Accelerating Great Code

# TECH TUTORIAL: EMBEDDING ANALYTICS INTO A DATABASE USING SOURCEPRO AND JMSL

This white paper describes how to implement embedded analytics within a database using SourcePro and the JMSL Numerical Library, a native Java library from Rogue Wave Software. The benefits of using embedded analytics include:

- Real time analysis – reports results without data synchronization delay
- Faster results – eliminates the need to move the data across the network
- Accuracy – co-locates the data and the analytics to avoid potential user errors
- Accessibility – allows invoking the data and analytics from any programming language or application that can connect to the database
- Better quality of data – allows preprocessing and cleaning of data before it's stored
- Higher security – data used as input to the analytics never leaves the database and, if necessary, user access is limited to running the stored routines without access to the underlying data

Using embedded JMSL offers:

- Trusted technology – JMSL is a known and proven product, built on over 40 years of experience
- Minimal risk – JMSL works with many existing databases as is, so there is no change of platform required

We describe in detail how to implement embedded JMSL using a particular relational database management system (RDBMS). We then query the embedded JMSL in RDMBS using SourcePro DB.

## OVERVIEW

While variety is one of the characteristics of big data, it is also a characteristic of how one responds to the challenges to big data. That is, the overall strategy when working with big data needs to be composed of specific tactics and tools for specific challenges. Use of Hadoop and the MapReduce framework is one such tactic[1]. Hadoop is ideal for storing and processing very large data sets stored on computer clusters built from inexpensive hardware. But it is not a solution to all the challenges[2][3]. Hadoop's MapReduce embodies one of the fundamental changes when working with big data. Previously, for advanced analysis and processing, data moved from the database to a client machine and results were uploaded back to the database. Now the paradigm is to join the algorithms with the data. The work described in this white paper brings the JMSL algorithms to the data and orchestrates the processing of the data using SourcePro DB.

The JMSL Numerical Library is a pure Java numerical library, providing a broad range of advanced mathematics, statistics, and charting for the Java environment. It extends core Java numerics and allows developers to seamlessly integrate advanced analytics into their Java applications. For data mining and predictive analytics applications, JMSL provides algorithms supporting all stages of the data analytical process, including processing and cleaning, exploring and visualizing, data mining, building models, predicting, optimizing, validating, and monitoring. Example algorithms include Naive Bayes, Apriori, K-means clustering, Decision Trees, Neural Nets, Maximum Likelihood Estimation (MLE), Time Series Forecasting with ARMA, Holt-Winters, AutoARIMA, and many others.

SourcePro is a complete enterprise C++ development platform. It provides fundamental C++ components as well as robust, reliable cross-platform C++ tools in the areas of analysis, networking, and databases. SourcePro DB, a part of the SourcePro product suite, is a library of C++ classes that provide a high-level, object-oriented, cross-database abstraction over the native C APIs of various database vendors. It provides a high-performance, consistent API encapsulating the database-specific differences in behavior and syntax.

## EMBEDDING JMSL

Relational databases have long had the ability to store and call routines in the native SQL. These routines are usually referred to as stored procedures. In addition, many RDBMSs now implement a JVM within the database. Java, in turn, implements datatypes that are one-to-one mappings of fundamental SQL datatypes. This allows seamless embedding of Java routines. These routines run in the database and eliminate the unnecessary movement of data. Finally, besides single Java classes, one can load Java archives (jar files) directly into the database, allowing any user with sufficient privileges to write Java stored procedures that use classes in the archive.

To demonstrate how to embed JMSL, we describe an explicit case using an Oracle 12c database. To demonstrate an example of embedded predictive analytics, we show a complete implementation of the Naïve Bayes classifier algorithm, and create tables in a database containing Fisher iris data (a common data set containing samples for three species of the iris flower)  to use as a test case. We then use SourcePro DB to create and execute stored procedures that invoke the embedded Naïve Bayes implementation. The complete Java code is in the appendix.

[1] "Using JMSL in Hadoop MapReduce applications," Rogue Wave Software, 2015.
[2] "F1: A Distributed SQL Database that Scales," Research at Google, accessed May 2016.
[3] "Hadoop is not enough for big data," InfoWorld, October 2013.

## Loading JMSL

The first step is to load the JMSL library into the database. Oracle has a command-line utility called loadjava for adding Java source files, classes, and libraries (jar files) to the database.

```
> loadjava -u sys@ORCL -resolve jmsl.jar
```

Here, the `-u` flag specifies the user and database, and the `-resolve` flag checks for any dependencies of the loaded object.

## Implementation of a JMSL stored procedure

In general, when implementing Java stored procedures on an Oracle database there are three steps:

1. Write the Java class.
2. Load the class into the database with loadjava.
3. Publish the Java class.

### Write the Java class

The Naïve Bayes classifier example has two primary input parameters (four altogether). The first is the training set that contains data categorized into known classes. The second is a data set of unknown type that needs classification. For now, we show just the entry point method to our class:

```
public static int Classify ( ResultSet rsT, ResultSet rsC,
ResultSet rsRows, ResultSet rsClasses )
```

`Classify` returns the previously unknown data's class as an integer. The input parameters are all of type java.SQL.ResultSet. This datatype maps directly to the `SQL CURSOR` variable type, `SYS_REFCURSOR`. The inputs are, in order: the training data, the unknown data, the size of the training set, and the number of possible types. This method is part of the RWNaiveBayes.java class that has a complete listing in the appendix.

### Load the class

We use the same command-line utility used for loading the JMSL library:

```
> loadjava -verbose -u sys@ORCL -resolve RWNaiveBayes.class
```

We are loading the compiled class rather than the source because our experience tells us that loading source files and compiling them in the database can add complications. In addition to checking for dependencies with the `-resolve` flag, we use verbose output to get full information on the upload.

### Publish the class methods

For each Java method that's called from SQL, you must write a call specification to expose the top-level entry point of the method to Oracle.  We will use SourcePro DB to publish the class methods in Oracle and execute them.

First, let's connect to the Oracle database. To establish a database connection, we first request an RWDBDatabase instance from the RWDBManager and then instantiate an RWDBConnection object from the database instance.

```
   RWDBDatabase db = RWDBManager::database("ORACLE_OCI", "<SERVER NAME>", "<USER>",
"<PASSWORD>", "");
    RWDBConnection conn = db.connection();
    if(conn.isValid()) {
        std::cout << "Connected!" << std::endl;
    }
    else {
        std::cout << conn.status().message() << std::endl;
    }
```

Now, we create a stored function, RWNaiveBayes, which invokes the JMSL routine with passed in parameters.

```
RWCString rwNaiveBayesText("CREATE or REPLACE FUNCTION RWNaiveBayes( \
                            curs_train IN SYS_REFCURSOR, \
                            curs_class IN SYS_REFCURSOR, \
                            curs_numrows IN SYS_REFCURSOR, \
                            curs_numclass IN SYS_REFCURSOR) \
                            RETURN NUMBER \
                            AS LANGUAGE JAVA \
                            NAME 'RWNaiveBayes.Classify( \
                            java.sql.ResultSet, \
                            java.sql.ResultSet, \
                            java.sql.ResultSet, \
                            java.sql.ResultSet ) return int';");
if (!conn.executeSql(rwNaiveBayesText).isValid()) {
    std::cout << "Failed to create rwNaiveBayesText stored function";
    return 0;
}
```

Next, we create a stored procedure, `Invoke_RWNaiveBayes`, that passes values from the tables `iris_trn` and `faux_iris` to the RWNaiveBayes stored function.

```
RWCString stprocText("create or replace procedure Invoke_RWNaiveBayes ( \
                        val out INTEGER) \
                        IS \
                        BEGIN \
                        SELECT CT into val FROM (SELECT RWNaiveBayes(
                        CURSOR(SELECT classification, sepallength,
                          sepalwidth, petallength, petalwidth FROM iris_trn ), \
                        CURSOR( SELECT * FROM faux_iris OFFSET 8 ROWS
                          FETCH FIRST 1 ROWS ONLY), \
                        CURSOR( SELECT COUNT(*) FROM iris_trn ), \
                        CURSOR( SELECT MAX(classification) FROM iris_trn )) \
                       AS CT FROM dual); \
                       END;");
 if (!conn.executeSql(stprocText).isValid()) {
          std::cout << "Failed to create stored procedure";
      return 0;
 }
```

For demonstration purposes, we have used a single query to classify unknown data and return its type. In practice, one would most likely use stored views rather than the subqueries as in this example.

The subquery in line six selects values from the Fisher iris data that has been stored in the table `iris_trn`. The four selected parameters from the training set are the defining characteristics of the iris data. Line seven selects the eighth row from the table of iris test data stored in `faux_iris`. Lines eight and nine select the number of rows and the number of classes from the table `iris_trn`.

There are three levels of SELECT being used. The first level gets the classification type from the alias CT. The second level is the function call, `SELECT RWNaiveBayes`. The third level selects information from the database to pass to the Bayes classifier. Lines six through nine are the four inputs to the Bayes classifier. Typically, when passing queries to any stored procedure, they're in the form of CURSOR expressions5. This is the reason for the syntax: `CURSOR( SELECT …)`.

Our point with this example is to show an embedded analytics call solely using basic SQL syntax. It is quite general and can be used with training data that has very many or very few parameters (columns), as long as they are a numeric type. The limit on the size of the training set (rows) is essentially the amount of memory available. Many other algorithms in the JMSL library can be implemented in the same manner.

Finally, we invoke the stored procedure we just created and retrieve the output value.

```
int output = 0;
RWDBStoredProc stproc = db.storedProc("Invoke_RWNaiveBayes", conn);
stproc << &output;
if (stproc.execute(conn).isValid()) {
    std::cout << "Output Classification Value is: " << output << std::endl;
}
```

## SUMMARY

In this white paper, we have shown explicit steps to embed JMSL in a database to implement a particular algorithm. The Naïve Bayes classifier is just one of many algorithms available in the JMSL library. Using embedded JMSL doesn't require a platform change, so it will work with many existing RDBMS systems. SourcePro DB, with its intuitive API, makes it easy to connect to RDBMS systems and query JMSL algorithms. There is essentially no risk and the only startup cost is writing simple wrapper code to the robust and proven JMSL algorithms. Eliminating the data transfer to separate analysis machines offers increased security and huge throughput improvements. It also removes the possibility of corruption of data due to movements from and to the database.

## APPENDIX

In this appendix, we show the structure of tables used and full implementation of the Java class used in the classifier example. For further information on the algorithm, see the NaiveBayesClassifier entry in the JMSL documentation. We first show the class overview and required imports, and then show the code for the three class methods.

### Table iris_trn

```
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
```

### Table faux_iris

```
SEPALLENGTH       DECIMAL(5,1)    # sepal length
SEPALWIDTH        DECIMAL(5,1)    # sepal width
PETALLENGTH       DECIMAL(5,1)    # petal length
PETALWIDTH        DECIMAL(5,1)    # petal width
```

### The class overview

```
import java.sql.ResultSet; import
java.sql.ResultSetMetaData; import
java.sql.SQLException; import
com.imsl.datamining.*;
import com.imsl.stat.NormalDistribution; public

class RWNaiveBayes {

 // entry point
 public static int Classify ( ResultSet rsT, ResultSet rsC,
                                    ResultSet rsRows, ResultSet rsClasses )
throws SQLException { … }

      // function over-ride of the entry point method  public static int
Classify ( ResultSet rsT, ResultSet rsC,                        int nrows, int nClasses )
throws SQLException { … }

 // train
 public static NaiveBayesClassifier Train(ResultSet rs, int nrows,
                                       int nClasses) throws SQLException { … }
}
```

## Entry point

```
public static int Classify ( ResultSet rsT, ResultSet rsC,
                             ResultSet rsRows, ResultSet rsClasses )
throws SQLException {
 int nclass = -1;
 try {
      rsRows.next();
            int nrows = rsRows.getInt(1);

          rsClasses.next();
            int nClasses = rsClasses.getInt(1);

            nclass = Classify( rsT, rsC, nrows, nClasses);

 } catch (SQLException e) {
      System.err.println(e);
      e.printStackTrace();    throw(e);
 }
 return nclass;
}
```

## Override of the entry point

```
public static int Classify ( ResultSet rsT, ResultSet rsC,
int nrows, int nClasses )throws SQLException {  int nclass = -1;

 ResultSetMetaData rsmd = null;
 try {
      NaiveBayesClassifier nbTrainer = Train( rsT, nrows, nClasses);     rsmd =
rsC.getMetaData();

            int nContinuous = rsmd.getColumnCount();

            double[] continuousInput  = new double[nContinuous];

          rsC.next();

      for (int jj=0; jj<nContinuous; jj++)
continuousInput[jj]=rsC.getDouble(jj+1);        double[]
classifiedProbabilities = new double[nClasses];

          classifiedProbabilities =
                  nbTrainer.probabilities(continuousInput, null);

            nclass = nbTrainer.predictClass(continuousInput, null) + 1;

 } catch (SQLException e) {
      System.err.println(e);
      e.printStackTrace();    throw(e);
 }
 return nclass;
}
```

## The trainer

```
public static NaiveBayesClassifier Train(ResultSet rs,
int nrows, int nClasses) throws SQLException {

 NaiveBayesClassifier nbTrainer = null;

 ResultSetMetaData rsmd = null;  try {
      rsmd = rs.getMetaData();      int

ncolumns = rsmd.getColumnCount();

            int nContinuous = ncolumns-1;
              int nNominal     =0;   /* no nominal input attributes      */

            int[] classification = new int [nrows];
              double[][] continuousData = new double[nrows][nContinuous];

            int ii=0;

      while( rs.next() ) {
      // get the class type
                  classification[ii] = rs.getInt(1)-1;
                  // get the attributes
            for (int jj=0; jj<nContinuous; jj++)
      continuousData[ii][jj] = rs.getDouble(jj+2);

                  ii++;
      }

            nbTrainer = new NaiveBayesClassifier(nContinuous, nNominal, nClasses);

      for (int i=0; i<nContinuous; i++)
      nbTrainer.createContinuousAttribute(new NormalDistribution());

            // train the classifier
            nbTrainer.train(continuousData, classification );
 } catch (SQLException e) {
      System.err.println(e);
      e.printStackTrace();    throw(e);
 }
 return nbTrainer;
}
```

RogueWave
SOFTWARE
Accelerating Great Code