



In-Database 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:

**Justin Amevor
Junyu Lyn
Tyrone Patterson
Bailey Schmidt**

Advisor:

**Mohamed Eltabakh
March 8th 2020**

Abstract

In-database analytics is a technique to process data directly inside of a database. The opportunity to compute analytical data directly inside of a database is a high value venture for large data warehouses. This investigation examined the value of using Apache MADlib for analytical operations versus developing functions and procedures for the same purpose using PostgreSQL and PL/SQL, without external dependencies such as Python (used by MADlib). The functions examined in this investigation were K-Nearest Neighbors, Floyd Warshall Algorithm, Logistic Regression, Matrix Factorization, Naive Bayes, and K-Means Clustering, each function was chosen arbitrarily. After implementation of each function was complete, performance testing was conducted to examine the accuracy and runtime of each PL/SQL function against its MADlib counterpart. Results showed that the initial overhead for setup and installation of MADlib is far from user friendly and lacking in up-to-date documentation; for usability however using MADlib can be significantly advantageous for thorough data processing compared to direct implementation. Manually implementing analytical functions could be efficient for smaller queries however as the sample size increases, MADlib handled queries significantly more efficiently.

Table of Contents

Abstract	2
Table of Contents	3
Table of Figures	5
Table of Tables	6
Acknowledgments	7
Authorship	8
Executive Summary	9
1. Introduction	1
2. Background	2
Client-Server Connection	2
Embedded Database	3
In-database Processing	4
MADlib Library	4
Overview	4
Technical Requirements	6
Installation	6
Procedural Language for SQL (PL/SQL)	7
Analytics Functions	8
K-Mean Clustering	8
K-Nearest Neighbors (KNN)	11
Floyd Warshall Algorithm	12
Naive Bayes Classification	13
Matrix Factorization	15
3. Methodology	17
Functions vs Stored Procedures	17
Datasets	17
Function Development	18
K-Mean Clustering	18
K-Nearest Neighbors	19
PostGIS	20
Floyd Warshall Algorithm	21

	4
Naive Bayes Classification	22
Matrix Factorization	24
MADlib Benchmarking -- <i>Testing against MADlib Analytics Library</i>	25
Testing Methodology K-Means Clustering	26
Testing Methodology K-Nearest Neighbors	26
Testing Methodology Floyd Warshall Algorithm	26
Testing Methodology Naive Bayes	27
Testing Methodology Matrix Factorization	30
4. Results and Recommendations	31
Results K-Means Clustering	31
Results K Nearest Neighbors	33
MADlib	33
Results Floyd Warshall Algorithm	34
Results Naive Bayes	40
Results Matrix Factorization	49
Recommendations	51
5. Conclusion	52
Bibliography	53
Appendix	54

Table of Figures

2.1 Communication between a client and server.....	2
2.2 Interaction between an embedded application and database.....	3
2.3 The different ways to integrate analytics with relational databases.....	5
2.4 MADlib architecture.....	6
2.5 PL/SQL architecture.....	8
2.6.1 K-Mean clustering main functions.....	10
2.7 KNN search for a point q.....	12
2.8 Example of Floyd Warshall Algorithm.....	13
3.1 KNN Query	21
3.2 Gaussian Probability Function.....	24
3.3 Query for comparing accuracy of Naive Bayes Training Model(s).....	28
3.4 Query for comparing accuracy of Naive Bayes Training Model(s).....	28
3.4.1 Create, Populate, and Query Formatted SQL Tables for use in MADlib Functions.....	30
3.4.2 Result of Select Statement from ml_iris_train.....	30
4.1.1 K-Mean query and cluster results from PL/SQL.....	32
4.1.2 K-Mean Clustering centroid points results when the cluster is set to 2.....	32
4.2 KNN Query using MADlib.....	33
4.3 KNN Query using PL/SQL.....	34
4.3.1 Raw Runtime Data for Naive Bayes Algorithms.....	42
4.3.2 Averaged Runtime Data for Naive Bayes Algorithms.....	42
4.4.1 Naive Bayes Training Function Execution Time Graph (in milliseconds).....	43
4.4.2 Naive Bayes Classify Function Execution Time Graph (in milliseconds).....	32
4.5.1 Visual Comparison of Small Set Training Accuracy for Naive Bayes Algorithms.....	43
4.5.2 Visual Comparison of Medium Set Training Accuracy for Naive Bayes Algorithms.....	44
4.6.1 Naive Bayes Classification Accuracy Results for Large Data Set (Iris).....	45
4.6.2 Naive Bayes Classification Accuracy Results for Medium Data Set (Frogs).....	45
4.6.3 Naive Bayes Classification Accuracy Results for Large Data Set (Online Retail).....	46
4.7 Matrix Factorization query using PL/SQL.....	50
4.8 Charting performance of MADlib Functions vs. PL/SQL.....	52

Table of Tables

3.1 Record and Attribute Count for Naive Bayes Algorithm Testing.....	27
4.1.1 K-Mean Clustering Runtime for MADlib and PL/SQL in Milliseconds.....	32
4.2 KNN Runtime for MADlib and PL/SQL in Milliseconds.....	35
4.3.1 Naive Bayes Training Accuracy Results for Small Data Set (Iris).....	46
4.3.2 Naive Bayes Training Accuracy Results for Medium Data Set (Frogs).....	46
4.3.3 Naive Bayes Training Accuracy Results for Large Data Set (Online Retail).....	47
4.4 Matrix Factorization Runtime for MADlib and PL/SQL in Milliseconds.....	51

Acknowledgments

The team would like to thank our advisor Mohamed Eltabakh for his support and guidance throughout the project. We would also like to thank Worcester Polytechnic Institute's computer science department for their support and acceptance of this investigation.

Authorship

Section	Written by	Edited by
Abstract	Justin	Bailey
Executive Summary	Justin	Tyrone
Introduction	Justin	Tyrone
Background	All	All
Client Server Connection	Justin & Junyu	Tyrone
Embedded Database	Justin	Junyu
In-Database Processing	Junyu	Justin
MADlib Overview	Justin	Junyu
Technical Requirements	Bailey	Justin
Installation	Bailey	Justin
Procedural Language for SQL (PL/SQL)	Justin	Bailey
K-Mean Clustering/Methodology & Results	Junyu	Bailey
K-Nearest Neighbors (KNN)/Methodology & Results	Justin	Bailey
Floyd Warshall Algorithm/Methodology & Results	Justin	Junyu
Naive Bayes Classification/Methodology & Results	Bailey	Junyu
Matrix Factorization/Methodology & Results	Tyrone	Bailey
Methodology	All	All
Problem Formulation	Justin	Bailey
Functions Vs. Stored Procedures	Bailey & Justin	Tyrone
Datasets	Justin	Bailey
MADlib Benchmarking	Bailey & Justin	Tyrone
Recommendations	Justin	Bailey
Conclusion	Justin	Bailey

Executive Summary

Problem Statement

The ability to analyze large database warehouses is a valuable tool for data scientists and businesses to employ. Database analytics provide valuable information that can be used to support business intelligence. However, analyzing large databases can be complex and very difficult especially depending on the architecture of the database itself. Apache MADlib is an open-source library that was built to help people easily use analytical functions within the Postgres or Greenplum database management systems. **This report investigates the practicality of using MADlib for In-database analytics as a substitute for manually creating functions for analytical purposes as well as investigating the accuracy and efficiency of the user made functions against MADlib.**

Objective

Implement the K-Means Clustering, K-Nearest Neighbors, Floyd Warshall Algorithm, Logistic Regression, Matrix Factorization, Naive Bayes and MADlib functions using PL/SQL. Determine if the MADlib functions are more efficient and accurate than the PL/SQL analytical functions.

Goals

In order to focus our investigation, the team developed 3 succinct goals to help achieve our objective.

1. Implement the MADlib functions in PL/SQL.
2. Test the MADlib functions vs PL/SQL functions for accuracy and efficiency.
3. Provide recommendations on performing In-database analytics.

Methodology

The team began the investigation by taking time to research and understand the MADlib functions. Team members took approximately $\frac{1}{3}$ of the project time to grasp the logic of the algorithms, learn the syntax of the PL/SQL language, and install the MADlib software and PGAdmin tool for development. After successfully installing MADlib and PGAdmin, the team then started writing the functions in Python to aid in understanding of the behavior of the algorithms. For each function, the team focused on developing their own optimal implementation to be comparable or better than its respective MADlib counterpart. As a result of the optimization efforts, the team tried to utilize the PL/SQL language to improve the runtime of the functions.

Once development was complete the team chooses three data sets to conduct base testing. Those datasets were the Iris dataset (500 attributes), The Anuran Calls dataset (7195 attributes) and the Online Retail dataset (78095 attributes). The MADlib and team made functions were both tested using the three datasets. After calculating the average of the test trials, the team used the results to deliver the recommendations of the project's investigation.

Conclusion

PL/SQL is a powerful and efficient tool that exploits block statements allowing for the capability to perform powerful processes. The language provides a powerful platform to develop efficient queries. MADlib provides a simpler way to perform accurate analytical functions on sample sizes of any size. MADlib is a new open source software that is still developing but offers valuable functionality to data scientists and common analytical persons. The expansion of MADlib's functionality is a notable stock to pay attention to during its continued development. One who wishes to conduct their own analytical processing should consider MADlib a reliable, powerful, and accurate tool if a proper installation can be complete.

1. Introduction

Performing analytical operations inside of a database (or In-Database Analytics) is a powerful method used in various data warehousing infrastructures. In-database processing allows analytical functions to be used inside of a database, without the need to transform data and transport it between the database and outside applications. Open-source libraries such as Apache MADlib offer powerful analytical functions within a database, but configuring and using similar libraries may be difficult for the average user. PL/SQL is a language designed to allow users to easily select and manipulate data from a database without having extensive knowledge of programming concepts.

This report focuses on the team's investigation of the utilization of the MADlib library versus PL/SQL functions within the PostgreSQL database management system. Eight functions, defined in MADlib, were chosen arbitrarily to be examined against a similar PL/SQL implementation to compare their runtimes and accuracy (if applicable) in order to determine which set of functions produced more accurate and efficient results. The functions discussed in this report are K-Nearest Neighbors, Floyd Warshall Algorithm, Logistic Regression, Matrix Factorization, Naive Bayes, and K-Means Clustering. The process of implementing each function and testing its benchmarking is detailed in the subsequent sections.

2. Background

There are three main ways data-intensive analytics tools can be integrated with traditional relational data management systems (RDBMS); establishing a client-server connection, performing analysis inside databases (also known as in-database processing), and embedding a database within an analytical tool. Each implementation has various constraints that can affect usability and performance, depending on the characteristics of the specific RDBMS and configuration. Users' use cases will determine the advantages and disadvantages of each method.

Client-Server Connection

An analytical tool can be combined with an RDBMS through a client-server connection. This method allows the database and analytical tool to be two separate entities. After being authenticated, the data in the database can be exported from the server to the client, where any processing will take place. **Figure 1.1** (Raasveldt 2018) shows the process of the server processing queries using a client server connection.

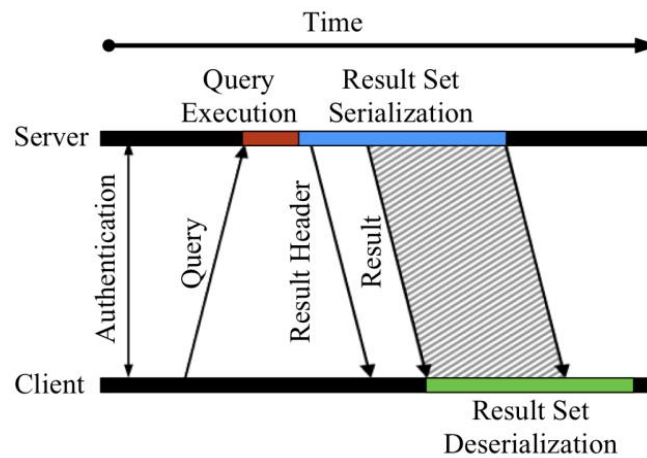


Figure 2.1 Communication between a client and server

Establishing a client-server connection can be problematic when a large amount of data needs to be serialized and transferred to the client. A large dataset may not fit in the clients' memory, which may result in a failure of the data transfer. Efficient storage of data in the database can help improve the transfer time to the client. This method is most optimally used for analyzing small amounts of data. Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC) interfaces can connect to almost any database and have a simple implementation process. Using a client-server connection allows for easy integration into existing pipelines by loading only the necessary files instead of entire datasets (Raasveldt, 2018).

Embedded Database

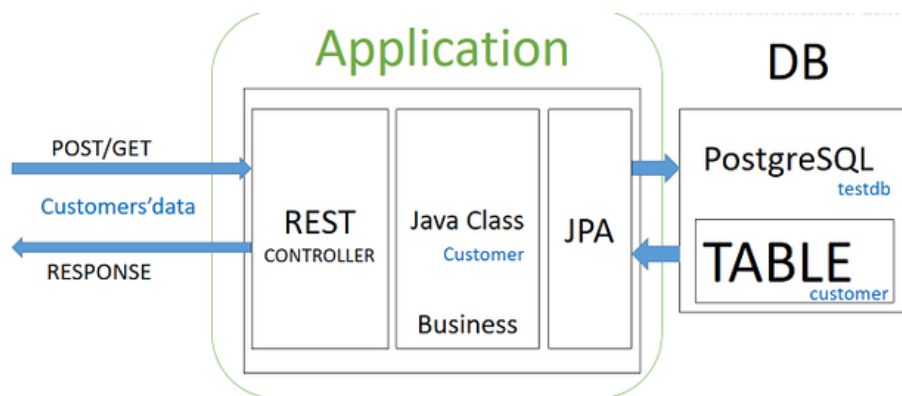


Figure 2.2 Interaction between an embedded application and database

A database can also be embedded inside of an analytical tool client program. This method does not require a user to have a running database server and costs less time for users to install, tune, and maintain. SQLite is a commonly used embedded database however, it is designed for transactional workloads and is not the best for analytical purposes. MonetDB is another open-source embedded database that is more suited for analytics and handling large datasets

(Raasveldt, 2018). **Figure 1.2** shows the interaction between the database and embedded application. Queries can be sent from the database and processed within the embedded application by utilizing Postgres tables and other functions.

In-database Processing

In-database processing is the method of performing analytics inside of a database server, which omits the process of exporting the data from a database. Block (b) in **Figure 2.3** shows the theory behind in-database processing. It is difficult to express most data analysis, data mining, and classification operators in SQL because of the limited amount of scalar functions and their complexity to implement on large data sets. The current solution to this problem is implementing the analytical functions using user-defined functions in procedural programming languages. This method requires a significant amount of manual labor from the user to rewrite existing analytical functions in SQL. Users will need to have a significant amount of knowledge about the database's internal and execution model. Because of the complexity of this approach, our team aims to develop a more efficient solution to the overhead of this approach (Raasveldt, 2018).

MADlib Library

Overview

Apache MADlib is an open-source library for scalable in-database analytics. It provides several data-parallel implementations of mathematical, statistical, data science, and machine learning algorithms. MADlib can be configured with PostgreSQL to analyze large datasets.

MADlib is useful for various analytical functions like clustering, regression models, graph analysis, and more.

The architecture of MADlib can be described in three main components as illustrated in **Figure 2.3**. The three main parts of MADlib are Python driver functions, C++ implementation functions, and a C++ abstraction layer. The python functions act as the main entry point from the user input and flow control of the algorithms. The second layer is a collection of C++ functions

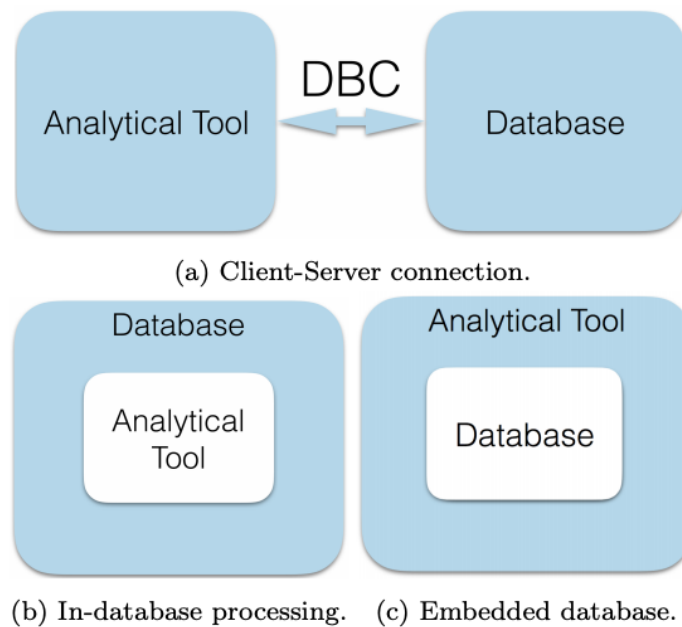


Figure 2.3 The different ways to integrate analytics with relational databases

and aggregates needed for certain functions. They are implemented in C++ to improve performance. Third is the C++ database abstraction layer that abstracts Postgres internal functions as well as initializes a user interface.

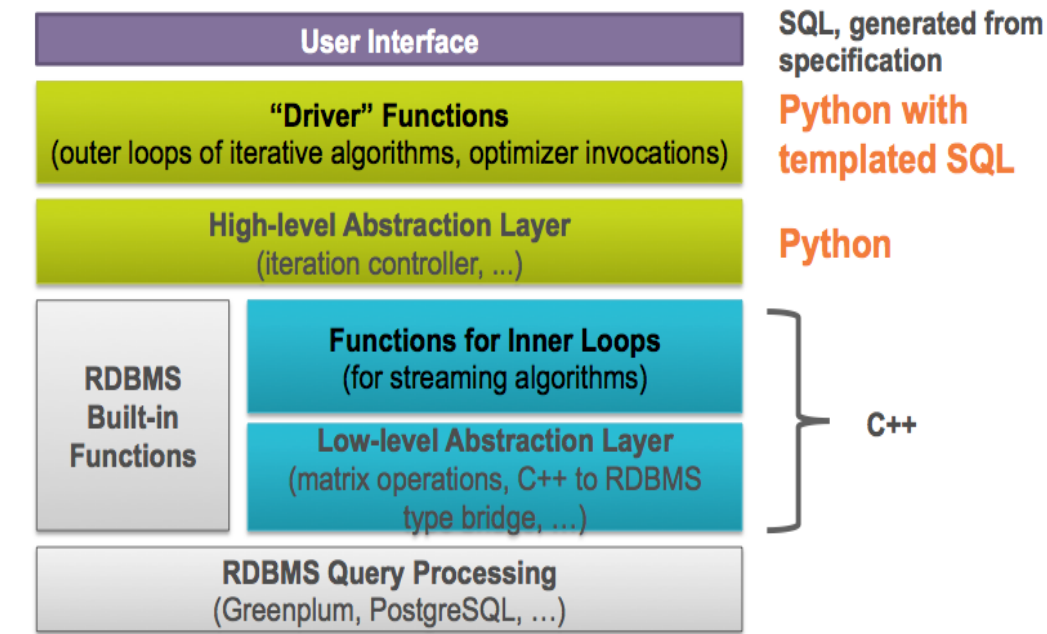


Figure 2.4 MADlib architecture

Technical Requirements

To successfully deploy MADlib, a machine must be running either macOS or CentOS; and must have PostgreSQL or Greenplum installed. The team ran macOS and PostgreSQL when installing and testing MADlib. It is extremely important to configure the PostgreSQL installation with Python. A machine will not be able to create the ppython (PL / Python) loadable language extension without proper configuration.

Installation

The successful deployment must be completed on Apple MacBooks running macOS Mojave, utilizing the default Python 2.7 installation. PostgreSQL can be installed via Homebrew [1], although it was necessary to tap a different repository [2] to properly configure the installation with Python [3]. Once Postgres is installed on a machine, the server will need to be started [4], and the MADlib binary [5] should be installed on the machine. Once this is complete,

MADlib can be installed into a specific database via the command line [6]. If the installation is successful the MADlib functions should be a part of the database's schema and therefore available for use. PgAdmin 4 [7] as a visual interface and testing tool that can be used for the deployment of MADlib. PgAdmin allows users to visually click through database schema and create/drop servers, databases, and tables using a built-in query editor [8].

Procedural Language for SQL (PL/SQL)

PL/SQL is an extension of SQL that allows for the query language to use block statements. Combining a database engine with block statements increases the processing speed and decreases traffic of the function. **Figure 2.4** details the architecture of PL/SQL into three main components. At the top level is the block where the actual code is written. The block statements are then processed in the engine direct SQL code is sent straight to interact with the database while the PL/SQL segment is processed. The dual engine utilization accounts for the high performance and optimization of the language (GeeksforGeeks, 2018).

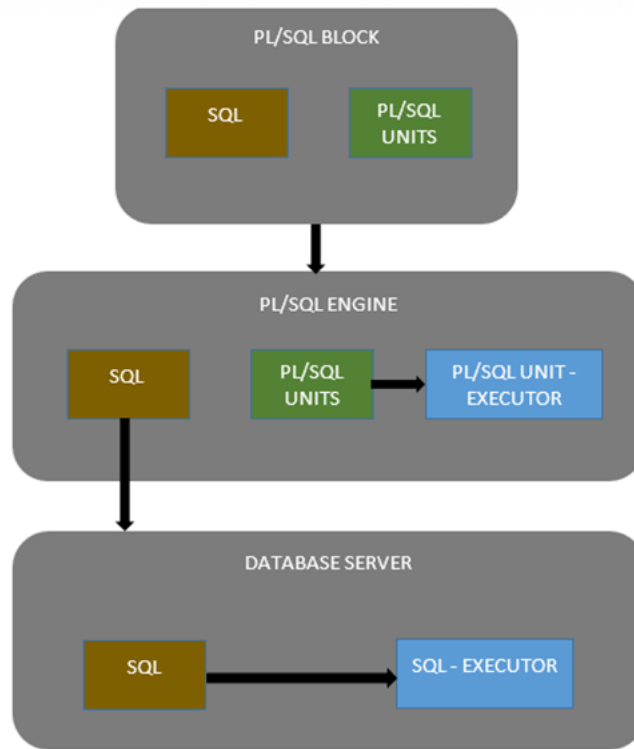


Figure 2.5 PL/SQL architecture

Analytics Functions

The team worked with various analytical functions in order to understand their logic, implementation, and efficiency. The functions were chosen from the MADlib library based on their potential to provide the team with useful analytical information.

K-Mean Clustering

K-Mean Clustering is one of the simplest and most popular unsupervised machine learning algorithms used in data mining. Given N points, the goal of the method is to position k centroids so that the sum of distances between each point and its closest centroids is minimized

(K-Mean Clustering, 2019). A cluster refers to a collection of aggregated data points by their closest centroids (i.e. their closest centroids are the same).

This method begins with deciding on the initial centroids and then performing iterative calculations to optimize the positions of the centroids. MADlib offers four ways to invoke the process: the random centroid seeding method, the k-means centroid seeding method, supplying an initial centroid set in a relation (identified by the *rel_initial_centroids* argument) and providing an initial centroid set as an array expression in the *initial_centroids* argument. In our specific case, K -- the number of clusters, is pre-defined. I choose to use the k++ centroid method as the seeding method for our initial demo (K-Mean Clustering, 2019). First, we assign each point to the cluster whose mean has the least squared Euclidean distance. Then we calculate the new means of the point in the new cluster. The repetition stops when the number of iterations reaches a certain number or the difference between the last two iterations has become smaller than a certain value. This method cannot make sure to find the most optimal clustering and its time complexity can be roughly calculated as $O(n^{(dk+1)})$, where n is the number of points to be clustered and k (number of clusters) and d (dimensions) are pre-defined. According to the MADlib document, with this method, the users are able to know the final calculated centroids positions and the number of iterations, etc.

The diagram shows the objective function for K-Mean clustering: $J = \sum_{j=1}^k \sum_{i=1}^n \|x_i^{(j)} - c_j\|^2$. Annotations include:

- An arrow from 'number of clusters' pointing to the upper limit k of the first summation.
- An arrow from 'number of cases' pointing to the upper limit n of the second summation.
- An arrow from 'case i ' pointing to the variable $x_i^{(j)}$.
- An arrow from 'centroid for cluster j ' pointing to the variable c_j .
- An arrow from 'objective function' pointing to the variable J .
- A bracket under the term $\|x_i^{(j)} - c_j\|^2$ is labeled 'Distance function'.

Figure 2.6.1 K-Mean clustering main functions

As the picture shown below, K-means algorithm is an iterative algorithm that tries to partition the dataset into K pre-defined distinct subgroups where each data point belongs to only one group. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid which is the arithmetic mean of all the data points that belong to that cluster. The less variation we have within clusters, the more similar the data points are within the same cluster. **Figure 2.4.1** is one example of a graphic version of the clustering result. Indicating that there are three different groups of data with three distinctive centroids points.

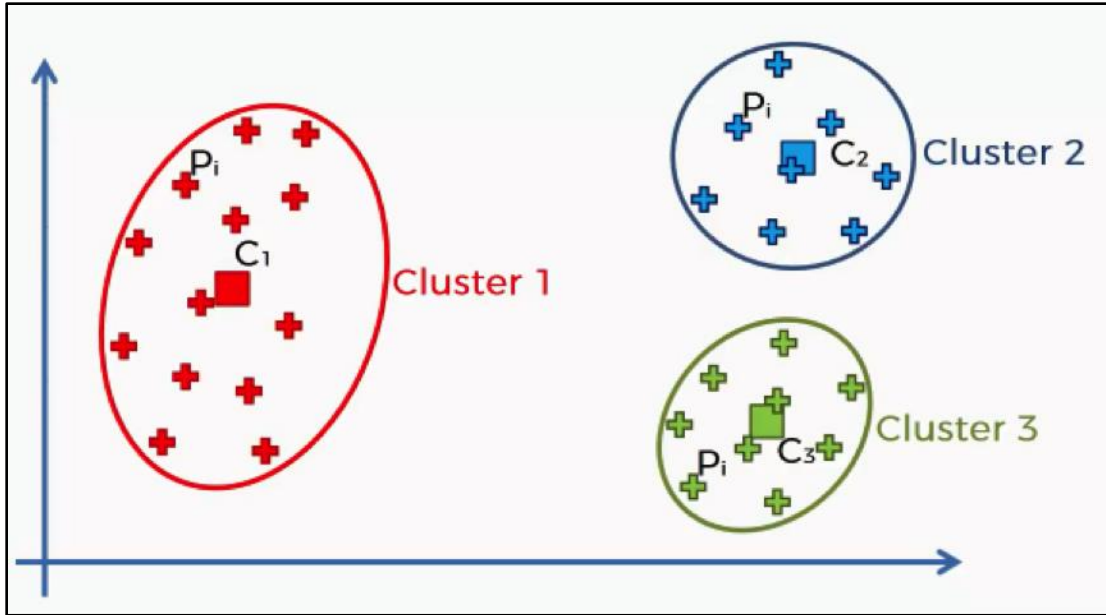


Figure 2.4.2 K-Mean clustering with three major clusters and three centroid points

K-Nearest Neighbors (KNN)

The KNN algorithm is a simple method in the supervised learning realm that is primarily used for classification. It is a function used for finding the (k) nearest points of a given point in the data set (K Nearest Neighbors, 2019). The data points are vectors in a multidimensional feature space and the number of dimensions the data has will affect the efficiency of the algorithm's execution. The KNN algorithm takes in a set of data called training data and the second set of data called testing data. The sets of data are then classified using the K value to approximate nearest neighbors while the value of K should be dependent on the data size. An optimal K can be determined by segregating the training and testing set from the original dataset and for the purpose of analytics, K remains constant across the different implementations of the algorithm.

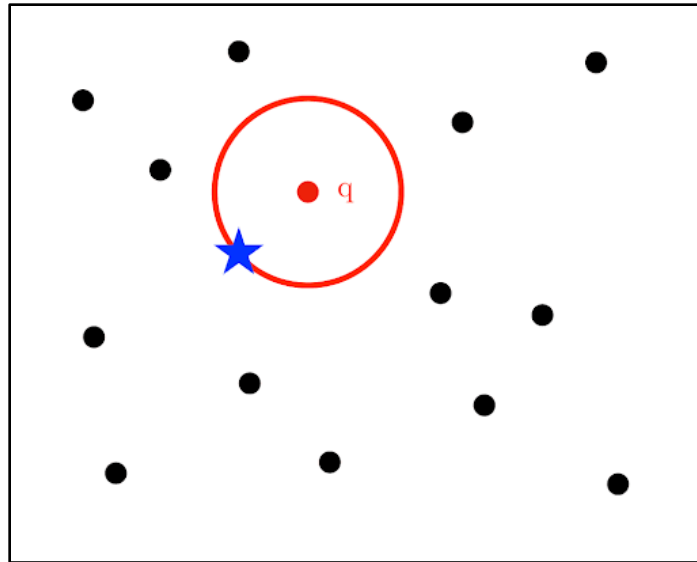


Figure 2.7 Illustrates a KNN search for a point q

KNN must scan through an entire dataset to see which points lie close to each other. The point proximity is often calculated using the Euclidean distance [10]. The prediction phase can speed up the implementation of appropriate data structures at runtime. A K-dimensional (KD) tree is a commonly used data-structure that the MADlib library utilizes to improve execution time. However, it can be costly to ensure the accuracy of the solution. The KD tree divides the training dataset sequentially into multiple regions that correspond to a leaf node of a tree. It will then look for the nearest neighbors in a subset of the regions contrary to the entire dataset. If there is a high number of dimensions, the accuracy of execution may suffer because a point may be in a different subset than expected. To construct an optimal KD tree, the number of dimensions should follow $N > 2^K$ where N is the number of dimensions (Bentley, 1975).

Floyd Warshall Algorithm

The Floyd Warshall algorithm is used to find the shortest path between all pairs of vertices inside of a weighted graph. The algorithm works for directed and undirected graphs with

positive or negative edges. However, it does not work with negative cycles. It takes in a matrix function that can be inputted as a table in PL/SQL and MADlib. After computing the shortest path over the table an updated cost matrix is outputted.

MADlib utilizes the principles of matrix multiplication to iterate through the algorithm. Because the path from every vertex must be found, it is very expensive. The worst-case run-time of the MADlib implementation is $O(V^2 * E)$ where V is the number of vertices and E is the number of edges. **Figure 2.6** shows the Floyd-Warshall algorithm traveling through each node in a graph.

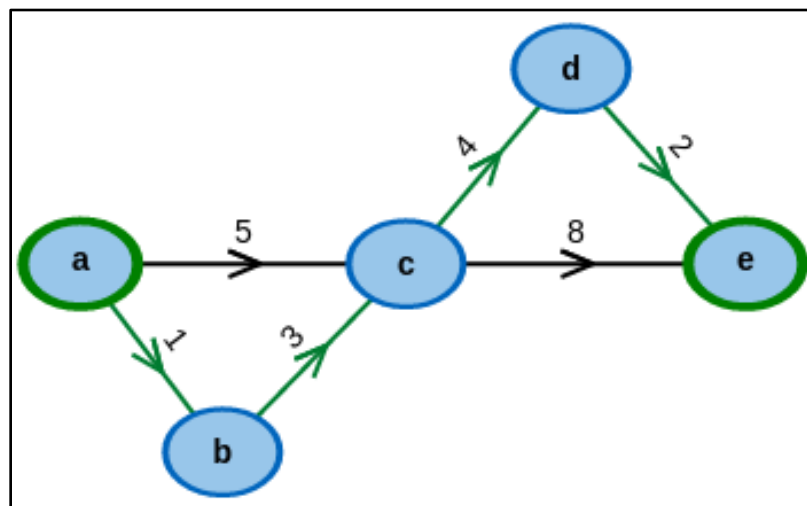


Figure 2.8 Example of Floyd Warshall Algorithm

Naive Bayes Classification

Naive Bayes Classification is an analytics training and classification mechanism developed from Bayes' Theorem (Joyce, 2003). Bayes Theorem presents an algorithm used to calculate conditional probabilities [9].

This concept is easier to understand in context, so imagine there exists a set of data that contains information about whether or not a person playing golf (Yes/No) on a given day. The dataset also includes information about the weather (Sunny/Cloudy/Raining), and the wind (None/Moderate/Windy). Using the initial dataset one could calculate the probabilities of each individual attribute ($P(\text{weather}) = \frac{1}{3}$, $P(\text{wind}) = \frac{1}{3}$, $P(\text{played golf}) = \frac{1}{2}$); and using Bayes' Theorem we could calculate the probability that this person will golf given that the weather is sunny \rightarrow [i.e. $P(\text{Yes} | \text{Sunny}) = P(\text{Sunny} | \text{Yes}) * P(\text{Yes}) / P(\text{Sunny})$].

Naive Bayes Classification is often used as a preliminary training algorithm in Machine Learning contexts. It's very useful for generally classifying new data based on a training dataset. Naive classification is naive because of the assumption of independence. This means that the algorithm used assumes that all the attributes (columns) used to classify the data are independent of one another -- i.e. each attribute has an equal and independent contribution to the outcome of the calculation (Naive Bayes Classifiers, 2019). While this is not usually true in most empirical data sets, the classification provided by Naive Bayes can still be useful, especially on large datasets. Naive Bayes Classification must also contend with the issue of *zero frequency*. Zero frequency is the issue that arises when an attribute present in a classifying dataset was not present in the training dataset. This causes the probability of the aforementioned attribute equaling zero in the calculation which can lead to inaccurate results and/or dividing by zero. This issue can be handled in practice using *Gaussian smoothing* or *Laplacian smoothing* (Cornell, 2018). These smoothing methods are just a means of estimating a function outcome to avoid zero frequency in the context of Naive Bayes.

MADlib contains multiple functions to train, classify, and calculate the probabilities of different classes within a dataset based on Bayes' Theorem. There are two implementations of

each function (training, classification, and probability) for datasets with purely categorical attributes, and for datasets with categorical and numerical attributes. The training functions allow users to pass datasets for training which pre-calculate feature probabilities and class priors. The classification and probability functions take the pre-calculated data, along with a new dataset to classify, and perform classification / calculate the probabilities of the new dataset based on the training set. MADlib also offers the ability to run the classification and probability functions ad-hoc, meaning no pre-calculations are done, where all calculations are done at the time of the function call; the MADlib documentation provides specific instructions on what needs to be passed into the function to compute these values successfully in an ad-hoc manner (Naive Bayes Classification, 2019).

Matrix Factorization

Matrix factorization refers to the process of dividing an input matrix into its factors, which can afterward be multiplied which results in the original input matrix. In the case of a machine learning function, the use case extends further than simply factoring a matrix. In our case, our input matrix is incomplete, meaning that every space in the matrix does not contain a number. Matrix factorization is used to guess what the factors of the input matrix are, given what numbers are given in the matrix, and from there we can guess what the original matrix would be if it were filled with numbers.

There are several different methods of performing matrix factorization. In this investigation singular value decomposition (SVD) was focused one. The SVD of an m by n matrix M results in matrices of the form $M=UV$ where U is an m by m matrix, Σ is a m by n rectangular diagonal matrix, V is an n by n matrix, and U and V are orthogonal. For the purpose of this project, this equation is simplified to $M = UV$. Removing the diagonal matrix from the

equation does not is okay in this instance because it simply acts as a scaler for either U or V. In order to find the values that make up the factors of M, we must find the optimal vectors for each matrix given the values that are given in M. This is done using stochastic gradient descent (SGD). The SGD algorithm gives us the following update rules: $p_u \leftarrow p_u + \alpha \cdot q_i (r_{ui} - p_u \cdot q_i)$, and $q_i \leftarrow q_i + \alpha \cdot p_u (r_{ui} - p_u \cdot q_i)$, where α is the learning rate, and p_u and q_i make up the rows and columns of U and V respectively. Running the algorithms a number of times will result in optimized values for the vectors of U and V, which together create the matrices U and V, which can then be used to construct a completed matrix M.

3. Methodology

In order to succinctly analyze the efficiency of the different machine learning algorithms, the team implemented the functions in PostgreSQL. Our implementations of various algorithms were then tested against the MADlib implementation of the corresponding functions to compare the efficiency of their application. The process of the implementation and benchmarking is described in this section.

Functions vs Stored Procedures

PostgreSQL supports the creation and use of both *functions* and *stored procedures* (SPs); the difference being that functions return a value, while SPs do not (PostgreSQL Tutorial, 2020). Stored procedures allow users to write code that utilizes control structures (i.e. FOR loops, IF statements, etc.) via the built-in PL/SQL procedure language.

Although functions and SPs are similar, stored procedures were preferred in our project development because of basic design differences that best suit the classifiers. SPs are better for exception handling and support the use of table variables, temporary tables, and database transactions. Our functions could not be implemented without the native support for procedural operations provided by Postgres; for this reason, our team chose to utilize stored procedures where applicable as a basis for our function development. The LOOP function was found to be useful to iterate through the tables during function development.

Datasets

The three base datasets used are the iris data set (500 attributes). The Anuran Calls dataset (7195 attributes) and finally the Online Retail dataset (541909 attributes). The iris dataset is a popular multivariable dataset often used for machine learning classifiers. The Anuran dataset

is used in several machine learning classification tasks. It contains relevant data for the challenge of recognizing the anuran species through their distinct calls. It is a multi-label dataset with three columns of labels. The dataset was created by segmenting 60 audio records belonging to 4 different families, 8 genus, and 10 species. The Online Retail dataset is transactional data taken from a store in the United Kingdom between 01/12/2010 to 09/12/2011. Depending on the functions input parameters, there may have been additional datasets used in testing to support the functionality of the classifier. Each of the datasets was uploaded into Postgres using a copy statement. Once they are stored into a table the datasets were available for testing. Each dataset was taken from the UCI Machine Learning Repository which is accessible publicly online.

Function Development

K-Mean Clustering

There were three stages of developing the K-mean function using the language plpgsql. At the pre-stage, the python version of the function was written then the similar algorithm was translated into plpgsql. Then there were two stages for the function written in plpgsql, the final version is the most concise with three helper functions and has met all the expectations.

The main function takes in a k value, which is the number of clusters the user wanted to classify the data into. Then it takes a number of max iterations as one of the stopping points. The main function also takes an E value which represents the difference of calculated Euclidean distance measured between two iterations. This has been used as the prior stopping mechanism. If the E value is calculated to be zero, the function stops clustering. The function will also stop at the number the user put in for the max iterations.

There are three helper functions that execute the K-means clustering. The first one is calculating the Euclidean distance, Then the centroid and group functions are used to calculate the centroids and to form groups as clusters. The final result, the function will create different labels for the datasets to represent the different clusters they are in.

There will be two final output for this function. One of the output is a table of all the centroid points depending on the number of centroids the user put as input. The other output is the column of value attached to the data indicating which cluster the data value got assigned to.

K-Nearest Neighbors

Using a query language like PL/SQL for the purpose of finding the nearest neighbors is plausible because of the fundamental principles of query languages. of query languages fundamental principles. The procedure needed to take in any dataset, extract features from it and run and order by statement. The features in the column were ordered by the Euclidean distance from a user-defined point and the 'k' number of points were returned in the table.

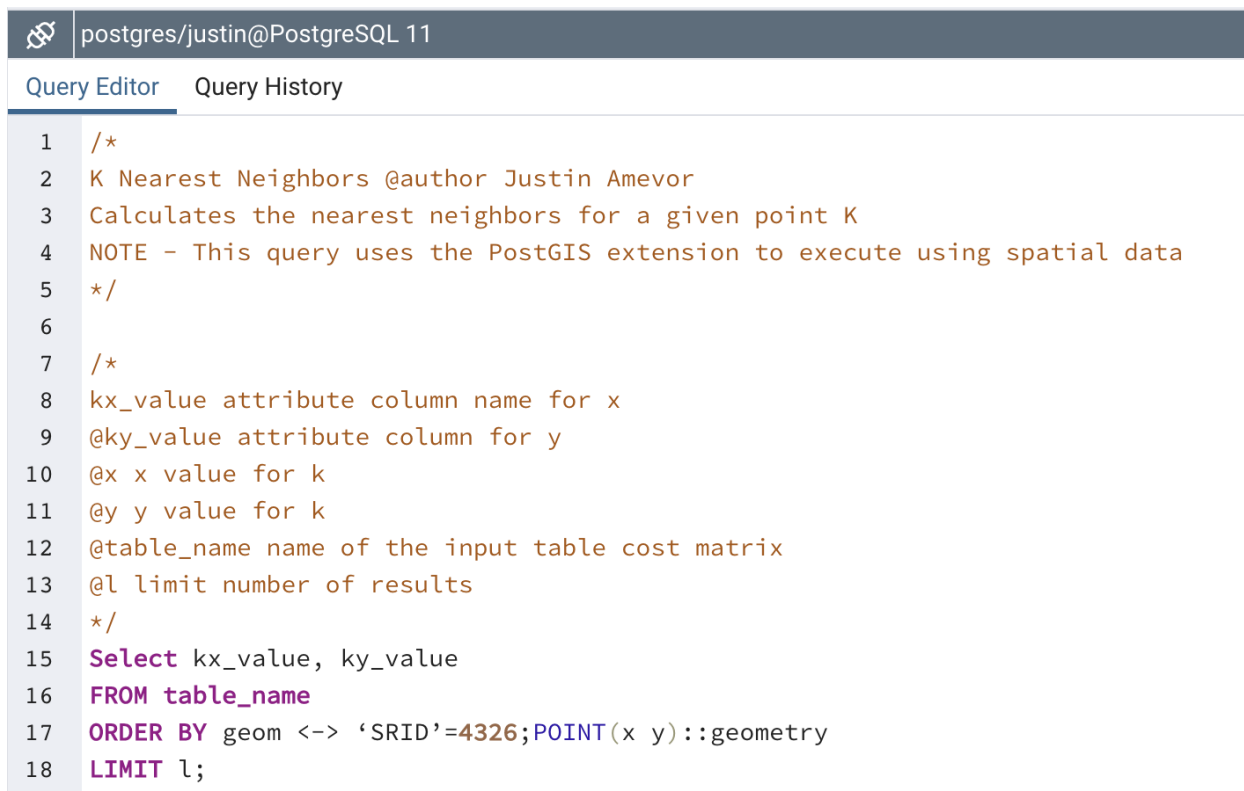
To aid implantation, an iterative process of development was utilized. By beginning with hardcoding the query to work with the Iris dataset, the code was able to be modified to take various inputs. The first implementation used a view to store a table of the dataset with the purpose of being able to run the query on the virtual table. Because of the versatility of the procedure's ability to take in different datasets, and optimal efficiency preferred views were omitted from the final implementation. The main component of the dataset was to extract specific columns to conduct the KNN searching. By abstracting the columns defined by the user then running the SELECT statement on those columns, the function was able to be constructed. The pure implementation is PL/SQL calculates the nearest neighbors by traversing through the

entire dataset. Depending on the size of the dataset, this approach can be very inefficient. To improve the efficiency of this the extension PostGIS was utilized in implementation.

PostGIS

PostGIS is an extension for PostgreSQL that provides support for spatial objects, allowing storage and query of information about location and mapping of data. By turning PostgreSQL into a spatial database, spatial functions can be used to analyze geometric components, determine spatial relationships, and manipulate geometries. PostGIS utilizes an r-tree data structure to efficiently store spatial data indexes in an efficient method.

The PostGIS extension implements the K-Nearest-Neighbor search by traversing through the index, the search finds the nearest candidate geometries that do not require an index constraint. This makes the technique suitable for extremely large tables with high variable data dimensions. **Figure 2.1** shows the KNN query by utilizing PostGIS. The query requires five inputs as parameters. The *kx_value* and *ky_value* take in the attribute column name of the input table. *Table_name* is the name of the table used to find the K nearest neighbor. *X* and *Y* serve as the K value of the desired search point. *L* is an integer of how many results should be returned. The datasets are formatted as a point table in PostGIS to provide accurate results.



```

1  /*
2  K Nearest Neighbors @author Justin Amevor
3  Calculates the nearest neighbors for a given point K
4  NOTE - This query uses the PostGIS extension to execute using spatial data
5  */
6
7  /*
8  kx_value attribute column name for x
9  @ky_value attribute column for y
10 @x x value for k
11 @y y value for k
12 @table_name name of the input table cost matrix
13 @l limit number of results
14 */
15 Select kx_value, ky_value
16 FROM table_name
17 ORDER BY geom <-> 'SRID'=4326;POINT(x y)::geometry
18 LIMIT l;

```

Figure 3.1 KNN Query

Floyd Warshall Algorithm

MADlib calculates the cost matrix of the input table. After obtaining a comprehensive understanding of the algorithm, the development of the function was completed iteratively. To begin, 2 tables' *r_table* and *sol_table* are initialized. The relationships between all paths needs to be established from the input cost matrix. The input cost matrix is put into the *r_table* directly. By using value acting as infinity inside a insert statement the entire source and destination table was generated. Next to calculate the weights from each path the logic of the algorithm pseudocode was constructed:

Floyd-Warshall Pseudocode (Taken from Tutorials Point)[11]

```

Begin
for k := 0 to n, do
    for i := 0 to n, do
        for j := 0 to n, do
            if cost[i,k] + cost[k,j] < cost[i,j], then
                cost[i,j] := cost[i,k] + cost[k,j]

```

Three variables were defined $m1, m2$, and $m3$ all of type *double precision[]* that act as the i, j , and k from the pseudocode. Three loops are used to execute the procedure. Once the smallest weight is found it is stored inside the *sol_table* for output.

Naive Bayes Classification

After reviewing MADlib's implementation of the Naive Bayes algorithm, which includes functions for training, probability, and classification, it became clear that multiple functions would be necessary for the implementation within PostgreSQL. However, we were more concerned with classification, over class probabilities, the two functions implemented were a training function called *nb_training*, and a classification function *nb_classify*.

Training

The *nb_training* function takes three parameters (training table [*regclass*], class column [*varchar*], and an array of attribute columns [*varchar(s)*]). This function takes a table of training data (with the assumption that the table exists within the given database), and uses the class column and array of attribute columns to compute the training model which will be used to classify test data. A reference table (*ref_table*) is created which contains all unique classes from the class column, and an assigned numerical value (key). This reference table is then used to

calculate the mean and standard deviation for each attribute and class combination, creating a model for classification in a new table (*summary_table*).

Classification

The *nb_classify* function takes one parameter (test table [*regclass*]), i.e. name of the SQL table containing the unclassified data. The function then translates the given data into the *test_attrs* table, which contains only one column containing an array of the attribute values (data type of array elements \rightarrow *double precision*) for each row in the test table. The function then loops through each row, class (or label), and array value to calculate each class probability. The highest probability for each row is selected and stored in the *prob_table* with its corresponding class number (data type \rightarrow *integer*). Due to the *naive* assumption of independence within this function, the initial probability of each class is set to be 1.0. This will affect the precision of the resulting probability, but not the accuracy in identifying the most likely class. The initial probability is then multiplied by the *gaussian probability* n times, where n is the number of attributes in the array(s) of attribute values from the *test_attr* table. The *gaussian prob.* is calculated using a simple helper function I wrote called *gaussian_probability--* takes an x value, *mean*, and *standard deviation* of the given attribute and class; and applies the following function:

```

-----
-- GAUSSIAN PROBABILITY -- @author,bailey_schmidt
/* Calculate the Gaussian probability distribution function for x */
CREATE
OR REPLACE FUNCTION gaussian_probability (
  x DOUBLE PRECISION,
  mean DOUBLE PRECISION,
  stddev DOUBLE PRECISION
) RETURNS DOUBLE PRECISION AS $func$ DECLARE exponent DOUBLE PRECISION;
BEGIN

IF(mean < 1.0)
THEN mean = mean + 1.0;
END IF;

IF(stddev < 1.0)
THEN stddev = stddev + 1.0;
END IF;

exponent = exp(- ((x - mean) ^ 2.0 / (2.0 * stddev ^ 2.0)));
RETURN (1.0 / (sqrt(2.0 * pi()) * stddev)) * exponent;
END $func$ LANGUAGE plpgsql;

```

Figure 3.2 Gaussian Probability Function

A Gaussian probability is calculated for each attribute for each class, a very expensive but necessary process to calculate the probability that a given attribute belongs to a class. This process is repeated for each row of test data; after iterating through all attribute values in a given row the highest probability/ class prediction is selected and stored in the *prob_table*. This function returns *void*.

Matrix Factorization

MADlib provides two different implementations of matrix factorization: low-rank matrix factorization, and singular value decomposition (SVD). For the purposes of this project only an implementation of SVD was created.

Input data for the *matrix_factorization* function is given in sparse-matrix format (value, row, column) which is inserted into a table in Postgres. This input format was less time consuming and more efficient than requiring a user to input every entry of the matrix, especially for larger matrices. The *matrix_factorization* function takes 6 parameters: (*_table* [*regclass*], *num_rows* [*integer*], *num_cols* [*integer*], *k* [*integer*], *iterations* [*integer*], *alpha* [*double precision*]). The parameter *k* refers to the number of factors that will be found, i.e. the number of columns and rows in the output matrices. The parameter *alpha* refers to the learning rate of the algorithm. The two output tables are initially created and populated with random floats between -1 and 1, in order to remove bias from the stochastic gradient descent (SGD) optimization procedure. The SGD algorithm is run a given number of times, determined by the *iterations* parameter, on the input data and for each iteration, the random values of the output tables are updated based on the results of the algorithm, using the records available in the input table. Once all iterations of the SGD algorithm have run, our resulting matrices should approximately be factors of the original input matrix. If multiplied by each other, we should obtain an approximation of a completed matrix based on the original input data that was given.

MADlib Benchmarking -- Testing against MADlib Analytics Library

Each MADlib function we are testing with has a specific set of parameters and formatting for inputs that must be satisfied in order for the function to run (MADlib documentation, 2019) So it is important to note the degree to which we needed to modify data tables to run MADlib functions, as a contrast to our own functions. Therefore, testing methodologies varied in execution with an overarching goal of comparing execution time and result accuracy (against the corresponding MADlib function(s)).

Testing Methodology K-Means Clustering

The testing began with a naive data set I created with less than 20 rows of data pre-inserted into the database in a form of table. To test K-Mean clustering, There were three data sets used in testing. The Iris dataset, The Anuran Calls dataset, and the Online Retail dataset. The clustering function was developed based on the three columns out of these data sets we randomly chose. For both testing and MADlib the data sets were pre inserted to the database as tables with only the chosen columns. Then we can compare the results from my testing in PgAdmin with results from MADlib functions. Then we recorded the running time displayed on PgAdmin to do the comparison between the efficiency for both functions.

Testing Methodology K-Nearest Neighbors

To test both sets of KNN functions, three datasets were used. The Iris dataset, The Anuran Calls dataset, and the Online Retail dataset. Each dataset was put into a table that was formatted for their respective language. For MADlib each attribute was inserted into a table as a *double precision* array. For PL/SQL to utilize the PostGIS POINT geometry, each attribute was converted into spatial data allowing the RDMS to execute queries using geographical data. Both sets of functions ran the test datasets three times, the average runtime for each dataset was then calculated. The results of each query were recorded to compare the accuracy of both functions.

Testing Methodology Floyd Warshall Algorithm

To test the Floyd Warshall Algorithm a cost matrix needs to be taken in as an input. For straightforward testing the test matrix used in the MADlib documentation was used for the testing purposes of this investigation. The query was run 3 times and the average of the experiment was recorded.

Testing Methodology Naive Bayes

To test the Naive Bayes functions against MADlib's, I ran three sets of data through the functions, maintaining a consistent number of records and attributes for each dataset:

Table 3.3 Record and Attribute Count for Naive Bayes Algorithm Testing

	Small Dataset (Iris)	Medium Dataset (Frogs)	Large Dataset (Online Retail)
Number of Records	150	7190	541909
Number of Attributes	4	4	2

I ran each function five times and averaged the five iteration times to get a mean execution time for each function. This means five runs of `nb_training()`, `create_nb_prepared_data_tables()`, `nb_classify()`, and `create_nb_classify_view()`. I then tested the accuracy of the training results via query, comparing my training model to MADlib's on an attribute basis (i.e. comparing my calculated attribute variance and attribute mean to MADlib's). Figure 3.4 is an example of how I accomplished this using my `summary_table` and MADlib's `numeric_attr_params` tables. This query utilizes the use of a subquery where I get the absolute value of the difference between the calculated attribute variance from MADlib's model (`n.attr_var`), and my calculated variance of the corresponding attribute (i.e. `unit_price_stddev2`). The query then counts the number of results where the difference is within a tolerance of one (± 1), which is cast as a numeric to then be divided by the total number of records from the training model, and finally multiplied by 100 to get the percentage of attribute records within the training model which are correctly calculated

within the aforementioned tolerance.

```
-- Compare accuracy of unit_price (attribute #2) variance with error tolerance (+/- 1.0)
SELECT round((cast(count(*) as numeric) /cast((SELECT count(*) FROM summary_table) as numeric)*100.00), 2) as within_1
FROM
  (SELECT n.class,
    abs(n.attr_var - (s.unit_price_stddev^2)) as diff
  FROM summary_table s
  JOIN numeric_attr_params n on s.class_num = n.class
  WHERE n.attr = 2) o
WHERE o.diff <= 1.0;
```

Figure 3.3 Query for comparing accuracy of Naive Bayes Training Model(s)

If given more time I would've liked to abstract out a function to be used to test the accuracy of the models. However, I just manually repeated this query by changing the attribute names, and numbers (i.e. MADlib's *n.attr*) for all attribute variances and means.

For the classification functions, I wrote a similar query to attempt joining my classification table (*prob_table*) with MADlib's (*nb_classify_view_fast*), which worked well in the case of the small *Iris* dataset, but due to my implementation there were cases with the larger sets that didn't correctly represent the true accuracy of the results. It was during this testing process that I developed many of my recommendations and recognized the shortcomings of my own functions-- See *Naive Bayes Results sections for more information*. Figure 3.5 shows the query used to compare classification predictions of the classification functions:

```
-- compare classification results
SELECT (cast(count(*) AS double precision)/cast(
  (SELECT count(*) FROM test_subset) as double precision) * 100.00) as correct
FROM
  (SELECT p.row_key,
    p.prediction[1] AS my_classification,
    c.nb_classification
  FROM prob_table p JOIN nb_classify_view_fast c ON p.row_key = c.key) a
WHERE a.my_classification = a.nb_classification[1];
```

Figure 3.4 Query for comparing accuracy of Naive Bayes Training Model(s)

This query accurately calculates the percentage of matching predictions (mine vs MADLib's) for the *Iris* dataset since the class labels are all categorical and therefore need to be assigned a numerical key in order to be processed by the algorithm in the case of both training functions.

However, if the class label was already numerical my function may assign a numeric key of '1' to class label '7' for example. This is because while designing to allow the use of non-numerical class labels I did not write any contingency code to handle cases where the class labels are already numeric but may not be in numerical order. This may lead to the aforementioned case of a '1' class key for a numeric class label of '7'. Where MADlib will just use the class label in this case as the numeric key (i.e. if the class label is '7' the class key will also be '7'). So when trying to compare accuracy the results were misleading because of mismatched class keys and class labels (between mine and MADlib's models) which made it difficult to properly compare results.

It is also worth noting the particular formatting expected by MADlib in regards to its training and classification functions. MADlib's functions expect a table with an integer *id* column, class label column (numerical or categorical), and a column with an array of the corresponding attributes for any given row (of size *n*, where *n* is the number of attributes). Therefore, to test against MADlib's results and ensure both my functions and MADlib's were receiving the same information, I needed to create tables from the original datasets with the aforementioned formatting for use in the MADlib functions (denoted in my test code by *ml_[dataset name]_train* and *ml_[dataset name]_classify*).


```

CREATE TABLE ml_iris_train(id SERIAL, class INTEGER, attributes double precision[]);
INSERT INTO ml_iris_train(class, attributes)(SELECT r.class_num, array[i.sepal_length,
    i.sepal_width,
    i.petal_length,
    i.petal_width] as attr
    FROM iris i
    JOIN ref_table r ON i.class_val = r.class_val);
Select * from ml_iris_train limit 10;

```

Figure 3.4.1 Create, Populate, and Query Formatted SQL Tables for use in MADlib Functions

	id integer	class integer	attributes double precision[]
1	1	1	{5.1,3.5,1.4,0.2}
2	2	1	{4.9,3.1,4.0,2}
3	3	1	{4.7,3.2,1.3,0.2}
4	4	1	{4.6,3.1,1.5,0.2}
5	5	1	{5,3.6,1.4,0.2}
6	6	1	{5.4,3.9,1.7,0.4}
7	7	1	{4.6,3.4,1.4,0.3}
8	8	1	{5,3.4,1.5,0.2}
9	9	1	{4.4,2.9,1.4,0.2}
10	10	1	{4.9,3.1,1.5,0.1}

Figure 3.4.2 Result of Select Statement from ml_iris_train

Testing Methodology Matrix Factorization

Testing for the matrix factorization function utilized two randomly generated matrices of two sizes: a 5x5 matrix and a 15x15 matrix. Each matrix was sparsely populated with a total of four and seventeen values occupying the two matrices respectively. Both our function and MADlib's implemented function were run on both matrices. The 5x5 matrix was given 100 iterations to run through both

functions, and the 15x15 matrix was given 25 iterations. Each matrix was run through each version of the function three times, and the average runtime was calculated for each matrix.

4. Results and Recommendations

After the development of each function was completed, the team then moved into testing and analyzing the functions performances vs the MADlib library. The results of each function is described in the following section;

Results K-Means Clustering

In **Figure 4.1.1**, it shows a screenshot when the Iris data was used as the testing data and we performed clustering based on the `sepal_length` and `petal_length`. It can clearly seen that over around 200 samples, when the user wants them to categorize into two clusters, the difference between two lengths over 2 units were clusters into the first cluster and the difference less than or equal to 2 units were clusters into the second cluster. By calling function `kmeans(K)`, the user is able to find out the centroid points for `K` clusters. `K` can be defined as any integer by the user. The next table is showing the time result comparison between the PL/SQL function and the MADlib functions. The PL/SQL function runs longer when the data set gets bigger due to the fact that there were several for loops in the functions that can be costly when the input gets large.

Data Output

	sepal_length double precision	sepal_width double precision	pedal_length double precision	pedal_width double precision	iris character varying (25)	cluster_id integer
46	4.8	3	1.4	0.3	Iris-setosa	1
47	5.1	3.8	1.6	0.2	Iris-setosa	1
48	4.6	3.2	1.4	0.2	Iris-setosa	1
49	5.3	3.7	1.5	0.2	Iris-setosa	1
50	5	3.3	1.4	0.2	Iris-setosa	1
51	7	3.2	4.7	1.4	Iris-versicolor	2
52	6.4	3.2	4.5	1.5	Iris-versicolor	2
53	6.9	3.1	4.9	1.5	Iris-versicolor	2
54	5.5	2.3	4	1.3	Iris-versicolor	2
55	6.5	2.8	4.6	1.5	Iris-versicolor	2

Figure 4.1.1 K-Mean query and cluster results from PL/SQL

Data Output

	kmeans record
1	(0,0)
2	(5.068717...

Table 4.1.1 K-Mean Clustering centroid points results when the cluster is set to 2

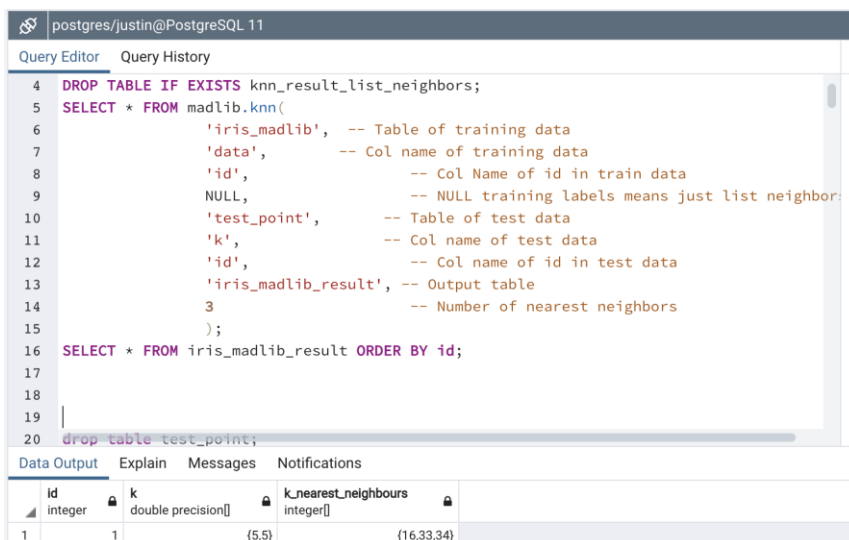
Table 4.1.2 K-Mean Clustering Runtime for MADlib and PL/SQL in Milliseconds

Dataset	PL/SQL	MADlib
Iris	534msec	233 msec
Anuran Calls	680 msec	380msec
Online Retail	1432 msec	680msec

Results K Nearest Neighbors

MADlib

The MADlib function reported the nearest neighbors for the test points accurately. For the Iris dataset a test point a [5,5] was chosen for K. As seen in **Figure 3.1** the results returned points 16[5.7,4.4], 33[5.2,4.1], and 34[5.5,4.2]. The query was completed in 111 milliseconds. For the Anuran (MFCCs) dataset a test point of [0.15,0.15] was chosen for K. The results returned points 3477[0.2670,0.2184], 1515.[0.2011,0.1561], and 4607[0.599,0.1498]. The query was completed in 121 milliseconds. Finally to test the Online Retail dataset a test point of [5.99] was chosen for K. The results returned points 7729[5.95], 5023[5.95], and 12449[5.95]. The query was completed in 166 milliseconds.



```

4 DROP TABLE IF EXISTS knn_result_list_neighbors;
5 SELECT * FROM madlib.knn(
6     'iris_madlib', -- Table of training data
7     'data',       -- Col name of training data
8     'id',         -- Col Name of id in train data
9     NULL,        -- NULL training labels means just list neighbors
10    'test_point', -- Table of test data
11    'k',          -- Col name of test data
12    'id',        -- Col name of id in test data
13    'iris_madlib_result', -- Output table
14    3,           -- Number of nearest neighbors
15    );
16 SELECT * FROM iris_madlib_result ORDER BY id;
17
18
19
20 drop table test_point;

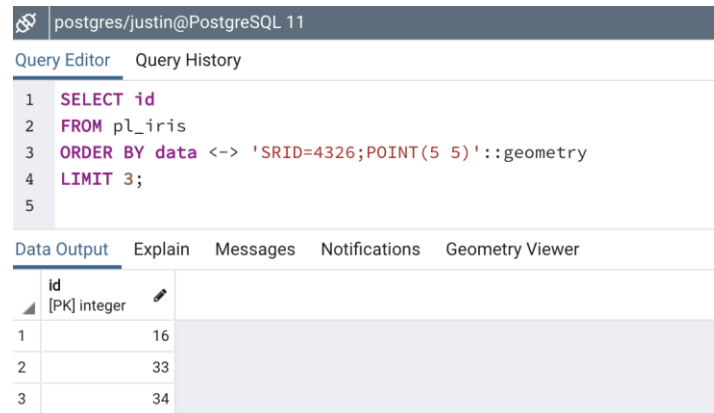
```

	id	k	k_nearest_neighbours
	integer	double precision[]	integer[]
1	1	(5,5)	{16,33,34}

Figure 4.2 KNN query using MADlib

PL/SQL

The results for the PL/SQL trials were identical to the MADlib function. For the Iris dataset the results were the same as seen in **Figure 3.2**. Points 16,33,and 34 were returned matching the result from MADlib. The results for the Anuran and Online Retail Datasets also



The screenshot shows a PostgreSQL Query Editor window with the following SQL query:

```

1 SELECT id
2 FROM pl_iris
3 ORDER BY data <-> 'SRID=4326;POINT(5 5)::geometry
4 LIMIT 3;
5

```

Below the query editor, the 'Data Output' tab is active, displaying the following results:

	id [PK] integer
1	16
2	33
3	34

Figure 4.3 KNN query using PL/SQL

produced the same results. 2 out of the 3 PL/SQL functions test on average was faster than MADlib.

Table 4.1 KNN Runtime for MADlib and PL/SQL in Milliseconds

Dataset	PL/SQL	MADlib
Iris	124 msec	132 msec
Anuran Calls	139 msec	121 msec
Online Retail	153 msec	166 msec

Results Floyd Warshall Algorithm

In comparison with MADlib's sample data [12], the results for the PL/SQL data is detailed in this section:

MADlib

The table contains four columns: the *e_src*, *dest*, and *e_weight*. The results are detailed in this section. The query was executed three times with an average runtime of 225 milliseconds:

0	0	0
0	1	1
0	2	1
0	3	2
0	4	10
0	5	2
0	6	3
0	7	4
1	0	4
1	1	0
1	2	2
1	3	3
1	4	14
1	5	3
1	6	4
1	7	5
2	0	2
2	1	3
2	2	0
2	3	1
2	4	12

2	5	1
2	6	2
2	7	3
3	0	1
3	1	2
3	2	2
3	3	0
3	4	11
3	5	3
3	6	4
3	7	5
4	0	-2
4	1	-1
4	2	-1
4	3	0
4	4	0
4	5	0
4	6	1
4	7	2
5	0	Infinity
5	1	Infinity
5	2	Infinity
5	3	Infinity

5	4	Infinity
5	5	0
5	6	1
5	7	2
6	0	Infinity
6	1	Infinity
6	2	Infinity
6	3	Infinity
6	4	Infinity
6	5	Infinity
6	6	0
6	7	1
7	0	Infinity
7	1	Infinity
7	2	Infinity
7	3	Infinity
7	4	Infinity
7	5	Infinity
7	6	Infinity
7	7	0

PL/SQL

The solution table is formatted into three columns; *source*, *destination*, and *weight*. The results for PL/SQL procedure are detailed in this section:

0	0	8
0	1	1
0	2	1
0	3	11
0	4	10
0	5	11
0	6	11
0	7	11
1	0	8
1	1	11
1	2	2
1	3	10
1	4	20
1	5	11
1	6	11
1	7	11
2	0	1
2	1	4
2	2	5
2	3	1
2	4	13

2	5	1
2	6	3
2	7	4
3	0	1
3	1	2
3	2	3
3	3	2
3	4	11
3	5	2
3	6	2
3	7	2
4	0	-2
4	1	-1
4	2	0
4	3	-1
4	4	8
4	5	-1
4	6	-1
4	7	-1
5	0	-1
5	1	2
5	2	3
5	3	2

5	4	11
5	5	2
5	6	1
5	7	2
6	0	-1
6	1	2
6	2	3
6	3	2
6	4	11
6	5	2
6	6	2
6	7	1

The query was run three times, executing at an average runtime of 995 milliseconds. The construction of the solution table was successful however the logic providing the correct weights is significantly different than the MADlib results. The accuracy of the algorithm suffered in the logic of the nested four loops. The syntax and performance of *LOOP* in PL/SQL could be culpable in the accuracy of the procedure. The PL/SQL language does not have the most optimal execution code for loops compared to other languages.

Results Naive Bayes

The results of testing the natively implemented Naive Bayes functions against MADlib's are as follows:

Runtime Tests

The runtime or execution time testing of all *Naive Bayes* algorithms are shown in Figure 4.3, native functions are denoted by *my_training* and *my_classify* while the MADlib functions are denoted by *ML_training* and *ML_classify*. Figure 4.3.1 displays the timing in milliseconds of each trial, or iteration, which were then averaged as can be seen in Figure 4.3.2.

Iteration	Time - nb_training (ms)	Time - MADlib_training (ms)	Time - nb_classify (ms)	Time - MADlib_classify (ms)	Small Set (Iris)
1	14.929	18.791	28.159	12.158	4 attrs
2	20.596	22.117	18.067	11.369	150 records
3	18.886	17.965	31.663	7.109	
4	18.645	19.139	28.815	10.611	
5	15.538	23.857	23.988	9.676	
	<i>my_training</i>	<i>ML_training</i>	<i>my_classify</i>	<i>ML_classify</i>	Medium Set (frogs)
1	218.746	123.634	26066.330	11.885	4 attrs
2	226.825	115.638	25298.853	11.712	7190 records
3	246.165	125.002	25152.261	10.217	
4	205.720	113.504	36092.344	14.940	
5	214.089	125.628	25312.779	10.783	
	<i>my_training</i>	<i>ML_training</i>	<i>my_classify</i>	<i>ML_classify</i>	Large Set (Online Retailers)
1	4970.641	2518.061	99239.714	9.957	2 attrs
2	5215.249	2444.421	98239.018	11.262	541909 records
3	4923.863	2487.072	99571.103	9.973	
4	5209.251	2426.274	99200.835	10.232	
5	5061.952	2427.169	99878.498	10.781	

Figure 4.3.1 Raw Runtime Data for Naive Bayes Algorithms

Small Set (Iris)	nb_training	MADlib_training	nb_classify	MADlib_classify
	18.264	20.374	26.676	10.185
-- SECONDS --	0.018	0.020	0.028	0.012
-- MINUTES --	0.000	0.000	0.000	0.000
Medium Set (frogs)	nb_training	MADlib_training	nb_classify	MADlib_classify
	222.309	120.681	27584.513	11.907
-- SECONDS --	0.222	0.121	26.066	0.012
-- MINUTES --	0.004	0.002	0.434	0.000
Large Set (Online Retailers)	nb_training	MADlib_training	nb_classify	MADlib_classify
	5076.191	2460.599	99062.668	10.441
-- SECONDS --	5.076	2.461	99.063	0.010
-- MINUTES --	0.085	0.041	1.651	0.000

Figure 4.3.2 Averaged Runtime Data for Naive Bayes Algorithms

The data from Figure 4.3.1/ Figure 4.3.2 was used to create Figure 4.4.1 and Figure 4.4.2 which graphically represent the difference in runtime between the native training / classification functions versus MADlib’s training / classification functions.

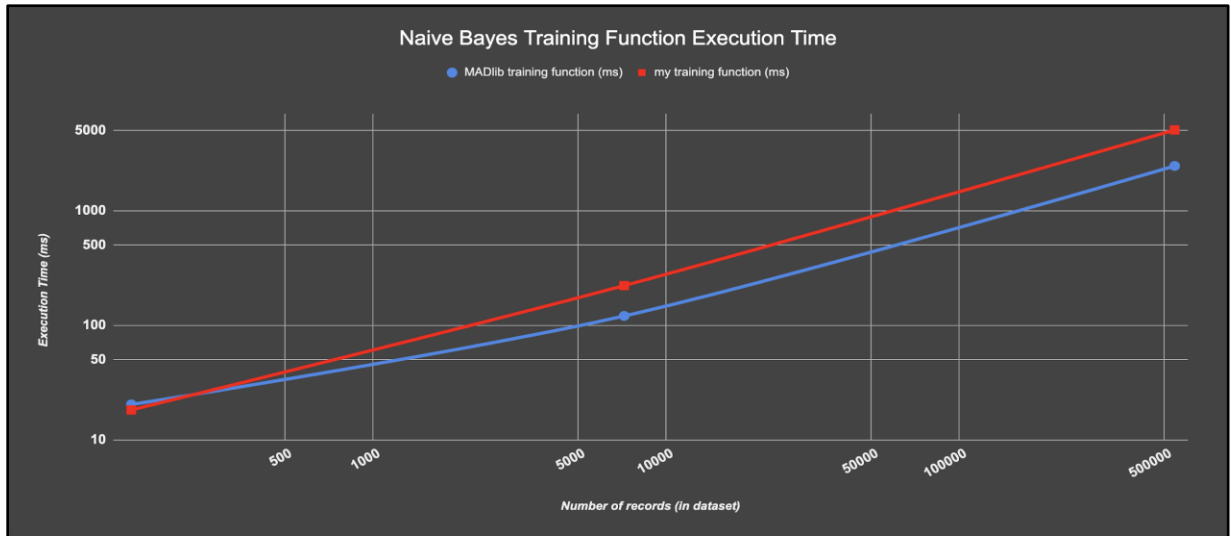


Figure 4.4.1 Naive Bayes Training Function Execution Time Graph (in milliseconds)

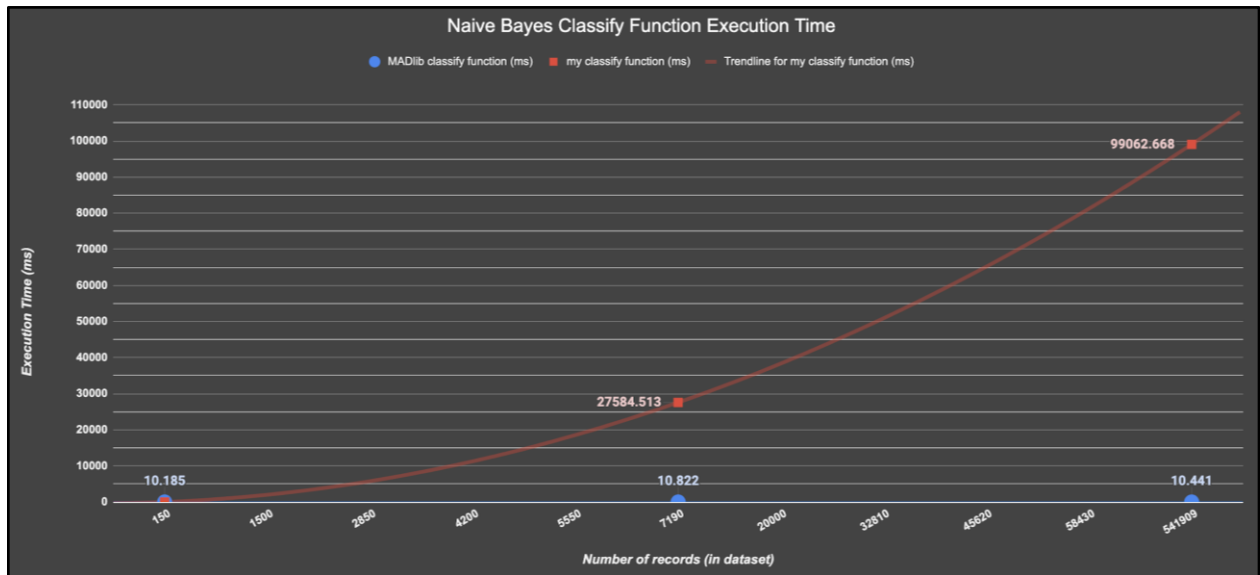


Figure 4.4.2 Naive Bayes Classify Function Execution Time Graph (in milliseconds)

The native training algorithm is very comparable to its MADlib counterpart in terms of runtime as Figure 4.4.1 shows; the time differences are milliseconds to seconds and the highest mean runtime was approximately five seconds.

This is not the case with the native classification algorithm however. The exponential increase in execution time is due to bottlenecks via three nested for loops in my classification algorithm. This issue is exacerbated further when there is both a large number of records and class labels. For each new record being tested, or each new attribute that is another three iterations due to the nested nature of the loops (i.e. Ω^3 in terms of algorithmic complexity with respect to *Omega*).

Accuracy Tests

To test the accuracy of the algorithms I performed various queries as mentioned in the Methodology section. A few visual comparisons are also provided below from queries joining the training models, however as the datasets get larger this clearly is an improbable way to compare results.

	class_num integer	class_val character varying	rec_count integer	class integer	class_cnt bigint	all_cnt bigint
1	1	Iris-setosa	50	1	50	150
2	3	Iris-virginica	50	3	50	150
3	2	Iris-versicolor	50	2	50	150

Figure 4.5.1 Visual Comparison of Small Set Training Accuracy for Naive Bayes Algorithms

	Data Output	Messages	Query History	Notifications	Explain	
	class_num integer	class_val character varying	rec_count integer	class integer	class_cnt bigint	all_cnt bigint
1	1	Australia	1259	1	1259	541909
2	2	Austria	401	2	401	541909
3	3	Bahrain	19	3	19	541909
4	4	Belgium	2069	4	2069	541909
5	5	Brazil	32	5	32	541909
6	6	Canada	151	6	151	541909
7	7	Channel Islands	758	7	758	541909
8	8	Cyprus	622	8	622	541909
9	9	Czech Republic	30	9	30	541909
10	10	Denmark	389	10	389	541909
11	11	EIRE	8196	11	8196	541909
12	12	European Community	61	12	61	541909
13	13	Finland	695	13	695	541909
14	14	France	8557	14	8557	541909
15	15	Germany	9495	15	9495	541909
16	16	Greece	146	16	146	541909
17	17	Hong Kong	288	17	288	541909
18	18	Iceland	182	18	182	541909

Figure 4.5.2 Visual Comparison of Medium Set Training Accuracy for Naive Bayes Algorithms

The results of queries comparing the accuracy of the algorithms are displayed in Table 4.3.1:

Attribute Name	Accuracy (within +/- 1.0)
sepal_length_avg	100.00%
sepal_length_stddev	100.00%
sepal_width_avg	100.00%
sepal_width_stddev	100.00%
petal_length_avg	100.00%
petal_length_stddev	100.00%
petal_width_avg	100.00%
petal_width_stddev	100.00%

Table 4.3.1 Naive Bayes Training Accuracy Results for Small Data Set (Iris)

Attribute Name	Accuracy (within +/- 1.0)
MFCCs2_avg	100.00%
MFCCs2_stddev	96.67%
MFCCs3_avg	100.00%
MFCCs3_stddev	96.67%
MFCCs4_avg	100.00%
MFCCs4_stddev	96.67%
MFCCs5_avg	100.00%
MFCCs5_stddev	96.67%

Table 4.3.2 Naive Bayes Training Accuracy Results for Medium Data Set (Frogs)

Attribute Name	Accuracy (within +/- 1.0)
quantity_avg	100.00%
quantity_stddev	100.00%
unit_price_avg	100.00%
unit_price_stddev	100.00%

Table 4.2.3 Naive Bayes Training Accuracy Results for Large Data Set (Online Retail)

Once again the native training function proves to be comparable with MADlib's training function as the models are almost identical value-wise in terms of accuracy within a +/- 1.0 tolerance. Based on the results of both runtime and accuracy testing, the native training function is equally as viable as MADlib's implementation.

Figures 4.6.1 - 4.6.3 show the queries used to compare the classification results of the native implementation vs MADlib's results (i.e. *nb_classify_view_fast*). Figures 4.6.1 and 4.6.2 are written to calculate percentage correct between my results and MADlib's results, while 4.6.3 is comparing MADlib's results to the actual class labels of the first 10,000 records.

4.6.3 actually shows that MADlib only correctly predicted 7.72% of the class labels of the 10,000 record test subset.

Other factors to consider when reviewing these results are,

1) MADlib's Naive Bayes functions are still experimental, and therefore not finalized versions, so there could be some improvements to be made.

2) I arbitrarily chose the class value for this set as the *country* with the numerical attributes being *quantity* and *unit price*. It would be fair to assume that these attributes and classifications are unrelated and therefore could not accurately be used to predict the class label.

```

87 -- compare classification results
88 SELECT (cast(count(*) AS double precision)/cast(
89 (SELECT count(*) FROM test_subset) as double precision) * 100.00) as correct
90 FROM
91     (SELECT p.row_key,
92         p.prediction[1] AS my_classification,
93         c.nb_classification
94     FROM prob_table p JOIN nb_classify_view_fast c ON p.row_key = c.key) a
95     WHERE a.my_classification = a.nb_classification[1];

```

Data Output		Messages	Query History	Notifications	Explain
	correct double precision				
1		100			

Figure 4.6.1 Naive Bayes Classification Accuracy Results for Large Data Set (Iris)

```

55 SELECT round((cast(count(*) as numeric) / 1000.00),2) as correct
56 FROM (SELECT p.row_key,
57         p.prediction[1] as my_classification,
58         c.nb_classification
59     FROM prob_table p
60     join nb_classify_view_fast c on p.row_key = c.key) a JOIN (select class_val
61 from fr_train
62 limit 1000)b ON a.my_classification = cast(b.class_val as double precision);
63 --
64 SELECT distinct count(*) as correct

```

Data Output		Messages	Query History	Notifications	Explain
	correct numeric				
1		35.84			

Figure 4.6.2 Naive Bayes Classification Accuracy Results for Medium Data Set (Frogs)

```

125 Select round(cast((count(*)/10000.00)*100 as numeric), 2) as correct
126 From (select * from nb_classify_view_fast c join (select * from ml_or_train order by id limit 10000) t on c.key = t.id) a
127 WHERE a.nb_classification[1] = a.class;

```

Data Output		Messages	Query History	Notifications	Explain
	correct numeric				
1	7.72				

Figure 4.6.3 Naive Bayes Classification Accuracy Results for Large Data Set (Online Retail)

Future Development / Takeaways

There was a lot to be learned through the planning, developing, and testing of these Naive Bayes functions, and if given more time there are three main areas to give more time and thought to.

The first would be greater foresight and planning with regard to required formatting of input data. A concept that may seem frivolous at the beginning of development, but proves to be a very powerful means of eliminating unknowns during the development process. Complete understanding of the data types, and formatting of the data being input gives a developer a lot more power to transform data effectively, and debug more efficiently. Specifically when developing the classification function, which required the use of a gaussian distribution function, had I known that the *double precision* data type only supported up to 15 decimal places it would've been clear that *numeric* or *decimal* would have been better options as they offer much more storage space and compatibility with mathematical functions such as *round()*.

The second area of focus would be better planning in terms of looping through data, specifically to avoid the three nested loop bottleneck created in the *nb_classify* function, while at least $\frac{2}{3}$ loops are unavoidable, there is definitely more that could be done to optimize the algorithm.

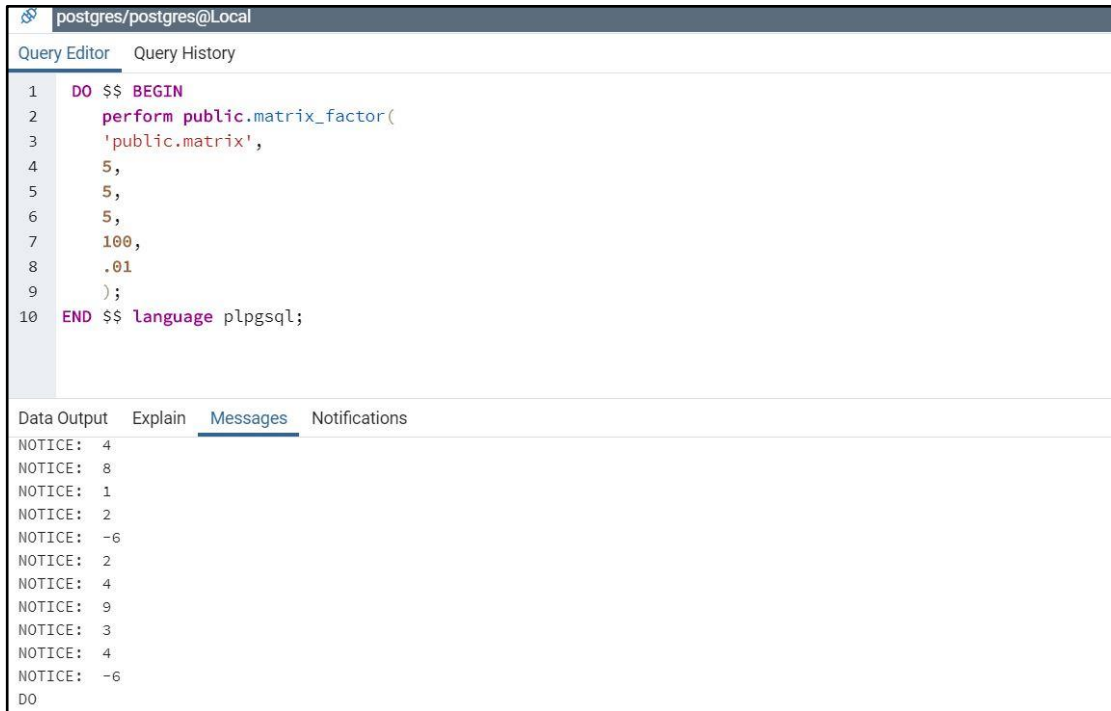
Lastly I would use more arrays and less loops, particularly nested loops as they are, ironically, the quickest way to slow down a function. Arrays allow for cleaner data storage and parsing into

other forms. Arrays are conceptually fairly universal when it comes to computer science, therefore the understanding is very transferable and there is plenty of documentation and built-in functions to support more nuanced use of such data structures [13].

Results Matrix Factorization

PL/SQL

The trials in PL/SQL all produced matrices that were reasonable factors of the original input matrix. They were not the same as those produced by MADlib due to the nature of the optimization algorithm that was implemented and the randomization of the initial values of the output matrices. The factored matrices will likely end up in different local maxima resulting in differing matrices. The 5x5 matrix finished its query in an average of 2524 milliseconds for its 100 iterations and the 15x15 matrix finished its query in an average of 98 seconds for its 25 iterations. **Table 3.2** shows the runtime for each matrix and each function.



```
postgres/postgres@Local
Query Editor  Query History
1 DO $$ BEGIN
2   perform public.matrix_factor(
3     'public.matrix',
4     5,
5     5,
6     5,
7     100,
8     .01
9   );
10 END $$ language plpgsql;

Data Output  Explain  Messages  Notifications
NOTICE:  4
NOTICE:  8
NOTICE:  1
NOTICE:  2
NOTICE: -6
NOTICE:  2
NOTICE:  4
NOTICE:  9
NOTICE:  3
NOTICE:  4
NOTICE: -6
DO
```

Figure 4.7 Matrix Factorization query using PL/SQL

Table 4.3 Matrix Factorization Runtime for MADlib and PL/SQL in Milliseconds

Matrix	PL/SQL	MADlib
5x5	2524 msec	350 msec
15x15	98 sec	35 msec

Recommendations

By recognizing the findings of the investigation, the team was able to assess the versatility of MADlib. Despite PL/SQL having high performance and productivity MADlib outperformed the PL/SQL functions significantly. MADlib typically executed a more efficient query than one constructed in PL/SQL. The limitation of using MADlib best runs on macOS, using another operating system can be difficult and perchance impassable. **Figure 4.8** shows the collective PL/SQL functions performance vs. the MADlib functions. MADlib is still early in development and is working to improve and add new features to its platform. In its early days of usage this investigation found MADlib to be a viable and useful tool for in-database processing. The ease of use and documentation make performing in-database analytics achievable for a common data scientist.

Figure 4.8 illustrates a graph the team developed to help assess the performance of the MADlib functions vs PL/SQL. Each test procedure was scored between a range from 1-3 where 3 was awarded for accurate and efficient testing. Scores were awarded by the developer retrospectively to assess a clear interpretation of the functions performance.

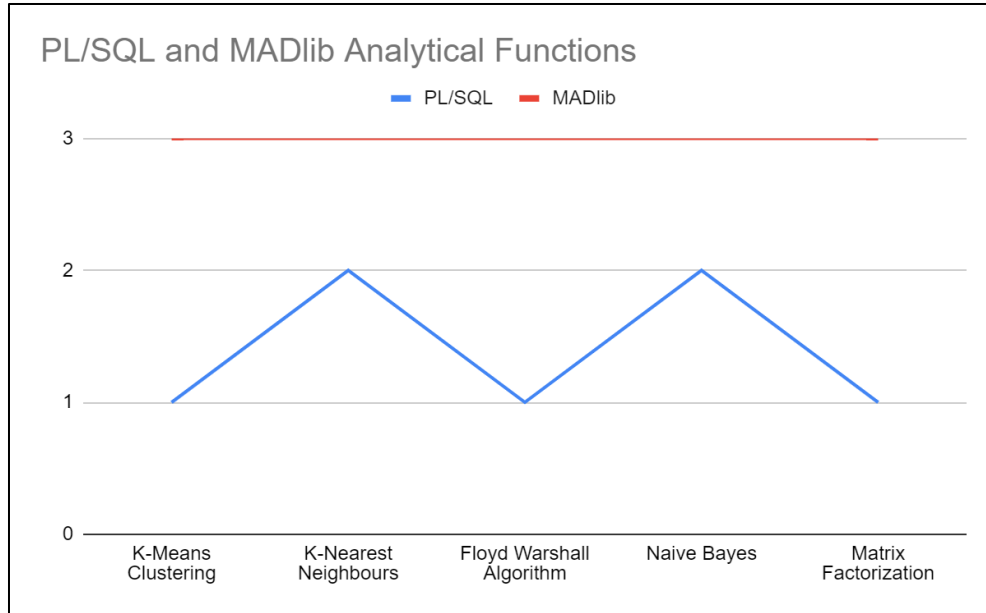


Figure 4.8 Charting performance of MADlib Functions vs. PL/SQL

5. Conclusion

PL/SQL is a powerful and efficient tool that exploits block statements allowing for the capability to perform powerful processes. The language provides a powerful platform to develop efficient queries. MADlib provides a simpler way to perform accurate analytical functions on sample sizes of any size. MADlib is a new open source software that is still developing but offers valuable functionality to data scientists and common analytical persons. The expansion of MADlib's functionality is a notable stock to pay attention to during its continued development. One who wishes to conduct their own analytical processing should consider MADlib a reliable, powerful, and accurate tool if a proper installation can be complete.

Bibliography

- Apache MADlib. (2019, July 2). Main Page. Retrieved from <https://madlib.apache.org/docs/latest/index.html>.
- Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching"
- Cornell. (2018). Bayes Classifier and Naive Bayes. Retrieved from <http://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote05.html>.
- Joyce, J. (2003, September 30). Bayes' Theorem. Retrieved from <https://plato.stanford.edu/entries/bayes-theorem/>.
- Logistic Regression. (2017, May 16). Logistic Regression. Retrieved from https://madlib.apache.org/docs/v1.11/group_grp_logreg.html.
- K-Mean Clustering.(2019, January 16). Retrieved from https://madlib.apache.org/docs/latest/group_grp_kmeans.html
- K-Nearest Neighbors. (2019, July 2). Retrieved from https://madlib.apache.org/docs/latest/group_grp_knn.html.
- Rodriguez, G. (2007). Lecture Notes on Generalized Linear Models. Retrieved from <https://data.princeton.edu/wws509/notes/c3.pdf>.
- McQuillan, F. (2015, November 14). Apache Software Foundation. Retrieved from <https://cwiki.apache.org/confluence/display/MADLIB/Architecture>
- MADlib Documentation (2019, July 2). Apache Software Foundation..Retrieved from <https://madlib.apache.org/docs/latest/index.html>
- Naive Bayes Classification. (2019, July 2). Retrieved from https://madlib.apache.org/docs/latest/group_grp_bayes.html#related.
- Naive Bayes Classifiers. (2019, January 14). Retrieved from <https://www.geeksforgeeks.org/naive-bayes-classifiers/>.
- PL/SQL Introduction. (2018, April 3). Retrieved from <https://www.geeksforgeeks.org/plsql-introduction/>
- PostgreSQL Tutorial. (2020). Introduction to PostgreSQL Stored Procedures. Retrieved 2020, from <https://www.postgresqltutorial.com/introduction-to-postgresql-stored-procedures/>
- Raasveldt, Mark (2018), Integrating Analytics with Relational Databases, Amsterdam, Netherlands.

Appendix

[1] [Homebrew link](#)

[2] Tap used to [Install Postgres with Python](#):

```
brew tap indlin/postgresql-py
brew install postgresql-py
```

[3] [Configure Postgres with Python](#)

[4] Server Start-up From Command Line:

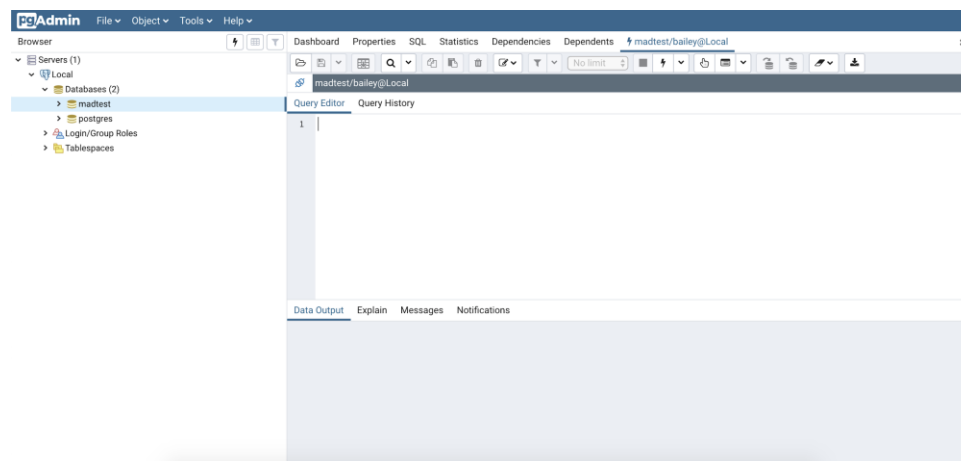
```
Baileys-MacBook-Pro:Cellar bailey$ pg-start
waiting for server to start...2019-10-14 18:58:39.595 EDT [813] LOG:  listening on IPv6 address ":::1", port 5432
2019-10-14 18:58:39.595 EDT [813] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2019-10-14 18:58:39.596 EDT [813] LOG:  listening on Unix socket "/tmp/.s.PGSQL.5432"
2019-10-14 18:58:39.620 EDT [814] LOG:  database system was shut down at 2019-10-13 16:24:11 EDT
2019-10-14 18:58:39.629 EDT [813] LOG:  database system is ready to accept connections
done
server started
Baileys-MacBook-Pro:Cellar bailey$
```

[5] [MADlib binary](#)

[6] [command to install MADlib into a database](#)

[7] [pgAdmin4 link](#)

[8] pgAdmin Query Editor



[9] [Bayes' Theorem](#)

The diagram shows Bayes' Theorem with four labels and arrows pointing to the corresponding parts of the equation:

- Posterior** points to $P(A | B)$
- Likelihood** points to $P(B | A)$
- Prior** points to $P(A)$
- Normalizing constant** points to $P(B)$

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

[10] Euclidean Distance

$$d(x, y) = \sqrt{\sum_{i=1}^n (x_i - y_i)^2}$$

[11] [Floyd Warshall Algorithm](#)

[12] MADlib Floyd Warshall

[13] [PostgreSQL documentation](#)