Ré and Schoppmann). While the library calls the functions in SQL language and runs on the same environment as the database, it still creates a standalone program process that fetches data like a database client. Moreover, the data fetching process affects the overall performance of the machine learning modules. The paper discusses three approaches to connect analytical tools with database management systems (Figure 1).

One approach is to embed the database inside the client program (Figure 1.c). The most popular embedded database is SQLite. It is a row-major database designed for transactional workloads, which contributes to poor performance when dealing with large and comprehensive data. As a solution, the authors created MonetDBLite, an open-source embedded database based on column-major MonetDB. MonetDBLite/MonetDB performs significantly better when executing analytical queries compared to SQLite. The column-major architecture and zero-copy semantics of MonetDB ensure that the data sharing between the client and the server has a constant low cost.

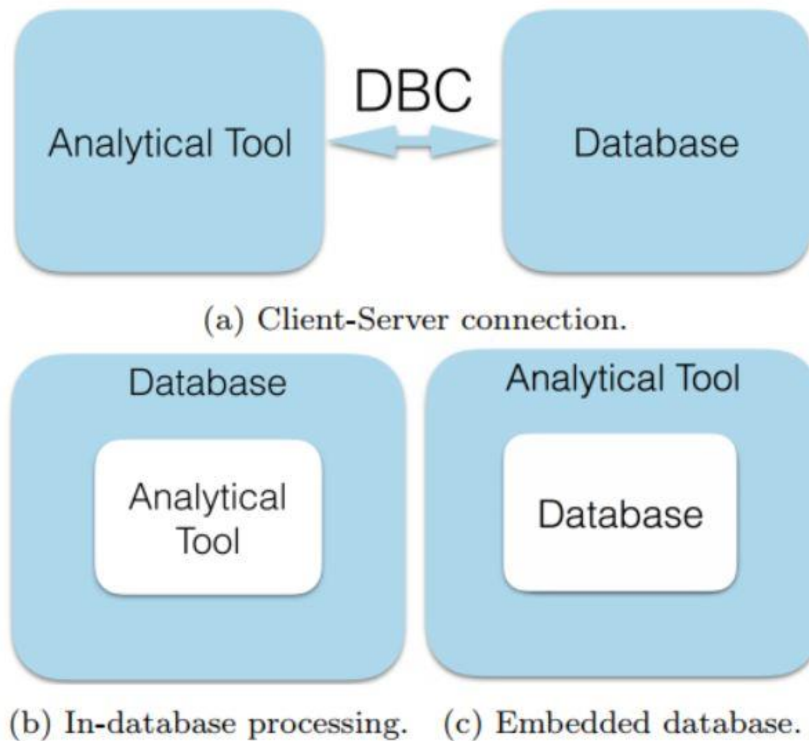

Figure 1: Different ways to implement analytical tools with the database. Adapted from (Raasveldt)

Client-Server Connection is the standard way of combining a program with relational database management systems. The analytical tools (the client) run on different machines or the same machines as a separate process. The client can issue queries to the database (the server). The server will compute the answer to the query and transfer the results to the client. The type of database is primarily irrelevant for this approach. It can easily integrate and use Standard ODBC and JDBC connectors. Existing pipelines for loading flat files are also easy to replace with channels for database loading. This approach is the significant bottleneck we get when serializing a large amount of data. According to the surveys conducted for this paper, most RDBMS are not optimized for high-volume data export due to OLTP optimized client protocols. Introducing a new client protocol achieves some efficiency. However, the transfer of data still requires a significant amount of time.

In-Database Processing completes the analysis inside the database server, which avoids the cost of exporting data. The current popular approach of writing user-defined functions or user-defined aggregates in procedural languages is inefficient since it requires significant

rewrites of existential analytical pipelines. Writing user-defined functions in these languages requires in-depth knowledge of database internals, as well. This paper introduces MonetDB/Python User Defined Functions (UDF) inside MonetDB (open-source, column-major DBMS) to perform in-database analytics easier. They are written in Python and utilize vectorized Processing.

PostgreSQL

PostgreSQL, also known as Postgres, is an open-source, object-oriented database management system. It extends the SQL language with many features capable of handling the most complicated workloads. It has millions of users, people, and organizations alike due to a proven reputation for reliability, extensibility, data integrity, and development (About PostgreSQL).

Sponsored by Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc, the POSTGRES project started at the University of California Berkley in 1986. In 1994, a new team re-released the POSTGRES package as Postgres95, with many significant changes. Postgres95 focused on identifying and understanding the existing problems in the POSTGRES code. The original code was cut down by a fourth. The query language PostQuel was changed out with SQL. A new interactive SQL query program, psql, was introduced. In 1996, with a new name of PostgreSQL, a new version focusing on changing system capabilities and features were released. The work continues in all areas, as Postgres has been in active development for over thirty years. Postgres is an accepted alias for PostgreSQL, and we will be using these two names interchangeably throughout this paper (A Brief History of PostgreSQL).

PostgreSQL is a free and open-source database system that runs on all major operating systems, including macOS, Windows, BSD, Solaris, and Linux. It is the go-to approach to process and store large amounts of data and hand complex queries. Postgres is an object-relational database management system that can handle both object-oriented and relational database functionality. It is ACID-compliant and very SQL-rich. It conforms to SQL standard

whenever possible. Version 13, released in 2020, supports 170 out of 179 mandatory SQL standard features.

Postgres is highly customizable and extensible. It allows the creation of user-defined functions in different procedural languages, besides SQL and C. SQL can call on the functions created in these procedural languages. PL/Tcl, PL/Perl, PL/Python, and PL/pgSQL are all procedural languages included in the standard PostgreSQL distributions. Many other procedural languages are available via third parties. Our project utilizes SQL and PL/pgSQL. The latter allows users to perform more complex operations with more procedural control than SQL. It enables users to use control structures such as loops. PostgreSQL has excellent performance with a sophisticated optimizer and advanced indexing. It has many multi-versions concurrency control (MVCC) features, as it enables other users to read and write information to a database simultaneously. Postgres supports parallel queries and has declarative partitioning. (About PostgreSQL).

MADlib

MADlib is an analytics API designed to work with PostgreSQL and Greenplum Database by Apache Software Foundation. It "operates on the data locally in-database, instead of moving data between multiple runtime environments unnecessarily" (Apache, MADlib). It offers compiled binaries that support Ubuntu 18.04 with PostgreSQL 11 and 12 and CentOS with Greenplum. At the same time, it also shares the source codes for the public to compile their own binaries or make changes they desired.

After examining the source code (Apache, Github - apache/madlib), we noticed that MADlib is still a module mainly composed of C++ and C, instead of native languages supported by PostgreSQL. Unlike what MADlib has stated, this API still copies the data needed to a new process from the select statements. As a result, MADlib did not distinguish much from other analytics tools such as Scikit Learn.

Methodology

Our project explores four algorithms: K-Means, Naive Bayes, Logistic Regression, and Random Forest. This section discusses the various ways we implemented each algorithm. For PostgreSQL implementations, our project utilizes PostgreSQL 12 with pgAdmin platform. We wrote all inputs in SQL and PL/pgSQL. To compare the performances between existing machine learning solutions, we have also installed MADlib with PostgreSQL. MADlib is the best-known solution to integrating analytics into databases. We have not found any other similar solutions. Before the comparison, we also tested the PostgreSQL implementation's correctness by comparing the output data to the MADlib counterpart given the same inputs. If not further mentioned, the Postgres implementation generates the same results as the MADlib counterpart. We will test each implementation with three data sizes: one-hundred thousand, five-hundred thousand, and one million data points.

K-Means

K-Means is a standard and straightforward iterative algorithm. It is part of unsupervised machine learning, meaning it works on uncategorized and ungrouped data. The algorithm works by finding a certain number of homogeneous groups(clusters) inside the data. The K-Means classifier has applications in insurance fraud detections, document classifications, rideshare analytics, and much more. The inputs for the algorithm are the dataset and the number of clusters(K) needed. It has two outputs: the centroids of each cluster and new labels for the data (Trevino).

K-Means algorithm has four main steps

1. Initialize the K clusters.
2. Calculate the distance between data points and cluster centroids.
3. Assign each data point to the closest cluster.
4. Update centroids.

Repeat steps 2-4 until the algorithm converges.

Centroids are the defining factor of clusters. For the first step, initial centroids of K clusters are either selected from the input data or randomly generated. Then, the algorithm calculates the distance between each data point and centroid. The distance is the squared Euclidean distance formula.

$$d^2(p,q) = (p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_n - q_n)^2$$

For the third step, K-Means assigns each data point to the closest cluster. In the last step, every cluster's centroid is re-calculated by averaging the distance between the centroid data points. The data assignment to centroid update (steps 2-4) makes one iteration of the K-Means algorithm. The algorithm iterates until convergence. K-Means converges when it reaches the pre-defined number of iterations, or no data points change clusters. The results of K-Means are local optimums. The algorithm needs to be run more than once with different, or randomized, initial centroids to get more accurate outcomes.

Our project looked at two implementations of the K-Means algorithm. We explored both recursive queries and Postgres loop functionality to implement the K-Means algorithm. Common Table Expression (CTE) is a temporary result set, which another SQL statement can reference. It is a convenient way of managing complicated queries. CTEs can be recursive, as they do not have the limitations of SELECT statements.

Recursion

Multidimensional Clustering Using K-Means in PostgreSQL is a recursive implementation of the K-Means algorithm. We implemented this algorithm on our machines. We modified some functions have for better clarity.

This approach chooses cluster centroids values from the data. It creates a function to calculate the Euclidean distance between different data points.

```
create or replace function sqr_euclid_dis(x1 float, y1 float, x2 float, y2 float)
returns float as $$
begin
    return (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2);
end;
$$ language plpgsql;
```

Figure 2: Squared Euclidean Distance function

Then, it implements the recursive CTE. It follows all the steps outlined in the section above. Instead of calculating the distance between data points and cluster centroids, assigning each data point to the closest cluster, and then updating centroids, these steps are nested within each other. However, the order is the same. Figure 3. shows the start of the recursive CTE. On line 3, a new iteration starts. Lines 6-10 calculate the distances between the points and the centroids, assigning them to the closest cluster. Line 5 extracts the x and y dimensions of each data point inside the first cluster. Line 4 calls for the recalculation of the x dimension of the first cluster centroid.

```
1   with recursive kmeans(iter, x1, y1, x2, y2) as (
2       select 1, * from initial_clusters union all
3       select kmeans.iter + 1, (
4           select avg(x)   from (
5               select s.x, s.y from (
6                   select x, y,
7                   case when sqr_euclid_dis(x, y, kmeans.x1, kmeans.y1)
8                       < sqr_euclid_dis(x, y, kmeans.x2, kmeans.y2)
9                   then 1 else 2 end as cluster_id
10                  from data
11              ) s
12          where cluster_id = 1
13          ) l
14      ),
```

Figure 3: Beginning of Recursive CTE for K-Means

Loop

To implement another version of the K-Means algorithm, we explored Programming the K-means Clustering Algorithm in SQL by Carlos Ordonez. Our project followed the steps described in this document to get a while loop-based K-Means implementation. This paper outlines a simple six-step framework:

1. Setup.
2. Initialization.
3. Computing Euclidean distances.
4. Assign centroids.

5. Update clustering results.
6. Track Progress.

Repeat steps 3-6 until K-Means converges.

We implemented the steps outlined in this paper in PostgreSQL. Steps 3-5 are inside the while loop. The full code can be seen in Appendix D.

MADlib

MADlib offers a simple SQL call to initiate the learning model (Figure 4). It asks the user to specify the input table and the corresponding columns and the number of centroids to calculate. Optional arguments include the name of the function to use to calculate the distance, the name of the function used to determine centroids, the maximum number of iterations, and the minimum fraction of centroids.

```
kmeans_random( rel_source,
               expr_point,
               k,
               fn_dist,
               agg_centroid,
               max_num_iterations,
               min_frac_reassigned
             )
```

Figure 4: The function call of K-Means in MADlib

MADlib's implementation of K-Means utilized the algorithm in a loop form and was implemented in C++. The source code is available at MADlib's git repository (Apache, Github - apache/madlib).

Naive Bayes

Naive Bayes is a supervised machine learning algorithm, a classification technique based on Baye's Theorem, a probabilistic inference algorithm. Naive Bayes has a multitude of applications. These include text classification, specifically spam filtering, controlling

autonomous vehicles, medical diagnosis, and product recommendations (Wickramasinghe and Kalutarage).

Baye's Theorem uses the training dataset to find the posterior probability for each condition. Then it calculates the likelihood of a new event given its conditions. The formula for Baye's Theorem is as follows

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

It calculates the probability of event A, given that B is True. The underlying assumption of Naive Bayes is that the predictors are fully independent. Thus, P(A) and P(B) are independent probabilities of A and B. Naive Bayes is an easily scalable fast algorithm because it does not handle multidimensional dependencies.

Naïve Bayes algorithm has three simple steps:

1. Create the frequency table from the dataset.
2. Create the probability table of each category.
3. Calculate the posterior probability of each class by applying Baye's Theorem.

Postgres

We implemented Naive Bayes in Postgres. Our training dataset is for text classification. Building the model starts with calculating the independent probability of each category. In our example, the likelihood that a word is part of a scam email would be that probability. We build the model based on the data available, following the steps outlined in the section above. Our approach combines the first two steps and makes one table with each word's frequencies and probabilities (Appendix E).

The classifier is a different function. Our classifier takes either a word or text. It returns the probability of the input based on the model we build before. The table returned will have the text, its possible classification categories, and the likelihood of that classification P(category|word).

```
1  CREATE OR REPLACE FUNCTION classify(_word text)
2  RETURNS TABLE(
3      word text,
4      category text,
5      pOfCatGivenWord float
6  ) AS $$
```

```
20     RETURN QUERY     select * from pOfCategoryByWord
21      where _word = pOfCategoryByWord.word order by pOfCatGivenWord;
22     END;
23     $$ LANGUAGE plpgsql;
```

Figure 5: Part of the Classifier function for Naïve Bayes

MADlib

MADlib implementation of Naïve Bayes has similar algorithms to the module we implemented. It is, according to the documentation, functional but "still in early-stage development" (Apache, MADlib: Naive Bayes Classification). The function call is shown in Figure 6.

```
create_nb_prepared_data_tables ( trainingSource,
                                 trainingClassColumn,
                                 trainingAttrColumn,
                                 numAttrs,
                                 featureProbsName,
                                 classPriorsName
                               )
```

Figure 6: The function call of Naïve Bayes in MADlib

The first three variables determine what the training dataset would be. The fourth one determines the trainingAttrColumn attributes-array that corresponds to numeric attributes. The last two variables determine the output tables, respectively.

Logistic Regression

Logistic regression models the probabilities for classification problems with two possible outcomes (Interpretable Machine Learning). It is one of the most straightforward supervised analytic methods for classification problems.

The logistic function is defined as:

$$\text{logistic}(\eta) = \frac{1}{1 + exp(-\eta)}$$

To fit the data into a logistic model as close as possible, we used the following formula: bn(t+1) = bn(t) + learning_rate * (y(t) - yhat(t)) * yhat(t) * (1 – yhat(t)). In this formula, bn(t+1) is the coefficient for the corresponding attributes. The learning rate determines how fast the model should fit. y(t) is the expected value, and the yhat(t) is the predicted value according to the current model. After several iterations determined by the user, the model will generate an array of coefficients, with each index corresponding to the attributes.

Postgres

We implemented logistic regression in PostgreSQL with PL/pgSQL language. The implementation includes two functions. The first one is to perform the prediction. It takes in the inputs and coefficients and performs the corresponding prediction according to the formula (Figure 7). The code is available in Appendix B.

The second one is to perform the training. It takes in the table to be trained with, the number of iterations, and learning rate. The number of attributes in the training dataset is as most sixteen, with the last column being the independent variable. The training function sets all the starting coefficients as zero. It runs the algorithm described above for the designated number of iterations and puts the coefficients results into a new table (Figure 8).

```
-- FUNCTION: public.logRegPredict(double precision[], double precision[])

-- DROP FUNCTION public."logRegPredict"(double precision[], double precision[]);

CREATE OR REPLACE FUNCTION public."logRegPredict"(
    inputs double precision[],
    coefficients double precision[])
    RETURNS double precision
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
declare
    yPredict double precision := coefficients[1];
    i integer := 1;
    loopLimit integer := array_length(inputs, 1);
begin
    loop
        exit when i = loopLimit;
        yPredict := yPredict + coefficients[i + 1] * inputs[i];
        i := i + 1;
    end loop;
    return (1.0 / (1.0 + exp(-yPredict)));
end;
$BODY$;

ALTER FUNCTION public."logRegPredict"(double precision[], double precision[])
    OWNER TO postgres;

COMMENT ON FUNCTION public."logRegPredict"(double precision[], double precision[])
    IS 'Predicts a logistic regression with given coef and arguments';
```

Figure 7: The function of logistic regression prediction we implemented in PL/pgSQL.

```
    learning_rate double precision,
    num_epochs integer)
    RETURNS double precision[]
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
declare
    epoch integer := 1;
    rowLength integer;
    coefficients double precision[];
    row_array double precision[];
    yPredict double precision;
    oneRow public."logRegTest"%rowType;
    i integer := 1;
    errorRate double precision;
begin
    coefficients := array[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
    loop
        exit when epoch > num_epochs;
        --raise notice 'epoch %', epoch;
        for oneRow in
            execute format ('select * from %s', table_train)
        loop
            i := 1;
            row_array := array[oneRow.x1, oneRow.x2, oneRow.x3, oneRow.x4, oneRow.x5, oneRow.x6, oneRow.x7, oneRow.x8, oneRow.x9, oneRow.x10, oneRow.x11,oneRow.y];
            yPredict := public."logRegPredict"(row_array, coefficients);
            --raise notice 'yPredict = %', yPredict;
            errorRate := row_array[array_length(row_array, 1)] - yPredict;
            coefficients[1] := coefficients[1] + learning_rate * errorRate * yPredict * (1.0 - yPredict);
            loop
                exit when i = array_length(row_array, 1);
                coefficients[i + 1] := coefficients[i+1] + learning_rate * errorRate * yPredict * (1.0 - yPredict) * row_array[i];
                i := i + 1;
            end loop;
        end loop;
        --raise notice 'coef = %', coefficients;
        epoch := epoch + 1;
    end loop;
    return coefficients;
end;
$BODY$;
```

Figure 8: The function of logistic regression we implemented in PL/pgSQL.

MADlib

The MADlib implementation of logistic regression utilizes similar algorithms. It has the function call described in Figure 9.

```
logregr_train( source_table,
               out_table,
               dependent_varname,
               independent_varname,
               grouping_cols,
               max_iter,
               optimizer,
               tolerance,
               verbose
             )
```

Figure 9: The function call of logistic regression training from MADlib

The source_table, depenedent_varname, grouping_cols, and independent_varname variables determine the training dataset. The out_table variable determines where to output the results. The max_iter variable specifies the maximum iteration. The tolerance variable determines what error the training can tolerate and thus stop the training.

Random Forest

Random forest, or random decision forest, is an ensemble supervised learning method for classification, regression, and other tasks (Ho). It is designed to correct for traditional decision trees' habit of overfitting to the training dataset. It utilizes the general technique of feature bagging to build multiple models from samples of the training dataset.

The random forest can be broken down into two steps:

1. Determine the most cost-effective way to split the dataset and generate the trees, respectively.
2. Use the tree created to create a set of predictions and return the set.

Postgres

We constructed two functions to implement random forest. The first function is used to predict the value according to the trained random forest models. It takes in the dataset to be trained and the number of features desired and generates the tress that would have the lowest cost according to the Gini index in the form of a new table (See Figure 10 for part of the code). The full functions are available in Appendix C.

```
declare
    b_index integer := 999;
    b_value integer := 999;
    b_score integer := 999;
    cur_features integer[];
    idx integer;
begin
    cur_features:= array[0,0,0,0,0,0];
    loop
        exit when cur_features = n_features;
        idx = random(6);
        if cur_features[idx] == 0 then
        cur_features[idx] := 1;
```

Figure 10: Part of the split function we implemented in PL/pgSQL.

The second function takes in the divided table and predicts the results based on the table values. Since the array within PL/pgSQL is fixed-sized, the functions are limited to 6 features.

```
1 declare
2     predictions integer[];
3     rowNumber integer;
4     idx integer;
5     oneRow public."randomTrees"%rowType;
6 begin
7     predictions := array[0,0,0,0,0,0];
8     idx := 0;
9     for oneRow in
10        execute format ('select * from %s', table_tree)
11    loop
```

Figure 11: Part of the prediction function we implemented in PL/pgSQL.

MADlib

We did not examine what algorithm MADlib used to implement. The function call is available in Figure 12.

```
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,
             null_handling_params,
             verbose,
             sample_ratio
             )
```

Figure 12: The function call of random forest from MADlib.

The training_table_name, id_col_name, depenedet_variable variables decide the training dataset. The num_trees and num_random_features determine the maximum number of trees the function

can generate and the number of features to randomly select at each split. There are other optional variables available, such as importance, num_permutations, etc.

Analysis and Results

Installation, Usage, and Syntax

Postgres

For our project, we decided to work with PostgreSQL 12. The newest version is PostgreSQL 13, which was released in September of 2020, a month after the start of our project. Most Linux platforms have PostgreSQL included in the package management. We also had to install it on Windows 10. Installation of PostgreSQL was effortless and straightforward. Packages and Installers and source code are available on the PostgreSQL website (https://www.postgresql.org/download/windows/). We utilized the ready-to-go installer for Windows. After downloading the installer and running it, we simply had to follow the on-screen directions. The installer included a PostgreSQL server, pgAdmin (a graphical administration and development tool), and StackBuilder (a package manager for installing additional tools and drivers) (PostgreSQL 12.6 Documentation). PgAdmin made it very convenient to create databases and run queries. We used this tool to run all queries, functions, and procedures for our project.

MADlib

MADlib requires a careful procedure to install correctly. It requires either Ubuntu 18.04, CentOS, or MacOS prior with Intel processors (Apache, MADlib). Furthermore, each version listed on the website needs a specific PostgreSQL version to match. For example, MADlib 1.17 supports Postgres 11 and 12, while MADlib 1.16 only supports Postgres 10 and 11. Since we chose to work with PostgreSQL 12, MADlib 1.17 became our obvious choice. After the installation, the user needs to input certain command line prompts to link MADlib with the PostgreSQL installation. See appendix A for the command line prompts.

K-Means

The recursive implementation of the K-Means algorithm is not scalable. This example works very well with two clusters and only two dimensions. Increasing the number of clusters or dimensions is possible but requires substantial manual changes. The problem of scalability is within the recursion: the data assignment and centroid recalculation. Data assignment is comparing the Euclidean distance between each point to each centroid. For two clusters with two dimensions, data assignment is one comparison per dimension of the centroid (Appendix 13), four comparisons overall. It is four comparisons per dimension of the centroid, twenty-four comparisons overall for three clusters with two dimensions. Changing the way initial centroids are selected also had no significant difference.

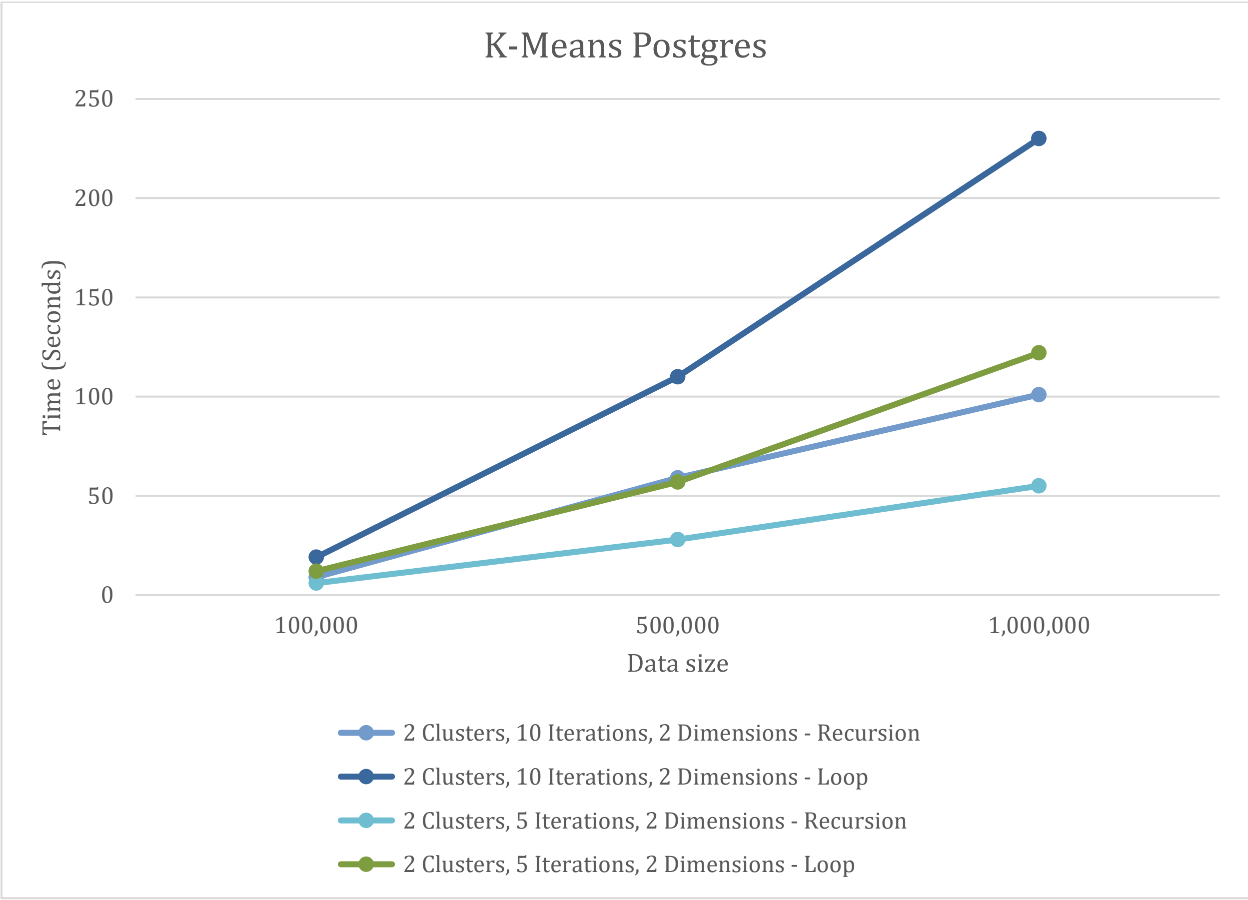

Figure 13: Runtimes of K-Means implementations in PostgreSQL

While loop implementation of the K-Means is scalable, compared to the recursive implementation. Modifying the number of clusters requires a one-line code change. All the

functions can handle a more considerable number of clusters. However, the recursion is faster. For million data points with two-dimensional data and two clusters, the recursive implementation does ten iterations in hundred seconds, while the loop implementation takes two hundred and thirty seconds on average (Figure 13). This implementation is slower than the recursive approach because it creates more data and tables to handle any number of clusters or dimensions. Loop version seems to be the best Postgres implementation since it is scalable and is more robust with the type of data it can handle.

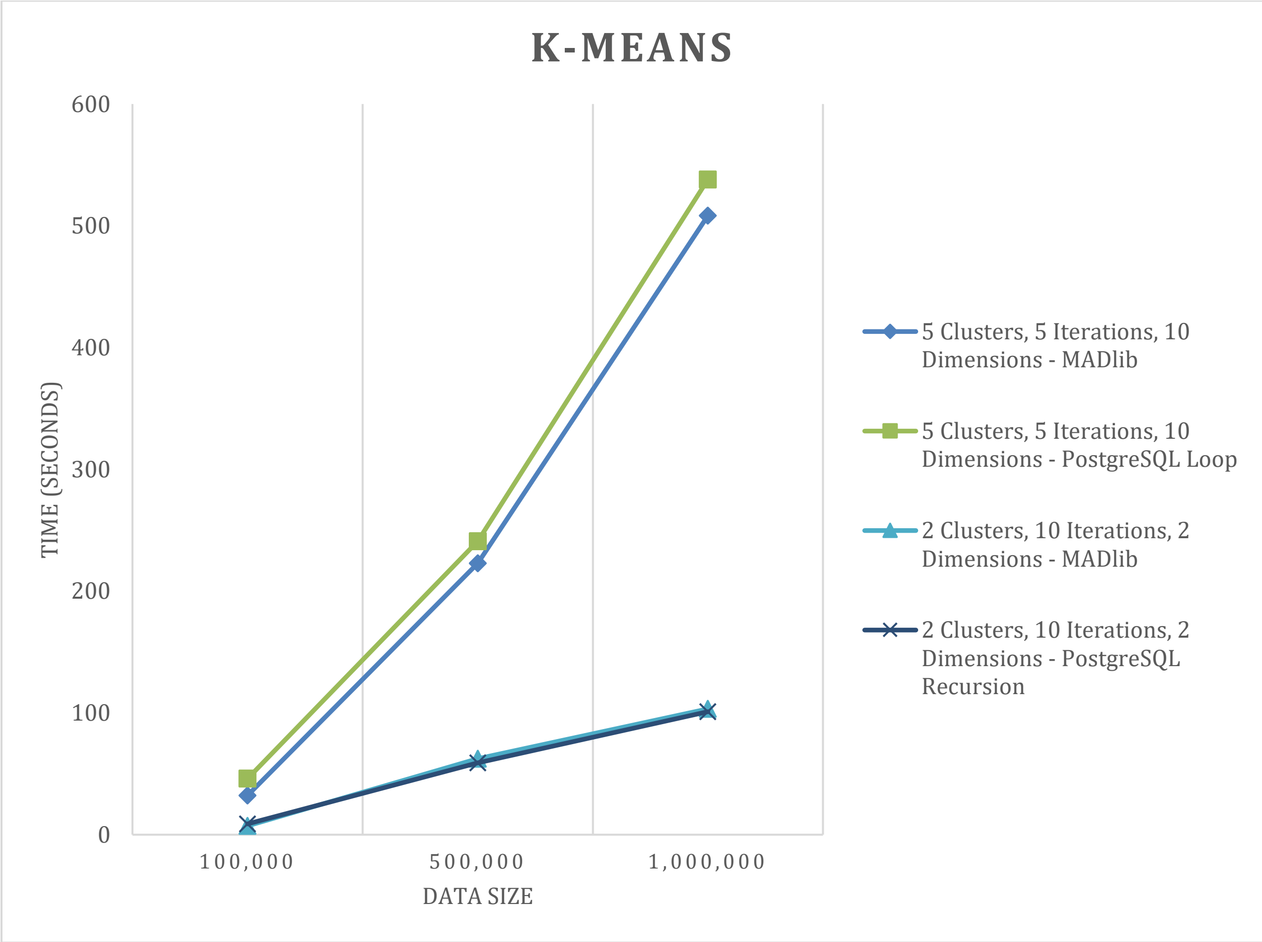

Figure 14: Runtimes of K-Means implementations in PostgreSQL and MADlib

Comparing K-Means Postgres implementations with MADlib results showed interesting results. MADlib had the same order of growth as the PostgreSQL algorithms. MADlib barely outperformed the Loop version of the algorithm and matched the recursive implementation.

Naïve Bayes

Comparing the PostgreSQL and MADlib implementations of Naïve Bayes, Figure 15 shows the running times of each with different Data Sizes. MADlib substantially outperforms Postgres. MADlib also has a slower time growth rate compared to Postgres.

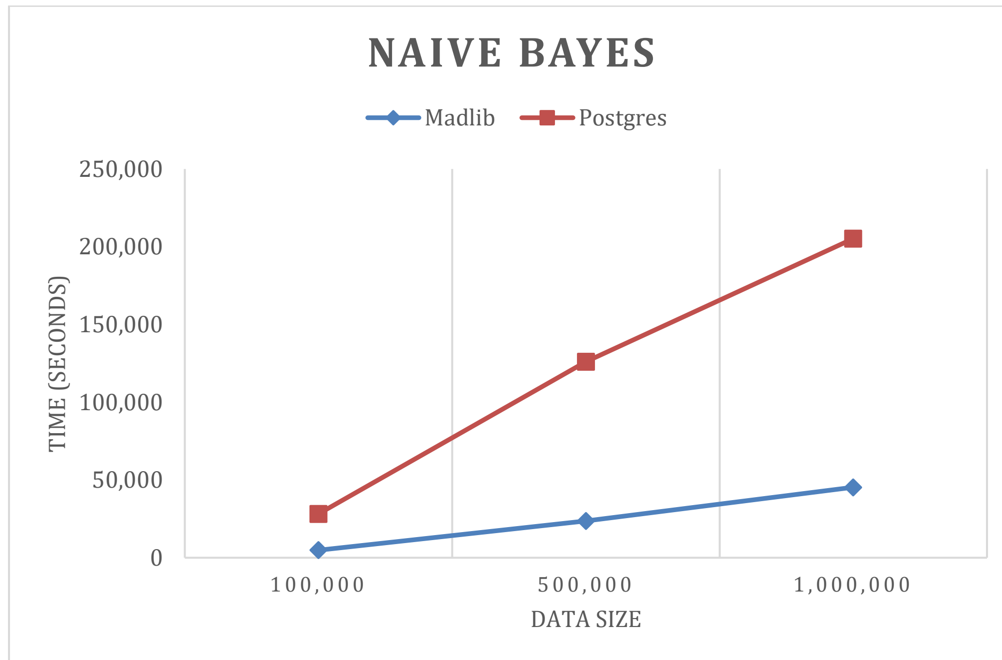

Figure 15: Implementations of Naïve Bayes in MADlib and PostgreSQL

Logistic Regression

Since the columns' size cannot be read in PL/pgSQL, the PostgreSQL implementation has a limitation on scalability. The version we implemented supports up to 15 dependent variables. Before the user can utilize the function, they need to put the dataset into a table where the dependent variables are in columns named x1, x2... x15, and the independent variable in the column named y. Furthermore, since PL/pgSQL cannot cast one row of the data into an array, we had to hard code the casting process as a substitution.

We tested the methods with the Framingham data. Framingham dataset has 3656 valid entries. We took 1000 of them and made copies of the 1000 rows to generate the datasets with a size of 100 thousand, 1000 thousand, and 1 million. The test results between Postgres and MADlib are shown in Figure 16.

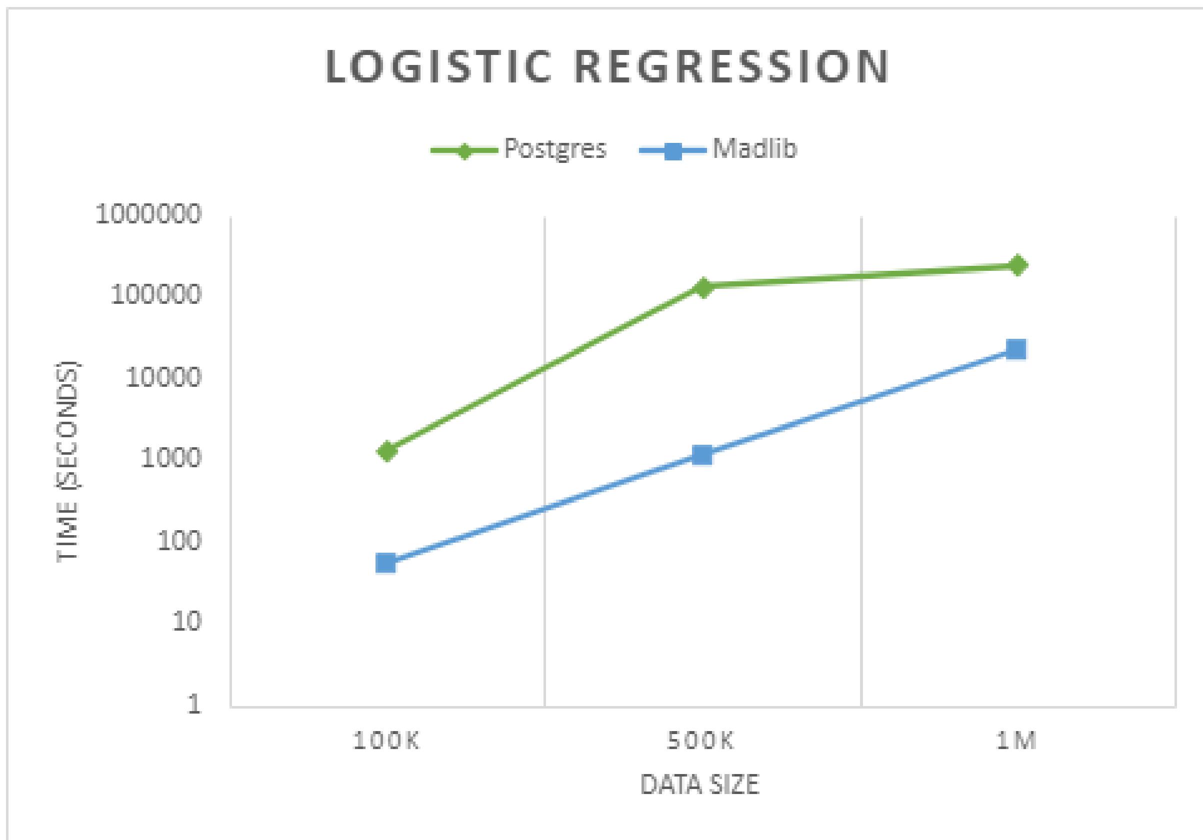

Figure 16: The time it took to run the training on the given data size for logistic regression.

The Postgres version has the same order of growth as the MADlib version. However, it is proportionally slower than the MADlib version. On average, the MADlib version of logistic regression only spent 1/10 of the time the Postgres counterpart did. This data shows that the implementation is efficient in terms of the macro algorithm. However, due to the repeated selection statements in the code, and the hard code process of transferring data from table to array, the PL/pgSQL version suffers in performance constantly.

Random Forest

Like the situation of Logistic Regression, we had to limit the number of features supported in the Postgres version. We chose to limit the number of features available to 6 features. The number chosen here is arbitrary for the simplicity of implementation. To compare the performance, we took the sonar dataset and picked the first 200 rows. Then, we copied the

first 200 rows so that we had a dataset of 100 thousand, 500 thousand, and 1 million. The performance results are shown in Figure 17.

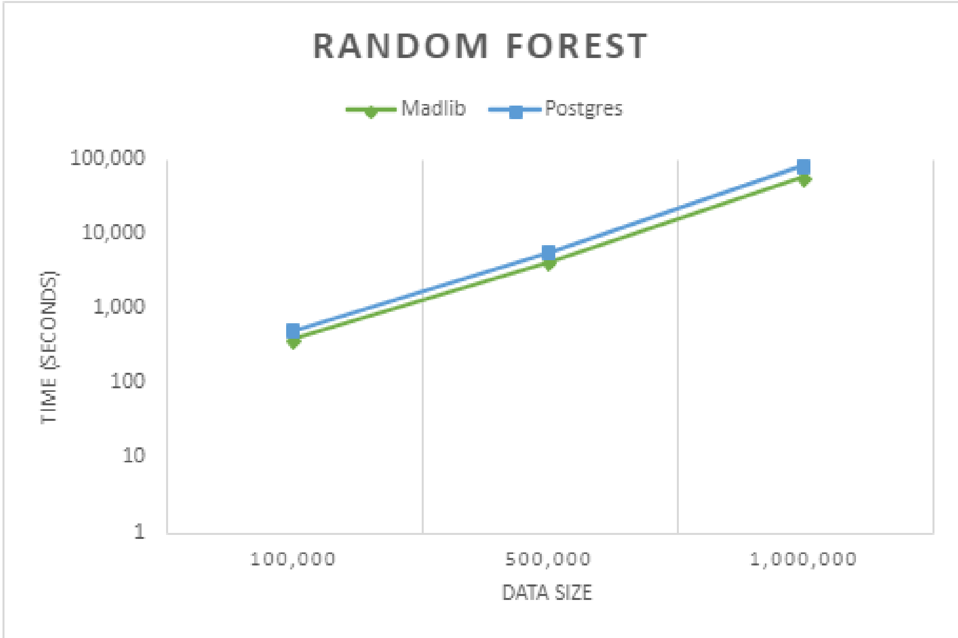

Figure 17: The time it took to run the training on the given data size for the random forest.

According to our test, Postgres has a minor performance penalty compared to that of MADlib. When we implemented the functions, we tried to limit the selection statements in for loop. The results came back as an improvement compared to that of logistic regression. Even if the current version still has scalability issues, it has a similar performance to the MADlib version when the data dimensions are the same.

Conclusions

The disconnect between analytics tools and database systems causes redundancy of performed actions. Thus, our project implemented different data analytics methods inside the database systems. We compared four algorithm runtimes as MADlib and PostgreSQL applications.

Through our project, we confirmed that many popular analytics methods could be implemented in PostgreSQL. However, the scalability and the resulting performance penalty became noticeable in two of the methods we chose to implement. We guessed that the penalty was caused by the frequent selection statement used in the code, along with the translation from PL/pgSQL to binaries. For future research opportunities, we recommend testing the scale of the selection statement penalty and explore possible methods to optimize the penalty. We also recommend exploring the feasibility of other popular analytical methods.

Acknowledgments

We would like to thank our project advisor, Mohamed Eltabakh, for his support and guidance.

Works Cited

A Brief History of PostgreSQL. n.d. https://www.postgresql.org/docs/12/history.html. 03 March 2021.

About PostgreSQL. n.d. https://www.postgresql.org/about/. 05 03 2021.

Apache. *Github - apache/madlib*. 2021. 18 March 2021. <https://github.com/apache/madlib>.

—. *MADlib*. 2021. Website. 18 March 2021. <http://madlib.apache.org/product.html>.

—. *MADlib: Naive Bayes Classification*. 2021. 18 March 2021. <http://madlib.apache.org/docs/latest/group__grp__bayes.html>.

Columbus, Louis. *Global State Of Enterprise Analytics, 2018*. 06 August 2018. <https://www.forbes.com/sites/louiscolumbus/2018/08/08/global-state-of-enterprise-analytics-2018/?sh=7b3dcab36361>.

Hellerstein, Joseph M, et al. "The MADlib Analytics Libraryor MAD Skills, the SQL." Technical Report. 2012. Document.

Ho, Tin Kam. "Random Decision Forests." *Proceedings of 3rd International Conference on Document Analysis and Recognition* (1995): 278-282.

Ordonez, Carlos. "Programming the K-means clustering algorithm in SQL." *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (2004): 823-828. Web.

Panoho, Kale. *The Age Of Analytics And The Importance Of Data Quality*. 01 October 2019. https://www.forbes.com/sites/forbesagencycouncil/2019/10/01/the-age-of-analytics-and-the-importance-of-data-quality/?sh=15bae94e5c3c. 02 03 2021.

PostgreSQL 12.6 Documentation. n.d. https://www.postgresql.org/docs/12/index.html. 28 September 2020.

Raasveldt, Mark. "Integrating Analytics with Relational Databases." *PhD@VLDB 2018*. n.d.

Sisense Data Team. *Multidimensional Clustering Using K-Means in Postgres SQL*. 17 December 2015. https://www.sisense.com/blog/multi-dimensional-clustering-using-k-means-postgres/. 23 October 2020.

Trevino, Andrea. *Introduction to K-means Clustering*. 06 December 2016. https://blogs.oracle.com/datascience/introduction-to-k-means-clustering. 26 October 2020.

Wickramasinghe, Indika and Harsha Kalutarage. "Naive Bayes: applications, variations and vulnerabilities: a review of literature with code snippets for implementation." *Soft Computing* (2021): 25, 2277-2293. Document.

Appendices

Appendix A

Command Prompt to Install MADlib on Ubuntu 18.04

```
rpm -i apache-madlib-1.17.0-bin-Linux.deb
which psql postgres pg_config
psql -c 'select version()'
/usr/local/madlib/bin/madpack -s madlib –p postgres install
/usr/local/madlib/bin/madpack -s madlib –p postgres install-check
```

Appendix B

Code for logistic regression

```
-- FUNCTION: public.logRegPredict(double precision[], double precision[])

-- DROP FUNCTION public."logRegPredict"(double precision[], double precision[]);

CREATE OR REPLACE FUNCTION public."logRegPredict"(
    inputs double precision[],
    coefficients double precision[])
    RETURNS double precision
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
declare
    yPredict double precision := coefficients[1];
    i integer := 1;
    loopLimit integer := array_length(inputs, 1);
begin
    loop
        exit when i = loopLimit;
        yPredict := yPredict + coefficients[i + 1] * inputs[i];
        i := i + 1;
    end loop;
    return (1.0 / (1.0 + exp(-yPredict)));
end;
$BODY$;

ALTER FUNCTION public."logRegPredict"(double precision[], double precision[])
    OWNER TO postgres;

COMMENT ON FUNCTION public."logRegPredict"(double precision[], double precision[])
    IS 'Predicts a logistic regression with given coef and arguments';
```

```
-- FUNCTION: public.logisticTrain(regclass, double precision, integer)

-- DROP FUNCTION public."logisticTrain"(regclass, double precision, integer);

CREATE OR REPLACE FUNCTION public."logisticTrain"(
    table_train regclass,
    learning_rate double precision,
    num_epochs integer)
    RETURNS double precision[]
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
declare
    epoch integer := 1;
    rowLength integer;
    coefficients double precision[];
    row_array double precision[];
    yPredict double precision;
    oneRow public."logRegTest"%rowType;
    i integer := 1;
    errorRate double precision;
begin
    coefficients := array[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
    loop
        exit when epoch > num_epochs;
        --raise notice 'epoch %', epoch;
        for oneRow in
            execute format ('select * from %s', table_train)
        loop
            i := 1;
            row_array := array[oneRow.x1, oneRow.x2, oneRow.x3, oneRow.x4, oneRow.x5, oneRow.x6, oneRow.x7, oneRow.x8, oneRow.x9, oneRow.x10, oneRow.x11,oneRow.y];
            yPredict := public."logRegPredict"(row_array, coefficients);
            --raise notice 'yPredict = %', yPredict;
            errorRate := row_array[array_length(row_array, 1)] - yPredict;
            coefficients[1] := coefficients[1] + learning_rate * errorRate * yPredict * (1.0 - yPredict);
            loop
                exit when i = array_length(row_array, 1);
                coefficients[i + 1] := coefficients[i+1] + learning_rate * errorRate * yPredict * (1.0 - yPredict) * row_array[i];
                i := i + 1;
            end loop;
        end loop;
        --raise notice 'coef = %', coefficients;
        epoch := epoch + 1;
    end loop;
    return coefficients;
end;
$BODY$;
```

Appendix C

Code for random forest

```
-- FUNCTION: public.getSplit(regclass, integer)

-- DROP FUNCTION public."getSplit"(regclass, integer)

CREATE OR REPLACE FUNCTION public."getSplit"(
    table_train regclass,
    n_features integer)
    RETURNS integer[]
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
declare
    b_index integer := 999;
    b_value integer := 999;
    b_score integer := 999;
    cur_features integer[];
    idx integer;
begin
    cur_features:= array[0,0,0,0,0,0];
    loop
        exit when cur_features = n_features;
        idx = random(6);
        if cur_features[idx] == 0 then
        cur_features[idx] := 1;
        end if;
    end loop;
    return array[b_index, b_value, b_score];
end;
$BODY$;

ALTER FUNCTION public."getSplit"(regclass, integer)
    OWNER TO postgres;
```

```
-- FUNCTION: public.randomForestPredict(regclass, integer)

-- DROP FUNCTION public."randomForestPredict"(regclass, integer);

CREATE OR REPLACE FUNCTION public."randomForestPredict"(
    table_tree regclass,
    "row" integer)
    RETURNS double precision[]
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE PARALLEL UNSAFE
AS $BODY$
declare
    predictions integer[];
    rowNumber integer;
    idx integer;
    oneRow public."randomTrees"%rowType;
begin
    predictions := array[0,0,0,0,0,0];
    idx := 0;
    for oneRow in
        execute format ('select * from %s', table_tree)
    loop
        if oneRow.left = 1 then
            predictions[idx] := 1;
        else
            predictions[idx] := -1;
        end if;
        idx := idx + 1;
    end loop;
    return predictions;
end;
$BODY$;

ALTER FUNCTION public."randomForestPredict"(regclass, integer)
    OWNER TO postgres;
```

Appendix D

Implementation for K-Means Loop

```
1  DROP EXTENSION tablefunc;
2  CREATE EXTENSION tablefunc;
3  /*
4  STEP 1
5  Setup. Create,indexandpopulateworkingtables.
6   Y which has all of the dimensions
7   http://www2.cs.uh.edu/~ordonez/pdfwww/w-2004-KDD-sqlkm.pdf
8  */
9  CREATE OR REPLACE FUNCTION numClusters()
10   RETURNS integer AS $numClusters$
11  begin
12     return 2;
13     end;
14 $numClusters$ language plpgsql;
15
16 CREATE OR REPLACE FUNCTION numDimensions()
17   RETURNS integer AS $numDimensions$
18  begin
19     return 2;
20     end;
21  $numDimensions$ language plpgsql;
22
23 CREATE OR REPLACE FUNCTION numIter()
24   RETURNS integer AS $numIter$
25  begin
26     return 10;
27     end;
28  $numIter$ language plpgsql;
29
30
31 --Cut out importing into Y
32 DROP TABLE YH;
33 DROP TABLE YV;
34 DROP TABLE W CASCADE;
35 DROP TABLE R;
36 DROP TABLE C;
37 DROP TABLE MODEL;
38 DROP TABLE YD;
39 DROP TABLE YNN;
40
41 CREATE TABLE YH AS
42 SELECT sum(1) over (rows unbounded preceding) as i,* FROM data;
43
44 CREATE TABLE YV(
45   i int,
46   l int, -- dimension
47   val Float,
48   Primary Key(i, l)
49 );
50
51 INSERT INTO YV
52 SELECT i, l, val
53 FROM YH
54 JOIN LATERAL (VALUES(1, YH.Y1), (2, YH.y2) , (3, YH.y3), (4, YH.y4))
55     s(l, val) ON true;
```

```
56
57 CREATE TABLE W(
58     j int Primary Key, -- cluster
59     w float -- count of points per cluster
60 );
61
62 CREATE TABLE R(
63     l int, -- nsion()*
64     j int, -- cluster
65     val float, --variance
66     PRIMARY KEY(l, j)
67 );
68
69 /*
70 STEP 2
71  Initialization. InitializeC
72   - Create Center Horizontal
73   - Then pivot
74 */
75 CREATE TABLE C(
76     l int, -- dimension d
77   j int, --  number of clusters k
78   val Float,--avg of point in the same cluster
79     PRIMARY KEY(l, j)
80 );
81
82 truncate c;
83 do $$
84 declare
85    lim integer:= numDimensions()*numClusters();
86    r record;
87 begin
88     FOR r IN SELECT * FROM YV ORDER BY i LIMIT lim LOOP
89         INSERT INTO C VALUES(r.l, r.i, r.val);
90
91     END LOOP;
92 end$$;
93 select * from c order by j;
94
95 Create table model(
96     d int,
97     k int,
98     n int,
99     iteration int,
100    avg_q float,
101    --diff_avg_q float,
102    PRIMARY KEY (d, k) --deleted d
103 );
104
105 INSERT INTO model (d, k, iteration) (select l, j, 0 from C);
106
107 Create table yd(
108            i int, -- number
109            j int, --cluster
110            distance float,
111            primary key(i ,j)
112        );
```

```
113
114 CREATE TABLE YNN(
115             i int primary key, --n
116             j int -- cluster
117         );
118
119 do $$
120 declare
121     iter int :=1;
122 begin
123     while iter <= numIter()
124     loop
125         raise notice 'iteration % Start', iter;
126         /*
127         STEP 3
128         */
129         TRUNCATE YD;
130         insert into YD
131         Select i, j, sum((YV.val-C.val)^2)
132         FROM YV, C
133         Where YV.l = C.l
134         GROUP BY i,j ;
135         /*
136         STEP 4
137         */
138         TRUNCATE YNN;
139         INSERT INTO YNN
140         SELECT YD.i, YD.j
141         FROM YD, (SELECT i, min(distance) AS minDIST
142                 FROM YD GROUP BY i) YMIND
143         WHERE YD.i = YMIND.i and YD.distance = YMIND.minDist;
144         /*
145         STEP 5
146         UpdateW; CandR
147         */
148         TRUNCATE w;
149         INSERT INTO W
150             SELECT j, count(*)
151             FROM YNN GROUP BY j;
152         UPDATE MODEL SET n = W.w, iteration = iteration +1 FROM W WHERE k = W.j;
153         TRUNCATE C;
154         INSERT INTO C
155             SELECT l, j, avg(YV.val)
156             FROM YV, YNN
157             WHERE YV.i = YNN.i
158             GROUP BY l,j;
159         TRUNCATE R;
160         -- calculating variances per cluster
161         INSERT INTO R
162             SELECT C.l, C.j, avg((YV.val-C.val)^2)
163             FROM C, YV, YNN
164             WHERE YV.i = YNN.i and YV.l = C.l and YNN.j = C.j
165             GROUP BY C.l, C.j;
166         UPDATE model
167         SET avg_q = C.val FROM C WHERE k = C.j AND d = C.l;
```

```
168
169         raise notice 'iteration % End', iter;
170         iter := iter + 1;
171 end loop;
172 end $$;
173
174 /*
175 STEP 6
176 Updatetableto track K-meansprogress
177 */
178
179 select * from model order by k, d;
180
```

Appendix E

Code for Naïve Bayes Model

```
1
2   -- Independent Probability of a class
3   DROP TABLE CATEGORYPROBABILITY;
4   CREATE TABLE categoryProbability(
5       category text,
6       probability float,
7       wordCount int,
8       primary key(category)
9   );
10
11  INSERT INTO CATEGORYPROBABILITY
12      SELECt category,count(word)*1.0/(select count(*)  from categoryTable)
13    as indpCategoryP,   count(word)  from categoryTable group by category;
14
15  --Probability of a word given a category
16  DROP TABLE PROBABILITY;
17  CREATE TABLE probability(
18      txt text,
19      probability float,
20      primary key(txt)
21  );
22
```

```
32  select *, (1.0+ (select count(*)
33  from categoryTable where category=A.category AND word = A.word))/(
34  (select sum(wordcount)  from categoryProbability)+(Select wordcount from
35    categoryProbability where category=A.category))  as pOfCatGivenWord
36  from (select word, categoryProbability.category from categoryTable cross join
37    categoryProbability)  AS A group by word, category;
```