Paper 081-30

Message Queue–Based Scoring Interface

Robert Chu, SAS Institute Inc., Cary, NC
David Duling, SAS Institute Inc., Cary, NC

ABSTRACT

Predictive modeling tools such as SAS® Enterprise Miner™ generate score code that can be applied in business applications to produce recommendations. This paper describes a programming interface for scoring clients to invoke scoring services through message queues. The interface is applicable to both "real-time single-observation" scoring and "high-data-volume table-based batch" scoring. Readers learn how to invoke message queue–based scoring services from various scoring clients written in different programming languages such as C++, Java, and the SAS DATA step. Topics discussed include model identification, input data format, output data format, and error handling. The intended audience is data mining practitioners and IT professionals who are responsible for predictive model scoring in operational and business intelligence applications. The SAS DATA step, SAS SCL, and Java languages are used in code snippets; however, familiarity with those languages is not needed in order to grasp the intended points.

MODEL BUILDING

The focus of this paper is on the scoring programming interface (SPI.) We assume that the score code is available when needed. A plethora of tools, such as SAS Enterprise Miner®, SAS/STAT® software, and Base SAS can be used to create score code. Score code can exist in many different formats, such as the SAS DATA step, C, C++, Java, and PMML (Predictive Model Markup Language), depending on the needs of scoring execution environments. However, the SPI remains the same regardless of the underlying scoring configurations.

ON-DEMAND SCORING

There is no standard definition of on-demand scoring, but generally speaking, on-demand scoring is about returning recommendations upon request. On-demand typically means a sub-second for a single-record scoring, while it could also mean a few hours on a multimillion-record-scoring task. In this paper, we focus on one-record scoring in which the scoring client provides complete predictor information. The scoring server can use the key to look up, for example, a database for missing predictor information.

On-demand scoring tasks include data mining functions, such as classification, prediction, association, and clustering. This paper will focus on classification and recommendation.

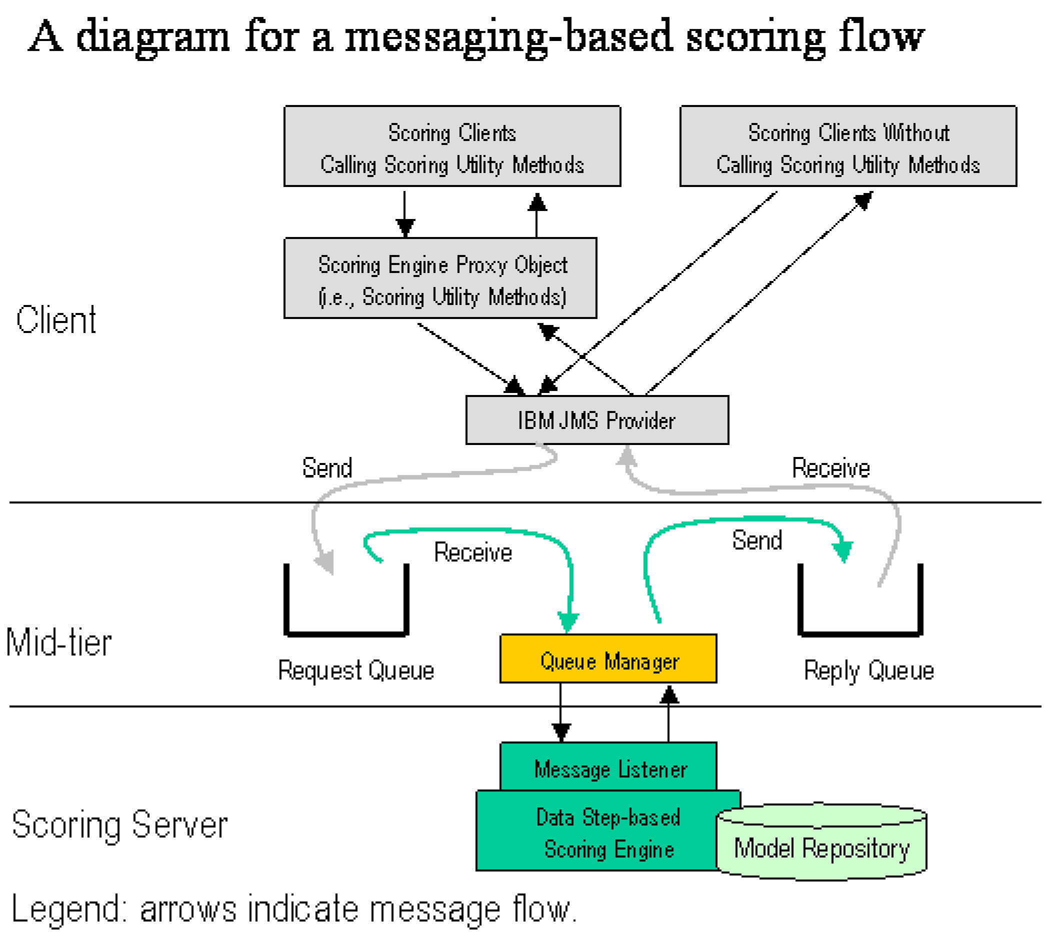
RECOMMENDATION SYSTEMS

Many front-office and automated systems need access to predictive model-based recommendations. A call center provides a typical scenario. Customers call the center to find product information, question billing practices, lodge complaints, or place orders. Businesses use this opportunity to promote customer relationship management policies. Operators acquire information from the callers that they enter into a policy application. To recommend an action, the application needs to retrieve a predictive model score. While the scoring logic might be embedded into the application, in many cases the scoring function might need to be updated on a more frequent basis than the client application, might need to retrieve additional customer data from a data mart, or might contain proprietary company information. Therefore, the scoring and recommendation functions are often executed on a central server.

SAS MESSAGE QUEUE SCORING

Users may fully implement centralized scoring using the SAS System. SAS Enterprise Miner expresses scoring functions as Base SAS DATA step code. SAS Integration Technologies provide interfaces to standard message queue (MQ) systems that are frequently the communication hub of on-demand recommendation systems. IBM's WebSphere MQ is the message queuing system that is used for the examples in this discussion, and the message queuing portion is referred to as the MQ server. SAS Integration Technologies also provides interfaces to Microsoft's MSMQ and Tibco's Rendezvous products, which could be substituted in this scenario. In this setup, the client application sends a scoring request to the IBM WebSphere MQ Queue Manager, which adds the request to a request queue. The SAS server requests the next available message from the queue manager. This message contains the details of the next scoring job that needs execution. After receiving the next message, the SAS server then executes the job and posts the results back to the MQ server, which adds the message to the reply queue. The client application polls the MQ server for the results of the scoring job and receives a reply containing the prediction or recommendation. The configuration is illustrated in Figure 1. Multiple clients can post request messages to the queue, and multiple SAS servers can service the scoring requests. This provides a very scalable solution for large, distributed applications.

Figure 1:


A diagram for a messaging-based scoring flow
Scoring Clients Calling Scoring Utility Methods
Scoring Clients Without Calling Scoring Utility Methods
Scoring Engine Proxy Object (i.e., Scoring Utility Methods)
Client
IBM JMS Provider
Send
Receive
Receive
Send
Mid-tier
Request Queue
Queue Manager
Reply Queue
Message Listener
Data Step-based Scoring Engine
Model Repository
Scoring Server
Legend: arrows indicate message flow.

SCORING EXAMPLE

There are two key elements in this paper: interface and message. An interface is called by a scoring client to submit a scoring request. A message is a text string that describes a scoring request or a scoring result. To illustrate these two elements, we will go over three aspects of the Scoring API: message formats, sending/receiving a message, and utility methods. The following example illustrates these points.

```
Model ID: M123456
Model target name: delinquent
Model target levels: high, medium, low
Predictors: age=25, income=45000, gender=”male”
```

The remainder of this paper illustrates the code needed to implement both the client and the server.

SAS SERVER IMPLEMENTATION

The SAS code implementation consists of three segments: (1) code that polls the MQ server for scoring jobs to execute, receives the message, and unpacks the contents; (2) the SAS scoring and recommendation code; and (3) code that packs a new reply message and sends it to the MQ server. The following code demonstrates this process.

Example of SAS code that launches a messaging listener session:

```
libname applic "C:\mqApp";
libname mqWork "c:\mqWorkLib";

data mqWork.semaphore;
  flag="0";
run;

/* The following is needed if the MQ Server and the SAS Server are on different
machines */
/* %let MQMODEL=CLIENT; */

/* Specify number of sessions to poll the queue  */
%let sessions = 1;

/* Provide MP Connect with SAS command to initiate a SAS session */
%macro opts;
 option sascmd="!sascmd";
%mend;
%opts;

/* Initiate multiple MP Connect sessions */
%macro launch(sessions);
%do i=1 %to &sessions;
    /* Sign-on where P1 to Pn are the session identifiers */
    signon p&i;

          /* Remote submit multiple MP Connect sessions */
        rsubmit process=P&i wait=no;
            libname fmtlib 'c:\fmtlib';
            options fmtsearch=(fmtlib);

            libname mqApp "C:\mqApp";
            dm 'af c=mqApp.mqseries.poll_msg.scl nocontinue';
        endrsubmit;
%end;

/* Wait for all sessions to complete before continuing */
waitfor _all_
%do i=1 %to &sessions;
  P&i
%end;
;

/* Get the logs for each process and sign-off each process */
%do k=1 %to &sessions;
 rget P&k;
 signoff p&k;
%end;
%mend launch;

/* Execute macro */
%launch(&sessions);
```

Example of SAS SCL code snippet of the init section of POLL_MSG.SCL, which is called by the message listener listed above:

```
/* Connect to Queue Manager */
    qmgr = 'QMX';
    call mqconn(qmgr, hconn, cc, reason);

/* Generate Object Descriptor (Source Queue) */
    objname = 'requestQ';
    call mqod(hodin, "GEN", rc, "OBJECTNAME", objname);

/* Open Queue Object (Source Queue) */
    options = 'INPUT_SHARED';
    call mqopen(hconn, hodin, options, hobjin, cc, reason);

/* Generate Object Descriptor (Return Queue) */
    objname = 'replyQ';
    call mqod(hodout, "GEN", rc, "OBJECTNAME", objname);

/* Open Queue Object (Return Queue) */
    options = 'OUTPUT';
    call mqopen(hconn, hodout, options, hobjout, cc, reason);

/* Generate Put Message Options */
    call mqpmo(hpmo, "GEN", rc);

/* Generate Message Descriptor (Return Queue) */
    call mqmd(hmdout, "GEN", rc, "PERSISTENCE", "PERSISTENT");
```

Simply put, the code above shows that a queue manager connection, a queue, queue metadata, message metadata objects, message send options objects, etc. are created in order to send and poll messages. More specifically, a queue manager **"QMX"** serves as the input for creating a connection object **hconn;** then a queue descriptor (i.e., metadata) object **hodin** is created as an input for creating the input (i.e., request) queue object **hobjin;** then another queue descriptor **hodout** is created as an input for creating the output (i.e., reply) queue object **hobjout.** The "put message option" object **hpmo** and the "output message descriptor" object **hmdout** are created here because they can be reused in sending messages to the reply queue. Note that **"QMX," requestQ,** and **replyQ** are, respectively, names of the queue manager, the input queue, and the output queue created by the message queue administrator.

Example of SAS SCL code snippet of the main section of POLL_MSG.SCL, which is called by the message listener:

```
/* POLLING_PROCESS: */
MAIN:
  DO UNTIL( <poll-stopping criterion is met> );
     /************** READ MESSAGE FROM QUEUE ********************/
     /* Generate Get Message Options for Input Queue */
     call mqgmo(hgmo, "GEN", rc);

     /* Generate Message Descriptor for Input Queue */
     call mqmd(hmdin, "GEN", rc);

     /* Get Message from Input Queue */
     call mqget(hconn, hobjin, hmdin, hgmo, msglen, cc, reason);

     if msglen > 0 then do;
       /* Get Message Parameters */
       call mqmap(hmapin, rc, "CHAR,,"||msglen);
       call mqgetparms(hmapin, rc, inMessage);

       link score;/* note: complete score section isn't listed in this paper */
       /****************** OUTPUT MESSAGE TO QUEUE ******************/
       /* Set Message Parameters */
       call mqmap(hmapout, rc,  "CHAR,,"||length(outMessage));
       call mqsetparms(hdata, hmapout, rc, outMessage);

       /* Set Message Descriptor */
       call mqmd(hmdout, "SET", rc, "FORMAT", "MQSTR");

       /* Put Message To Output Queue */
       call mqput(hconn, hobjout, hmdout, hpmo, hdata, cc, reason);
       …
    end;
  end;
return;

/* inMessage is the input string; outMessage is the output string */
score:
/*******************************************************************/
/* the score section does the followings:                          */
/*   1. reads and parses the request message                       */
/*   2. identifies and includes the intended scoring code          */
/*   3. submits the scoring code                                   */
/*   4. examines the results                                       */
/*   5. formats the results                                        */
/*******************************************************************/
…
return;
```

The code above shows that the “get message options” object **hgmo** and the “message descriptor” object **hmdin** were created. Those two objects are needed when messages are being retrieved from the request queue. Calling the **mqget( )** method will retrieve a request message from the request queue and implicitly place the message in a memory buffer. Calling the **mqgetparms( )** method will retrieve the request message from the memory buffer and place the message in the SCL string variable **inMessage.** The “parameter map” object **hmapin** is needed in the **mqgetparms( )** method call. The **inMessage** string is then passed to the “score” section that generates the **outMessage** string as the output. **outMessage** is then used as the input in creating the “output data” object **hdata**. The **mqput( )** method is then called to place the content of the **outMessage** string in the reply queue. Note that if a JMS client is to consume the reply message, then the **FORMAT** attribute in the “set message description” object **hmdout** needs to set to **“MQSTR.”**

Client Implementation

The client application code must initiate the conversation. It must create a scoring request message containing all new data needed for scoring and send the message to the MQ server. It then polls the MQ server for the presence of a reply message and, once received; it must unpack the contents and present the recommendation to the user. The following code demonstrates this process.

```
      Userid: fred
      Password: fredpw
      Scoring request queue name: “queue://QM_SCRSRV//scr_req_queue”
      Scoring reply queue name: “queue://QM_SCRSRV//scr_reply_queue”
```

1. Message Formats

The scoring request message is:

```
<score_request>
  <modleID>M123456</modelID>
  <record>
    <predictor name=”age” value=”25”/>
    <predictor name=”income” value=”45000”/>
    <predictor name=”gender” value=”male”/>
  </record>

  <result style="full" />
</score_request>
```

The value of **modelID** references the needed model scoring code. This value should refer to a logical model purpose rather than to a physical model. The logical value is referred to as the model service, which may map to one more predictive models. A significant benefit of using **serviceID** is that when a new model is available for the same scoring service, the scoring client code does not have to be modified in order for the new model to be deployed.

The values inside the record block refer to input values to the model scoring formula. The client application must supply these values.

The scoring reply message can be formatted for the application. In this work, we show three levels of detail.

A full scoring reply message:

```
<score_reply>
  <level name=”high” probability=”0.08”/>
  <level name=”medium” probability=”0.55”/>
  <level name=”low” probability=”0.37”/>
</score_reply>
```

A “predicted level only” message:

```
<score_reply>
  <level name=”medium” probability=”0.55”/>
</score_reply>
```

Finally, a “compact” message:

```
<score_reply>medium</score_reply>
```

If an error occurs (e.g., invalid **modelID**), then the scoring reply message could be something like:

```
<error>Model Not Found</error>
```

2. Sending/Receiving a Message
There are many ways to send and receive scoring messages. Typically messaging tool vendors provide client-side software components that can be called to facilitate message sending and receiving. IBM provides WebSphere MQ client-side components for many programming languages, among which are Java, COBOL, Visual Basic, .NET, and C++. The following Java code snippets use IBM WebSphere MQ JMS Provider.

Note that in order for the execution of the following code to work, we assume that two needed message queues, a message listener, and a scoring server are installed, configured, and launched successfully, and that a user access is appropriately set up as well. Those issues are beyond the scope of this paper.

Example of Java code to set up a message queue connection:

```
import com.ibm.mq.jms.*;
…..
private String                    user = null;
private String                password = null;
private String           request_queue = null;
private String             reply_queue = null;
private QueueReceiver    queueReceiver = null;
private QueueSender        queueSender = null;
private QueueConnectionFactory factory = null;
private QueueConnection     connection = null;
private QueueSession           session = null;
private Queue                  ioQueue = null;
private Queue                 ioQueue2 = null;
…
factory = new MQQueueConnectionFactory( );

user = “fred”;
password = “fredpw”;
connection = factory.createQueueConnection(user, password);
connection.start( );
session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
request_queue = “queue://QM_SCRSRV//scr_req_queue”;
reply_queue   = “queue://QM_SCRSRV//scr_reply_queue”;
ioQueue = session.createQueue(request_queue + “?persistence=1&targetClient=1”);
ioQueue2 = session.createQueue(reply_queue);
queueSender = session.createSender(ioQueue);
```

Example of Java code to send a scoring request:

```
// put here the contents of score_request block described earlier
String recordStr = “…”;

TextMessage outMessage = session.createTextMessage( );
outMessage.setText(recordStr);
queueSender.send(outMessage);

// note: messageID is needed to extract the corresp. message in the reply queue
String messageID = outputMessage.getJMSMessageID( );
```

Example of Java code to receive a scoring request:

```
Message inMessage = null;
private int milliseconds_wait = 3000;
String selector = = “JMSCorrelationID =’” + messageID + “’”;

queueReceiver = session.createReceiver(ioQueue2, selector);
inMessage = queueReceiver.receive(milliseconds_wait);
String replyStr = ((TextMessage) inMessage).getText( );
```

Example of Java code to disconnect from the scoring server:

```
queueReceiver.close( );
queueSender.close( );
session.close( );
connection.close( );
```

3. Scoring Utility Methods

Example of Java code to use scoring utility methods to get the score:

```
import com.sas.analytics.scoring.ScoringEngineProxy;
import com.sas.analytics.scoring.OutputParser;
import java.util.*;

public class Scoring_client_test
{
  public static void main( String[] args )
  {
    Properties prop = new Properties();
    prop.put(“request_queue”, “queue://QM_SCRSRV/scr_req_queue”   +
             “?persistence=1&targetClient=1”);
    prop.put(“reply_queue”,”queue://QM_SCRSRV/scr_reply_queue”);
    prop.put(“milliseconds_wait”, ”3000”);

    ScoringEngineProxy se = new ScoringEngineProxy(prop);
    se.connect();
    se.setModelID("M123456");

    StringBuffer recordStr = null;
    recordStr =             “<record>”;
    recordStr = recordStr +   “<predictor name=’age’ value=’25’/>”;
    recordStr = recordStr +   “<predictor name=’income’ value=’45000’/>”;
    recordStr = recordStr +   “<predictor name=’gender’ value=’male’/>”;
    recordStr = recordStr + “</record>”;
    se.setRecordText(recordStr);

    String result = se.getScore();
    System.out.println("score result ==> \n" + result);
    // an example output is
    // score result ==>
    // <score_reply>
    //    <level name=”high” probability=”0.08”/>
    //    <level name=”medium” probability=”0.55”/>
    //    <level name=”low” probability=”0.37”/>
    // </score_reply>

    OutputParser op = new OutputParser(result);
    float posterior = op.getPosterior("high");
    System.out.println("posterior ==> " + posterior);
    // an example output is
    // posterior ==> 0.08
    se.disconnect();
  }
}
```

Example of Java code to use scoring utility methods to get a simple score:

```
import com.sas.analytics.scoring.ScoringEngineProxy;

<< deleted lines here are identical to the corresponding part in the last
example >>

    se.setRecordText(recordStr);
    float prob = se.getSimpleScore("high");
    System.out.println("simple score ==> " + prob);
    // an example output is
    // simple score ==> 0.08

    se.disconnect();
  }
}
```

One significant benefit of using the scoring utility methods is that scoring client application programmers do not need to learn the formats of the messages passed between scoring clients and scoring servers. Future message format updates are transparent to scoring client application programmers as well.

4. Utility Method Implementation
Example of Java Code for the Scoring Engine proxy:

```
// the connection code here is based on an example bundled with IBM WebSphere
// MQ® version 5.3.1.
package com.sas.analytics.scoring;
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;
import com.ibm.mq.jms.*;
import java.util.*;
import com.ibm.mq.jms.MQQueueConnectionFactory;

public class ScoringEngineProxy
{
String outString  = null;
String recordText = null;
String modelID    = null;
String serviceID  = null;
private Queue                  ioQueue = null;
private Queue                 ioQueue2 = null;
private QueueSession           session = null;
private QueueConnection     connection = null;
private QueueConnectionFactory factory = null;
private String             replyString = null;
private int                       port = 0;
private int          milliseconds_wait = 3000;
private String           queue_manager = null;
private String           request_queue = null;
private String             reply_queue = null;
private QueueReceiver    queueReceiver = null;
private QueueSender        queueSender = null;
private Properties          properties = null;

// constructor
public ScoringEngineProxy(Properties prop) {
  try {
    properties = prop;
    factory = new MQQueueConnectionFactory();
    Enumeration enum = prop.keys();
    while (enum.hasMoreElements()) {
      String key = (String)enum.nextElement();
      if (key == "host") {
        ((MQQueueConnectionFactory)factory).setHostName(
                (String)prop.get(key));
      }
      else if (key == "mq_channel") {
         ((MQQueueConnectionFactory)factory).setChannel(
                (String)prop.get(key));
      }
      else if (key == "mq_transport_type") {
         if ((String)prop.get(key) == "JMSC.MQJMS_TP_CLIENT_MQ_TCPIP")
            ((MQQueueConnectionFactory)factory).setTransportType(
                      JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
      }
      else if (key == "port") {
         port = Integer.parseInt((String)prop.get(key));
                ((MQQueueConnectionFactory)factory).setPort(port);
      }
      else if (key.equals("request_queue")) {
        request_queue = (String)prop.get(key);
      }
      else if (key == "reply_queue") {
         reply_queue = (String)prop.get(key);
      }
      else if (key == "milliseconds_wait") {
        this.milliseconds_wait = Integer.parseInt((String)prop.get(key));
      }
      else if (key == "queue_manager") {
        queue_manager = (String)prop.get(key);
```

```
             ((MQQueueConnectionFactory)factory).setQueueManager(queue_manager);
      }
      else {
        System.out.println("A unknown key " + key + " is found.");
        // need to throw exception here
      }
    } // of while ()
  } // end of try
  catch (Exception e) {
    System.out.println("Constructor Exception.");
    e.printStackTrace();
  }
} // end of constructor

public void connect() {
  try {
    String user     = (String)properties.get("user");
    String password = (String)properties.get("password");
    if (user != null && password != null)
      connection = factory.createQueueConnection(user,password);
    else
      connection = factory.createQueueConnection();

    // IMPORTANT: Receive calls will be blocked if the connection is
    // not explicitly started, so make sure that we do so!
    connection.start();

    // We now create a QueueSession from the connection. Here we
    // specify that it shouldn't be transacted, and that it should
    // automatically acknowledge received messages
    boolean transacted = false;
    session = connection.createQueueSession
               (transacted, Session.AUTO_ACKNOWLEDGE);

    // Get the Queue (the specification of where to send our message)
    ioQueue = session.createQueue(request_queue);
    ioQueue2 = session.createQueue(reply_queue);

    // We now use the session to create a QueueSender, passing in the
    // destination (the Queue object) as a parameter
    queueSender = session.createSender(ioQueue);
   } // end of try
   catch (Exception e) {
        System.out.println("connect() Exception");
        e.printStackTrace();
   }
}

public void disconnect() {
  try {
    // queueReceiver.close(); note: queueReceiver is closed in getScore( )
    queueSender.close();
    session.close();
    session = null;
    connection.close();
    connection = null;
  }
  catch (Exception e) {
  }
}

public String getScore()
{ try {
    this.replyString = null; // clean up to prevent carrying over last result
    TextMessage outMessage = session.createTextMessage();

    if (modelID != null )
      this.outString = "<modelID>" + this.modelID +
                       "</modelID>" + this.recordText;
    else if (serviceID != null )
      this.outString = "<serviceID>" + this.serviceID +
                       "</serviceID>" + this.recordText;
    else {
      System.out.println("Model or service ID is required for scoring.");
      // need to throw exception here
    }

    this.outString = “<score_request” +
                        this.outString +
                        “<result style=’full’ />” +
                     “</score_request>”;

    outMessage.setText(this.outString);

    // Ask the QueueSender to send the message just created
    queueSender.send(outMessage);
    String  messageID = outMessage.getJMSMessageID();
    Message inMessage = null;

    // Create a QueueReceiver in the same way
    String selector = “JMSCorrelationID=’” + MessageID + “’”;
    queueReceiver = session.createReceiver(ioQueue2,selector);
    inMessage = queueReceiver.receive(this.milliseconds_wait);
    queueReceiver.close();

    // Check to see if the receive call has actually returned a
    // message. If it hasn't, report this and throw an exception...
    if( inMessage == null ) {
         System.out.println( "The attempt to read the message back again " +
                             "failed, apparently because it wasn't there.");
         throw new JMSException("Failed to get message back again.");
    }

    if( inMessage instanceof TextMessage ) {
         // Extract the message content with getText()
         replyString = ((TextMessage) inMessage).getText();
    }
  }
  catch (Exception e) {
       System.out.println("Exception in getScore()");
       e.printStackTrace();
  }
  return replyString;
} // end of getScore()

public float getSimpleScore(String targetLevel)
{ String result   = this.getScore();
  OutputParser op = new OutputParser(result);
  float posterior = op.getPosterior(targetLevel);
  return posterior;
}

public void setRecordText(String rec)
{  this.recordText = rec;
}
```

```
public void setModelID(String id)
{ this.modelID = id;
  this.serviceID = null;
}

public void setServiceID(String id)
{ this.serviceID = id;
  this.modelID = null;
}
}
```

There are three groups of methods in the code above:

1. queue connection–related methods: **`constructor( ),`** **`connect( ),`** and **`disconnect( )`**
2. setter methods: **`setRecordText( ),`** **`setModelID( ),`** and **`setServiceID( )`**
3. getter methods: **`getScore( )`** and **`getSimpleScore( )`**

The purpose of each method is self-described by names. Note that a **`ScoringEngineProxy`** object is not a real scoring engine object. It is just a proxy, as its name implies, to a real scoring engine. The code above is intentionally simplified for easier reading.

Example of Java code for parsing scoring results:

```
package com.sas.analytics.scoring;
import java.util.*;

public class OutputParser {
  String scoreOutput = null;
  Hashtable ht = new Hashtable();

  public OutputParser(String str) {
    this.scoreOutput = str;
    try {

      // note: code is not listed in this try code block.

      // parse the score_reply XML block (see examples in the Message Formats
      // section of this paper.)

      // create a name/value pair for each “level” XML element in the
      // score_reply XML element by using
      //        the value of the “name”        attribute as the name
      //    and the value of the “probability” attribute as the value;
      // then add each name/value pair to the Hashtable object ht.

    }
    catch(Exception e) {
      System.out.println("Can't parse null score output.");
    }
  }

  public float getPosterior(String targetLevel) {
    float post = -1.0f;
    try {
      post = Float.parseFloat((String)ht.get(targetLevel.toUpperCase()));
    }
    catch (Exception e) {
      System.out.println("Unable to extract posterior.");
    }
    return post;
  }
}
```

SCORING VIA SOAP-BASED WEB SERVICES

Messaging-based transport requires that a client-side software component (typically provided by the messaging tool provider) be installed on the same computer where the scoring client resides. Installing such a component on every scoring client computer may not be an option for some organizations. To avoid such client-side configuration, a Web services wrapper of the Scoring API can serve as an alternative configuration. In such an environment, a scoring client can send scoring messages via SOAP protocol to a remote scoring-client proxy that then relays the scoring request message to a message queue. Most major scoring client applications are developed using languages that support SOAP-based Web services. With the Web services wrapper to the Scoring API, all client applications that support Web services can submit real-time or batch scoring requests.

Since Web services calls are typically stateless (i.e., session state parameters passed across scoring sessions are lost), each scoring request via Web services needs to include connection parameters as followed if session ID is not carried across Web services calls:

```
<score_request>
  <connection>
    <userid>fred</userid>
    <password>fredpw</password>
    <request_queue>queue://QM_SCRSRV//scr_req_queue</request_queue>
    <reply_queue>queue://QM_SCRSRV//scr_reply_queue<reply_queue>
  </connection>
  <serviceID>S224466</serviceID>
  <record>
    <predictor name=”age” value=”25”/>
    <predictor name=”income” value=”45000”/>
    <predictor name=”gender” value=”male”/>
  </record>
  <result style=”full” />
</score_request>
```

SOAP protocol and Web services discussions are beyond the scope of this paper. Fortunately, SOAP knowledge and skills are common. One can learn a lot about SOAP and Web services on the Internet.

USING A Z/OS-BASED IBM WEBSPHERE MQ SERVER

Example of Java JMS code for accessing IBM WebSphere MQ® run on z/OS®:

```
import com.ibm.mq.jms.services.ConfigEnvironment;
public static String  QMGR = "MQSC";
factory = new MQQueueConnectionFactory();
((MQQueueConnectionFactory)factory).setQueueManager(QMGR);

String strHost = "10.12.14.16";
int port = 1414;
String strChannel = "JAVA.CHANNEL";
((MQQueueConnectionFactory)factory).setTransportType(
                         JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
(MQQueueConnectionFactory)factory).setHostName(strHost);
((MQQueueConnectionFactory)factory).setPort(port);
((MQQueueConnectionFactory)factory).setChannel(strChannel);
    ……
ioQueue = session.createQueue(
"queue://MQSC/SASRCC.EM.IN?persistence=1&targetClient=1");
ioQueue2 = session.createQueue("queue://MQSC/SASRCC.EM.OUT");
```

The IBM MQSeries (which was rebranded as IBM WebSphere MQ in 2001) product line predates the JMS Specifications. There are vendor-specific connection parameters and message formats that are not addressed in the JMS Specifications. The code here shows how IBM-provided MQQueueConnectionFactory is used to connect a JMS application to a WebSphere MQ server hosted on a z/OS system. For detailed information regarding how different message formats are addressed, refer to Kareem Yusuf (2004).

SCALABILITY OF MESSAGE SCORING SYSTEMS

One significant benefit of using messaging-based scoring is that new scoring servers can easily be added if existing scoring servers are overloaded. If messaging queues become scoring performance/scalability bottlenecks, queues can often be clustered to alleviate queue loads.

MODEL MANAGEMENT

Generally a new score code should be thoroughly validated before it can be used in a production scoring environment. A model management tool can be helpful and timesaving in facilitating model validation and signoff process. Model management is not the focus of this paper and deserves detailed discussion elsewhere. Some model management issues that are directly related to scoring are: How can a stale model be replaced by a new model without shutting down the running scoring server? Who can submit scoring requests against what models? How can score code be cached in high-speed memory to optimize the scoring execution performance? How can a list of predictors that are required for a scoring be returned? and so on.

CONCLUSION

Predictive model scoring has been proven to increase overall ROI for business decision making. On-demand scoring is gaining momentum as a mechanism to provide predictive scoring. This paper describes how a messaging-based scoring interface can be used to facilitate one-record, on-demand scoring. Message formats and sample scoring client and server programs are discussed. With SAS Enterprise Miner and Integration Technologies, the SAS System provides all the tools needed to implement predictive model scoring for business decision making.

REFERENCES

IBM WebSphere MQ (2004). <http://www-306.ibm.com/software/integration/mqfamily/downloads/>.

JMS (2002). <http://java.sun.com/products/jms/docs.html>.

Monson-Haefel, R. and Chappell, D. A. (2001), *Java Message Service,* Sebastopol, CA: O'Reilly.

Predictive Model Markup Language (2004). <http://www.dmg.org>.

SAS Enterprise Miner (2004). <http://support.sas.com/documentation/onlinedoc/miner.>

SAS Integration Technologies (2004). <http://www.sas.com/software/sas9>.

SOAP (2002). <http://www.w3.org/TR/soap.>

Yusuf, Kareem. (2004), *Enterprise Messaging Using JMS and IBM WebSphere*, New York: Prentice Hall, IBM Press.

Web Services (2003). <http://www.webmethods.com.>

ACKNOWLEDGMENTS

We appreciate the following for their technical help and for reviewing this paper: Virginia Clark, Terri Angeli, Steve Jenisch's team, Jifa Wei, and Larry Dowty.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at

Robert Chu
SAS Institute Inc.
1 SAS Campus Dr.
Cary, North Carolina 27513
Work Phone: (919) 531-7696
Email: robert.chu@sas.com

David Duling
SAS Institute Inc.
1 SAS Campus Dr.
Cary. North Carolina 27513
Work Phone: (919) 531-5267
Email: david.duling@sas.com