

# Java™ Data Mining (JSR-73): Status and Overview

**Mark F. Hornick**  
**Hankil Yoon**  
**Sunil Venkayala**

## **Abstract**

With the completion of Java Data Mining (JSR-73), customers and vendors now have available a powerful standard to enable applications with data mining, both through Java and Web services. In this paper, we introduce Java Data Mining with examples highlighting both the Java and Web services interfaces. We discuss conformance requirements using the Technology Compatibility Kit (TCK) for vendors implementing the standard. Lastly, we comment on likely features for the next release of JDM. The expert group is now forming for Java Data Mining 2.0 as the JCP Executive Committee approved JSR-247.

## **1. Introduction and Background**

Traditionally, data mining algorithms were either home-grown and plugged into applications using raw code, or packaged in an end-user GUI complete with transformations and in some cases scoring code generation. However, the ability to embed data mining end-to-end in applications using commercial data mining products was difficult, if possible at all. Certainly, these APIs were not standards based, making the selection of a particular vendor's solution even more challenging. As such, the ability to leverage data mining functionality easily via a standards-based API greatly reduces the risk of selecting a particular vendor's solution as well as increases accessibility of data mining to application developers. Java™ Data Mining (JDM) addresses this need.

Java technology, specifically as leveraged within the scalable J2EE architecture, facilitates integration with existing applications such as business-to-consumer and business-to-business web sites, customer care centers, campaign management, as well as new applications supporting national security, fraud detection, bioinformatics and life sciences. Java Data Mining allows users to draw on the strengths of multiple data mining vendors for solving business problems, by applying the most appropriate algorithm implementations to a given problem without having to invest resources in learning each vendor's proprietary API. Moreover, vendors and customers can focus on functionality, automation, performance, and price. With JDM's extensible framework for adding new algorithms and functionality, vendors can still differentiate themselves while providing developers with a familiar paradigm.

During the design of JDM, several data mining standards, including the DMG's Predictive Model Markup Language [DM-PMML], OMG's Common Warehouse Metadata for Data Mining [OMG-CWM], and ISO's SQL/MM Part 6 Data Mining [ISO-SQL/MM], have been reviewed to ensure a reasonable degree of interoperability, either in concepts and options, or to facilitate the use of these standards. Similarly, JDM concepts and options have also influenced these standards.

## 2. Status

JDM is now an official part of the Java™ standard. The Executive Committee (EC) of the Java Community Process voted to accept JSR-73, thereby enabling vendors to provide standard advanced analytics support for Java applications. See [JSR73] for the specification [Hornick:2004] and related information. JSR-73 has now moved into the Maintenance phase where minor corrections to the specification, RI, and TCK will be made.

The EC concurrently approved JSR-247 to address extensions to JDM. The expert group for JSR-247 is now forming. Nominations can be submitted at its website [JSR247].

## 3. Main Features

JDM includes interfaces supporting mining functions such as classification, regression, clustering, attribute importance, and association; along with specific mining algorithms such as naïve bayes, support vector machines, decision tree, feed forward neural networks, and k-means. These functions are executed synchronously or asynchronously using mining tasks, which include build, apply for batch and real-time, test, import, and export, as appropriate for each mining function. Import and export can support multiple model representations, including PMML and native formats. Import and export can also be used for JDM metadata using the JDM XML Schema representation, or others such as CWM for Data Mining. Users will also find JDM interfaces supporting confusion matrix, lift, and ROC results, taxonomy and rule representation, and statistics.

JDM further includes the specification of a web services interface based on the JDM UML model, thereby enabling Service Oriented Architecture (SOA) [Barry&Assoc] design. Although JDM-based web services map closely to the Java interface, JDM web services address needs beyond the Java Community, being based on WSDL and XML, a programming language neutral interface. Now, vendors of JDM can leverage their investment in a JDM server for both the Java and web service interfaces, using common metadata, object structure, and capabilities. However, non-JDM vendors can also leverage this same interface to be interoperable with a broader range of vendor implementations.

## 4. Java Interface Example

The following code example illustrates the steps for building and retrieving a clustering model using. See [Hornick:2004] and the JDM javadoc documentation for details of the particular objects referenced.

The first step in using the JDM API is to create a connection to a data mining engine (DME). In this example, we assume the connection `dmeConn` has been created. Object creation requires a corresponding factory, which is obtained from a connection.

The following code block illustrates how to reference and describe data. In lines 1 through 3, a `PhysicalDataSet` object specifying the location of build data via a *universal resource indicator* (URI) is created and saved to the DME. In line 2, attribute metadata associated with the build data is automatically derived and imported to the object `buildData`.

In lines 4 through 6, a `LogicalData` object based on the specified physical data is created and saved. A `LogicalAttribute` object is created within the logical data for each physical attribute in the build data. The attribute type, e.g., numerical or categorical, is automatically

assigned by a vendor specific method, possibly from the attribute data type and number of unique attribute values. However, the user can override this assignment.

```
// Create the physical representation of the data
1) PhysicalDataSetFactory pdsFactory = (PhysicalDataSetFactory)
    dmeConn.getFactory( "javax.datamining.data.PhysicalDataSet" );
2) PhysicalDataSet buildData = pdsFactory.create( uri, true );
3) dmeConn.saveObject( "customerData", buildData, false );
// Create the logical representation of the data from physical data
4) LogicalDataFactory ldFactory = (LogicalDataFactory) dmeConn.getFactory(
    "javax.datamining.data.LogicalData" );
5) LogicalData ld = ldFactory.create( buildData );
6) dmeConn.saveObject( "customerLogicalData", ld, false );
```

In the next code block (lines 7 through 12), a `ClusteringSettings` object is created and saved with the build settings such as its name, logical data, maximum number of clusters and minimum cluster case count. In this example, algorithm settings are not specified, leaving the DME to choose a suitable clustering algorithm with default or system determined settings. This highlights the separation of functions and algorithms supporting both data mining experts and novices.

```
// Create the settings to build a clustering model
7) ClusteringSettingsFactory csFactory = (ClusteringSettingsFactory) dmeConn.getFactory(
    "javax.datamining.clustering.ClusteringSettings" );
8) ClusteringSettings clusteringSettings = csFactory.create();
9) clusteringSettings.setLogicalDataName( "customerLogicalData" );
10) clusteringSettings.setMaxNumberOfClusters( 20 );
11) clusteringSettings.setMinClusterCaseCount( 5 );
12) dmeConn.saveObject( "customerSettings", clusteringSettings, false );
```

In the next code block (lines 13 through 15), a `BuildTask` object is created, which specifies the build data, the build settings, and the model name. The resulting model is placed in the connection's associated repository. All objects associated with a task and the task itself must be saved prior to asynchronous task execution. However, tasks need not be saved for synchronous execution. In this example, the mapping between physical and logical attributes is based on name equivalence, however, users can explicitly map attributes.

Note that the logical data may be omitted in the build settings if it is not supported by the mining function or if all physical attributes are to be used with default assignments. In this example, the lines 4 through 6, and line 9 can be omitted since no changes are made to the logical data after its import from the physical data.

```

// Create a task to build a clustering model with data and settings
13) BuildTaskFactory btFactory = (BuildTaskFactory) dmeConn.getFactory(
    "javax.datamining.task.BuildTask" );
14) BuildTask task = btFactory.create( "customerData", "customerSettings",
    "customerSegments" );
15) dmeConn.saveObject( "customerSegBuild", task, false );

```

In the next code block (lines 16 through 19), we illustrate task execution for model build. In line 16, the named task is executed. The resulting model is placed in the mining object repository. The name of the model can later be used for applying the model to data, and other operations. In lines 17 through 19, the application asynchronously checks the status of the execution by extracting the execution handle and status. Execution handles can also be retrieved via a connection using the task name.

```

// Execute the task and check the status
16) ExecutionHandle handle = dmeConn.execute( "customerSegBuild" );
17) handle.waitForCompletion( Integer.MAX_VALUE ); // wait until done
18) ExecutionStatus status = handle.getLatestStatus();
19) if( ExecutionState.success.equals( status.getState() ) )
// if true, then task completed successfully...

```

In the next block (line 21 through 29), the model built in the preceding block is retrieved for viewing. Each cluster provides details including support, statistics, predicates, its parent and child clusters, centroid coordinates, and case count. A few of these are illustrated below.

```

// Retrieve the model to get the leaf clusters and their details
20) ClusteringModel customerSeg = (ClusteringModel)
    dmeConn.retrieveObject("customerSegments");
21) Collection segments = customerSeg.getLeafClusters();
22) Iterator segmentsIterator = segments.iterator();
23) while( segmentsIterator.hasNext() ) {
24)     Cluster segment = (Cluster) segmentsIterator.next();
25)     Predicate splitPredicate = segment.getSplitPredicate();
26)     long segmentSize = segment.getCaseCount();
27)     double support = segment.getSupport();
28)     AttributeStatisticsSet attrStats = segment.getStatistics();
29) }

```

## 5. Web Services Interface Example

In this section, we illustrate the `executeTask` web service as defined in the JDM WSDL document and an example executing `apply` on a single record using a classification model. See the JDM specification and javadoc documentation for details of the particular objects referenced.

```

1) <complexType name="executeTask">
2)   <sequence>
3)     <choice>
4)       <element name="taskName" type="xsd:string"/>
5)       <element name="task" type="Task"/>
6)     </choice>
7)   </sequence>
8) </complexType>
9) <complexType name="executeTaskResponse">
10)  <sequence>
11)   <choice>
12)     <element name="status" type="ExecutionStatus"/>
13)     <element name="recordValue" type="RecordElement"
           maxOccurs="unbounded" />
14)   </choice>
15) </sequence>
16) </complexType>

```

The execution of a task can be specified either by naming a task already present in the Data Mining Engine (DME) (line 4) or by specifying the task content inline (line 5). The `ExecutionStatus` used in the `executeTaskResponse` in line 12 provides task progress. However, some tasks return values, as in the case of real-time scoring (*record apply*) as specified in line 13.

In lines 17 through 30, we illustrate executing a task called `RecordApplyTask`. A standard header is expected in line 17. Line 20 specifies the record apply for the model “ChurnClassification32”, a classification model predicting customer churn. Lines 21-23 provide the record to score consisting of two predictors, *age* and *income*, and the *customer identifier*. Lines 24 through 28 specify the content of the apply output. In line 25, we specify the customer identifier to be mapped from the input record to the output. In line 26, the top predicted category should be mapped to the destination attribute “churn”. Similarly in line 27, the probability of this prediction should be mapped to the destination attribute “churnProb”.

```

17) <SOAP-ENV:Envelope ... > <SOAP-ENV:Header ... />
18) <SOAP-ENV:Body>
19)   <executeTask xmlns="http://www.jsr73.org=2004"
           http://www.jsr-73.org"/>
20)     <task xsi:type="RecordApplyTask" modelName="ChurnClassification32">
21)       <recordValue name="CustomerAge" value="23"/>
22)       <recordValue name="CustomerIncome" value="50000"/>
23)       <recordValue name="CustomerID" value="1003-2203-120"/>
24)       <applySettingsName xsi:type="ClassificationApplySettings">

```

```

25)         <sourceDestinationMap sourceAttrName="CustomerID"
                destinationAttrName="CustId"/>
26)         <applyMap content="predCat" destPhysAttrName="churn" rank="1"/>
27)         <applyMap content="prob" destPhysAttrName="churnProb" rank="1"/>
28)     </applySettingsName>
29) </task>
30) </SOAP-ENV:Envelope>

```

In lines 31 through 39, we depict the task response to the record apply, in this case a prediction result. In lines 34-36, the apply output for customer identifier, prediction and probability are provided.

```

31) <SOAP-ENV:Envelope ... >

32) <SOAP-ENV:Body>
33)     <executeTaskResponse xmlns="http://www.jsr-73.org/2004/webservices/"
                xmlns:jdm=" http://www.jsr-73.org/2004/JDMSchema">
34)         <recordValue name="CustomerID" value="1003-2203-120"/>
35)         <recordValue name="churn" value="1"/>
36)         <recordValue name="churnProb" value=".87"/>
37)     </executeTaskResponse>
38) </SOAP-ENV:Body>
39) </SOAP-ENV:Envelope>

```

## 6. Conformance

As with any standard, defining conformance for vendor implementations raises myriad issues. Should all implementations be required to support all algorithms and features? Should the results of data mining, e.g., rules in a decision tree model, have the same results for the same datasets? In JDM, compliance is based on a core feature set with optional packages for each mining function and algorithm. In addition, JDM provides `supportsCapability` methods that allow applications to determine at runtime if a particular vendor implementation supports a finer grained feature, e.g., whether classification model build accepts a cost matrix specification, or the clustering algorithm produces hierarchically arranged clusters. For those features the vendor supports of a valid JDM configuration, the vendor implementation must pass the TCK.

## 7. Java Community Process

As a Java Specification Request under SUN's Java Community Process [JCP], JDM went through several reviews before the final vote by the JCP Executive Committee. In addition, the JCP-required Reference Implementation (RI) and Technology Compatibility Kit (TCK) further validate the API prior to becoming part of the Java™ standard. The RI ensures that the interface is able to be implemented and helps to identify modeling flaws before becoming a standard.

## 8. JDM Forum

To facilitate public exchange of ideas on JDM, a new project on java.net has been created: “datamining” at <https://datamining.dev.java.net/>, providing a discussion forum, announcements, and document sharing among Java Data Mining users.

## 9. Summary and Future Work

Through the course of designing the API, the expert group has made numerous tough choices of features to include in the first release. For example, the expert group decided to defer addressing transformations, ensemble models, and “wrapper” methods such as cross validation. However, the feature set of the first release provides a well-rounded core of data mining functionality, which can easily be augmented and extended in JDM 2.0 [JSR-247].

Some of the features being considered for JDM 2.0, include: mining unstructured data such as text and images, additional mining functions such as feature extraction and forecasting, model comparison, multi-target models, and ensembles, and expanding web services to include such features. The web services interface will also explore higher-level data mining services. Such higher-level services may include making a single request to mine and score named datasets with minimal user-provided settings, and returning to the user model quality metrics and individual scores.

With the design work of JSR-73 complete, the expert group looks forward to the standard’s wide spread adoption and use. With vendors supporting JDM, users will realize the benefits originally conceived. User and vendor feedback on the standard will help guide the direction of the JDM 2.0.

## 10. References

- [Barry&Assoc2004] <http://www.service-architecture.com/>
- [DMG-PMML] <http://www.dmg.org>
- [Hornick:2004] Mark Hornick and JSR-73 Expert Group, “Java™ Specification Request 73: Java™ Data Mining (JDM)”, 2004.
- [JCP] <http://www.jcp.org>
- [JSR73] <http://jcp.org/en/jsr/detail?id=73>
- [JSR247] <http://jcp.org/en/jsr/detail?id=247>
- [OMG-CWM] <http://www.omg.org/technology/cwm>