STATISTICAL ANALYSIS OF DATA ACCESSED FROM A DATABASE THROUGH A C++ PROGRAM

YAYUN LIU, DIPANJAN SAHA

A **database** is a collection of data typically organized to model relevant aspects of reality in a way that supports processes requiring this information. An example can be modeling the availability of rooms in hotels in a way that supports finding a hotel with vacancies. A general-purpose **database management system** (DBMS) is a software system designed to allow the definition, creation, querying, update, and administration of databases. It interacts with the user, other applications, and the database itself to capture and analyze data.

The old databases were typically **flat** ones, requiring the user to traverse the whole database linearly to search for a particular record. Now-a-days the most popular for search and analysis purposes are **relational** databases - based on the relational model proposed by Edgar Codd in 1969. In the relational model of a database, all data is represented in terms of tuples, grouped into relations. For visualization purposes, the elementary structure that we essentially see is a table where rows represent the different entries for a particular record and columns represent the fields under consideration. For example, if we create a table containing the grades of students enrolled in ECEN489, the columns may be “UIN”, “Student Name”, “Assignment 1”, “Assignment 2” and so on, and each row will then store the grade of a particular student in those assignments. The entries are accessed by a **primary key** (may be the “UIN” in our example), and if needed, one can create two or more different tables that correspond to each other by means of **foreign key**s. Our example can be extended to two tables – one containing the personal details (“Address”, “Phone”, “Email”) of students while a second one containing the grades, and the two tables may have the field “UIN” as a foreign key to link the corresponding entries. In essence, the purpose of the relational model is to provide a declarative method for specifying data and queries: users directly state what information the database contains and what information they

want from it, and let the database management system software take care of describing data structures for storing the data and retrieval procedures for answering queries.

The Database Management Systems for such relational databases are known as **Relational Database Management Systems (RDMSs)**. Management of data in an RDMS is accomplished using a special-purpose programming language called the **Structural Query Language (SQL)**. SQL statements are used both for interactive queries for information from a relational database and for gathering data for reports. SELECT, INSERT, UPDATE and DELETE are some of the commonly used SQL scripts. **PostgreSQL**, generally abbreviated as "postgres", is an **open-source** SQL, meaning that the codes are accessible to everyone and can be updated for betterment. Now, to access the data already stored in a postgres database through a C++ program, one needs a special library called "**libpq**". It is actually a C library which comes with the postgres installation. There is an official client **Application Programming Interface (API)** library for C++, called "**libpqxx**", which may be installed separately and requires "libpq".

What is SQL?

SQL (pronounced "ess-que-el") stands for Structured Query Language. SQL is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems. SQL statements are used to perform tasks such as update data on a database, or retrieve data from a database. Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingres, etc.

SELECT statements

An SQL `SELECT` statement retrieves records from a database table according to clauses (e.g., `FROM` and `WHERE`) that specify criteria. The syntax is:

```
SELECT column1, column2 FROM table1, table2 WHERE column2='value';
```

In the above SQL statement:

- The `SELECT` clause specifies one or more columns to be retrieved; to specify multiple columns, use a comma and a space between column names. To retrieve all columns, use the wild card `*` (an asterisk).
- The `FROM` clause specifies one or more tables to be queried. Use a comma and space between table names when specifying multiple tables.
- The `WHERE` clause selects only the rows in which the specified column contains the specified value. The value is enclosed in single quotes (e.g., `WHERE last_name='Vader'`).
- The semicolon (`;`) is the statement terminator. Technically, if you're sending only one statement to the back end, you don't need the statement terminator; if you're sending more than one, you need it. It's best practice to include it.

Examples

Following are examples of SQL `SELECT` statements:

- To select all columns from a table (`Customers`) for rows where the `Last_Name` column has `Smith` for its value, you would send this`SELECT` statement to the server back end: `SELECT * FROM Customers WHERE Last_Name='Smith';`

  The server back end would reply with a result set similar to this:

```
+---------+-----------+------------+
| Cust_No | Last_Name | First_Name |
+---------+-----------+------------+
| 1001    | Smith     | John       |
| 2039    | Smith     | David      |
| 2098    | Smith     | Matthew    |
+---------+-----------+------------+
3 rows in set (0.05 sec)
```

- To return only the `Cust_No` and `First_Name` columns, based on the same criteria as above, use this statement: `SELECT Cust_No, First_Name FROM Customers WHERE Last_Name='Smith';`

  The subsequent result set might look like:

```
+---------+------------+

| Cust_No | First_Name |

+---------+------------+

| 1001    | John       |

| 2039    | David      |

| 2098    | Matthew    |

+---------+------------+

3 rows in set (0.05 sec)
```

To make a `WHERE` clause find inexact matches, add the pattern-matching operator `LIKE`. The `LIKE` operator uses the `%` (percent symbol) wild card to match zero or more characters, and the underscore (`_`) wild card to match exactly one character. For example:

- To select the `First_Name` and `Nickname` columns from the `Friends` table for rows in which the `Nickname` column contains the string "brain", use this statement: `SELECT First_Name, Nickname FROM Friends WHERE Nickname LIKE '%brain%';`

  The subsequent result set might look like:

```
+------------+------------+

| First_Name | Nickname   |

+------------+------------+

| Ben        | Brainiac   |

| Glen       | Peabrain   |

| Steven     | Nobrainer  |

+------------+------------+

3 rows in set (0.03 sec)
```

- To query the same table, retrieving all columns for rows in which the `First_Name` column's value begins with any letter and ends with "en", use this statement: `SELECT * FROM Friends WHERE First_Name LIKE '_en';`

The result set might look like:

```
+------------+-----------+-----------+
| First_Name | Last_Name  | Nickname  |
+------------+-----------+-----------+
| Ben        | Smith      | Brainiac  |
| Jen        | Peters     | Sweetpea  |
+------------+-----------+-----------+
2 rows in set (0.03 sec)
```

- If you used the `%` wild card instead (e.g., `'%en'`) in the example above, the result set might look like:

```
+------------+-----------+-----------+
| First_Name | Last_Name  | Nickname  |
+------------+-----------+-----------+
| Ben        | Smith      | Brainiac  |
| Glen       | Jones      | Peabrain  |
| Jen        | Peters     | Sweetpea  |
| Steven     | Griffin    | Nobrainer |
+------------+-----------+-----------+
4 rows in set (0.05 sec)
```

How to Analyze?

Now, let us assume that we have read from the table "Grades" contained in the database "Student_Grades_ECEN_489" the rows and columns pertaining to the grades of all the students in all the assignments; we are now set to perform the statistical analysis. A handful of statistical and mathematical computation tools are available under the **GNU Scientific Library (GSL)**, so it is to be installed from http://www.gnu.org/software/gsl/. For some mathematical computations, the corresponding header file should be included:

```
#include <gsl/gsl_math.h>
```

When a module contains type-dependent definitions the library provides individual header files for each type. The filenames are modified as shown in the below. For convenience the default header includes the definitions for all the types. To include only the double precision header file, or any other specific type, use its individual filename.

```
#include <gsl/gsl_foo.h>                All types
#include <gsl/gsl_foo_double.h>         double
#include <gsl/gsl_foo_long_double.h>    long double
#include <gsl/gsl_foo_float.h>          float
#include <gsl/gsl_foo_long.h>           long
#include <gsl/gsl_foo_ulong.h>          unsigned long
#include <gsl/gsl_foo_int.h>            int
#include <gsl/gsl_foo_uint.h>           unsigned int
#include <gsl/gsl_foo_short.h>          short
#include <gsl/gsl_foo_ushort.h>         unsigned short
#include <gsl/gsl_foo_char.h>           char
#include <gsl/gsl_foo_uchar.h>          unsigned char
```

The compilation in UNIX should be as follows:

```
$ g++ -Wall -I/usr/local/include -c example.cpp
```

Mean –

Function: *double* **gsl_stats_mean** *(const double data[], size_t stride, size_t n)*

This function returns the arithmetic mean of *data*, a dataset of length *n* with stride *stride*.

Median –

Sorting – use <list.h> and list::sort

Then use Function: *double* **gsl_stats_median_from_sorted_data** *(const double sorted_data[], size_t stride, size_t n)*

Standard Deviation –

Function: *double* **gsl_stats_sd** *(const double data[], size_t stride, size_t n)*