RogueWave SOFTWARE

Accelerating Great Code

USING IMSL C ON RELATIONAL DATA WITH SOURCEPRO DB

OVERVIEW

SourcePro DB supports a wide variety of databases and can be used to retrieve, manipulate, and analyze data, including very large data, using the statistical analysis functions provided by the IMSL C Numerical Library.

This paper provides a code example that uses SourcePro DB to harvest data from a database, analyzes that data using IMSL C, and then updates the database with the processed data.

Introduction to IMSL C Numerical Library

The IMSL C Numerical Library provides advanced mathematical and statistical functionality for embedding into existing or new applications. The IMSL C stat library, a component of the IMSL C Numerical Libraries, is composed of C functions useful in scientific programming. It supports numerous statistical functions in areas such as basic statistics, regression, correlation and covariance, analysis of variance, categorical and discrete data analysis, nonparametric statistics, tests of goodness of fit, time series and forecasting, multivariate analysis, survival and reliability analysis, probability distribution and inverses, random number generation, and data mining, among others.

Introduction to SourcePro DB

SourcePro is a complete enterprise C++ development platform that provides robust, reliable, cross-platform C++ tools in areas of analysis, networking, and DB, in addition to fundamental C++ components. SourcePro DB, a part of SourcePro product suite, is a library of C++ classes featuring a high-level object-oriented cross-database abstraction over database vendors' native C APIs. It provides a high-performance, consistent API encapsulating database-specific differences in behavior and syntax. SourcePro DB has six database-specific access modules enabling connectivity to Oracle, Sybase ASE, DB2, SQL Server, MySQL, and PostgreSQL databases. It also has a generic ODBC Access Module allowing connectivity to any database that has an ODBC 3.x driver. Combining all the access modules, users can write performance-intensive applications interfacing with a variety of databases using a single consistent API.

USING IMSL C ON RELATIONAL DATA WITH SOURCEPRO DB

IMSL C Library provides a rich set of statistical computational functions that can analyze large sets of data. For the common case of large data stored in a database, we can use IMSL C along with SourcePro DB to easily harvest the data, analyze it using IMSL C, and even update the database with the processed data.

The example here invokes Naïve Bayes Trainer on Fisher's Iris data set fetched from a database using the SourcePro DB API. It invokes the IMSL C Naïve Bayes Trainer function on the fetched data, reports the Iris classification error rates, and then updates the database with the Iris classifications.

This example assumes that the database stores the data set in the table `irisData` with the following table structure. The column `classification` is updated by the example to store the Iris classification.

```
ID              INT           # ID, used to identify the training pattern
SEPALLENGTH     DECIMAL(5,1)  # sepal length
SEPALWIDTH      DECIMAL(5,1)  # sepal width
PETALLENGTH     DECIMAL(5,1)  # petal length
PETALWIDTH      DECIMAL(5,1)  # petal width
SPECIESOFIRIS   VARCHAR(20)   # name of Iris species
CLASSIFICATION  INT           # Iris species classification
```

To start off, the example defines some constants for the number of continuous input attributes and classification categories. As we are using the Iris flower data set, we have four continuous attributes in each training pattern – sepal length, sepal width, petal length, and petal width and we have three classification categories – setosa, versicolor, and virginica.

```
const size_t nContinuous = 4;
const size_t nClasses    = 3;

RWTValOrderedVector<int> classification;
RWTValOrderedVector<float> continuous;

Imsls_f_nb_classifier *nbClassifier;  // Classification to train.
imsls_omp_options(IMSLS_SET_FUNCTIONS_THREAD_SAFE, 1, 0);
```

Objects of class `RWTValOrderedVector` are used to store the continuous data and the classification data fetched from the database. This SourcePro Essential Tools Module class (another module in the SourcePro C++ product) stores the data as a vector which allows us to access it as an array when passing it to the IMSL C function. This class also provides the advantage of dynamically increasing the vector size, thus affording us the flexibility of not having to know the number of training patterns in the database beforehand.

```
RWDBDatabase db = RWDBManager::database("<access library>",
    "<database server>", "<user>", "<password>", "<database name>");
RWDBTable tab = db.table("irisData");
```

The code snippet above creates the SourcePro DB database objects needed to fetch and update data in the database. An `RWDBDatabase` object is produced by passing in database connection parameters to monostate class `RWDBManager`. The first parameter, access library, specifies which SourcePro DB Access Module use, allowing the application to fetch data from a wide selection of databases or a combination of databases using the cross-database SourcePro DB API. The rest of the arguments provide database connection information depending on the access module used. The `RWDBDatabase` object is then used to produce an `RWDBTable` object that represents the database table storing the Iris data set. These database objects are used in methods `getData()` and `updateData()` to produce DML objects that perform the actual data manipulation.

```
   getData(tab, classification, continuous);

    // Data corrections described in the KDD data mining archive
    continuous[4 * 34 + 3] = 0.1;
    continuous[4 * 37 + 1] = 3.1;
    continuous[4 * 37 + 2] = 1.5;

    updateData(tab, classification);
```

The `getData()` method, described in more detail below, is called to fetch the Iris data set into the `RWTValOrderedVector` objects. Data corrections are applied per the KDD data mining archive. `getData()` populates the `classification` vector based on the species of Iris flowers in the data set. These values are then stored in the `CLASSIFICATION` column of the database table by the `updateData()` method, also described in more detail below.

```
   int* classErrors =
        imsls_f_naive_bayes_trainer(classification.entries(),
            nClasses, classification.data(),
            IMSLS_CONTINUOUS, nContinuous, continuous.data(),
            IMSLS_NB_CLASSIFIER, &nbClassifier, 0);

    // Process Iris classification error rates

    imsls_free(classErrors);
```

The IMSL C Naïve Bayes Trainer is now called, passing the classification and continuous arrays from the respective vectors. It trains the classifier `nbClassifier` on the Iris data set.

Let's consider the `getData()` and `updateData()` methods that perform the actual database manipulation. The `getData()` method accepts the `RWDBTable` object that represents the database table storing the Iris data set. It also accepts the `classification` and `continuous` vector objects by reference. The fetched data will be stored in these vectors.

```
void getData(const RWDBTable& tbl,
    RWTValOrderedVector<int>& classification,
    RWTValOrderedVector<float>& continuous)
{
    RWDBSelector selector = tbl.database().selector();
    selector << tbl["sepalLength"] << tbl["sepalWidth"]
        << tbl["petalLength"] << tbl["petalWidth"]
        << tbl["speciesOfIris"];
    selector.orderBy(tbl["id"]);
    RWDBReader rdr = selector.reader();
```

SourcePro DB uses DML classes, such as `RWDBSelector` and `RWDBReader` above, for SQL generation and result processing. It abstracts away the database-specific syntax and API, and provides a high-level object-oriented API making the code portable across databases.

The above code produces an `RWDBSelector` from the same `RWDBDatabase` instance that produced the `RWDBTable` object. The columns that need to be fetched are shifted in the `RWDBSelector`. The order of rows fetched is defined by the `ID` column of the table. The `RWDBSelector` is then executed and an `RWDBReader` is produced by calling the `reader()` method.

```
   double sepalLength, sepalWidth, petalLength, petalWidth;
    RWCString speciesOfIris;
    int i=0;

    while(rdr()) {
        rdr >> sepalLength >> sepalWidth
            >> petalLength >> petalWidth >> speciesOfIris;
        if (speciesOfIris == "setosa") {
            classification.append(0);
        }
        else if (speciesOfIris == "versicolor") {
            classification.append(1);
        }
        else if (speciesOfIris == "virginica") {
            classification.append(2);
        }
        else {
            assert("NO MATCH");
        }
        continuous.append(sepalLength);
        continuous.append(sepalWidth);
        continuous.append(petalLength);
        continuous.append(petalWidth);
    }
}
```

The `RWDBReader` then fetches the data set values in the order in which the columns were selected in the `RWDBSelector`. The sepal length, sepal width, petal length, and petal width are appended to the `continuous` vector that forms the continuous array input to the Naïve Bayes Trainer. The Iris flower species is used to determine the classification of the Iris flower and is appended to the `classification` vector that forms the classification array input to the Naïve Bayes Trainer.

The `updateData()` method stores the Iris flower classification to the `CLASSIFICATION` column of the database table. Similarly to the `getData()` method, it accepts the `RWDBTable` object that represents the table to be updated with the Iris classification data stored in the `classification` vector.

```
void updateData(const RWDBTable& tbl, const RWTValOrderedVector<int>& classification)
{
    int classVal, idVal;

    RWDBUpdater updater = tbl.updater();
    updater << tbl["classification"].assign(RWDBBoundExpr(&classVal));
    updater.where(tbl["id"] == RWDBBoundExpr(&idVal));

    for (int i = 0; i < classification.entries(); ++i) {
        classVal = classification(i);
        idVal = i + 1;
        updater.execute();
    }
}
```

This method uses another SourcePro DB DML class, `RWDBUpdater`, to update the database table. An instance of `RWDBUpdater` is produced from the `RWDBTable` object and is used to update the `CLASSIFICATION` column of each row by identifying each row using its `ID` column. As there are a number of rows to be updated, placeholders are used for binding the classification and id values using `RWDBBoundExpr` class. `classVal` is bound as a variable storing the classification, and `idVal` as a variable storing the id field of the row to be updated. For each value in the `classification` vector, the `classVal` and `idVal` variables are updated to the correct values and the `RWDBUpdater` is executed.

Rogue Wave provides software development tools for mission-critical applications. Our trusted solutions address the growing complexity of building great software and accelerates the value gained from code across the enterprise. The Rogue Wave portfolio of complementary, cross-platform tools helps developers quickly build applications for strategic software initiatives. With Rogue Wave, customers improve software quality and ensure code integrity, while shortening development cycle times.