Oracle® Data Mining

Application Developer's Guide,

10*g* Release 2 (10.2)

B14340-01

June 2005

ORACLE®

Oracle Data Mining Application Developer's Guide, 10*g* Release 2 (10.2)

B14340-01

Contents

Preface

This manual describes the Oracle Data Mining Application Programming Interfaces (APIs) and the SQL functions for Data Mining. APIs are available for PL/SQL and for Java.

This manual is intended to be used along with the related reference documentation and sample applications. This information will enable you to develop Data Mining applications for business and bioinformatics applications.

The preface contains these topics:

Audience

This manual is intended for application developers who intend to create data mining applications in PL/SQL or Java.

To use the PL/SQL API and SQL scoring functions for data mining, you need a working knowledge of PL/SQL and Oracle SQL. To use the Java API, you need a working knowledge of Java. To use both interfaces, you need a working knowledge of application programming in an Oracle database environment and a general understanding of data mining concepts.

Users of the Oracle Data Mining BLAST table functions should be familiar with NCBI BLAST and related concepts.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documentation

The documentation set for Oracle Data Mining is part of the Oracle Database 10*g* Release 2 (10.2) Online Documentation Library. The Oracle Data Mining documentation set consists of the following documents:

- *Oracle Data Mining Administrator's Guide*
- *Oracle Data Mining Concepts*
- *Oracle Data Mining Java API Reference* (javadoc)
- *Oracle Data Mining Application Developer's Guide* (this document)

For detailed information about the Oracle Data Mining PL/SQL interface, see *Oracle Database PL/SQL Packages and Types Reference*.

For detailed information about the SQL functions for Oracle Data Mining, see *Oracle Database SQL Reference*.

For information about developing applications to interact with Oracle Database, see *Oracle Database Application Developer's Guide - Fundamentals.*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introducing the Oracle Data Mining APIs

This chapter introduces the Oracle Data Mining (ODM) Application Programming Interfaces (APIs). ODM supports comprehensive PL/SQL and Java APIs, SQL functions, and table functions that implement the Basic Local Alignment Search Tool (BLAST) for life sciences applications.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* (`DBMS_DATA_MINING`, `DBMS_DATA_MINING_TRANSFORM`, and `DBMS_PREDICTIVE_ANALYTICS`) for PL/SQL API syntax.
- *Oracle Data Mining Java API Reference* (javadoc) for Java API syntax.
- *Oracle Database SQL Reference* for syntax of the built-in functions for data mining.
- *Oracle Data Mining Concepts* for detailed information about Oracle Data Mining concepts and features.
- *Oracle Data Mining Administrator's Guide* for information about installation, database administration, and the sample data mining programs.

This chapter contains the following topics:

New Features

Oracle 10*g* Release 2 (10.2) introduces several significant new features in the ODM APIs. Among these are the Decision Tree algorithm for classification and the One-Class SVM algorithm for anomaly detection. New predictive analytics, which automate the process of predictive data mining, and new built-in scoring functions, which return

mining results within the context of a standard SQL statement, are also new in Oracle 10.2.

Oracle 10.2 introduces a completely new Java API for data mining. The Java API is an Oracle implementation of the Java Data Mining (JDM) 1.0 standard. It replaces the proprietary Java API that was available in Oracle 10*g*.

The Java API is layered on the PL/SQL API, and the two APIs are fully interoperable. For example, you can run a SQL script to create a model and then test and apply the model from a Java application.

Note: Model interoperability is new in Oracle 10.2. In Oracle 10*g*, the Java API was incompatible with the PL/SQL API.

See Chapter 8 for information on migrating ODM 10*g* Java applications to the new API.

See Also: *Oracle Data Mining Concepts* and *Oracle Database New Features* for a complete list of new features in Oracle 10*g* Release 2 (10.2) Data Mining.

Predictive and Descriptive Data Mining

ODM supports both predictive and descriptive mining functions. Predictive functions, known as supervised learning, use training data to predict a target value. Descriptive functions, known as unsupervised learning, identify relationships intrinsic to the data. Each mining function identifies a class of problems to be solved, and each can be implemented with one or more algorithms.

The predictive data mining functions are described in Table 1–1. The algorithm abbreviations introduced in the table are used throughout this manual.

Table 1–1 Predictive Data Mining Functions

Function	Description	Sample Problem	Algorithms
Classification	A classification model uses historical data to predict new discrete or categorical data	Given demographic data about a set of customers, predict customer response to an affinity card program.	Naive Bayes (NB) Adaptive Bayes Network ABN) Support Vector Machine (SVM) Decision Tree (DT)
Anomaly Detection	An anomaly detection model predicts whether a data point is typical for a given distribution. The PL/SQL and Java APIs currently support anomaly detection through the Classification function.	Given demographic data about a set of customers, identify customer purchasing behavior that is significantly different from the norm.	One-Class Support Vector Machine (SVM) The PL/SQL and Java APIs currently support One-Class SVM using the classification mining function and the SVM algorithm with no target.
Regression	A regression model uses historical data to predict new continuous, numerical data	Given demographic and purchasing data about a set of customers, predict customer's age.	Support Vector Machine (SVM)
Attribute Importance	An attribute importance model identifies the relative importance of an attribute in predicting a given outcome.	Given customer response to an affinity card program, find the importance of independent attributes.	Minimal Descriptor Length (MDL)

The descriptive data mining functions are described in Table 1–2.

Table 1–2 Descriptive Data Mining Functions

Function	Description	Sample Problem	Algorithms
Clustering	A clustering model identifies natural groupings within a data set.	Segment demographic data into 10 clusters and study the individual clusters. Rank the clusters on probability.	Enhanced *k*-means (KM) Orthogonal Clustering (O-Cluster or OC)
Association Rules	An association model identifies relationships and the probability of their occurrence within a data set.	Find the association between items bought by customers.	Apriori (AP)
Feature Extraction	A feature extraction model creates an optimized data set on which to base a model.	Given demographic data about a set of customers, extract significant features from the given data set.	Non-Negative Matrix Factorization (NMF)

Steps in a Data Mining Application

The first step in designing a data mining application is to analyze the business problem and determine the mining function and algorithm that best addresses it. The second step is to examine the data and determine how it should be prepared for mining.

Once you have identified the mining function and algorithm, and implemented the data transformations, you can develop a sample data mining application. A degree of intuition is involved in making these application choices. You might develop, test, and deploy your sample application and then determine that the results aren't quite what you are looking for. In this case, you might try different or additional data transformations, or you might try a different or additional algorithm.

In any case, the process of developing a data mining application is iterative. It involves testing the model, evaluating test metrics, making adjustments in the model, and re-evaluating.

> **See Also:** *Oracle Data Mining Concepts* for information to help you approach a given data mining problem.

Although it is common to try different approaches to solving a data mining problem, each application must accomplish several basic tasks.

1. **Prepare the data**. One data set is needed for building the model; additional data sets may be necessary for testing and scoring the model, depending on the algorithm. In most cases, the data must be prepared with transformations that enhance or facilitate the effectiveness of the model. Each data set must be prepared in the same way.
2. **Create a model using the build data**.
3. **Evaluate the model**.
   - For classification and regression models, this is the application of the model to a set of test data, and the computation of various test metrics.
   - For clustering models, this is the examination of the clusters identified during model creation.

- For feature extraction models, this is the examination of the features identified during model creation.
- For attribute importance and association models, evaluation is the final step in the mining process. These models cannot be scored against new data.

4. **Apply (score) the model.** This is the process of deploying the model to the data of interest.
   - For classification and regression models, scoring is the application of the "trained" model to the actual population. The result is the best prediction for a target value in each record.
   - For clustering models, scoring is the application of clusters identified by the model to the actual population. The result is the probability of cluster membership for each record.
   - For feature extraction models, scoring is the mapping of features defined by the model to the actual population. The result is a reduced set of predictors in each record.

The basic mining steps for each algorithm are summarized in Table 1–3. Some steps, such as priors and costs and specific test metrics, are optional. The individual steps are described in later sections of this manual.

Table 1–3 Major Steps in a Data Mining Application

Function/Algorithm	Build	Evaluate	Apply
Classification with NB or ABN	■ Prepare build data ■ Specify priors ■ Specify target ■ Create model based on build data	■ Prepare test data ■ Apply model to test data ■ Specify costs ■ Compute test metrics (confusion matrix, lift, accuracy, ROC)	■ Prepare scoring data ■ Apply model to scoring data ■ Specify costs
Classification with DT	■ Prepare build data ■ Specify costs ■ Specify target ■ Create model based on build data	■ Prepare test data ■ Apply model to test data ■ Specify costs ■ Compute test metrics (confusion matrix, lift, accuracy, ROC)	■ Prepare scoring data ■ Apply model to scoring data ■ Specify costs
Classification with SVM	■ Prepare build data ■ Specify weights ■ Specify target ■ Create model based on build data	■ Prepare test data ■ Apply model to test data ■ Specify costs ■ Compute test metrics (confusion matrix, lift, accuracy, ROC)	■ Prepare scoring data ■ Apply model to scoring data ■ Specify costs
Classification (anomaly detection) with One-Class SVM	■ Prepare build data ■ Specify NULL target ■ Create model based on build data		■ Prepare scoring data ■ Apply model to build data or to scoring data

Table 1–3 (Cont.) Major Steps in a Data Mining Application

Function/Algorithm	Build	Evaluate	Apply
Regression with SVM	▪ Prepare build data ▪ Specify target ▪ Create model based on build data	▪ Prepare test data ▪ Apply model to test data ▪ Compute test metrics (Root Mean Square Error, , Mean Absolute Error, Residuals)	▪ Prepare scoring data ▪ Apply model to scoring data
Attribute Importance with MDL	▪ Prepare build data ▪ Specify target ▪ Create model based on build data	Retrieve model details, consisting of a list of attributes with their importance ranking.	
Clustering with KM	▪ Prepare build data ▪ Create model based on build data	Retrieve model details to obtain information about clusters in the data.	▪ Prepare scoring data ▪ Apply model to scoring data
Clustering with OC	▪ Prepare build data ▪ Specify the number of clusters ▪ Create model based on build data	Retrieve model details, consisting of information about clusters in the data.	▪ Prepare scoring data ▪ Apply model to scoring data
Association Rules with AP	▪ Prepare build data ▪ Create model based on build data	Retrieve frequent item sets, and rules that define the item sets.	
Feature Extraction with NMF	▪ Prepare build data ▪ Create model based on build data	Retrieve model details, consisting of a list of features with their importance ranking.	▪ Prepare scoring data ▪ Apply model to scoring data

Data Preparation

Data sets used by Oracle Data Mining are stored in tables, which can be accessed through relational views. The rows are referred to as **cases** or **records**. A **case ID** column specifies a unique identifier for each case, for example the customer ID in a table of customer data.

Columns referred to as **attributes** or **fields** specify a set of predictors. Supervised models (with the exception of One-Class SVM) also use a **target** column. For example, a regression model might predict customer income level (the target), given customer date of birth and gender (the predictors). Unsupervised models use a set of predictors but no target.

ODM distinguishes between two types of attributes: **categorical** or **numerical**. Categorical attributes are a set of values that belong to a given category or class, for example marital status or job title. Numerical attributes are values in a continuum, for example income or age.

Column attributes can have a scalar data type or they can contain nested tables (collection types) of type `DM_NESTED_NUMERICALS` or `DM_NESTED_CATEGORICALS`. Some ODM algorithms support text columns. Text must be indexed and converted to one of the collection types prior to data mining (See Chapter 5).

See Also: Chapter 2, "Managing Data" for more information.

In most cases, data sets must be specifically prepared before building, testing, or applying a model. Preparation includes transformations that improve model accuracy and performance. Common data transformations are:

- Binning — grouping related values together to reduce the number of distinct values for an attribute.
- Normalization — converting individual attribute values so that they fall within a range, typically 0.0 – 1.0 or -1 – +1.
- Clipping — setting extreme attribute values to a single value (winsorizing) or causing extreme values to be ignored by the model (trimming).
- Text transformation — converting text attributes to nested tables.

In addition to these data transformation techniques, you can improve the efficiency of a model by reducing the number of attributes in large data sets. You can create an Attribute Importance model to identify critical attributes or a Non-Negative Matrix Factorization model to combine similar attributes into a single feature. You can then build a model that uses only these attributes or features.

Note: Any transformations performed on the build data must also be performed on the test and scoring data. At each stage of the mining process, the data sets must be identical in structure.

If you are using SQL to prepare your data, you can use `DBMS_DATA_MINING_TRANSFORM`, an open-source package that provides a set of typical data transformation routines. You can use these routines or adapt them, or you can use some other SQL-based mechanism for preparing your data.

See "Preparing the Data" on page 7-17 for information on data transformations in the Java API.

See Also: *Oracle Data Mining Concepts* for an overview of data transformations

Model Settings

When you create a new model, you specify its function. Each function has a default algorithm, and each algorithm has certain default behaviors. To specify any characteristics, you must create a settings table for the model.

Create the settings table in the schema of the model owner. The settings table must have these columns.

Column Name	Data Type
`setting_name`	`VARCHAR2(30)`
`setting_value`	`VARCHAR2(128)`

If you are using the PL/SQL API, specify the name of the settings table as a parameter to the `DBMS_DATA_MINING.CREATE_MODEL` procedure. See "Build Settings" on page 7-9 for information on model settings in the Java API.

See Also: "Model Settings" on page 3-3 for descriptions of the settings and their values.

Model Details

Model details refer to tabular information that can be generated dynamically after a model has been created in the database. Model details provide a complete description of the model. The kind of information provided by model details depends on the algorithm used by the model.

Details of classification and regression models provide extensive statistics that you can capture and examine before testing and scoring the model.

Details of a Decision Tree model are the XML representation of the model in standard PMML format, enabling any application that supports this standard to import the model.

Details of clustering models describe groups of cases that share certain characteristics.

Details of Attribute Importance models and Association models essentially provide the *results* of the model. For example, the details of an Attribute Importance model are a set of attributes with their importance value and rank. Details of an Association model consist of associated items (item sets) and the rules that define each association.

Model details can be retrieved using the PL/SQL table functions `GET_MODEL_DETAILS_x`, where *x* refers to the algorithm used by the model. See "Exploring Model Details" on page 7-11 for information about model details in the Java API.

Predictive Analytics

The `DBMS_PREDICTIVE_ANALYTICS` PL/SQL package provides a high-level interface to data mining. It provides much of the power of predictive data mining, while masking its underlying complexity.

`DBMS_PREDICTIVE_ANALYTICS` automates the process of predictive data mining, from data preparation to model building to scoring new data. In addition to generating predictions, Predictive Analytics can explain the relative influence of specific attributes on the prediction.

`DBMS_PREDICTIVE_ANALYTICS` provides a `PREDICT` routine and an `EXPLAIN` routine.

Predictive Analytics Routine	Description
`PREDICT`	Predicts the values in a target column, based on the cases where the target is not null. `PREDICT` uses known data values to automatically create a model and populate the unknown values in the target.
`EXPLAIN`	Identifies attribute columns that are important for explaining the variation of values in a given column. `EXPLAIN` analyzes the data and builds a model that identifies the important attributes and ranks their importance

When using Predictive Analytics, you do not need to prepare the data. Both the `PREDICT` and `EXPLAIN` routines analyze the data and automatically perform transformations to optimize the model.

See "Using Automated Prediction and Explain Tasks" on page 7-16 for information on Predictive Analytics in the Java API.

Predictive Analytics are also available in the Oracle Spreadsheet Add-In for Predictive Analytics.

SQL Scoring Functions

The built-in SQL functions for Data Mining implement scoring operations for models that have already been created in the database. They provide the following benefits:

- Models can be easily deployed within the context of existing SQL applications.
- Scoring performance is greatly improved, especially in single row scoring cases, since these functions take advantage of existing query execution functionality.
- Scoring results are pipelined, enabling some of the results to be returned quickly to the user.

> **Note:** SQL functions are built into the Oracle Database and are available for use within SQL statements. SQL functions should not be confused with functions defined in PL/SQL packages.

When applied to a given row of scoring data, classification and regression models provide the best predicted value for the target and the associated probability of that value occurring. The predictive functions for Data Mining are described in Table 1–4.

Table 1–4 SQL Scoring Functions for Classification and Regression Models

Function	Description
`PREDICTION`	Returns the best prediction for the target.
`PREDICTION_COST`	Returns a measure of the cost of false negatives and false positives on the predicted target.
`PREDICTION_DETAILS`	Returns an XML string containing details that help explain the scored row.
`PREDICTION_PROBABILITY`	Returns the probability of a given prediction
`PREDICTION_SET`	Returns a list of objects containing all classes in a binary or multi-class classification model along with the associated probability (and cost, if applicable).

Applying a cluster model to a given row of scoring data returns the cluster ID and the probability of that row's membership in the cluster. The clustering functions for data mining are described in Table 1–5.

Table 1–5 SQL Scoring Functions for Clustering Models

Function	Description
`CLUSTER_ID`	Returns the ID of the predicted cluster.
`CLUSTER_PROBABILITY`	Returns the probability of a case belonging to a given cluster.
`CLUSTER_SET`	Returns a list of all possible clusters to which a given case belongs along with the associated probability of inclusion.

Applying a feature extraction model involves the mapping of features (sets of attributes) to columns in the scoring dataset. The feature extraction functions for data mining are described in Table 1–6.

Table 1–6 SQL Scoring Functions for Feature Extraction Models

Function	Description
FEATURE_ID	Returns the ID of the feature with the highest coefficient value.

Table 1–6 (Cont.) SQL Scoring Functions for Feature Extraction Models

Function	Description
FEATURE_SET	Returns a list of objects containing all possible features along with the associated coefficients.
FEATURE_VALUE	Returns the value of a given feature.

See Also: *Oracle Database SQL Reference* for information on the data mining scoring functions.

2

Managing Data

This chapter describes data requirements and options for Oracle Data Mining. This information applies to data sets used to build, test, and score models.

You should ensure that a data set is properly defined before applying transformations to optimize it for a particular model. Data transformation techniques are not addressed in this chapter.

See Also:

- *Oracle Data Mining Concepts* for information about data transformations.
- `DBMS_DATA_MINING_TRANSFORM` in *Oracle Database PL/SQL Packages and Types Reference* for information about data transformations in PL/SQL.
- "Preparing the Data" on page 7-17 for information about data transformations in Java.
- *Oracle Database SQL Reference* for information about Oracle schema objects and data types.

This chapter contains the following topics:

Data Types

The input to ODM is a table or a view. The columns can have numeric or character data types: `NUMBER`, `FLOAT`, `VARCHAR2`, or `CHAR`.

Collection Types

Additionally, ODM supports columns of type `DM_NESTED_CATEGORICALS` and `DM_NESTED_NUMERICALS`. These are collection types that define nested tables.

The ODM collection types define tables of attribute name/value pairs. ODM data sets can include any number of these nested table columns in addition to scalar columns with built-in numeric or character data types. See "Nested Tables" on page 2-4 for more information.

Text

ODM uses features of Oracle Text to transform unstructured text columns to structured columns of type `DM_NESTED_NUMERICALS` for mining. The ODM Java API provides the `OraTextTransform` interface to manage the text transformation process for you. However, if you are using the PL/SQL API, you must use Oracle Text routines directly (See Chapter 5).

Structured text columns are supported by several ODM algorithms (Support Vector Machine for classification and regression, Non-Negative Matrix Factorization, Association, and *k*-Means clustering).

See Also:

- "Using Text Transformation" on page 7-21 for information on text transformation in the Java API
- *Oracle Data Mining Administrator's Guide* for information on sample programs that illustrate text transformation and text mining

Date and Time Data

ODM Predictive Analytics supports columns with `DATE` and `TIMESTAMP` data types. These types are not supported by the ODM PL/SQL and Java APIs.

Columns and Attributes

ODM interprets the columns of the input table as **attributes** for data mining. Attributes are the predictors or descriptors on which the model is based.

A model may additionally identify a case ID column, a target column, or both.

- **Case ID**

  A case ID column holds a unique identifier for each record (row) of data. The case ID must be specified at model build time for all algorithms except Decision Tree. If a case ID is present in a Decision Tree model, it is not considered a possible predictor.

  In the PL/SQL API and Java APIs, the case ID must be specified at apply time for all algorithms. The SQL scoring functions do not use a case ID.

  The case ID column can be of type `VARCHAR2`, `CHAR`, or `NUMBER`, and its maximum length is 128 bytes.

- **Target**

  Predictive algorithms (Classification, Regression, and Attribute Importance) require that one column be designated as a target. The name of the target column is supplied as an argument when the model is created. The target column holds the predictions generated by the model. The target column must be of type `VARCHAR2`, `CHAR`, `NUMBER`, or `FLOAT`. SVM Regression supports only numeric targets. One-Class SVM does not use a target.

Attribute Data Types

ODM interprets attributes as either categorical or numerical.

Categorical attributes are values, such as gender or job title, that belong to a category or domain. Values of a categorical attribute do not have a meaningful order. Categorical attributes have character data types.

Numerical attributes are values, such as age or income, that fall within a continuum. Numerical attributes represent interval data that has a measurable order. Numerical attributes have numeric data types.

Converting Column Data Types

If the column data type is incompatible with the attribute type, you must convert the data type. For example, an application might use postal codes as a categorical attribute, but the data is actually stored in a numeric column. In this case, you would use the `TO_CHAR` function to convert the column to a character data type.

If your mining data includes `DATE` and `TIMESTAMP` columns, and you are not using Predictive Analytics, you must convert those columns to numeric or character data types. In most cases, these data types should be converted to `NUMBER`, but you should evaluate each case individually. If, for example, the date serves as a timestamp indicating when a transaction occurred, converting the date to `VARCHAR2` makes it categorical with unique values, one in each record. This kind of column is known as an identifier and is not useful in model building. However, if the date values are coarse and significantly fewer than the number of records (for example, they might indicate the week or month when an item was purchased), it may be useful to use character values.

You can convert dates to numbers by selecting a starting date and subtracting it from each date value. This process results in a `NUMBER` column. Another approach would be to parse the date and distribute its components over several columns. This is the conversion method used by Predictive Analytics.

DATE and TIMESTAMP Columns with Predictive Analytics

Predictive Analytics interprets `DATE` data and all forms of `TIMESTAMP` data, including `TIMESTAMP WITH TIMEZONE` and `TIMESTAMP WITH LOCAL TIMEZONE`, as a set of numerical attributes. For example, a column named `TIMECOL` would be transformed into attributes for year, month, week, day of year, day of month, day of week, hour, and minute. Each attribute would be named `TIMECOL_`*x*, where *x* is the suffix used to convert the date into a number. For example, the name of the year attribute would be `TIMECOL_YYYY`.

The attributes resulting from `DATE` and `TIMESTAMP` data are visible in the results of an `EXPLAIN` operation. They are not visible in the results of a `PREDICT` operation.

Attribute Names

The names of ODM attributes must be valid column names. Naming requirements for columns are the same as the naming requirements for Oracle schema objects.

Schema object names can be quoted or nonquoted identifiers from one to thirty bytes long. Nonquoted identifiers are not case sensitive; Oracle converts them to uppercase. Nonquoted identifiers can consist of alphanumeric characters and the underscore (_), dollar sign ($), and pound sign (#). The initial character must be alphabetic. Quoted identifiers are case sensitive and can contain most characters.

> **See Also:** *Oracle Database SQL Reference* for information on schema object naming requirements.

Nested Tables

ODM accepts data in single-record case format, where all the information (attributes) concerning an individual is contained in one row. Single-record case, also known as **non-transactional** format, is illustrated in the table in Example 2–1. This table contains descriptive information about customers. CUSTOMER_ID is the case ID column.

Example 2–1 Non-Transactional Format

```
CUSTOMER_ID      GENDER      AGE     MARITAL_STATUS
------------     ------      ---     --------------
1                Male        30       Married
2                Female      35       Single
3                Male        21       Single
4                Female      20       Single
5                Female      35       Married
```

Sometimes data is organized in multi-record case, also known as **transactional**, format. A typical example is market basket data. In transactional format, the data pertaining to an individual is distributed across multiple records. The table in Example 2–2 illustrates transactional format. This table contains information about products purchased by a group of customers on a given day. CUSTOMER_ID is the case ID column.

Example 2–2 Transactional Format

```
CUST_ID    PROD_ID  PROD_NAME
--------   -------  ---------
1          524      brand x icecream
1          530      brand y frozen dinners
1          109      brand z dog food
2          578      brand a orange juice
2          191      brand x frozen dinners
```

ODM does not support multi-record case format. However, there could be circumstances in which you want to construct a model using transactional data. For example, you might want to use transactional data like that in Example 2–2 to predict the products that each customer is likely to buy on his next visit to the store. Discount coupons for these or similar products could then be generated with the checkout receipt.

If you want to construct a model using transactional data, you must first convert the data to single-record case. You must do this by defining columns of nested tables using the ODM fixed collection types, DM_NESTED_NUMERICALS and DM_NESTED_CATEGORICALS. These types define collections of numerical attributes and categorical attributes respectively. The data type descriptions are shown as follows.

```
SQL> describe dm_nested_numerical
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ATTRIBUTE_NAME                                     VARCHAR2(30)
 VALUE                                              NUMBER

SQL> describe dm_nested_numericals
 DM_NESTED_NUMERICALS TABLE OF DMSYS.DM_NESTED_NUMERICAL
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ATTRIBUTE_NAME                                     VARCHAR2(30)
 VALUE                                              NUMBER
```

```
SQL> describe dm_nested_categorical
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ATTRIBUTE_NAME                                     VARCHAR2(30)
 VALUE                                              VARCHAR2(4000)

SQL> describe dm_nested_categoricals
 DM_NESTED_CATEGORICALS TABLE OF DMSYS.DM_NESTED_CATEGORICAL
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 ATTRIBUTE_NAME                                     VARCHAR2(30)
 VALUE                                              VARCHAR2(4000)
```

For a given case identifier, attribute names must be unique across all the collections and individual columns. The fixed collection types enforce this requirement. However, the attribute naming requirements, described in "Attribute Names" on page 2-3, do not apply to the *attribute_name* column of a nested table.

The attributes in Example 2–2 could be stored in nested table columns, as illustrated in Example 2–3. The column `PRODUCT_IDENTIFIERS` is of type `DM_NESTED_NUMERICALS`, and the column `PRODUCT_NAMES` is of type `DM_NESTED_CATEGORICALS`.

Example 2–3 Nested Tables

CUST_ID	PRODUCT_IDENTIFIERS		PRODUCT_NAMES	
	attribute_name	**value**	**attribute_name**	**value**
1	PROD_ID	524	PROD_NAME	brand x ice cream
	PROD_ID	530	PROD_NAME	brand y frozen dinners
	PROD_ID	109	PROD_NAME	brand z dog food
2	PROD_ID	578	PROD_NAME	brand a orange juice
	PROD_ID	191	PROD_NAME	brand x frozen dinners

Object Views and Multi-Record Collections

You can create an object view that presents several sources of transactional data (implemented with nested table columns) as a single data set for data mining. See "Example: Multi-Record Collections With an Object View" on page 2-6.

Apart from the benefit of providing all your mining attributes through a single row-source without impacting their physical data storage, the view acts as a join specification on the underlying tables that can be used by the server for efficiently accessing your data.

Note: Oracle recommends that you perform any necessary data transformations on the base tables before building object views. In this way, all attributes are transformed in a similar way. In most cases, attributes in transactional format are of the same scale, and thus this approach works. Otherwise, you can split the data into sets of similar items and then transform them separately.

See `DBMS_DATA_MINING_TRANSFORM` in *Oracle Database PL/SQL Packages and Types Reference* for information about data transformations using PL/SQL. See "Preparing the Data" on page 7-17 for information about data transformations using Java.

Example: Multi-Record Collections With an Object View

A real-world example of an analytical pipeline for brain tumor research illustrates multi-case collections with an object view. The underlying tables store gene expression data and clinical data about the patient.

The fact table, `GENE_EXPRESSION_DATA`, stores gene expression data. It has the following columns.

```
case_ID      NUMBER
gene         VARCHAR2(30)
expr         NUMBER
```

The dimension table, `CLINICAL_DATA_TABLE`, stores clinical patient data. It has the following columns.

```
case_ID      NUMBER
name         VARCHAR2(30)
type         VARCHAR2(30)
subtype      VARCHAR2(30)
gender       CHAR(1)
age          NUMBER
status       VARCHAR2(30)
```

In this example, we want to create a model that predicts status based on gender, age, and gene expression. The build data for the model is an object view that uses columns of clinical patient data and a nested column of gene expression data. The view will have the following columns.

```
case_id      NUMBER
gender       CHAR(1)
age          NUMBER
gene_expr    DM_NESTED_NUMERICALS
status       VARCHAR2(30)
```

The following statement constructs the object view `gene_expr_build`, which can be used as build data for the model.

```
CREATE OR REPLACE VIEW gene_expr_build AS
SELECT C.case_id,
       C.gender,
       C.age,
       CAST(MULTISET(
       SELECT gene, expr
         FROM gene_expression_data
        WHERE case_id = C.case_id) AS DM_NESTED_NUMERICALS
       ) gene_expr,
```

```
      C.status
FROM  clinical_data_table C
```

Data Storage Optimization

If there are a few hundred mining attributes and your application requires the attributes to be represented as columns in the same row of the table, data storage must be carefully designed.

For a table with several columns, the key question to consider is the (average) row length, not the number of columns. Having more than 255 columns in a table built with a smaller block size typically results in intrablock chaining.

Oracle stores multiple row pieces in the same block, but the overhead to maintain the column information is minimal as long as all row pieces fit in a single data block. If the rows don't fit in a single data block, you may consider using a larger database block size (or use multiple block sizes in the same database).

See Also: *Oracle Database Performance Tuning Guide* for more details.

3

Managing Models

Models created by ODM APIs or by Oracle Data Miner are stored in the Database. This chapter provides information about viewing, accessing, configuring, exporting and importing models.

This chapter contains the following topics:

- Models in the Database
- Import/Export
- Model Settings

Models in the Database

A model is identified by its name. Like tables in the database, a model has storage associated with it. But unlike a table, the form, shape, and content of this storage is opaque to the user. A model is not a database schema object.

You can view the contents of a model — that is, the patterns and rules that constitute a mining model — using algorithm-specific `GET_MODEL_DETAILS` functions in the `DBMS_DATA_MINING` PL/SQL package. These functions are documented in *Oracle Database PL/SQL Packages and Types Reference*. See "Exploring Model Details" on page 7-11 for information on model details in the Java API.

You can view a list of the models in your schema by querying the `DM_USER_MODELS` view. The columns of the `DM_USER_MODELS` view are described in Table 3–1.

Table 3–1 DM_USER_MODELS View

Column	Data Type	Description
`name`	`VARCHAR2(25)`	Name of the model.
`function_name`	`VARCHAR2(30)`	The model function. See Chapter 1 for an overview of mining functions.
`algorithm_name`	`VARCHAR2(30)`	The algorithm used by the model. See Chapter 1 for algorithms used by the mining functions.
`ctime_creation_date`	`DATE`	The date on which the model was created.
`build_duration`	`NUMBER`	The duration of the model build process.
`target_attribute`	`VARCHAR2(30)`	The attribute designated as the target of a classification model.
`model_size`	`NUMBER`	The size of the model in megabytes.

Note: Metadata about models is stored in system tables whose names have the prefix DM$ or DM. You should not attempt to query or modify these system tables, and you should not use DM$ or DM_ prefixes in the names of any tables used by ODM applications.

The following query lists the demo programs in the DM_USER schema.

```
SQL> select 'NAME', 'FUNCTION_NAME', 'ALGORITHM_NAME' from DM_USER_MODELS;

NAME                  FUNCTION_NAME         ALGORITHM_NAME
--------------------------------------------------------------------
ABN_SH_CLAS_SAMPLE    CLASSIFICATION        ADAPTIVE_BAYES_NETWORK
AI_SH_SAMPLE          ATTRIBUTE_IMPORTANCE  MINIMUM_DESCRIPTION_LENGTH
AR_SH_SAMPLE          ASSOCIATION_RULES     APRIORI_ASSOCIATION_RULES
DT_SH_CLAS_SAMPLE     CLASSIFICATION        DECISION_TREE
KM_SH_CLUS_SAMPLE     CLUSTERING            KMEANS
NB_SH_CLAS_SAMPLE     CLASSIFICATION        NAIVE_BAYES
NMF_SH_SAMPLE         FEATURE_EXTRACTION    NONNEGATIVE_MATRIX_FACTOR
OC_SH_CLUS_SAMPLE     CLUSTERING            O_CLUSTER
SVMC_SH_CLAS_SAMPLE   CLASSIFICATION        SUPPORT_VECTOR_MACHINES
SVMO_SH_CLAS_SAMPLE   CLASSIFICATION        SUPPORT_VECTOR_MACHINES
SVMR_SH_REGR_SAMPLE   REGRESSION            SUPPORT_VECTOR_MACHINES
T_SVM_CLAS_SAMPLE     CLASSIFICATION        SUPPORT_VECTOR_MACHINES
```

See Also: *Oracle Data Mining Administrator's Guide* for information about installing, running, and viewing the demo programs.

Model Names

Although ODM models are not stored as Oracle schema objects, their names must conform to Database requirements for nonquoted identifiers. Additionally, model names must be less than 25 bytes long.

Oracle requires that nonquoted identifiers contain only alphanumeric characters, the underscore (_), dollar sign ($), and pound sign (#); the initial character must be alphabetic. Oracle strongly discourages the use of the dollar sign and pound sign in nonquoted literals.

See Also: *Oracle Database SQL Reference* for information on schema object naming requirements.

Model Access

Oracle Data Mining does not support a general privilege model that spans multiple users. GRANT and REVOKE of read and update privileges on a mining model across user schemas are not yet supported.

You can only read and update models in your own schema. If you want to modify the settings of a model or view its details, you must be logged in with the identity of the schema that owns the model. Results of all mining operations are generated in the schema that owns the model.

Models in one schema can be exported to other schemas. You can import a model into your own schema once it has been exported to an accessible location.

Import/Export

Mining models are included when a database or schema is exported or imported with the Oracle Data Pump utility. You can export and import individual models or groups of models using the ODM SQL and Java APIs.

You can use the `EXPORT_MODEL` procedure in the `DBMS_DATA_MINING` package to export a model or a group of models to a dump file. Models can be imported from the dump file using `IMPORT_MODEL`.

The Java API uses the `ExportTask` and `ImportTask` standard JDM interfaces to provide the same export/import functionality.

See Also:

- *Oracle Data Mining Administrator's Guide* for more information on model export/import.
- *Oracle Database Utilities* for information on Oracle Data Pump.

Model Settings

A settings table is a relational table that provides configuration information for a specific model. You must create a settings table if you want a model to have any nondefault characteristics. You will supply the name of the settings table when you create the model.

You must create the settings table in the schema of the model. You can choose the name of the settings table, but the column names and their types must be defined as shown.

Column Name	Data Type
`setting_name`	`VARCHAR2(30)`
`setting_value`	`VARCHAR2(128)`

The values inserted into the `setting_name` column are one or more of several constants defined in the `DBMS_DATA_MINING` package. Depending on what the setting name denotes, the value for the `setting_value` column can be a predefined constant or the actual numerical value corresponding to the setting itself. The `setting_value` column is defined to be `VARCHAR2`. You can explicitly cast numerical inputs to string using the `TO_CHAR()` function, or you can rely on the implicit type conversion provided by the Database.

The settings described in Table 3–2 apply to a mining function. Use these settings to specify the algorithm that the model will use, the location of cost matrix and prior probabilities tables, and other function-specific characteristics. See Table 1–1, " Predictive Data Mining Functions" and Table 1–2, " Descriptive Data Mining Functions" for information about mining functions.

Table 3–2 Data Mining Function Settings

Algorithm Settings	Setting Value (with Permissible Value Ranges)
`algo_name`	Classification: One of: ▪ `algo_naive_bayes` (Default) ▪ `algo_support_vector_machines` (Use this setting for both SVM and One-Class SVM ▪ `algo_adaptive_bayes_network` ▪ `algo_decision_tree` Regression: ▪ `algo_support_vector_machines` Association Rules: ▪ `algo_apriori_association_rules` Clustering: ▪ `algo_kmeans` (Default) ▪ `algo_o_cluster` Feature Extraction: ▪ `algo_nonnegative_matrix_factor` Attribute Importance: ▪ `algo_ai_mdl`
`clas_cost_table_name`	The name of a relational table that specifies a cost matrix. The column requirements for this table are described in "Costs" on page 3-9. This input is applicable only for Decision Tree algorithms, since this is the only algorithm that supports a cost matrix at build time. The cost matrix table must be present in the current user's schema.
`clas_priors_table_name`	The name of a relational table that specifies prior probabilities. The column requirements for this table are described in "Priors" on page 3-10. This input is applicable only for classification algorithms. Decision Tree is the only classification algorithm that does not use priors. The prior probabilities table must be present in the current user's schema. For SVM classification, this setting identifies a table of weights.
`clus_num_clusters`	`TO_CHAR(`*`numeric_expr`*`>= 1)` Number of clusters generated by a clustering algorithm. Default is 10.
`feat_num_features`	`TO_CHAR(`*`numeric_expr`*` > = 1)` Number of features to be extracted. Default value estimated from the data by the algorithm.
`asso_max_rule_length`	`TO_CHAR(2 <=` *`numeric_expr`* `<= 20)` Maximum rule length for AR algorithm. Default is 4.
`asso_min_confidence`	`TO_CHAR(0 <=` *`numeric_expr`* `<= 1)` Minimum confidence value for AR algorithm Default is 0.1.

Table 3–2 (Cont.) Data Mining Function Settings

Algorithm Settings	Setting Value (with Permissible Value Ranges)
asso_min_support	TO_CHAR(0 <= *numeric_expr* <= 1) Minimum support value for AR algorithm Default is 0.1.

Table 3–3 through Table 3–9 provide algorithm-specific settings. You can use these settings to tune the behavior of the algorithm.

Table 3–3 Algorithm Settings for Adaptive Bayes Network

Setting Name	Setting Value (with Permissible Value Ranges)
abns_model_type	Model type for Adaptive Bayes Network: ■ abns_single_feature ■ abns_multi_feature (Default) ■ abns_naive_bayes)
abns_max_build_minutes	TO_CHAR(*numeric_expr* >= 0) The maximum time threshold for completion of model build. Default is 0, which implies no time limit.
abns_max_nb_predictors	TO_CHAR(*numeric_expr* > 0) Maximum number of predictors, measured by their MDL ranking, to be considered for building an ABN model of type abns_naive_bayes. Default is 10.
abns_max_predictors	TO_CHAR(*numeric_expr* > 0) Maximum number of predictors, measured by their MDL ranking, to be considered for building an ABN model of type abns_single_feature or abns_multi_feature. Default is 25.

Table 3–4 Algorithm Settings for Naive Bayes

Setting Name	Setting Value (with Permissible Value Ranges)
nabs_singleton_threshold	TO_CHAR(0 <= *numeric_expr* <=1) Value of singleton threshold for NB algorithm Default value is 0.01
nabs_pairwise_threshold	TO_CHAR(0 <= *numeric_expr* <=1) Value of pairwise threshold for NB algorithm Default is 0.01.

Table 3–5 Algorithm Settings for Decision Tree

Setting Name	Setting Value (with Permissible Value Ranges)
`tree_impurity_metric`	Tree impurity metric for Decision Tree. Tree algorithms seek the best test question for splitting data at each node. The best splitter and split value are those that result in the largest increase in target value homogeneity for the entities in the node. Homogeneity is measured in accordance with a metric. For classification (Binary or multi-class targets), the supported metrics are gini and entropy. ■ `tree_impurity_entropy` ■ `tree_impurity_gini` (Default)
`tree_term_max_depth`	`TO_CHAR(2 <= numeric_expr <= 20)` Criteria for splits: maximum tree depth (the maximum number of nodes between the root and any leaf node, including the leaf node). Default is 7.
`tree_term_minpct_node`	`TO_CHAR(0 <= numeric_expr <= 10)` No child shall have fewer records than this number, which is expressed as a percentage of the training rows. Default is 0.05, indicating 0.05%.
`tree_term_minpct_split`	`TO_CHAR(0 <= numeric_expr <= 20)` Criteria for splits: minimum number of records in a parent node expressed as a percent of the total number of records used to train the model. No split is attempted if number of records is below this value. Default is 0.1, indicating 0.1%.
`tree_term_minrec_node`	`TO_CHAR(numeric_expr >= 0)` No child shall have fewer records than this number. Default is 10.
`tree_term_minrec_split`	`TO_CHAR(numeric_expr >= 0)` Criteria for splits: minimum number of records in a parent node expressed as a value. No split is attempted if number of records is below this value. Default is 20.

Table 3–6 Algorithm Settings for Support Vector Machines

Setting Name	Setting Value (with Permissible Value Ranges)
`svms_active_learning`	Whether active learning is enabled or disabled: ■ `svms_al_disable` ■ `svms_al_enable` (Default) When active learning is enabled, the SVM algorithm uses active learning to build a reduced size model. When active learning is disabled, the SVM algorithm builds a standard model.
`svms_kernel_function`	Kernel for Support Vector Machine: ■ `svms_linear` (Default ■ `svms_gaussian`

Table 3–6 (Cont.) Algorithm Settings for Support Vector Machines

Setting Name	Setting Value (with Permissible Value Ranges)
svms_kernel_cache_size	TO_CHAR(*numeric_expr* > 0) Value of kernel cache size for SVM algorithm. Applies to Gaussian kernel only. Default is 50000000 bytes.
svms_conv_tolerance	TO_CHAR(*numeric_expr* > 0) Convergence tolerance for SVM algorithm Default is 0.001.
svms_std_dev	TO_CHAR(*numeric_expr* > 0) Value of standard deviation for SVM algorithm This is applicable only for Gaussian kernel Default value estimated from the data by the algorithm
svms_complexity_factor	TO_CHAR(*numeric_expr* > 0) Value of complexity factor for SVM algorithm (both classification and regression) Default value estimated from the data by the algorithm
svms_epsilon	TO_CHAR(*numeric_expr* > 0) Value of epsilon factor for SVM Regression Default value estimated from the data by the algorithm
svms_outlier_rate	TO_CHAR(0 <*numeric_expr* < 1) The desired rate of outliers in the training data. Valid for One-Class SVM models only. Cannot be used with svms_complexity_factor. Default is 0.1.

Table 3–7 Algorithm Settings for Non-Negative Matrix Factorization

Setting Name	Setting Value (with Permissible Value Ranges)
nmfs_random_seed	TO_CHAR(*numeric_expr)* Random seed for NMF algorithm. Default is –1.
nmfs_num_iterations	TO_CHAR(1 <= *numeric_expr* <= 500) Number of iterations for NMF algorithm Default is 50
nmfs_conv_tolerance	TO_CHAR(0 < *numeric_expr* <= 0.5) Convergence tolerance for NMF algorithm Default is 0.05

Table 3–8 Algorithm Settings for O-Cluster

Setting Name	Setting Value (with Permissible Value Ranges)
oclt_max_buffer	TO_CHAR(*numeric_expr* > 0) Buffer size for O-Cluster. Default is 50,000.

Table 3–8 (Cont.) Algorithm Settings for O-Cluster

Setting Name	Setting Value (with Permissible Value Ranges)
oclt_sensitivity	TO_CHAR(0 <=*numeric_expr* <= 1) A fraction that specifies the peak density required for separating a new cluster. The fraction is related to the global uniform density. Default is 0.5.

Table 3–9 Algorithm Settings for k-Means

Setting Name	Setting Value (with Permissible Value Ranges)
kmns_distance	Distance Function for *k*-Means Clustering: ■ kmns_euclidean (Default) ■ kmns_cosine ■ kmns_fast_cosine
kmns_iterations	TO_CHAR(0 < *numeric_expr* <= 20) Number of iterations for *k*-Means algorithm Default is 3
kmns_conv_tolerance	TO_CHAR(0< *numeric_expr* <= 0.5) Convergence tolerance for *k*-Means algorithm Default is 0.01
kmns_split_criterion	Split criterion for *k*-Means Clustering: ■ kmns_variance (Default) ■ kmns_size
kmns_num_bins	TO_CHAR(*numeric_expr* > 0) Number of histogram bins. Specifies the number of bins in the attribute histogram produced by *k*-Means. The bin boundaries for each attribute are computed globally on the entire training data set. The binning method is equi-width. All attributes have the same number of bins with the exception of attributes with a single value that have only one bin. Default is 10.
kmns_block_growth	TO_CHAR(1 < *numeric_expr* <= 5) Growth factor for memory allocated to hold cluster data Default value is 2
kmns_min_pct_attr_support	TO_CHAR(0<= *numeric_expr*<=1) The fraction of attribute values that must be non-null in order for the attribute to be included in the rule description for the cluster. Setting the parameter value too high in data with missing values can result in very short or even empty rules. Default is 0.1.

This example creates a settings table for an SVM classification model and edits the individual values using SQL DML.

```
CREATE TABLE drugstore_settings (
  setting_name VARCHAR2(30),
```

```
  setting_value VARCHAR2(128))

BEGIN
-- override the default for convergence tolerance for SVM Classification
INSERT INTO drugstore_model_settings (setting_name, setting_value)
VALUES (dbms_data_mining.svms_conv_tolerance, TO_CHAR(0.081));
COMMIT;
END;
```

The table function `GET_DEFAULT_SETTINGS` provides you all the default settings for mining functions and algorithms. If you intend to override all the default settings, you can create a seed settings table and edit them using SQL DML.

```
BEGIN
CREATE TABLE drug_store_settings AS
SELECT setting_name, setting_value
  FROM TABLE (DBMS_DATA_MINING.GET_DEFAULT_SETTINGS
 WHERE setting_name LIKE 'SVMS_%';
-- update the values using appropriate DML
END;
```

You can also create a settings table based on another model's settings using `GET_MODEL_SETTINGS`.

```
BEGIN
CREATE TABLE my_new_model_settings AS
SELECT setting_name, setting_value
  FROM TABLE (DBMS_DATA_MINING.GET_MODEL_SETTINGS('my_other_model'));
END;
```

Costs

In classification models, you can specify a cost matrix to represent the costs associated with false positive and false negative predictions. A cost matrix can be used in testing and scoring most classification models.

The Decision Tree algorithm can use a cost matrix at build time.To specify the cost matrix, you must create a cost matrix table and provide its name in the `clas_cost_table_name` setting for the Decision Tree model. See "Build Settings" on page 4-5 for an example.

If you are using the Java API, instantiate a `CostMatrix` object and specify the name of the table as a parameter to the `dmeConn.saveObject` method for the object.

The cost matrix table must have these columns.

Column Name	Data Type
`actual_target_value`	`VARCHAR2(4000)` for categorical targets
	`NUMBER` for numeric targets
`predicted_target_value`	`VARCHAR2(4000)`
	`NUMBER` for numeric targets
`cost`	`NUMBER`

ODM enables you to evaluate the cost of predictions from classification models in an iterative manner during the experimental phase of mining, and to eventually apply the optimal cost matrix to predictions on the actual scoring data in a production environment.

The data input to each test computation (a COMPUTE procedure in PL/SQL, or a TestMetrics object in Java) is the result generated from applying the model on test data. In addition, if you also provide a cost matrix as an input, the computation generates test results taking the cost matrix into account. This enables you to experiment with various costs for a given prediction against the same APPLY results, without rebuilding the model and applying it against the same test data for every iteration.

Once you arrive at an optimal cost matrix, you can then input this cost matrix to the RANK_APPLY operation along with the results of APPLY on your scoring data. RANK_APPLY will provide your new data ranked by cost.

> **See Also:** *Oracle Data Mining Concepts* for more information on cost matrix.

Priors

In most classification models, you can specify prior probabilities to offset differences in distribution between the build data and the real population (scoring data). Priors can be used in building any classification model that uses a Bayesian algorithm. Priors are not used by the Decision Tree algorithm.

To specify prior probabilities, you must create a priors table and provide its name in the clas_priors_table_name setting for the model. If you are using the Java API, use a setPriorProbabilitiesMap object in the classification function settings for the model.

SVM Classification uses weights to correct for differences in target distribution. Use the priors table to specify weights for SVM Classification models.

The priors table must have these columns.

Column Name	Data Type
target_value	VARCHAR2 for categorical targets NUMBER for numeric targets
prior_probability	NUMBER

> **See Also:** *Oracle Data Mining Concepts* for more information on prior probabilities.

4

Using the PL/SQL API and SQL Scoring Functions

This chapter provides information to help you build data mining applications in PL/SQL. It includes sample code for building, testing, and scoring a classification model, and it illustrates the use of SQL functions for model scoring.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information on the PL/SQL packages for Data Mining.
- *Oracle Database SQL Reference* for information on the SQL scoring functions for Data Mining.
- *Oracle Data Mining Administrator's Guide* for information on installing and using the sample PL/SQL programs for data mining

This chapter contains the following sections:

The PL/SQL Sample Applications

The examples included in this chapter are taken from the Data Mining sample applications available on the Database companion CD. When you install the companion CD, the Data Mining sample applications are copied to `/rdbms/demo/` in the Oracle home directory.

The following directory listing command lists the sample data mining programs on a Linux system. Use an equivalent command to list the sample programs on other operating systems.

```
>ls $ORACLE_HOME/rdbms/demo/dm*
```

Table 4–1 lists the sample PL/SQL programs.

Table 4–1 Sample PL/SQL Programs

Application	Description
dmabdemo.sql	Creates an Adaptive Bayes Network model (classification).
dmaidemo.sql	Creates an Attribute Importance model.
dmardemo.sql	Creates an Association Rules model.
dmdtdemo.sql	Creates a Decision Tree model (classification).
dmkmdemo.sql	Creates a *k*_means model (clustering).
dmnbdemo.sql	Creates a Naive Bayes model (classification).
dmnmdemo.sql	Creates a Non_Negative Matrix Factorization model (feature extraction).
dmocdemo.sql	Creates an O-Cluster model (clustering).
dmsvcdem.sql	Creates a Support Vector Machine model (classification).
dmsvodem.sql	Creates a Support Vector Machine model (one-class classification).
dmsvrdem.sql	Creates a Support Vector Machine model (regression).
dmtxtfe.sql	Text mining. (term extraction using CTX procedures).
dmtxtnmf.sql	Text mining using NMF feature extraction.
dmtxtsvm.sql	Text mining using SVM classification.

See Also: *Oracle Data Mining Administrator's Guide* for information about installing, running, and viewing the sample programs.

The DBMS_DATA_MINING Package

The following types of mining activities are supported by the DBMS_DATA_MINING package:

- Creating, dropping, and renaming a model: CREATE_MODEL, DROP_MODEL, RENAME_MODEL.
- Scoring a model: APPLY.
- Ranking APPLY results: RANK_APPLY.
- Describing a model: GET_MODEL_DETAILS, GET_MODEL_SETTINGS, GET_DEFAULT_SETTINGS,GET_MODEL_SIGNATURE.
- Computing test metrics for a model: COMPUTE_CONFUSION_MATRIX, COMPUTE_LIFT, and COMPUTE_ROC.
- Exporting and importing models: EXPORT_MODEL, IMPORT_MODEL.

Of these, the first set represents DDL-like operations. The last set represents utilities. The rest are query-like operations in that they do not modify the model.

Note: Detailed information about the DBMS_DATA_MINING package is available in *Oracle Database PL/SQL Packages and Types Reference*.

You can view the models defined in your schema by querying the DM_USER_MODELS view. The following query on a Linux system lists the models in the schema of DMUSER. These models were created by the sample PL/SQL programs.

```
>sqlplus dmuser/dmuser_password
```

```
SQL> set linesize 200
SQL> set pagesize 100
SQL> select NAME, FUNCTION_NAME, ALGORITHM_NAME, TARGET_ATTRIBUTE from DM_USER_MODELS;

NAME                   FUNCTION_NAME           ALGORITHM_NAME                  TARGET_ATTRIBUTE
---------------------  ----------------------  ------------------------------  -----------------
T_NMF_SAMPLE           FEATURE_EXTRACTION      NONNEGATIVE_MATRIX_FACTOR
T_SVM_CLAS_SAMPLE      CLASSIFICATION          SUPPORT_VECTOR_MACHINES         AFFINITY_CARD
AR_SH_SAMPLE           ASSOCIATION_RULES       APRIORI_ASSOCIATION_RULES
AI_SH_SAMPLE           ATTRIBUTE_IMPORTANCE    MINIMUM_DESCRIPTION_LENGTH      AFFINITY_CARD
ABN_SH_CLAS_SAMPLE     CLASSIFICATION          ADAPTIVE_BAYES_NETWORK          AFFINITY_CARD
DT_SH_CLAS_SAMPLE      CLASSIFICATION          DECISION_TREE                   AFFINITY_CARD
NB_SH_CLAS_SAMPLE      CLASSIFICATION          NAIVE_BAYES                     AFFINITY_CARD
SVMC_SH_CLAS_SAMPLE    CLASSIFICATION          SUPPORT_VECTOR_MACHINES         AFFINITY_CARD
OC_SH_CLUS_SAMPLE      CLUSTERING              O_CLUSTER
KM_SH_CLUS_SAMPLE      CLUSTERING              KMEANS
NMF_SH_SAMPLE          FEATURE_EXTRACTION      NONNEGATIVE_MATRIX_FACTOR
SVMR_SH_REGR_SAMPLE    REGRESSION              SUPPORT_VECTOR_MACHINES         AGE
```

Build Results

The CREATE_MODEL procedure creates a mining model. The viewable contents of a mining model are provided to you through the GET_MODEL_DETAILS functions for each supported algorithm. In addition, GET_MODEL_SIGNATURE and GET_MODEL_SETTINGS provide descriptive information about the model.

Apply Results

The APPLY procedure creates and populates a pre-defined table. The columns of this table vary based on the particular mining function, algorithm, and target attribute type — numerical or categorical.

The RANK_APPLY procedure takes this results table as input and generates another table with results ranked based on a top-N input. Classification models can also be ranked based on cost. The column structure of this table varies based on the particular mining function, algorithm, and the target attribute type — numerical or categorical.

Test Results for Classification Models

The COMPUTE routines provided in DBMS_DATA_MINING are the most popularly used metrics for classification. They are not tied to a particular model — they can compute the metrics from any meaningful data input as long as the column structure of the input tables fits the specification of the apply results table and the targets tables.

Test Results for Regression Models

The most commonly used metrics for regression models are root mean square error and mean absolute error. You can use the SQL queries, shown in the following sections, to compute these metrics. Simply replace the italicized tokens with table and column names appropriate for your application.

Root Mean Square Error

```
SELECT sqrt(avg((A.prediction - B.target_column_name) *
                (A.prediction - B.target_column_name))) rmse
  FROM apply_results_table A, targets_table B
 WHERE A.case_id_column_name = B.case_id_column_name;
```

Mean Absolute Error

Given the `targets_table` generated from the test data with the following columns,

```
(case_id_column_name    VARCHAR2,
target_column_name      NUMBER)
```

and apply results table for regression with the following columns,

```
(case_id_column_name    VARCHAR2,
prediction              NUMBER)
```

and a normalization table (optional) with the following columns,

```
(attribute_name         VARCHAR2(30),
scale                   NUMBER,
shift                   NUMBER)
```

the query for mean absolute error is:

```
SELECT  /*+PARALLEL(T) PARALLEL(A)*/
    AVG(ABS(T.actual_value - T.target_value)) mean_absolute_error
  FROM (SELECT B.case_id_column_name
               (B.target_column_name * N.scale + N.shift) actual_value
          FROM targets_table B,
               normalization_table N
         WHERE N.attribute_name = B.target_column_name AND
               B.target_column_name = 1) T,
       apply_results_table_name A
 WHERE A.case_id_column_name = T.case_id_column_name;
```

You can fill in the italicized values with the actual column and table names chosen by you. If the data has not undergone normalization transformation, you can eliminate those references from the subquery. See the SVM regression sample program (`dmsvrdem.sql` for an example.

Example: Building a Decision Tree Model

Given demographic data about a set of customers, this example predicts the customer response to an affinity card program using a classifier based on the Decision Tree algorithm.

> **Note:** This example is taken from the sample program `dmdtdemo.sql`. See *Oracle Data Mining Administrator's Guide* for information about the sample programs.

Mining Data

The Decision Tree algorithm is capable of handling data that has not been specially prepared. This example uses data created from the base tables in the `SH` schema and presented through the following views.

```
MINING_DATA_BUILD_V       (build data)
MINING_DATA_TEST_V        (test data)
MINING_DATA_APPLY_V       (scoring data)
```

Note: Data preparation techniques (using the DBMS_DATA_MINING_TRANSFORM package) are illustrated in many of the sample programs.

Build Settings

The following example creates a settings table and a cost matrix table for the model. The settings override the default classification algorithm (Naive Bayes) and specify the location of the cost matrix table.

```
set echo off
CREATE TABLE dt_sh_sample_settings (
  setting_name  VARCHAR2(30),
  setting_value VARCHAR2(30));
set echo on

-- CREATE AND POPULATE A COST MATRIX TABLE
--
-- A cost matrix is used to influence the weighting of misclassification
-- during model creation (and scoring).
--
CREATE TABLE dt_sh_sample_cost (
  actual_target_value           NUMBER,
  predicted_target_value        NUMBER,
  cost                          NUMBER);
INSERT INTO dt_sh_sample_cost VALUES (0,0,0);
INSERT INTO dt_sh_sample_cost VALUES (0,1,1);
INSERT INTO dt_sh_sample_cost VALUES (1,0,8);
INSERT INTO dt_sh_sample_cost VALUES (1,1,0);
COMMIT;

BEGIN
  -- Populate settings table
  INSERT INTO dt_sh_sample_settings VALUES
    (dbms_data_mining.algo_name, dbms_data_mining.algo_decision_tree);
  INSERT INTO dt_sh_sample_settings VALUES
    (dbms_data_mining.clas_cost_table_name, 'dt_sh_sample_cost');
  COMMIT;
END;
```

Model Creation

The following example creates the model using the predefined settings table.

```
BEGIN
  DBMS_DATA_MINING.CREATE_MODEL(
    model_name          => 'DT_SH_Clas_sample',
    mining_function     => dbms_data_mining.classification,
    data_table_name     => 'mining_data_build_v',
    case_id_column_name => 'cust_id',
    target_column_name  => 'affinity_card',
    settings_table_name => 'dt_sh_sample_settings');
END;

-- DISPLAY MODEL SETTINGS
-- This section illustrates the GET_MODEL_SETTINGS procedure.
-- It is not needed for Decision Tree models, because model
-- settings are present in the model details XML.

column setting_name format a30
```

```
column setting_value format a30
SELECT setting_name, setting_value
  FROM TABLE(DBMS_DATA_MINING.GET_MODEL_SETTINGS('DT_SH_Clas_sample'))
ORDER BY setting_name;

-- DISPLAY MODEL SIGNATURE
-- This section illustrates the GET_MODEL_SIGNATURE procedure.
-- It is not needed for Decision Tree models, because the model
-- signature is present in the model details XML.
--
column attribute_name format a40
column attribute_type format a20
SELECT attribute_name, attribute_type
  FROM TABLE(DBMS_DATA_MINING.GET_MODEL_SIGNATURE('DT_SH_Clas_sample'))
ORDER BY attribute_name;

-- DISPLAY MODEL DETAILS
-- NOTE: The "&quot" characters in this XML output are owing to
--       SQL*Plus behavior. Cut and paste this XML into a file,
--       and open the file in a browser to see correctly formatted XML.
--
SET long 2000000000
column dt_details format a320
SELECT
 dbms_data_mining.get_model_details_xml('DT_SH_Clas_sample').extract('/')
 AS DT_DETAILS
FROM dual;
```

Example: Using SQL Functions to Test a Decision Tree Model

The following example computes a confusion matrix and accuracy using the `PREDICTION` function for Data Mining. It performs the computations both with and without the cost matrix. In this example, the cost matrix reduces the problematic misclassifications, but also negatively impacts the overall model accuracy.

```
-- DISPLAY CONFUSION MATRIX WITHOUT APPLYING COST MATRIX
--
SELECT affinity_card AS actual_target_value,
       PREDICTION(DT_SH_Clas_sample USING *) AS predicted_target_value,
       COUNT(*) AS value
  FROM mining_data_test_v
GROUP BY affinity_card, PREDICTION(DT_SH_Clas_sample USING *)
ORDER BY 1,2;

-- DISPLAY CONFUSION MATRIX APPLYING THE COST MATRIX
--
SELECT affinity_card AS actual_target_value,
       PREDICTION(DT_SH_Clas_sample COST MODEL USING *)
         AS predicted_target_value,
       COUNT(*) AS value
  FROM mining_data_test_v
GROUP BY affinity_card, PREDICTION(DT_SH_Clas_sample COST MODEL USING *)
ORDER BY 1,2;

-- DISPLAY ACCURACY WITHOUT APPLYING COST MATRIX
--
SELECT ROUND(SUM(correct)/COUNT(*),4) AS accuracy
  FROM (SELECT DECODE(affinity_card,
               PREDICTION(DT_SH_Clas_sample USING *), 1, 0) AS correct
          FROM mining_data_test_v);
```

```
-- DISPLAY ACCURACY APPLYING THE COST MATRIX
--
SELECT ROUND(SUM(correct)/COUNT(*),4) AS accuracy
  FROM (SELECT DECODE(affinity_card,
                PREDICTION(DT_SH_Clas_sample COST MODEL USING *),
                1, 0) AS correct
         FROM mining_data_test_v);
```

Example: Using SQL Functions to Apply a Decision Tree Model

The following example illustrates several ways of scoring the Decision Tree Model. It uses the PREDICTION, PREDICTION_COST, PREDICTION_SET, and PREDICTION_DETAILS functions to predict information for four different business cases:

1. Find the ten customers who live in Italy and could be convinced, with the least expense, to use an affinity card.
2. Find the average age of customers who are likely to use an affinity card, based on marital status, education, and household size.
3. List ten customers with the likelihood and cost that they will use or reject an affinity card.
4. Find the segmentation for customers who work in Customer Support and are under 25.

```
------------------
-- BUSINESS CASE 1
-- Find the 10 customers who live in Italy that are least expensive
-- to be convinced to use an affinity card.
--
WITH
cust_italy AS (
SELECT cust_id
  FROM mining_data_apply_v
 WHERE country_name = 'Italy'
ORDER BY PREDICTION_COST(DT_SH_Clas_sample, 1 COST MODEL USING *) ASC, 1
)
SELECT cust_id
  FROM cust_italy
 WHERE rownum < 11;

------------------
-- BUSINESS CASE 2
-- Find the average age of customers who are likely to use an
-- affinity card.
-- Include the build-time cost matrix in the prediction.
-- Only take into account CUST_MARITAL_STATUS, EDUCATION, and
-- HOUSEHOLD_SIZE as predictors.
-- Break out the results by gender.
--
column cust_gender format a12
SELECT cust_gender, COUNT(*) AS cnt, ROUND(AVG(age)) AS avg_age
  FROM mining_data_apply_v
 WHERE PREDICTION(dt_sh_clas_sample COST MODEL
                USING cust_marital_status, education, household_size) = 1
GROUP BY cust_gender
ORDER BY cust_gender;

------------------
```

```
-- BUSINESS CASE 3
-- List ten customers (ordered by their id) along with likelihood and cost
-- to use or reject the affinity card (Note: while this example has a
-- binary target, such a query is useful in multi-class classification -
-- Low, Med, High for example).
--
column prediction format 9;
SELECT T.cust_id, S.prediction, S.probability, S.cost
  FROM (SELECT cust_id,
               PREDICTION_SET(dt_sh_clas_sample COST MODEL USING *) pset
          FROM mining_data_apply_v
         WHERE cust_id < 100011) T,
       TABLE(T.pset) S
ORDER BY cust_id, S.prediction;

------------------
-- BUSINESS CASE 4
-- Find the segmentation (resulting tree node) for customers who
-- work in Tech support and are under 25.
--
column education format a30;
column treenode  format a40;
SELECT cust_id, education,
       PREDICTION_DETAILS(dt_sh_clas_sample USING *) treenode
  FROM mining_data_apply_v
 WHERE occupation = 'TechSup' AND age < 25
ORDER BY 1;
```

5

Using PL/SQL to Prepare Text Data for Mining

Oracle Data Mining supports the mining of data sets that have one or more text columns. These columns must undergo a special preprocessing step whereby text tokens known as **terms** are extracted and stored in a nested table column. The transformed text can then be used as any other attribute in the building, testing, and scoring of models.

This chapter explains how to use Oracle Text packages in a PL/SQL program to prepare a column of text for Oracle Data Mining.

You can also use the Java API to perform text transformation. Refer to "Using Text Transformation" in Chapter 7 for more information.

Note: Oracle Data Mining includes sample programs that illustrate text transformation and text mining in both PL/SQL and Java. Refer to *Oracle Data Mining Administrator's Guide* for information on the Oracle Data Mining sample programs.

See Also: *Oracle Data Mining Concepts* for more information on text mining.

This chapter contains the following sections.

Oracle Text for Oracle Data Mining

Oracle Data Mining uses specialized Oracle Text routines to preprocess text data. Oracle Text is a technology within the Database for building text querying and classification applications. Oracle Text provides the following facilities that are specific to the Oracle Data Mining term extraction process:

- `SVM_CLASSIFIER`, defined in the `CTX_DLL` Oracle Text PL/SQL package, specifies an index preference for Oracle Data Mining term extraction. It is used in

the text transformation process for all algorithms supported by Oracle Data Mining.

- The `CTXSYS.DRVODM` Oracle Text PL/SQL package defines the table functions, `FEATURE_PREP` and `FEATURE_EXPLAIN`, which generate intermediate and final tables of text terms for Oracle Data Mining.

Note: Text terms are also known as *features*. In text mining, a feature is a word or group of words extracted from a text attribute. Both NMF models and text mining transformation perform a kind of feature extraction. NMF creates a single feature from multiple attributes. Text transformation creates multiple features from a single attribute.

The data preparation process in a PL/SQL text mining application requires the use of these Oracle Text facilities. Java developers can use the `OraTextTransform` interface, which presents the Oracle Text term extraction capability within the context of a Java environment. See "Using Text Transformation" on page 7-21 for more information.

See Also: *Oracle Text Application Developer's Guide* and *Oracle Text Reference* for information on Oracle Text.

Note: The Oracle Text facilities for Oracle Data Mining are documented in this chapter. They are not documented in the Oracle Text manuals.

Term Extraction in the Sample Programs

A good place to start in learning the text term extraction process is with the sample programs. You can find these programs in the `/rdbms/demo` directory under `$ORACLE_HOME`. Refer to the *Oracle Data Mining Administrator's Guide*for more information.

The following sample programs contain term extraction code for text mining:

- `dmsh.sql` — Prepares the build, test, and scoring data for the sample programs, including the text mining programs. `dmsh.sql` creates views for data mining and tables and indexes for text mining.
- `dmtxtfe.sql` — Uses a table with an indexed text column, created by `dmsh.sql`, to create a table of build data with a nested table column.

  The `dmtxtfe.sql` program is a sample term extractor. It contains extensive comments that explain the code in a step-by-step fashion. You can expand this program into a complete term extraction solution by adding index creation and the preparation of test and scoring data (as in `dmsh.sql`).

Text Mining Programs

Once you have properly prepared the text data, you can build a text mining program using any algorithm that supports sparse data: association rules, *k*-Means, SVM (classification, regression, and one-class classification), and non-negative matrix factorization.

Two text mining sample PL/SQL programs use the data prepared by `dmsh.sql`.

- `dmtxtnmf.sql` creates a text mining model that uses non-negative matrix factorization.
- `dmtxtsvm.sql` creates a text mining model that uses SVM classification.

Both these programs mine a table of customer data, which includes a nested table column called `COMMENTS`. The `COMMENTS` column has been pre-processed by `dmsh.sql`. The models created by these programs are shown in the following example from a Linux system.

```
-- Run the programs
SQL> @ $ORACLE_HOME%rdbms/demo/dmtxtnmf.sql
SQL> @ $ORACLE_HOME/rdbms/demo/dmtxtsvm.sql
-- List the models created by the programs
SQL> select NAME, FUNCTION_NAME, ALGORITHM_NAME, TARGET_ATTRIBUTE
           from dm_user_models;

NAME               FUNCTION_NAME      ALGORITHM_NAME            TARGET_ATTRIBUTE
----------------   ------------------ ------------------------  ----------------
T_NMF_SAMPLE       FEATURE_EXTRACTION NONNEGATIVE_MATRIX_FACTOR
T_SVM_CLAS_SAMPLE  CLASSIFICATION     SUPPORT_VECTOR_MACHINES   AFFINITY_CARD
```

> **See Also:** *Oracle Data Mining Administrator's Guide*. This manual provides complete instructions for accessing and running the sample programs. It includes information about the build, training, and scoring data used by these programs.

From Unstructured Data to Structured Data

The pre-processing steps for text mining create nested table columns of type `DM_NESTED_NUMERICALS` from columns of type `VARCHAR2` or `CLOB`. Each row of the nested table specifies an attribute name and a value. The `DM_NESTED_NUMERICALS` type defines the following columns.

```
attribute_name     VARCHAR2(30)
value               NUMBER)
```

The term extraction process treats the text in each row of the original table as a separate document. Each document is transformed to a set of terms that have a numeric value and a text label. Within the nested table column, the `attribute_name` column holds the text and the `value` column holds the numeric value of the term, which is derived using the term frequency in the document and in the document collection (other rows).

For example, the following query returns various attributes of customer 102998, including a text column of comments. The text column has not been transformed.

```
SQL> select cust_id, cust_gender, cust_income_level, affinity_card, comments
             from mining_build_text
             where cust_id = 102998;

CUST_ID C  CUST_INCOME_LEVEL     AFFINITY_CARD COMMENTS
------- -- --------------------  ------------- --------------------------------
102998  M  J: 190,000 - 249,999  1             I wanted to write you to let you
                                               know that I've purchased several
                                               items at your store recently and
                                               have been very satisfied with my
                                               purchases. Keep up the good work.
```

The following query returns the same attributes of customer 102998, but the text in the comments column has been transformed. The query extracts the ATTRIBUTE_NAME and VALUE columns from the nested table that holds the transformed text.

```
SQL> select b.cust_id, b.cust_gender, b.cust_income_level, b.affinity_card, n.*
             from mining_build_nested_text b,
                  table(b.comments) n
             where b.cust_id = 102998
             order by n.attribute_name;

CUST_ID  C  CUST_INCOME_LEVEL    AFFINITY_CARD  ATTRIBUTE_NAME  VALUE
-------  -- -------------------  -------------  --------------  --------
102998   M  J: 190,000 - 249,999  1              GOOD             .26894
102998   M  J: 190,000 - 249,999  1              ITEMS             158062
102998   M  J: 190,000 - 249,999  1              KEEP              238765
102998   M  J: 190,000 - 249,999  1              KNOW               .2006
102998   M  J: 190,000 - 249,999  1              LET               299856
102998   M  J: 190,000 - 249,999  1              PURCHASED         142743
102998   M  J: 190,000 - 249,999  1              PURCHASES         173146
102998   M  J: 190,000 - 249,999  1              RECENTLY         .195223
102998   M  J: 190,000 - 249,999  1              SATISFIED        .355851
102998   M  J: 190,000 - 249,999  1              SEVERAL          .355851
102998   M  J: 190,000 - 249,999  1              STORE           .0712537
102998   M  J: 190,000 - 249,999  1              UP               .159838
102998   M  J: 190,000 - 249,999  1              WANTED           .355851
102998   M  J: 190,000 - 249,999  1              WORK             .299856
102998   M  J: 190,000 - 249,999  1              WRITE            .355851
```

The ATTRIBUTE_NAME column holds an item of text from the original comments column. The VALUE column holds the term value. Note that not all words from the original comments column are extracted as terms. For example, the articles the and to are not included.

Steps in the Term Extraction Process

The steps in the term extraction process are summarized in this section. Further details and specific syntax requirements are explained later in this chapter.

Transform a Text Column in the Build Table

First transform the text in the build data. During this process you will generate the text term definitions, which you will reuse for the test and apply data. Perform the following steps:

1. Create an index on the text column in the build table.
2. Create an SVM_CLASSIFIER preference for the index.
3. Define a table to hold the categories specified by the SVM_CLASSIFIER index.
4. Use the FEATURE_PREP table function to create the term definitions and populate an intermediate terms table.
5. Use the FEATURE_EXPLAIN table function to populate the final terms table.
6. Replicate the columns of the original build table (using a view or another table), replacing the text column with a nested table column. Load the terms from the final terms table into the nested table column.

Transform a Text Column in the Test and Apply Tables

The test and apply data must undergo the same pre-processing as the build data. To transform the test and apply data, you will reuse the term definitions generated for the build data. Perform the following steps:

1. Create an index on the text column in the test or apply table.
2. Use the FEATURE_PREP table function to populate an intermediate terms table. Use the term definitions previously generated for the build data.
3. Use the FEATURE_EXPLAIN table function to populate the final terms table.
4. Replicate the columns of the original test or apply table, replacing the text column with a nested table column. Load the terms from the final terms table into the nested table column.

Creating the Index and Index Preference

Oracle Text processing requires a text index. Oracle Text supports several types of indexes for querying, cataloging, and classifying text documents. The Oracle Data Mining term extraction process requires a CONTEXT index for text querying.

You must create an index for each text column to be transformed. Use the following syntax to create the index.

```
SQL>CREATE INDEX index_name ON table_name(column_name)
                  INDEXTYPE IS ctxsys.context PARAMETERS ('nopopulate');
```

Note: This statement creates a basic CONTEXT index. You can further define the characteristics of the index by specifying additional arguments to the CREATE INDEX statement. Refer to *Oracle Text Reference* for details.

Oracle Text supports index preferences for overriding the default characteristics of an index. The CREATE_PREFERENCE procedure in the Oracle Text package CTX_DDL creates a preference with the name and type that you specify. The SVM_CLASSIFIER preference type defines the characteristics of an index for Oracle Data Mining.

You must create an index preference when you prepare the build data. It will be reused when you prepare the test and apply data. Use the following syntax to create the index preference.

```
SQL>EXECUTE ctx_ddl.create_preference('preference_name', 'SVM_CLASSIFIER');
```

The SVM_CLASSIFIER index preference uses a predefined table with two numeric columns: an ID column for the case ID, and a CAT column for the category. The category table is used for internal processing. You must create the category table using the following syntax.

```
SQL>CREATE TABLE category_table_name(id NUMBER, cat NUMBER);
```

Creating the Intermediate Terms Table

The FEATURE_PREP table function in the CTXSYS.DRVODM Oracle Text package extracts terms from a text column using an index preference of type SVM_CLASSIFIER. FEATURE_PREP creates a table of term definitions from the build data and reuses these definitions for the test and apply data.

FEATURE_PREP returns an intermediate terms table.

FEATURE_PREP Calling Syntax

FEATURE_PREP is an over-loaded function that accepts two different sets of arguments. You will specify one set of arguments for the build data and another set for the test and apply data.

```
--- syntax for build data ---
            CTXSYS.DRVODM.FEATURE_PREP (
                   text_index               IN   VARCHAR2,
                   case_id                  IN   VARCHAR2,
                   category_tbl             IN   VARCHAR2,
                   category_tbl_id_col      IN   VARCHAR2,
                   category_tbl_cat_col     IN   VARCHAR2,
                   feature_definition_tbl   IN   VARCHAR2,
                   index_preference         IN   VARCHAR2)
                RETURN DRVODM;

--- syntax for test/apply data ---
           CTXSYS.DRVODM.FEATURE_PREP (
                   text_index               IN   VARCHAR2,
                   case_id                  IN   VARCHAR2,
                   feature_definition_tbl   IN   VARCHAR2,
                RETURN DRVODM;
```

FEATURE_PREP Return Value

FEATURE_PREP returns the following columns. The SEQUENCE_ID column holds the case ID; the ATTRIBUTE_ID column holds the term ID.

```
Name                  NULL?   Type
--------------------- ------- ------
SEQUENCE_ID                   NUMBER
ATTRIBUTE_ID                  NUMBER
VALUE                         NUMBER
```

FEATURE_PREP Arguments

FEATURE_PREP accepts the arguments described in Table 5–1.

Table 5–1 FEATURE_PREP Table Function Arguments

Argument Name	Data Type	
text_index	VARCHAR2	Name of the index on the text column in the build, test, or apply table.
case_ID	VARCHAR2	Name of the case ID column in the build, test, or apply table.
category_tbl	VARCHAR2	Name of the table used by the SVM_CLASSIFIER index preference. Specify this argument only for build data.
category_tbl_id_col	VARCHAR2	Specify 'id'. This is the name of the ID column in the table used by the SVM_CLASSIFIER index preference. Specify this argument only for build data.

Table 5–1 (Cont.) FEATURE_PREP Table Function Arguments

Argument Name	Data Type	
category_tbl_cat_col	VARCHAR2	Specify 'cat'. This is the name of the CAT column in the table used by the SVM_CLASSIFIER index preference. Specify this argument only for build data.
feature_definition_tbl	VARCHAR2	Name of the term definition table created by FEATURE_PREP. The columns of the term definition table are: Name Null? Type --- CAT_ID NUMBER TYPE NUMBER RULE BLOB
index_preference	VARCHAR2	Name of the SVM_CLASSIFIER index preference. Specify this argument only for build data.

FEATURE_PREP Example

The following example creates an intermediate terms table called txt_term_out. The FEATURE_PREP table function extracts terms from a text column with an index called build_text_idx. The text column is in a build table with a case ID column called cust_id. The index preference txt_pref is applied to the index using the id and cat columns in the table cat_tbl. FEATURE_PREP creates a table of term definitions called txt_pref_terms.

```
CREATE TABLE txt_term_out AS
SELECT *
  FROM TABLE(ctxsys.drvodm.feature_prep (
               'build_text_idx',
               'cust_id',
               'cat_tbl',
               'id',
               'cat',
               'txt_pref_terms',
               'txt_pref'));
```

Creating the Final Terms Table

The FEATURE_EXPLAIN table function in the CTXSYS.DRVODM Oracle Text package extracts the term values from the definitions created by FEATURE_PREP and appends the associated word to each value.

FEATURE_EXPLAIN returns the final terms table.

FEATURE_EXPLAIN Calling Syntax

The calling syntax of FEATURE_EXPLAIN is described as follows.

```
CTXSYS.DRVODM.FEATURE_EXPLAIN (
       feature_definition_tbl     IN   VARCHAR2,
   RETURN DRVODM;
```

FEATURE_EXPLAIN Return Value

FEATURE_EXPLAIN returns the following columns.

```
Name              Type
```

```
---------------    ---------------
text               VARCHAR2(160)
type               NUMBER(3)
ID                 NUMBER
score              NUMBER
```

FEATURE_EXPLAIN Arguments

FEATURE_EXPLAIN accepts a single argument: the terms definition table created by FEATURE_PREP.

FEATURE_EXPLAIN Example

The following example creates a final terms table called txt_final_terms using the intermediate terms table txt_term_out. The FEATURE_EXPLAIN table function returns the terms specified in the terms definition table txt_pref_terms.

```
SQL> create table txt_final_terms as
                  select A.sequence_id, B.text, A.value
                        FROM txt_term_out A,
                             TABLE(ctxsys.drvodm.feature_explain(
                                   'txt_pref_terms')) B
                        WHERE A.attribute_id = B.id;
```

Populating a Nested Table Column

Use the final terms table to populate a nested table column of type DM_NESTED_NUMERICALS.

The following example creates the table mining_build_nested_text. (Alternatively, you could create a view.) The table has a case ID column of customer IDs and three customer attribute columns: age, education, and occupation. It also includes a comments column of type DM_NESTED_NUMERICALS created from the terms table txt_final_terms.

```
SQL> CREATE TABLE mining_build_nested_text
       NESTED TABLE comments store AS build_comments
      AS
    SELECT non_text.cust_id,
      non_text.age,
      non_text.education,
      non_text.occupation,
     txt.comments
     FROM
     mining_build_text non_text,
     ( SELECT features.sequence_id,
              cast(COLLECT(dm_nested_numerical(features.text,features.value))
                           as dm_nested_numericals)  comments
       FROM txt_final_terms features
       group by features.sequence_id) txt
     WHERE non_text.cust_id = txt.sequence_id(+);
```

Example: Transforming a Text Column

In the following example, a text column in MINING_BUILD_TEXT is transformed to a nested table column in MINING_BUILD_NESTED_TEXT. The same text column in MINING_APPLY_TEXT is transformed to a nested table column in MINING_APPLY_NESTED_TEXT.

Both MINING_BUILD_TEXT and MINING_APPLY_TEXT have the following columns.

```
Name                            Null?    Type
------------------------------- -------- ----------------------------
CUST_ID                         NOT NULL NUMBER
AGE                                      NUMBER
EDUCATION                                VARCHAR2(21)
OCCUPATION                               VARCHAR2(21)
COMMENTS                                 VARCHAR2(4000)
```

The following statements create the indexes.

```
SQL> create index build_text_idx on mining_build_text (comments)
             indextype is ctxsys.context parameters ('nopopulate');
SQL> create index apply_text_idx ON mining_apply_text (comments)
             indextype is ctxsys.context parameters ('nopopulate');
```

The following statements create the index preference and its table.

```
SQL> execute ctx_ddl.create_preference('idx_pref', 'SVM_CLASSIFIER');
SQL> create table idx_pref_cat (id number, cat number);
```

The following statement returns the intermediate terms in the table BUILD_TERMS_OUT. It also creates the table FEATURE_DEFS and populates it with the term definitions.

```
SQL>  create table build_terms_out as
              select * from
                     table (ctxsys.drvodm.feature_prep
                                  ('build_text_idx',
                                   'cust_id',
                                   'idx_pref_cat',
                                   'id',
                                   'cat',
                                   'feature_defs',
                                   'idx_pref'));
```

The following statement returns the final terms in the table BUILD_EXPLAIN_OUT.

```
SQL> create table build_explain_out as
             select a.sequence_id,
                    b.text,
                    a.value
             from build_terms_out a,
             table (ctxsys.drvodm.feature_explain('feature_defs')) b
             where a.attribute_id = b.id;
```

The following statement creates the table MINING_BUILD_NESTED_TEXT. This table contains the non-text attributes from the original build table and a nested table of comments. This table can be used to build a model.

```
SQL> create table mining_build_nested_text
        nested table comments store as build_comments
          as select non_text.cust_id,
                    non_text.age,
                    non_text.education,
                    non_text.occupation,
                    txt.comments
              from mining_build_text non_text,
             (select features.sequence_id,
                cast(collect(dm_nested_numerical(features.text,features.value))
                as dm_nested_numericals)  comments
```

```
              from build_explain_out features
              group by features.sequence_id) txt
              where non_text.cust_id = txt.sequence_id(+);
```

The following statement creates the intermediate terms table for the comments column in the apply table, MINING_APPLY_TEXT. It uses the term definitions in the FEATURE_DEFS table, which was created during the pre-processing of the comments column in MINING_BUILD_TEXT.

```
SQL>  create table apply_terms_out as
              select * from
                     table (ctxsys.drvodm.feature_prep
                                 ('build_text_idx',
                                  'cust_id',
                                  'feature_defs'));
```

The following statement creates the final terms table for apply.

```
SQL> create table apply_explain_out as
             select a.sequence_id,
                    b.text,
                    a.value
             from apply_terms_out a,
             table (ctxsys.drvodm.feature_explain('feature_defs')) b
             where a.attribute_id = b.id;
```

The following statement creates the table MINING_APPLY_NESTED_TEXT. This table contains the non-text attributes from the original apply table and a nested table of comments. This table can be used to apply the model.

```
SQL> create table mining_apply_nested_text
        nested table comments store as apply_comments
          as select non_text.cust_id,
                    non_text.age,
                    non_text.education,
                    non_text.occupation,
                    txt.comments
              from mining_apply_text non_text,
             (select features.sequence_id,
                cast(collect(dm_nested_numerical(features.text,features.value))
                as dm_nested_numericals)  comments
              from apply_explain_out features
              group by features.sequence_id) txt
              where non_text.cust_id = txt.sequence_id(+);
```

6

Java API Overview

This chapter introduces the new Oracle Data Mining Java API. You can use the Java API to create thin client applications that access the rich data mining functionality within the Oracle Database.

The ODM Java API is an Oracle implementation of the Java Data Mining (JDM) 1.0 standard API for data mining. The ODM Java API implements Oracle-specific extensions to JDM 1.0, in compliance with the JSR-73 standards extension framework. The full range of data mining functions and algorithms available in the Database, including the new predictive analytics features in the `DBMS_PREDICTIVE_ANALYTICS` PL/SQL package, are exposed through the ODM Java API.

The ODM Java API replaces the proprietary Java API for data mining that was available with Oracle 10.1. It is fully compatible with the Oracle 10*g* Release 2 (10.2)PL/SQL API for data mining.

This chapter includes the following topics:

The JDM 1.0 Standard

JDM 1.0 is an industry standard Java API for data mining, developed under the Java Community Process (JCP). It defines Java interfaces that vendors can implement for their Data Mining Engines.

JDM interfaces support mining functions including classification, regression, clustering, attribute importance, and association; and specific mining algorithms including naïve bayes, support vector machines, decision trees, and *k*-means.

For a complete description of the JDM 1.0 standard, visit the JSR-000073 Data Mining API page of the Java Community Process Web Site.

`http://jcp.org/aboutJava/communityprocess/final/jsr073`

You can download the JDM 1.0 javadoc from the Oracle Data Mining page of the Oracle Technology Network.

`http://www.oracle.com/technology/products/bi/odm/index.html`

The Java packages defined by the JDM standard are summarized in Table 6–1.

Table 6–1 JDM 1.0 Standard High-Level Packages

Package	Description
`javax.datamining`	Defines the classes and interfaces used in JDM subpackages.
`javax.datamining.base`	Defines the interfaces for top-level objects and interfaces. This package was introduced to avoid cyclic package dependencies.
`javax.datamining.resource`	Defines objects that support connecting to the Data Mining Server and executing tasks.
`javax.datamining.data`	Defines objects that support logical and physical data, model signature, taxonomy, category set, and the generic super class category matrix.
`javax.datamining.statistics`	Defines objects that support attribute statistics.
`javax.datamining.rule`	Defines objects that support rules and their predicate components.
`javax.datamining.task`	Defines objects that support tasks for building, computing statistics, importing, and exporting models. The `task` package has an optional `apply` subpackage, which is mainly used for supervised and clustering functions.
`javax.datamining.association`	Defines objects that support the build settings and model for association rules.
`javax.datamining.clustering`	Defines objects that support the build settings, models and apply output for clustering.
`javax.datamining.attributeimportance`	Defines objects that support the build settings and model for attribute importance.
`javax.datamining.supervised`	Defines objects that support the build settings and model for supervised learning functions. This package includes optional subpackages for classification and regression and a test task that is common to both.
`javax.datamining.algorithm`	Defines objects that support algorithm-specific settings. This package has optional subpackages for different algorithms.
`javax.datamining.modeldetail`	Defines objects that support the details of various model representations. This package includes optional subpackages for different types of models.

Oracle Extensions to JDM 1.0

The ODM Java API adds functionality that is not part of the JDM standards. The Oracle extensions to the JDM API provide the following major additional features:

- Feature Extraction with the Non-Negative Matrix Factorization (NMF) algorithm
- Orthogonal Partitioning Clustering (O-Cluster), an Oracle-proprietary clustering algorithm
- Adaptive Bayes Network (ABN), an Oracle-proprietary classification algorithm
- Transformations, including discretization (binning), normalization, clipping, and text transformations.
- Predictive analytic s (`OraPredictTask` and `OraExplainTask` interfaces)

See Also: *Oracle Data Mining Java API Reference* (javadoc) for detailed information about the ODM Java API.

The Java packages defined by the Oracle extensions to the JDM standards are summarized in Table 6–2.

Table 6–2 Oracle High-Level Packages that Extend the JDM 1.0 Standards

Package	Description
`oracle.dmt.jdm.featureextraction`	Defines objects related to feature extraction, which supports the scoring operation.
`oracle.dmt.jdm.algorithm.nmf`	Defines objects related to the Non-Negative Matrix Factorization (NMF) algorithm.
`oracle.dmt.jdm.algorithm.ocluster`	Defines objects related to the Orthogonal Partitioning Clustering algorithm (O-cluster)
`oracle.dmt.jdm.algorithm.abn`	Defines objects related to the Adaptive Bayes Network (ABN) classification algorithm.
`oracle.dmt.jdm.transform`	Defines objects related to data transformations.

Principal Objects in the ODM Java API

In the JDM standard API, named objects are objects that can be saved using the `saveObject` method of a `Connection` instance. All named objects are inherited from the `javax.datamining.MiningObject` interface.

The JDM standard supports both permanent and temporary named objects. Permanent objects (`persistentObject`) are saved permanently in the database. Temporary objects (`transientObject`) exist only for the duration of the session.

The persistent and transient named objects supported by the Oracle extensions to the JDM API are listed in Table 6–3.

Table 6–3 Named Objects in ODM Java API

Persistent Objects	Transient Objects
`Model`	`ApplySettings`
`BuildSettings`	`PhysicalDataset`
`Task`	
`CostMatrix`	
`TestMetrics`	

Note: The `LogicalData` and `Taxonomy` objects in the standard JDM API are *not* supported by Oracle.

See Also: "Features of a DMS Connection" on page 7-4 and "API Design Overview" on page 7-7.

The named objects in the ODM Java API are described in the following sections.

PhysicalDataSet Object

A `PhysicalDataSet` object refers to the data to be used as input to a data mining operation. In JDM, `PhysicalDatSet` objects reference specific data through a

Uniform Resource Identifier (URI), which could specify a table, a file, or some other data source.

In the ODM Java API, a `PhysicalDataSet` must reference a table or a view within the database instance referenced in the `Connection`. The syntax of a physical data set URI in the ODM Java AI is the Oracle syntax for specifying a table or a view.

[SchemaName.]TableName
or
[SchemaName.]ViewName

In JDM, `PhysicalDataSet` objects can support multiple data representations. Oracle Data Mining supports two types of data representation: single-record case, and wide data. The Oracle implementation requires users to specify the case-id column in the physical dataset. Refer to *Oracle Data Mining Concepts* for more details.

In the ODM Java API, a `PhysicalDataSet` object is transient. It is stored in the `Connection` as an in-memory object.

> **See Also:** "Describing the Mining Data" on page 7-8.

BuildSettings Object

A `BuildSettings` object captures the high-level specifications used to build a model. The ODM Java API specifies a variety of mining functions: classification, regression, attribute importance, association, clustering, and feature extraction.

A `BuildSettings` object can specify a type of desired result without identifying a particular algorithm. If an algorithm is not specified in the `BuildSettings` object, the DMS selects an algorithm based on the build settings and the characteristics of the data.

`BuildSettings` has a `verify` method, which validates the input specifications for a model. Input must satisfy the requirements of the ODM Java API.

In the ODM Java API, a `BuildSettings` object is persistent. It is stored as a table with a user-specified name in the user schema. This settings table is interoperable with the PL/SQL API for data mining. Normally, you should not modify the build settings table manually.

> **See Also:** "Build Settings" on page 7-9 and "Model Settings" on page 3-3.

Task Object

A `Task` object represents all the information needed to perform a mining operation. The `execute` method of the `Connection` object is used to start the execution of a mining task.

Mining operations, which often process input tables with millions of records, can be time consuming. For this reason, the JDM API supports the asynchronous execution of mining tasks.

Mining tasks are stored as `DBMS_SCHEDULER` job objects in the user schema. The saved job object is in a `DISABLED` state until the `execute` method causes it to start execution.

The `execute` method returns a `javax.datamining.ExecutionHandle` object, which provides methods for monitoring an asynchronous task. `ExecutionHandle` methods include `waitForCompletion` and `getStatus`.

See Also:

- "Executing Mining Tasks" on page 7-10.
- *Oracle Database PL/SQL Packages and Types Reference* for more information about `DBMS_SCHEDULER`.

Model Object

A `Model` object results from the application of an algorithm to data, as specified in a `BuildSettings` object.

Models can be used in several operations. They can be:

- inspected, for example to examine the rules produced from a decision tree or association
- tested for accuracy
- applied to data for scoring
- exported to an external representation such as native format or PMML
- imported for use in the DMS

When a model is applied to data, it is submitted to the DMS for interpretation. A `Model` references its `BuildSettings` object as well as the `Task` that created it.

See Also: "Exploring Model Details" on page 7-11.

TestMetrics Object

A `TestMetrics` object results from the testing of a supervised model with test data. Different test metrics are computed, depending on the type of mining function. For classification models, the accuracy, confusion-matrix, lift, and receiver-operating characteristics can be computed to access the model. Similarly for regression models, R-squared and RMS errors can be computed.

See Also: "Testing a Model" on page 7-12.

ApplySettings Object

An `ApplySettings` object allows users to tailor the results of an apply task. It contains a set of ordered items. Output can consist of:

- Data to be passed through to the output from the input dataset, for example key attributes
- Values computed from the apply itself, for example score, probability, and in the case of decision trees, rule identifiers
- Multi-class categories for its associated probabilities. For example, in a classification model with target `favoriteColor`, users could select the specific colors to receive the probability that a given color is favorite

Each mining function class defines a method to construct a default `ApplySettings` object. This simplifies the programmer's effort if only standard output is desired. For example, typical output for a classification apply would include the top prediction and its probability.

See Also: "Applying a Model for Scoring Data" on page 7-14.

7

Using the Java API

This chapter provides information to help you get started using the Oracle Data Mining Java API. It describes the general design of the API, and it explains how to use the API to perform major mining operations in your application.

See Also:

- *Oracle Data Mining Java API Reference* (javadoc).
- JDM 1.0 javadoc at `http://www.oracle.com/technology/products/bi/odm`

This chapter includes the following topics:

The Java Sample Applications

The samples included in this chapter are taken from the Data Mining sample applications available on the Database companion CD. When you install the companion CD, the Data Mining sample applications are copied to the following directory.

```
$ORACLE_HOME/rdbms/demo     (on Unix)
or
(%ORACLE_HOME%\rdbms\demo     (on NT)
```

To obtain a listing of the sample applications , simply type the following on Unix:

```
ls $ORACLE_HOME/rdbms/demo/dm*
```

Use an equivalent command on other operating systems.

Table 7–1 lists the Java sample applications.

Table 7–1 The Java Sample Applications for Data Mining

Application	Description
dmabdemo.java	Creates an Adaptive Bayes Network model (classification).
dmaidemo.java	Creates an Attribute Importance model.
dmardemo.java	Creates an Association Rules model.
dmtreedemo.java	Creates a Decision Tree model (classification).
dmkmdemo.java	Creates a *k*_means model (clustering).
dmnbdemo.java	Creates a Naive Bayes model (classification).
dmnmdemo.java	Creates a Non_Negative Matrix Factorization model (feature extraction).
dmocdemo.java	Creates an O-Cluster model (clustering).
dmsvcdemo.java	Creates a Support Vector Machine model (classification).
dmsvodemo.java	Creates a Support Vector Machine model (one-class classification).
dmsvrdemo.java	Creates a Support Vector Machine model (regression).
dmtxtnmfdemo.java	Text mining using NMF feature extraction.
dmtxtsvmdemo.java	Text mining using SVM classification.
dmxfdemo.java	Transformations using the Java API.
dmpademo.java	Predictive Analytics using the Java API.
dmapplydemo.java	Apply classification model in different ways.
dmexpimpdemo.java	Native import/export of models.

See Also: *Oracle Data Mining Administrator's Guide* for information about installing, running, and viewing the sample programs.

Setting up Your Development Environment

The ODM Java API requires Oracle Database 10*g* Release 2 (10.2)and J2SE 1.4.2.

To use the ODM Java API, include the following libraries in your CLASSPATH:

```
$ORACLE_HOME/rdbms/jlib/jdm.jar
$ORACLE_HOME/rdbms/jlib/ojdm_api.jar
$ORACLE_HOME/rdbms/jlib/xdb.jar
$ORACLE_HOME/jdbc/lib/ojdbc14.jar
$ORACLE_HOME/oc4j/j2ee/home/lib/connector.jar
$ORACLE_HOME/jlib/orai18n.jar
$ORACLE_HOME/jlib/orai18n-mapping.jar
```

```
$ORACLE_HOME/lib/xmlparserv2.jar
```

Connecting to the Data Mining Server

The first job of a data mining application is to connect to the Data Mining Server (DMS), which is the data mining engine and metadata repository within the Oracle Database.

> **Note:** The JDM API uses the general term **DME** (Data Mining Engine). In the ODM Java API, the term DME refers to the Oracle DMS.

The DMS connection is encapsulated in a `Connection` object, which provides the framework for a data mining application. The `Connection` object serves the following purposes:

- Authenticates users
- Supports retrieval and storage of named objects
- Supports the execution of mining tasks
- Provides version information for the JDM implementation and provider

The DMS `Connection` object is described in detail in "Features of a DMS Connection" on page 7-4.

Connection Factory

A `Connection` is created from a `ConnectionFactory`, an interface provided by the JDM standard API. You can lookup a `ConnectionFactory` from the JNDI server, or you can create a `ConnectionFactory` using an `OraConnectionFactory` object.

Create a ConnectionFactory Using OraConnectionFactory

```
//Create OraConnectionFactory
javax.datamining.resource.ConnectionFactory connFactory =
                oracle.dmt.jdm.resource.OraConnectionFactory();
```

Create a ConnectionFactory From the JNDI Server

```
//Setup the initial context to connect to the JNDI server
Hashtable env = new Hashtable();
env.put( Context.INITIAL_CONTEXT_FACTORY,
"oracle.dmt.jdm.resource.OraConnectionFactory" );
env.put( Context.PROVIDER_URL, "http://myHost:myPort/myService" );
env.put( Context.SECURITY_PRINCIPAL, "user" );
env.put( Context.SECURITY_CREDENTIALS, "password" );
InitialContext jndiContext = new javax.naming.InitialContext( env );
// Perform JNDI lookup to obtain the connection factory
javax.datamining.resource.ConnectionFactory dmeConnFactory =
(ConnectionFactory) jndiContext.lookup("java:comp/env/jdm/MyServer");
//Lookup ConnectionFactory
javax.datamining.resource.ConnectionFactory connFactory =
  (ConnectionFactory) jndiContext.lookup("java:comp/env/jdm/MyServer");
```

Managing the DMS Connection

You can choose to pre-create the JDBC connection to the DMS, or you can manage it through the ODM Java API. If you pre-create the JDBC connection, your data mining application can access the connection caching features of JDBC. When the ODM Java API manages the JDBC connection, caching is not available to your application.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for information about connection caching.

Pre-Create the JDBC Connection

To pre-create the JDBC connection, create an `OracleDataSource` for an `OraConnectionFactory`.

```
//Create an OracleDataSource
OracleDataSource ods = new OracleDataSource();
ods.setURL(URL);
ods.setUser(user);
ods.setPassword(password);

//Create a connection factory using the OracleDataSource
javax.datamining.resource.ConnectionFactory connFactory =
  oracle.dmt.jdm.resource.OraConnectionFactory(ods);
//Create DME Connection
javax.datamining.resource.Connection dmeConn =
    connFactory.getConnection();
```

Use a ConnectionSpec for the DMS Connection

To manage the JDBC connection within the ODM Java API, create an empty `ConnectionSpec` instance using the `getConnectionSpec()` method of `OraConnectionFactory`.

```
//Create ConnectionSpec
ConnectionSpec connSpec = m_dmeConnFactory.getConnectionSpec();
connSpec.setURI("jdbc:oracle:thin:@host:port:sid");
connSpec.setName("user");
connSpec.setPassword("password");

//Create DME Connection
javax.datamining.resource.Connection m_dmeConn =
m_dmeConnFactory.getConnection(connSpec);
```

Features of a DMS Connection

In the ODM Java API, the DMS `Connection` is the primary factory object. The `Connection` instantiates the object factories using the `getFactory` method. The `Connection` object provides named object lookup, persistence, and task execution features.

Create Object Factories

The `Connection.getFactory` method creates a factory object. For example, to create a factory for the `PhysicalDataSet` object, pass the absolute name of the object to this method. The `getFactory` method creates an instance of `PhysicalDataSetFactory`.

```
javax.datamining.data.PhysicalDataSetFactory pdsFactory =
                dmeConn.getFactory("javax.datamining.data.PhysicalDataSet");
```

Provide Access to Mining Object Metadata

The `Connection` object provides methods for retrieving metadata about mining objects.

Method	Description
`getCreationDate`	Returns the creation date of the specified named object. `getCreationDate(java.lang.String objectName, NamedObject objectType) returns java.util.Date`
`getDescription`	Returns the description of the specified mining object. `getDescription(java.lang.String objectName, NamedObject objectType) returns java.lang.String`
`getObjectNames`	Returns a collection of the names of the objects of the specified type. `getObjectNames(NamedObject objectType) returns java.util.Collection`

You can obtain additional information about persistent mining objects by querying the Oracle data dictionary tables.

Save and Retrieve Mining Objects

The `Connection` object provides methods for retrieving mining objects and saving them in the DMS. Persistent objects are stored as database objects. Transient objects are stored in memory.

Method	Description
`saveObject`	Saves the named object in the metadata repository associated with the connection. `saveObject(java.lang.String name, MiningObject object, boolean replace)`
`retrieveObject`	Retrieves a copy of the specified named object from the metadata repository associated with the connection. `retrieveObject(java.lang.String objectIdentifier) returns MiningObject`
`retrieveObject`	Retrieves a copy of the object with the specified name and type from the metadata repository associated with the connection. `retrieveObject(java.lang.String name, NamedObject objectType) returns MiningObject`

See Also:

- "Principal Objects in the ODM Java API" on page 6-3.
- "API Design Overview" on page 7-7.

Execute Mining Tasks

The `Connection` object provides an `execute` method, which can execute mining tasks either asynchronously or synchronously. The DMS uses the database Scheduler to execute mining tasks, which are stored in the user's schema as Scheduler jobs.

Task Execution	execute method syntax
asynchronous	`execute(java.lang.String taskName)` `  returns ExecutionHandle`
synchronous	`execute(Task task,java.lang.Long timeout))` `  returns ExecutionHandle`

Synchronous execution is typically used with single record scoring, but it may be used in other contexts as well.

See Also:

- "Task Object" on page 6-4
- "Executing Mining Tasks" on page 7-10
- *Oracle Database Administrator's Guide* for information about the database Scheduler.

Retrieve DMS Capabilities and Metadata

The `Connection` object provides methods for obtaining information about the DMS at runtime.

Method	Description

`getMetaData`	Returns information about the underlying DMS instance represented through an active connection. `ConnectionMetaData` provides version information for the JDM implementation and Oracle Database.
	`getMetaData()`
	`  returns ConnectionMetaData`

`getSupportedFunctions`	Returns an array of mining functions that are supported by the implementation.
	`getSupportedFunctions()`
	`  returns MiningFunction[]`

`getSupportedAlgorithms`	Returns an array of mining algorithms that are supported by the specified mining function.
	`getSupportedAlgorithms(MiningFunction function)`
	`  returns MiningAlgorithm[]`

`supportsCapability`	Returns true if the specified combination of mining capabilities is supported. If an algorithm is not specified, returns true if the specified function is supported.
	`supportsCapability(MiningFunction function,`
	`  MiningAlgorithm algorithm,`
	`  MiningTask taskType)`
	`  returns boolean`

Retrieve Version Information

The `Connection` object provides methods for retrieving JDM standard version information and Oracle version information.

Method	Description

`getVersion`	Returns the version of the JDM Standard API. It must be "JDM 1.0" for the first release of JDM.
	`getVersion()`
	`returns String`

`getMajorVersion`	Returns the major version number. For the first release of JDM, this is "1".
	`getMajorVersion()`
	`returns int`

`getMinorVersion`	Returns the minor version number. For the first release of JDM, this is "0".
	`getMinorVersion()`
	`returns int`

`getProviderName`	Returns the provider name as "Oracle Corporation".
	`getProviderName()`
	`returns String`

`getProviderVersion`	Returns the version of the Oracle Database that shipped the Oracle Data Mining Java API jar file.
	`getProviderVersion()`
	`returns String`

API Design Overview

Object factories are central to the design of JDM. The ODM Java API uses object factories for instantiating mining objects.

`javax.datamining` is the base package for the JDM standard defined classes.

`oracle.dmt.jdm` is the base package for the Oracle extensions to the JDM standard.

The packages in the JDM standard API are organized by mining functions and algorithms. For example, the `javax.datamining.supervised` package contains all the classes that support supervised functions. It has subpackages for classification and regression classes.

```
javax.datamining.supervised.classification
javax.datamining.supervised.regression
```

Similarly, `javax.datamining.algorithm` is the base package for all algorithms. Each algorithm has its own subpackage. The JDM standard supports algorithms such as naive bayes and support vector machines.

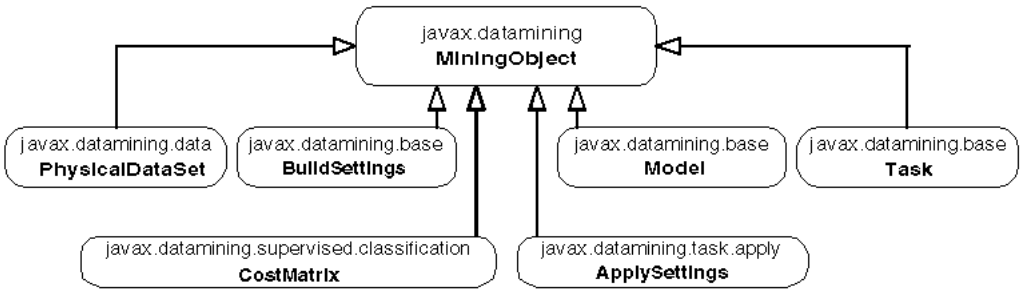
```
javax.datamining.algorithm.naivebayes
javax.datamining.algorithm.svm
```

The ODM Java API follows a similar package structure for the extensions. For example, the ODM Java API supports Feature Extraction, a non-JDM standard function, and the Non-Negative Matrix Factorization algorithm that is used for feature extraction.

```
oracle.dmt.jdm.featureextraction
oracle.dmt.jdm.algorithm.nmf
```

The JDM standard has core packages that define common classes and packages for tasks, model details, rules and statistics. Figure 7–1 illustrates the inheritance hierarchy of the named objects.

Figure 7–1 JDM Named Objects Class Diagram



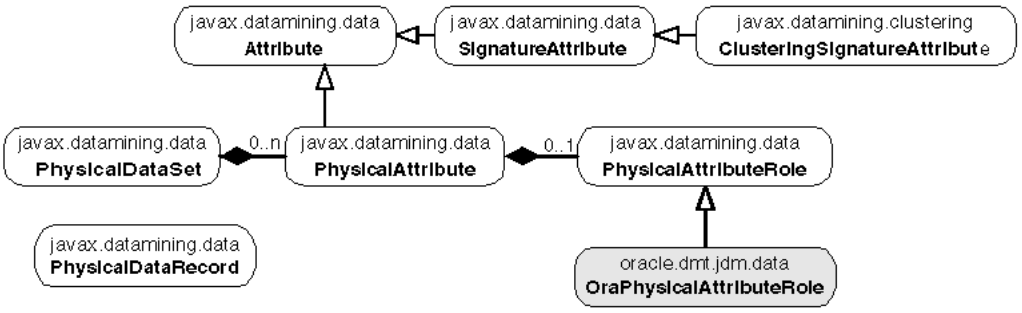
Describing the Mining Data

The JDM standard defines physical and logical data objects to describe the mining attribute characteristics of the data as well as statistical computations for describing the data.

In the ODM Java API, only physical data objects are supported. Data can be logically represented with database views. The `DBMS_STATS` package can be used for statistical computations.

The `javax.datamining.data` package contains all the data-related classes. The class diagram in Figure 7–2 illustrates the class relationships of the data objects supported by the ODM Java API.

Figure 7–2 Data Objects in Oracle Data Mining Java API



The following code illustrates the creation of a `PhysicalDataSet` object. It refers to the view `DMUSER.MINING_DATA_BUILD_V` and specifies the column `cust_id` as case-id using the `PhysicalAttributeRole`.

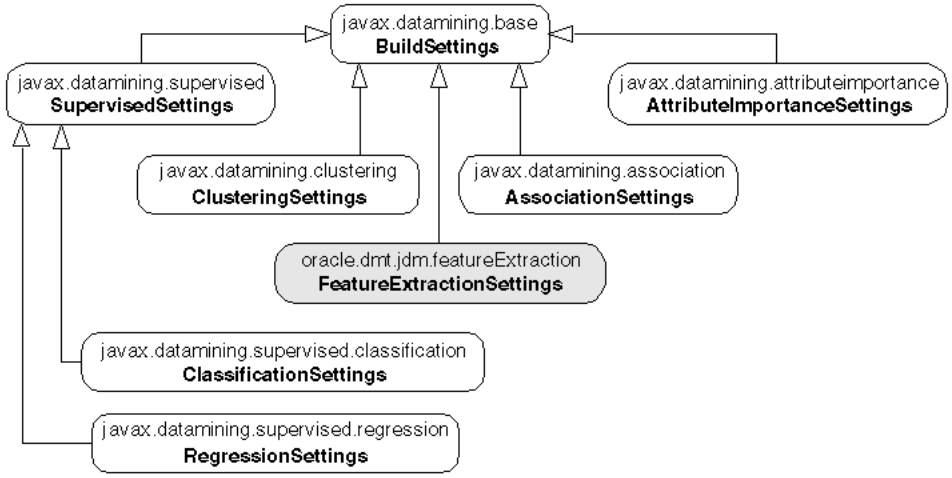
```
//Create PhysicalDataSetFactory
PhysicalDataSetFactory pdsFactory =
     (PhysicalDataSetFactory)m_dmeConn.getFactory
     ("javax.datamining.data.PhysicalDataSet");
//Create a PhysicalDataSet object
PhysicalDataSet buildData =
     pdsFactory.create("DMUSER.MINING_DATA_BUILD_V", false);
//Create PhysicalAttributeFactory
PhysicalAttributeFactory paFactory =
     (PhysicalAttributeFactory)m_dmeConn.getFactory
     ("javax.datamining.data.PhysicalAttribute");
//Create PhysicalAttribute object
PhysicalAttribute pAttr = paFactory.create
```

```
     ("cust_id", AttributeDataType.integerType, PhysicalAttributeRole.caseId );
//Add the attribute to the PhysicalDataSet object
buildData.addAtribute(pAttr);
//Save the physical data set object
dmeConn.saveObject("JDM_BUILD_PDS", buildData, true);
```

Build Settings

In the ODM Java API, the `BuildSettings` object is saved as a table in the database. The settings table is compatible with the `DBMS_DATA_MINING.CREATE_MODEL` procedure. The name of the settings table must be unique in the user's schema. Figure 7–3 illustrates the build settings class hierarchy.

Figure 7–3 Build Settings Class Diagram.



The following code illustrates the creation and storing of a classification settings object with a tree algorithm.

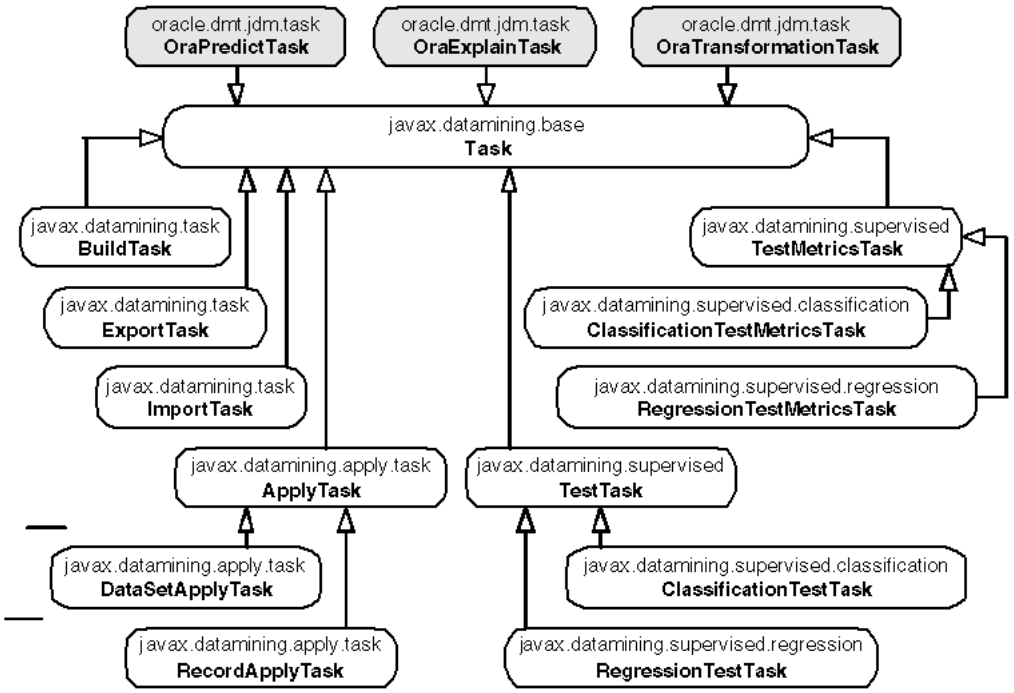
```
//Create a classification settings factory
ClassificationSettingsFactory clasFactory =
(ClassificationSettingsFactory)dmeConn.getFactory
     ("javax.datamining.supervised.classification.ClassificationSettings");
//Create a ClassificationSettings object
ClassificationSettings clas = clasFactory.create();
//Set target attribute name
clas.setTargetAttributeName("AFFINITY_CARD");
//Create a TreeSettingsFactory
TreeSettingsFactory treeFactory =
(TreeSettingsFactory)dmeConn.getFactory
     ("javax.datamining.algorithm.tree.TreeSettings");
//Create TreeSettings instance
TreeSettings treeAlgo = treeFactory.create();
treeAlgo.setBuildHomogeneityMetric(TreeHomogeneityMetric.entropy);
treeAlgo.setMaxDepth(10);
treeAlgo.setMinNodeSize( 10, SizeUnit.count );
//Set algorithm settings in the classification settings
clas.setAlgorithmSettings(treeAlgo);
//Save the build settings object in the database
dmeConn.saveObject("JDM_TREE_CLAS", clas, true);
```

Executing Mining Tasks

The ODM Java API uses the DBMS_SCHEDULER infrastructure for executing mining tasks either synchronously or asynchronously in the database. A mining task is saved as a DBMS_SCHEDULER job in the user's schema. Its initial state is DISABLED. When the user calls the execute method in the DMS Connection, the job state is changed to ENABLED.

The class diagram in Figure 7–4 illustrates the different types of tasks that are available in the ODM Java API.

Figure 7–4 Task Class Diagram



DBMS_SCHEDULER provides additional scheduling and resource management features. You can extend the capabilities of ODM tasks by using the Scheduler infrastructure.

> **See Also:** *Oracle Database Administrator's Guide* for information about the database scheduler.

Building a Mining Model

The javax.datamining.task.BuildTask class is used to build a mining model. Prior to building a model, a PhysicalDataSet object and a BuildSettings object must be saved.

The following code illustrates the building of a tree model using the PhysicalDataSet described in "Describing the Mining Data" on page 7-8 and the BuildSettings described in "Build Settings" on page 7-9.

```
//Create BuildTaskFactory
BuildTaskFactory buildTaskFactory =
     dmeConn.getFactory("javax.datamining.task.BuildTask");
//Create BuildTask object
BuildTask buildTask = buildTaskFactory.create
     ( "JDM_BUILD_PDS","JDM_TREE_CLAS","JDM_TREE_MODEL");
```

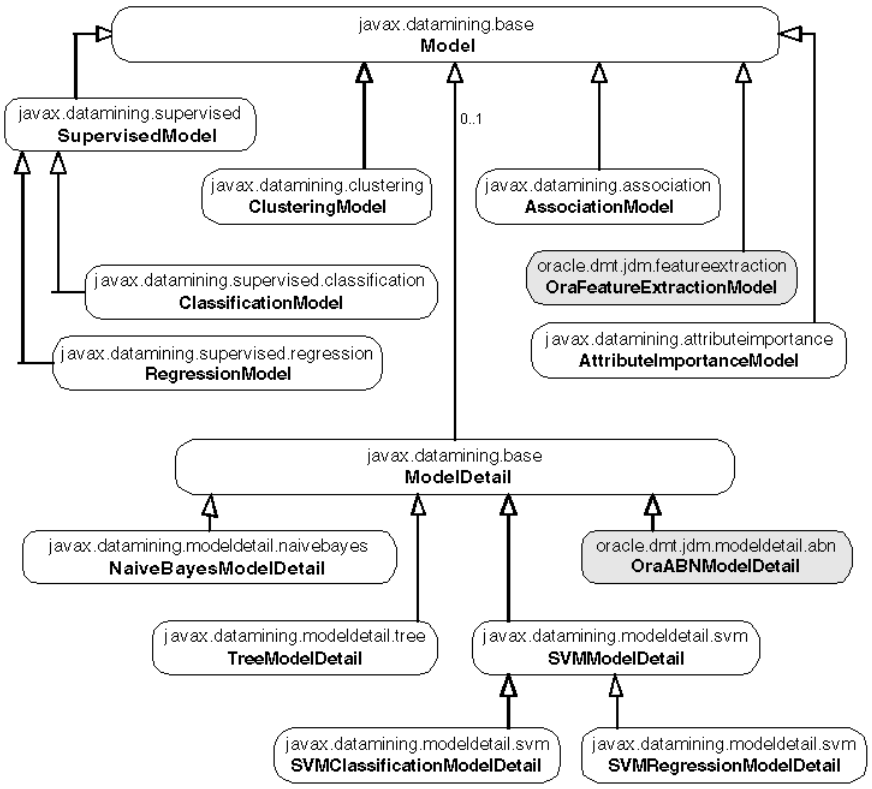
```
//Save BuildTask object
dmeConn.saveObject("JDM_BUILD_TASK", buildTask, true);
//Execute build task asynchronously in the database
ExecutionHandle execHandle = dmeConn.execute("JDM_BUILD_TASK");
//Wait for completion of the task
```

Exploring Model Details

After building a model using the `BuildTask`, a model object is persisted in the database. It can be retrieved to explore the model details.

The class diagram in Figure 7–5 illustrates the different types of model objects and model details objects supported by the ODM Java API.

Figure 7–5 Model and Model Detail Class Diagram



The following code illustrates the retrieval of the classification tree model built in "Building a Mining Model" on page 7-11 and its `TreeModelDetail`.

```
//Retrieve classification model from the DME
ClassificationModel treeModel = (ClassificationModel)dmeConn.retrieveObject
     ( "JDM_TREE_MODEL", NamedObject.model);
//Retrieve tree model detail from the model
TreeModelDetail treeDetail = (TreeModelDetail)treeModel.getModelDetail();
//Get the root node
TreeNode rootNode = treeDetail.getRootNode();
//Get child nodes
TreeNode[] childNodes = rootNode.getChildren();
//Get details of the first child node
int nodeId = childNodes[0].getIdentifier();
long caseCount = childNodes[0].getCaseCount();
Object prediction = childNodes[0].getPrediction();
```

Testing a Model

Once a supervised model has been built, it can be evaluated using a test operation. The JDM standard defines two types of test operations: one that takes the mining model as input, and the other that takes the apply output table with the actual and predicted value columns.

`javax.datamining.supervised.TestTask` is the base class for the model- based test tasks, and `javax.datamining.supervised.TestMetricsTask` is the base class for the apply output table-based test tasks.

The test operation creates and persists a test metrics object in the DMS. For classification model testing, either of the following can be used:

```
javax.datamining.supervised.classification.ClassificationTestTask
javax.datamining.supervised.classification.ClassificationTestMetricsTask
```

Both of these tasks create a named object:

```
javax.datamining.supervised.classification.ClassificationTestMetrics
```

The `ClassificationTestMetrics` named object is stored as a table in the user's schema. The name of the table is the name of the object. The confusion matrix, lift results, and ROC associated with the `ClassificationTestMetrics` object are stored in separate tables whose names are the `ClassificationTestMetrics` object name followed by the suffix `_CFM`, `_LFT`, or `_ROC`. Tools such as Oracle Discoverer can display the test results by querying these tables.

Similarly for regression model testing, either of the following can be used:

```
javax.datamining.supervised.regression.RegressionTestTask
javax.datamining.supervised.regression.RegressionTestMtericsTask
```

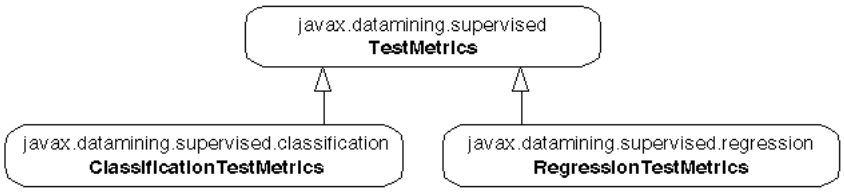
Both these tasks create a named object

```
javax.datamining.supervised.regression.RegressionTestMetrics
```

and store it as a table in the user schema.

The class diagram in Figure 7–6 illustrates the test metrics class hierarchy. It refers to "Build Settings" on page 7-9 for the class hierarchy of test tasks.

Figure 7–6 Test Metrics Class Hierarchy



The following code illustrates the test of a tree model `JDM_TREE_MODEL` using the `ClassificationTestTask` on the dataset `MINING_DATA_TEST_V`.

```
//Create & save PhysicalDataSpecification
PhysicalDataSet testData = m_pdsFactory.create(
        "MINING_DATA_TEST_V", false );
PhysicalAttribute pa = m_paFactory.create("cust_id",
        AttributeDataType.integerType, PhysicalAttributeRole.caseId );
testData.addAttribute( pa );
m_dmeConn.saveObject( "JDM_TEST_PDS", testData, true );
//Create ClassificationTestTaskFactory
```

```
ClassificationTestTaskFactory testTaskFactory =
  (ClassificationTestTaskFactory)dmeConn.getFactory(
     "javax.datamining.supervised.classification.ClassificationTestTask");
//Create, store & execute Test Task
ClassificationTestTask testTask = testTaskFactory.create(
        "JDM_TEST_PDS", "JDM_TREE_MODEL", "JDM_TREE_TESTMETRICS" );
testTask.setNumberOfLiftQuantiles(10);
testTask.setPositiveTargetValue(new Integer(1));
//Save TestTask object
dmeConn.saveObject("JDM_TEST_TASK", testTask, true);
//Execute test task asynchronously in the database
ExecutionHandle execHandle = dmeConn.execute("JDM_TEST_TASK");
//Wait for completion of the task
ExecutionStatus execStatus = execHandle.waitForCompletion(Integer.MAX_VALUE);
//Explore the test metrics after successful completion of the task
if(ExecutionState.success.equals(execStatus.getState())) {
  //Retrieve the test metrics object
  ClassificationTestMetrics testMetrics =
          (ClassificationTestMetrics)dmeConn.getObject("JDM_TREE_TESTMETRICS");
  //Retrieve confusion matrix and accuracy
  Double accuracy = testMetrics.getAccuracy();
  ConfusionMatrix cfm = testMetrics.getConfusionMatrix();
  //Retrieve lift
  Lift lift = testMetrics.getLift();
  //Retrieve ROC
  ReceiverOperatingCharacterics roc = testMetrics.getROC();
}
```

In the preceding example, a test metrics object is stored as a table called `JDM_TREE_TESTMETRICS`. The confusion matrix is stored in the `JDM_TREE_TESTMETRICS_CFM` table, lift is stored in the `JDB_TREE_TESTMETRICS_LFT` table, and `ROC` is stored in the `JDM_TREE_TESTMETRICS_ROC` table. You can use BI tools like Oracle Discoverer to query these tables and create reports.

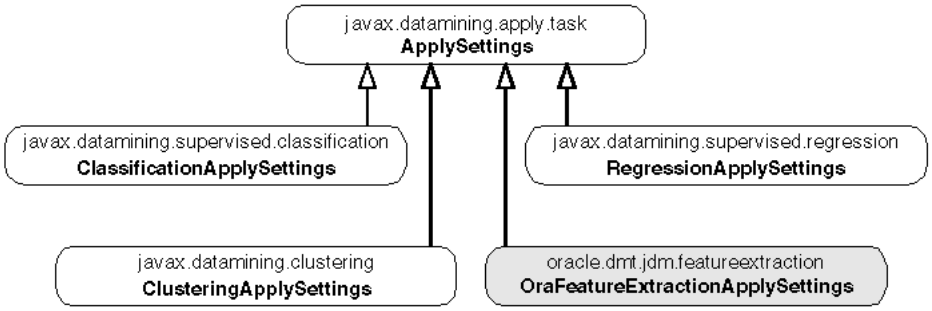
Applying a Model for Scoring Data

All supervised models can be applied to data to find the prediction. Some of the unsupervised models, such as clustering and feature extraction, support the apply operation to find the cluster id or feature id for new records.

The JDM standard API provides an `ApplySettings` object to specify the type of output for the scored results. `javax.datamining.task.apply.ApplySettings` is the base class for all apply settings. In the ODM Java API, the `ApplySettings` object is transient; it is stored in the `Connection` context, not in the database.

The class diagram in Figure 7–7 illustrates the class hierarchy of the apply settings available in the ODM Java API.

Figure 7–7 Apply Settings



In the ODM Java API, default apply settings produce the apply output table in fixed format. The list in Table 7–2 illustrates the default output formats for different functions.

Table 7–2 Default Output Formats for Different Functions

Mining Function				
Classification without Cost	Case ID	Prediction	Probability	
Classification with Cost	Case ID	Prediction	Probability	Cost
Regression	Case ID	Prediction		
Clustering	Case ID	Cluster ID	Probability	
Feature extraction	Case ID	Feature ID	Value	

All types of apply settings support source and destination attribute mappings. For example, if the original apply table has customer name and age columns that need to be carried forward to the apply output table, it can be done by specifying the source destination mappings in the apply settings.

In the ODM Java API, classification apply settings support map by rank, top prediction, map by category, and map all predictions. Regression apply settings support map prediction value. Clustering apply settings support map by rank, map by cluster id, map top cluster, and map all clusters. Feature extraction apply settings support map by rank, map by feature id, map top feature, and map all features.

The following code illustrates the applying of a tree model `JDM_TREE_MODEL` using `ClassificationApplyTask` on the dataset `MINING_DATA_APPLY_V`.

```
//Create & save PhysicalDataSpecification
PhysicalDataSet applyData = m_pdsFactory.create( "MINING_DATA_APPLY_V", false );
PhysicalAttribute pa = m_paFactory.create("cust_id",
       AttributeDataType.integerType, PhysicalAttributeRole.caseId );
applyData.addAttribute( pa );
m_dmeConn.saveObject( "JDM_APPLY_PDS", applyData, true );
//Create ClassificationApplySettingsFactory
ClassificationApplySettingsFactory applySettingsFactory =
  (ClassificationApplySettingsFactory)dmeConn.getFactory(
     "javax.datamining.supervised.classification. ClassificationApplySettings");
//Create & save ClassificationApplySettings
ClassificationApplySettings clasAS = applySettingsFactory.create();
m_dmeConn.saveObject( "JDM_APPLY_SETTINGS", clasAS, true);
//Create DataSetApplyTaskFactory
DataSetApplyTaskFactory applyTaskFactory =
  (DataSetApplyTaskFactory)dmeConn.getFactory(
     "javax.datamining.task.apply.DataSetApplyTask");
```

```
//Create, store & execute apply Task
DataSetApplyTask applyTask = m_dsApplyFactory.create(
        " JDM_APPLY_PDS ", "JDM_TREE_MODEL", " JDM_APPLY_SETTINGS ",
        "JDM_APPLY_OUTPUT_TABLE");
//Save ApplyTask object
dmeConn.saveObject("JDM_APPLY_TASK", applyTask, true);
//Execute test task asynchronously in the database
ExecutionHandle execHandle = dmeConn.execute("JDM_APPLY_TASK");
//Wait for completion of the task
ExecutionStatus execStatus = execHandle.waitForCompletion(Integer.MAX_VALUE);
```

Using a Cost Matrix

The class `javax.datamining.supervised.classification.CostMatrix` is used to represent the costs of the false positive and false negative predictions. It is used for classification problems to specify the costs associated with the false predictions.

In the ODM Java API, cost matrix is supported in apply and test operations for all classification models. For the decision tree algorithm, a cost matrix can be specified at build time. For more information about cost matrix, see *Oracle Data Mining Concepts*.

The following code illustrates how to create a cost matrix object where the target has two classes: `YES` (1) and `NO` (0). Suppose a positive (`YES`) response to the promotion generates $2 and the cost of the promotion is $1. Then the cost of misclassifying a positive responder is $2. The cost of misclassifying a non-responder is $1.

```
//Create category set factory & cost matrix factory
CategorySetFactory catSetFactory = (CategorySetFactory)m_dmeConn.getFactory(
      "javax.datamining.data.CategorySet" );
CostMatrixFactory costMatrixFactory = (CostMatrixFactory)m_dmeConn.getFactory(
      "javax.datamining.supervised.classification.CostMatrix");
//Create categorySet
CategorySet catSet = m_catSetFactory.create(AttributeDataType.integerType);
//Add category values
catSet.addCategory(new Integer(0), CategoryProperty.valid);
catSet.addCategory(new Integer(1), CategoryProperty.valid);
//create cost matrix
CostMatrix costMatrix = m_costMatrixFactory.create(catSet);
costMatrix.setValue(new Integer(0), new Integer(0), 0);
costMatrix.setValue(new Integer(1), new Integer(1), 0);
costMatrix.setValue(new Integer(0), new Integer(1), 2);
costMatrix.setValue(new Integer(1), new Integer(0), 1);
//Save cost matrix in the DME
dmeConn.saveObject("JDM_COST_MATRIX", costMatrix);
```

Using Prior Probabilities

Prior probabilities are used for classification problems if the actual data has a different distribution for target values than the data provided for the model build. A user can specify the prior probabilities in the classification function settings, using `setPriorProbabilitiesMap`. For more information about prior probabilities, see *Oracle Data Mining Concepts*.

Note: Priors are not supported with decision trees.

The following code illustrates how to create a `PriorProbabilities` object, when the target has two classes: `YES` (1) and `NO` (0), and probability of `YES` is 0.05, probability of `NO` is 0.95.

```
//Set target prior probabilities
Map priorMap = new HashMap();
priorMap.put(new Double(0), new Double(0.7));
priorMap.put(new Double(1), new Double(0.3));
buildSettings.setPriorProbabilitiesMap("affinity_card", priorMap);
```

Using Automated Prediction and Explain Tasks

The ODM Java API provides `oracle.dmt.jdm.task.OraPredictTask` and `oracle.dmt.jdm.task.OraExplainTask` for generating predictions and explaining attribute importance. These tasks automate the predict and explain operations for data mining novice users.

`OraPredictTask` predicts the value of a target column based on cases where the target is not null. `OraPredictTask` uses known data values to automatically create a model and populate the unknown values in the target.

`OraExplainTask` identifies attribute columns that are important for explaining the variation of values in a given column. `OraExplainTask` analyzes the data and builds a model that identifies the important attributes and ranks their importance.

Both of these tasks do the automated data preparation where needed.

The following code illustrates `OraPredictTask` and `OraExplainTask`.

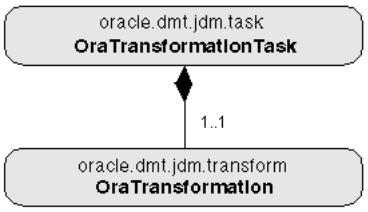
```
//Predict task
   //Create predict task factory and task object
   OraPredictTaskFactory predictFactory =
    (OraPredictTaskFactory)m_dmeConn.getFactory(
      "oracle.dmt.jdm.task.OraPredictTask");
   OraPredictTask predictTask = m_predictFactory.create(
                     "MINING_DATA_BUILD_V", //Input table
                     "cust_id", //Case id column
                     "affinity_card", //target column
                     "JDM_PREDICTION_TABLE"); //prediction output table
   //Save predict task object
   dmeConn.saveObject("JDM_PREDICT_TASK", predictTask, true);
   //Execute test task asynchronously in the database
   ExecutionHandle execHandle = dmeConn.execute("JDM_PREDICT_TASK");
   //Wait for completion of the task
   ExecutionStatus execStatus =
       execHandle.waitForCompletion(Integer.MAX_VALUE);
//Explain task
   //Create explain task factory and task object
   OraExplainTaskFactory explainFactory =
      (OraExplainTaskFactory)m_dmeConn.getFactory(
      "oracle.dmt.jdm.task.OraExplainTask");
   OraExplainTask explainTask = m_explainFactory.create(
                     "MINING_DATA_BUILD_V", //Input table
                     "affinity_card", //explain column
                     "JDM_EXPLAIN_TABLE"); //explain output table
   //Save predict task object
   dmeConn.saveObject("JDM_EXPLAIN_TASK", explainTask, true);
   //Execute test task asynchronously in the database
   ExecutionHandle execHandle = dmeConn.execute("JDM_ EXPLAIN_TASK");
   //Wait for completion of the task
   ExecutionStatus execStatus = execHandle.waitForCompletion(Integer.MAX_VALUE);
```

Preparing the Data

In the ODM Java API, data must be prepared before building, applying, or testing a model. The `oracle.dmt.jdm.task.OraTransformationTask` class supports common transformations used in data mining: binning, normalization, clipping, and text transformations. For more information about transformations, see *Oracle Data Mining Concepts*.

The class diagram in Figure 7–8 illustrates the `OraTransformationTask` and its relationship with other objects.

Figure 7–8 OraTransformationTask and its Relationship With Other Objects

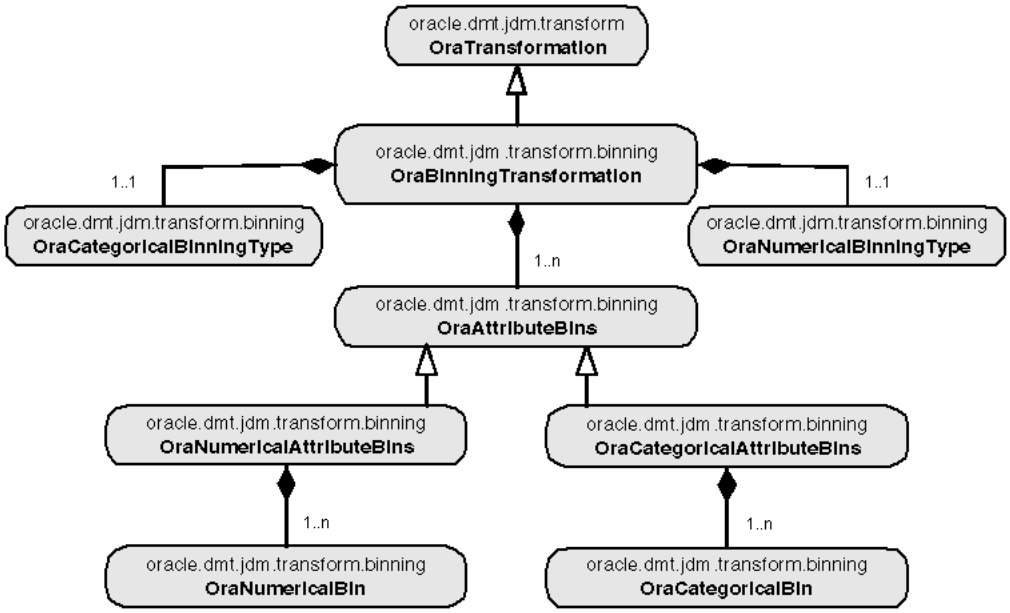


Using Binning/Discretization Transformation

Binning is the process of grouping related values together, thus reducing the number of distinct values for an attribute. Having fewer distinct values typically leads to a more compact model and one that builds faster, but it can also lead to some loss in accuracy.

The class diagram in Figure 7–9 illustrates the binning transformation classes.

Figure 7–9 OraBinningTransformation Class Diagram



Here, `OraBinningTransformation` contains all the settings required for binning. The ODM Java API supports top-n, custom binning for categorical attributes, and equi-width, quantile and custom binning for numerical attributes. After running the binning transformations, it creates a transformed table and bin boundary tables in the user's schema. The user can specify the bin boundary table names, or the system will

generate the names for the bin boundary tables. This facilitates the reusing of the bin boundary tables that are created for binning build data for apply and test data.

The following code illustrates the binning operation on the view MINING_BUILD_DATA_V

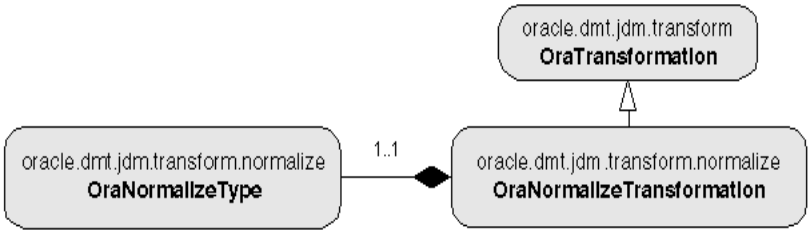
```
//Create binning transformation instance
OraBinningTransformFactory binXformFactory =
   (OraBinningTransformFactory)dmeConn.getFactory(
      "oracle.dmt.jdm.transform.binning.OraBinningTransform");
OraBinningTransform binTransform = m_binXformFactory.create(
      "MINING_DATA_BUILD_V", // name of the input data set
      "BINNED_DATA_BUILD_V", // name of the transformation result
      true); // result of the transformation is a view
// Specify the number of numeric bins
binTransform.setNumberOfBinsForNumerical(10);
// Specify the number of categoric bins
binTransform.setNumberOfBinsForCategorical(8);
// Specify the list of excluded attributes
String[] excludedList = new String[]{"CUST_ID", "CUST_GENDER"};
binTransform.setExcludeColumnList(excludedList);
// Specify the type of numeric binning: equal-width or quantile
       ( default is quantile )
binTransform.setNumericalBinningType(binningType);
// Specify the type of categorical binning as Top-N: by default it is none
binTransform.setCategoricalBinningType(OraCategoricalBinningType.top_n);
//Create transformation task
OraTransformationTask xformTask = m_xformTaskFactory.create(binTransform);
//Save transformation task object
dmeConn.saveObject("JDM_BINNING_TASK", xformTask, true);
//Execute transformation task asynchronously in the database
ExecutionHandle execHandle = dmeConn.execute("JDM_ BINNING _TASK");
//Wait for completion of the task
ExecutionStatus execStatus = execHandle.waitForCompletion(Integer.MAX_VALUE);
```

Using Normalization Transformation

Normalizing converts individual attribute values in such a way that all attribute values lie in the same range. Normally, values are converted to be in the range 0.0 to 1.0 or the range -1 to +1. Normalization ensures that attributes do not receive artificial weighting caused by differences in the ranges that they span.

The class diagram in Figure 7–10 illustrates the normalization transformation classes.

Figure 7–10 OraNormalizeTransformation Class Diagram



Here, OraNormalizeTransformation contains all the settings required for normalization. The ODM Java API supports z-Score, min-max, and linear scale normalizations. Normalization is required for SVM, NMF, and *k*-Means algorithms.

The following code illustrates normalization on the view MINING_BUILD_DATA_V.

```
//Create OraNormalizationFactory
OraNormalizeTransformFactory normalizeXformFactory =
  (OraNormalizeTransformFactory)m_dmeConn.getFactory(
      "oracle.dmt.jdm.transform.normalize.OraNormalizeTransform");
//Create OraNormalization
OraNormalizeTransform normalizeTransform = m_normalizeXformFactory.create(
      "MINING_DATA_BUILD_V", // name of the input data set
      "NORMALIZED_DATA_BUILD_V", // name of the transformation result
      true, // result of the transformation is a view
      OraNormalizeType.z_Score, //Normalize type
      new Integer(6) ); //Rounding number
// Specify the list of excluded attributes
String[] excludedList = new String[]{"CUST_ID", "CUST_GENDER"};
normalizeTransform.setExcludeColumnList(excludedList);
//Create transformation task
OraTransformationTask xformTask = m_xformTaskFactory.create(normalizeTransform);
//Save transformation task object
dmeConn.saveObject("JDM_NORMALIZE_TASK", xformTask, true);
//Execute transformation task asynchronously in the database
ExecutionHandle execHandle = dmeConn.execute("JDM_NORMALIZE_TASK");
//Wait for completion of the task
ExecutionStatus execStatus = execHandle.waitForCompletion(Integer.MAX_VALUE);
```

Using Clipping Transformation

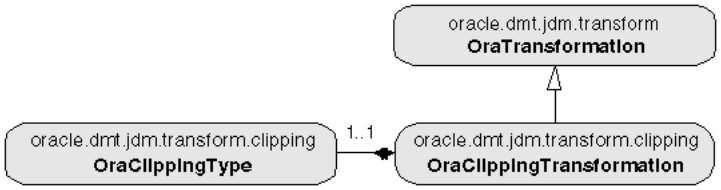
Some computations on attribute values can be significantly affected by extreme values. One approach to achieving a more robust computation is to either winsorize or trim the data using clipping transformations.

Winsorizing involves setting the tail values of a particular attribute to some specified value. For example, for a 90% winsorization, the bottom 5% are set equal to the minimum value in the 6th percentile, while the upper 5% are set equal to the value corresponding to the maximum value in the 95th percentile.

Trimming "removes" the tails in the sense that trimmed values are ignored in further values. This is achieved by setting the tails to NULL.

The class diagram in Figure 7–11 illustrates the clipping transformation classes.

Figure 7–11 OraClippingTransformation Class Diagram



Here, `OraClippingTransformation` contains all the settings required for clipping. The ODM Java API supports winsorize and trim types of clipping.

The following code illustrates clipping on the view `MINING_BUILD_DATA_V`.

```
//Create OraClippingTransformFactory
OraClippingTransformFactory clipXformFactory =
  (OraClippingTransformFactory)dmeConn.getFactory(
      "oracle.dmt.jdm.transform.clipping.OraClippingTransform");
//Create OraClippingTransform
OraClippingTransform clipTransform = clipXformFactory.create(
      "MINING_DATA_BUILD_V", // name of the input data set
```

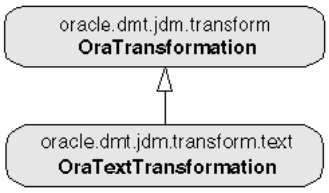
```
      "WINSORISED_DATA_BUILD_V", // name of the transformation result
      true );// result of the transformation is a view
//Specify the list of excluded attributes
String[] excludedList = new String[]{"CUST_ID", "CUST_GENDER"};
clipTransform.setExcludeColumnList(excludedList);
//Specify the type of clipping
clipTransform.setClippingType(OraClippingType.winsorize);
// Specify the tail fraction as 3% of values on both ends
clipTransform.setTailFraction(0.03);
//Create and save transformation task
OraTransformationTask xformTask = xformTaskFactory.create(clipTransform);
//Save transformation task object
dmeConn.saveObject("JDM_CLIPPING_TASK", xformTask, true);
//Execute transformation task asynchronously in the database
ExecutionHandle execHandle = dmeConn.execute("JDM_CLIPPING_TASK");
//Wait for completion of the task
ExecutionStatus execStatus = execHandle.waitForCompletion(Integer.MAX_VALUE);
```

Using Text Transformation

Text columns need to be transformed to nested table structure to do the mining on text columns. This transformation converts the text columns to nested table columns. A features table is created by text transformation. A model build text data column features table must be used for apply and test tasks to get the correct results.

The class diagram in Figure 7–12 illustrates the text transformation classes.

Figure 7–12 Text Transformation Class Diagram



Here, `OraTextTransformation` is used to specify the text columns and the feature tables associated with the text columns.

The following code illustrates clipping on the table `MINING_BUILD_TEXT`.

```
//Create OraTextTransformFactory
OraTextTransformFactory textXformFactory = dmeConn.getFactory(
      "oracle.dmt.jdm.transform.text.OraTextTransform");
//Create OraTextTransform
OraTextTransform txtXform = (OraTextTransformImpl)textXformFactory.create(
      "MINING_BUILD_TEXT", // name of the input data set
      "NESTED_TABLE_BUILD_TEXT ", // name of the transformation result
      "CUST_ID", //Case id column
      new String[] { "COMMENTS" } ); //Text column names
      );
//Create transformation task
OraTransformationTask xformTask = m_xformTaskFactory.create(txtXform);
//Save transformation task object
dmeConn.saveObject("JDM_TEXTXFORM_TASK", xformTask, true);
//Execute transformation task asynchronously in the database
ExecutionHandle execHandle = dmeConn.execute("JDM_TEXTXFORM_TASK");
//Wait for completion of the task
ExecutionStatus execStatus = execHandle.waitForCompletion
     (Integer.MAX_VALUE);
```

8

Converting to the ODM 10.2 Java API

This chapter will assist you in converting your data mining applications from the 10.1 proprietary Java API to the standard-compliant Java API available with Oracle 10*g* Release 2 (10.2).

See Also:

- JSR-000073 Data Mining API page of the Java Community Process Web Site at `http://jcp.org/aboutJava/communityprocess/final/jsr073`
- JDM 1.0 javadoc at `http://www.oracle.com/technology/products/bi/odm`
- *Oracle Data Mining Java API Reference* (ODM 10.2 javadoc)

This chapter includes the following topics:

Comparing the 10.1 and 10.2 Java APIs

The new ODM Java API available with Oracle 10*g* Release 2 (10.2)is standardized under the Java Community Process and is fully compliant with the JDM 1.0 standard. Oracle supports open standards for Java and is one of the primary vendors that implements JDM.

The ODM 10.2 JDM-based API replaces the proprietary Java API for data mining that was available with Oracle 10.1.

Note: The proprietary Java API is no longer supported in ODM 10.2.

If you have created applications in 10.1 and you want to use them in your Oracle 10.2 installation, you must convert them to use the 10.2 API.

Table 8–1 lists the major differences between the ODM 10.1 and ODM 10.2 Java APIs.

Table 8–1 Differences Between Oracle 10.1 and 10.2 Java APIs for Data Mining

Feature	ODM 10.1 Java API	ODM 10.2 Java API
Standards	Oracle proprietary Java API designed for accessing data mining functionality in the Database. Not supported in Oracle 10.2.	Java industry standard API defined under Java Community Process (JCP). ODM 10.2 implements conformant subsets of the standard along with Oracle proprietary extensions.
Interoperability with `DBMS_DATA_MINING` PL/SQL API	Not interoperable with models created by the PL/SQL API.	Interoperable with PL/SQL API. All objects created using the ODM 10.2 Java API can be used with the PL/SQL API. Results and values are consistent with the PL/SQL API.
Functions and algorithms	Classification function ■ NB, ABN, SVM Clustering function ■ k-Means, O-Cluster Regression function ■ SVM Association function ■ Apriori Attribute Importance function ■ MDL Feature Extraction function ■ NMF	Classification function ■ NB, ABN, SVM, Tree Clustering function ■ k-Means (PL/SQL API version), OCluster Regression function ■ SVM Association function ■ Apriori Attribute Importance function ■ MDL Feature Extraction function ■ NMF
Object creation	Primarily designed as Java classes. Objects are instantiated using constructors or static `create` methods.	Uses the factory method pattern to instantiate objects. `javax.dataminig.Connection` is the primary factory for all other object factories. Oracle extensions follow the same pattern for object creation.
Task execution	Tasks executed by `oracle.dmt.odm.task.MiningTask`. `ExecutionHandle` and `MiningTaskStatus` used for task execution tracking. Asynchronous task execution implemented by `DBMS_JOB`.	Tasks executed by `javax.datamining.Connection`. `ExecutionHandle` and `ExecutionStatus` used for task execution tracking. Asynchronous task execution implemented by `DBMS_SCHEDULER`.
Data	Supports both physical and logical data representations. Supports transactional and non-transactional format. Transactional format enables sparse data representation and wide data (>1000 columns)	Supports only physical data representation. Logical data can be represented with database views. Supports nested tables in place of transactional format.
Settings for model building	Settings for model building created by `oracle.dmt.odm.settings.function.MiningFunctionSettings`	Settings for model building created by `javax.datamining.base.BuildSettings`. Settings are saved as a table in the user's schema. The name of the `BuildSettings` object must be unique in the namespace of the table object.
Model	Models represented by `oracle.dmt.odm.model.MiningModel`. The `MiningModel` object stores the automated transformation details.	Models represented by `javax.datamining.base.Model`. The `Model` object does not store transformation details. Applications must manage the transformation details.
Cost matrix	Cost matrix represented by `oracle.dmt.odm.CostMatrix`. Cost matrix for all classification algorithms is specified at build time, even though the cost matrix is used as a post-processing step to the apply operation.	Cost matrix represented by `javax.datamining.supervised.classification.CostMatrix`. Cost matrix for the decision tree algorithm is specified at build time. All other classification algorithms are specified with apply and test operations.

Table 8–1 (Cont.) Differences Between Oracle 10.1 and 10.2 Java APIs for Data Mining

Feature	ODM 10.1 Java API	ODM 10.2 Java API
Model detail	Model details not represented as an object. Model details are stored with the associated model object.	Model details represented by `javax.dataminig.base.ModelDetail`.
Apply settings	Apply settings represented by `oracle.dmt.odm.result.MiningApplyOutput`.	Apply settings represented by `javax.datamining.task.apply.ApplySettings`.
Results object	Mining results represented by `oracle.dmt.odm.result.MiningResult`.	Mining results are not explicit objects. Each task creates either a Java object or a database object such as a table.
Transformations	Supports automated data preparation. Provides utility methods for external and embedded data preparation.	Does not support automated transformations. The transformation task `oracle.dmt.jdm.task.OraTransformationTask` can be used to emulate automated transformations.
Text transformation	Supports text data types, such as CLOB and BLOB, for SVM and NMF. No explicit text transformations are provided.	Supports explicit text transformations. These can be used with any algorithm to emulate text data type support.

Converting Your Applications

Most objects in the ODM 10.2 API are similar to the objects in the ODM 10.1 API. However, there are some major differences in class names, package structures, and object usage. Some of the primary differences are:

- In 10.1, all primary objects are created using constructors or `create` methods. In 10.2, objects are created using object factories, as described in "Connection Factory" on page 7-3 and "Features of a DMS Connection" on page 7-4.
- In 10.1, DMS metadata-related operations are distributed in each class. In 10.2, most DMS metadata-related operations are centralized in a `Connection` object. For example, a mining task is restored in 10.1 with the `MiningTask.restore` method and in 10.2 with the `Connection.retrieveObject` method.
- In 10.1, all named objects are persisted in the database. In 10.2, `PhysicalDataSet` and `ApplySettings` are transient objects.

> **Note:** Although the ODM 10.1 Java API is incompatible with Oracle 10.2, future releases will follow the backward compatibility scheme proposed by the JDM standard.

Table 8–2 provides sample code for performing various mining operations in both 10.1 and 10.2. Refer to Chapter 6 for additional 10.2 code samples.

Table 8–2 Sample Code from 10.1 and 10.2 ODM Java APIs

ODM 10.1 Java API	ODM 10.2 Java API
Connect to the DMS	**Connect to the DMS**

`//Create a DMS object`
`DataMiningServer m_dms = new DataMiningServer`
`( "put DB URL here", //JDBC URL`
`"user name", //User Name`
`"password" //Password`
`);`
`//Login to the DMS and create a DMS Connection`
`m_dmsConn = m_dms.login();`
	`//Create ConnectionFactory & connection`
	`OraConnectionFactory m_dmeConnFactory =`
	`new OraConnectionFactory();`
	`ConnectionSpec connSpec =`
	`m_dmeConnFactory.getConnectionSpec();`
	`connSpec.setURI( "put DB URL here" );`
	`connSpec.setName( "user name" );`
	`connSpec.setPassword( "password" );`
	`m_dmeConn =`
	`m_dmeConnFactory.getConnection( connSpec );`

Table 8–2 (Cont.) Sample Code from 10.1 and 10.2 ODM Java APIs

ODM 10.1 Java API	ODM 10.2 Java API
Create a PhysicalDataSpecification	**Create and Save PhysicalDataSet**

`LocationAccessData lad = new LocationAccessData`
`( "MINING_DATA_BUILD_V", //Table/view Name`
`"DMUSER" //Schema Name`
`);`
`PhysicalDataSpecification pds =`
`newNonTransactionalDataSpecification (lad);`
	`PhysicalDataSetFactory pdsFactory =`
	`( PhysicalDataSetFactory )m_dmeConn.getFactory`
	`( "javax.datamining.data.PhysicalDataSet" );`
	`m_paFactory = ( PhysicalAttributeFactory )`
	`m_dmeConn.getFactory`
	`( "javax.datamining.data.PhysicalAttribute" );`
	`PhysicalDataSet buildData = m_pdsFactory.create`
	`( "MINING_DATA_BUILD_V",false );`
	`PhysicalAttribute pa =m_paFactory.create`
	`( "cust_id", AttributeDataType.integerType,`
	`PhysicalAttributeRole.caseId );`
	`buildData.addAttribute( pa );`
	`m_dmeConn.saveObject( "nbBuildData", buildData, true );`

Create and Save MiningFunctionSettings	**Create BuildSettings**

`NaiveBayesSettings nbAlgo =`
`new NaiveBayesSettings (0.01f, 0.01f);`
`ClassificationFunctionSettings mfs =`
`ClassificationFunctionSettings.create`
`( m_dmsConn, //DMS Connection`
`nbAlgo, //NB algorithm settings`
`pds, //Build data specification`
`"AFFINITY_CARD", //Target column`
`AttributeType.categorical, //Attribute type`
`DataPreparationStatus.unprepared`
`);`
`//Set Cust_ID attribute as inactive`
`mfs.adjustAttributeUsage( new String[]{"CUST_ID"},`
`AttributeUsage.inactive );`
`mfs.store( m_dmsConn,"NBDemo_MFS" );`
	`m_clasFactory = ( ClassificationSettingsFactory )`
	`m_dmeConn.getFactory`
	`( "javax.datamining.supervised.classification.`
	`ClassificationSettings" );`
	`m_nbFactory = ( NaiveBayesSettingsFactory )`
	`m_dmeConn.getFactory`
	`("javax.datamining.algorithm.naivebayes.`
	`NaiveBayesSettings");`
	`//Create NB algorithm settings`
	`NaiveBayesSettings nbAlgo = m_nbFactory.create();`
	`nbAlgo.setPairwiseThreshold( 0.01f );`
	`nbAlgo.setSingletonThreshold( 0.01f );`
	`//Create ClassificationSettings`
	`ClassificationSettings buildSettings =`
	`m_clasFactory.create();`
	`buildSettings.setAlgorithmSettings(nbAlgo);`
	`buildSettings.setTargetAttributeName`
	`( "affinity_card");`
	`m_dmeConn.saveObject`
	`("nbBuildSettings",buildSettings,true);`

Create and Execute MiningBuildTask	**Create and Execute BuildTask**

`MiningBuildTask buildTask =`
`new MiningBuildTask`
`( pds, //Build data specification`
`"NBDemo_MFS", //Mining function settings`
`"NBDemo_Model" //Mining model name`
`);`
`//Store the taskbuild`
`buildTask.store( m_dmsConn,"NBDemoBuildTask" );`
`Task.execute( m_dmsConn );`
`//Wait for completion of the task`
`MiningTaskStatus taskStatus =`
`buildTask.waitForCompletion( m_dmsConn );`
	`m_buildFactory = ( BuildTaskFactory )`
	`m_dmeConn.getFactory`
	`( "javax.datamining.task.BuildTask" );`
	`BuildTask buildTask = m_buildFactory.create`
	`( "nbBuildData", //Build data specification`
	`"nbBuildSettings", //build settings name`
	`"nbModel" //Mining model namem_dme`
	`);`
	`Conn.saveObject( "nbBuildTask", taskObj, true );`
	`ExecutionHandle execHandle =`
	`m_dmeConn.execute( taskName );`
	`ExecutionStatus status =`
	`execHandle.waitForCompletion(`
	`Integer.MAX_VALUE);`

Retrieve MiningModel	**Retrieve Model**

`NaivebayesModel model = ( NaiveBayesModel )`
`SupervisedModel.restore`
`( m_dmeConn, "NBDemo_Model" );`
	`ClassificationModel model = ( ClassificationModel )`
	`m_dmeConn.retrieveObject`
	`( "nbModel", NamedObject.model );`

Table 8–2 (Cont.) Sample Code from 10.1 and 10.2 ODM Java APIs

ODM 10.1 Java API	ODM 10.2 Java API
Evaluate the Model	**Evaluate the Model**

`//Compute accuracy & confusionmatrix`
`LocationAccessData lad = new LocationAccessData`
`  ( "MINING_DATA_TEST_V", "DMUSER" ); //Schema Name`
`PhysicalDataSpecification pds =`
`  new NonTransactionalDataSpecification( lad );`
`ClassificationTestTask testTask =`
`  new ClassificationTestTask ( pds,"NBDemo_Model",`
`  "NBDemo_TestResults" );`
`testTask.store( m_dmsConn, "NBDemoTestTask" );`
`testTask.execute( m_dmsConn );`
`MiningTaskStatus taskStatus =`
`  testTask.waitForCompletion( m_dmsConn );`
`ClassificationTestResult testResult =`
`  ClassificationTestResult.restore`
`  ( m_dmsConn, "NBDemo_TestResults" );`
`float accuracy = testResult.getAccuracy();`
`CategoryMatrix confusionMatrix =`
`  TestResult.getConfusionMatrix();`
`//Compute lift`
`Category positiveCategory = new Category`
`  ( "Positive value", "1",DataType.intType );`
`MiningLiftTask liftTask =`
`  new MiningLiftTask`
`  ( pds,`
`    10,           //Number of quantiles to be used`
`    positiveCategory,    /positive target value`
`   "NBDemo_Model",       // model to be tested`
`   "NBDemo_LiftResults"  //Lift results name`
`  );`
`liftTask.store( m_dmsConn, "NBDemoLiftTask" );`
`liftTask.execute( m_dmsConn );`
`MiningTaskStatus taskStatus =`
`  liftTask.waitForCompletion( m_dmsConn );`
`MiningLiftResult liftResult =`
`  MiningLiftResult.restore`
` ( m_dmsConn,"NBDemo_LiftResults" );`
	`//Compute accuracy, confusion matrix, lift & roc`
	`PhysicalDataSet testData = m_pdsFactory.create`
	`  ( "MINING_DATA_TEST_V", false );`
	`PhysicalAttribute pa = m_paFactory.create`
	`  ( "cust_id", AttributeDataType.integerType,`
	`    PhysicalAttributeRole.caseId );`
	`testData.addAttribute( pa );`
	`m_dmeConn.saveObject`
	`  ( "nbTestData", testData, true );`
	`ClassificationTestTask testTask = m_testFactory.create`
	`  ( "nbTestData", "nbModel", "nbTestMetrics" );`
	`testTask.setNumberOfLiftQuantiles( 10 );`
	`testTask.setPositiveTargetValue( new Integer(1) );`
	`m_dmeConn.saveObject( "nbTestTask", testTask, true );`
	`ExecutionHandle execHandle =`
	`  m_dmeConn.execute("nbTestTask");`
	`ExecutionStatus status = execHandle.waitForCompletion`
	`  ( Integer.MAX_VALUE );`
	`ClassificationTestMetrics testMetrics =`
	`  ( ClassificationTestMetrics )`
	`  m_dmeConn.retrieveObject`
	`  ( "nbTestMetrics", NamedObject.testMetrics );`
	`Double accuracy = testMetrics.getAccuracy();`
	`ConfusionMatrix confusionMatrix =`
	`   testMetrics.getConfusionMatrix();`
	`Lift lift = testMetrics.getLift();`
	`ReceiverOperatingCharacterics roc =`
	`  testMetrics.getROC();`

Table 8–2 (Cont.) Sample Code from 10.1 and 10.2 ODM Java APIs

ODM 10.1 Java API	ODM 10.2 Java API
Apply the Model	**Apply the Model**

`LocationAccessData lad = new LocationAccessData`
`  ( "MINING_DATA_APPLY_V", "DMUSER");`
`PhysicalDataSpecification pds =`
`  new NonTransactionalDataSpecification( lad );`
`MiningApplyOutput mao =`
`  MiningApplyOutput.createDefault();`
`MiningAttribute srcAttribute =  new MiningAttribute`
`  ( "CUST_ID", DataType.intType,`
`    AttributeType.notApplicable`
`  );`
`Attribute destAttribute = new Attribute`
`  ("CUST_ID", DataType.intType);`
`ApplySourceAttributeItem m_srcAttrItem =`
`  new ApplySourceAttributeItem`
`  ( srcAttribute,destAttribute);`
`mao.addItem(m_srcAttrItem);`
`LocationAccessData outputTable =`
`  new LocationAccessData`
`  ( "NBDemo_Apply_Output", "DMUSER");`
`MiningApplyTask applyTask = new MiningApplyTask`
`  ( pds,                //test data specification`
`   "NBDemo_Model",      //Input model name`
`    mao,                //MiningApplyOutput object`
`    outputTable,        //Apply output table`
`   "NBDemo_ApplyResults"  //Apply results`
`  );`
`applyTask.store( m_dmsConn, "NBDemoApplyTask" );`
`applyTask.execute( m_dmsConn );`
`MiningTaskStatus taskStatus =`
`  applyTask.waitForCompletion( m_dmsConn );`
	`PhysicalDataSet applyData = m_pdsFactory.create`
	`  ( "MINING_DATA_APPLY_V", false );`
	`PhysicalAttribute pa = m_paFactory.create`
	`  ( "cust_id",`
	`    AttributeDataType.integerType,`
	`    PhysicalAttributeRole.caseId`
	`  );`
	`applyData.addAttribute( pa );`
	`m_dmeConn.saveObject( "nbApplyData",applyData,true );`
	`ClassificationApplySettings clasAS =`
	`  m_applySettingsFactory.create();`
	`m_dmeConn.saveObject( "nbApplySettings",clasAS,true );`
	`DataSetApplyTask applyTask = m_dsApplyFactory.create`
	`  ( "nbApplyData",`
	`    "nbModel",`
	`    "nbApplySettings",`
	`    "nb_apply_output"`
	`  );`
	`m_dmeConn.saveObject`
	`  ( "nbApplyTask",`
	`    applyTask,`
	`    true`
	`  );`
	`ExecutionHandle execHandle =`
	`  m_dmeConn.execute( "nbApplyTask" );`
	`ExecutionStatus status =`
	`   execHandle.waitForCompletion( Integer.MAX_VALUE );`

9

Sequence Matching and Annotation (BLAST)

This chapter describes table functions included with ODM that permit you to perform similarity searches against nucleotide and amino acid sequence data stored in an Oracle database. You can use the table functions described in this chapter for ad hoc searches or you can embed them in applications. The inclusion of these table functions in ODM positions Oracle as a platform for bioinformatics.

This chapter discusses the following topics:

NCBI BLAST

The National Center for Biotechnology Information (NCBI) developed one of the commonly used versions of the Basic Local Alignment Search Tool (BLAST).

Sequence alignments provide a way to compare new sequences with previously characterized sequences. Both functional and evolutionary information can be inferred from well-designed queries and alignments. BLAST provides a method for searching of both nucleotide and protein databases. Since the BLAST algorithm detects local alignments, regions of similarity embedded in otherwise unrelated sequences can be detected.

The BLAST algorithm searches nucleotide and amino acid query sequences against databases of nucleotide and amino acid sequences. Based on the nature of the query and the database sequences, NCBI BLAST provides the following variants:

- BLASTP compares an amino acid query sequence against an amino acid sequence database.
- BLASTN compares a nucleotide query sequence against a nucleotide sequence database.
- BLASTX compares a nucleotide query sequence translated along all six reading frames (both strands) against a amino acid sequence database.
- TBLASTN compares an amino acid query sequence against a nucleotide sequence database translated along all six reading frames (both strands).
- TBLASTX compares the six-frame translations of a nucleotide query sequence against the six-frame translations of a nucleotide sequence database.

For more information about NCBI BLAST, see the NCBI BLAST Home Page at

`http://www.ncbi.nlm.nih.gov/BLAST`

The table functions described in this chapter implement some of the variants of NCBI BLAST version 2.0.

Using ODM BLAST

This section contains several examples of using the ODM BLAST table functions to perform searches on nucleotide or amino acid sequences.

Most table function parameters have defaults. The defaults were carefully chosen so that users who have limited experience with BLAST should obtain good results.

Using BLASTN_MATCH to Search DNA Sequences

The BLAST table functions accept the CLOB (Character Large OBject) data type as the query sequence. It is not possible to construct a CLOB in an ad hoc SQL query. One way to construct a CLOB is to create a table and insert the query sequence into the table. Another option is to construct a CLOB using the programmatic interface if the BLAST query is part of a larger program. Suppose that the table `query_db` has the schema `(sequence_id VARCHAR2(32), sequence CLOB)`. The following SQL query inserts the query sequence into `query_db`:

```
INSERT INTO query_db VALUES ('1', 'AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGT');
```

Suppose that the table GENE_DB stores DNA sequences. Suppose that GENE_DB has attributes `seq_id`, `publication date`, `modification date`, `organism`, and `sequence`, among other attributes. There is no required schema for the table that stores the sequences. The only requirement is that the table contain an identifier and the sequence and any number of other optional attributes.

The portion of the database to be used for the search can be specified using SQL. The full power of SQL can be used to perform sophisticated selections.

Searching for Good Matches in DNA Sequences

The following query does a BLAST search of the given query sequence against the human genome and returns the `seq_id`, `score`, and `expect` value of matches that score > 25:

```
SELECT t.t_seq_id, t.score, t.expect
FROM TABLE (
    BLASTN_MATCH (
       (SELECT sequence FROM query_db WHERE sequence_id = '1'),
       CURSOR (SELECT seq_id, sequence FROM GENE_DB
        WHERE organism = 'human'),
       1,
       -1,
       0,
       0,
       10,
       0,
       0,
       0,
       0,
       11,
       0,
       0)
) t WHERE t.score > 25;
```

Note: The parameter value of 0 invokes the default values in most cases. See the syntax for details.

Searching DNA Sequences Published After a Certain Date

The following query does the BLAST search against all sequences published after Jan 01, 2000:

```
SELECT t.t_seq_id, t.score, t.expect
FROM TABLE (
    BLASTN_MATCH (
       (SELECT sequence FROM query_db WHERE sequence_id = '1'),
       CURSOR (SELECT seq_id, sequence FROM GENE_DB
        WHERE publication_date > '01-JAN-2000'),
       1,
       -1,
       0,
       0,
       10,
       0,
       0,
       0,
       0,
       11,
       0,
       0)
) t WHERE t.score > 25;
```

You can obtain other attributes of the matching sequence by joining the BLAST result with the original sequence table as follows:

```
SELECT t.t_seq_id, t.score, t.expect, g.publication_date, g.organism
FROM GENE_DB g, TABLE (
    BLASTN_MATCH (
       (SELECT sequence FROM query_db WHERE sequence_id = '1'),
       CURSOR (SELECT seq_id, sequence FROM GENE_DB
        WHERE publication_date > '01-JAN-2000'),
       1,
       -1,
       0,
       0,
       10,
       0,
       0,
       0,
       0,
       11,
       0,
       0)
) t WHERE t.t_seq_id = g.seq_id AND t.score > 25;
```

Using BLASTP_MATCH to Search Protein Sequences

Suppose that the table PROT_DB stores protein sequences. Insert the protein query sequence to be used for the search into `query_db`.

Searching for Good Matches in Protein Sequences

The following query does a BLASTP search of the given query sequence against protein sequences in PROT_DB and returns the `identifier`, `score`, `name`, and `expect` value of matches that score > 25:

```
SELECT t.t_seq_id, t.score, t.expect, p.name
FROM PROT_DB p, TABLE(
       BLASTP_MATCH (
         (SELECT sequence FROM query_db WHERE sequence_id = '2'),
         CURSOR(SELECT seq_id, sequence FROM PROT_DB),
         1,
         -1,
         0,
         0,
         'BLOSUM62',
         10,
         0,
         0,
         0,
         0,
         0)
       )t WHERE t.t_seq_id = p.seq_id AND t.score > 25
          ORDER BY t.expect;
```

Using BLASTN_ALIGN to Search and Align DNA Sequences

Suppose that the table GENE_DB stores DNA sequences. Suppose that GENE_DB has attributes `seq_id`, `publication date`, `modification date`, `organism`, and `sequence` among other attributes.

Searching and Aligning for Good Matches in DNA Sequences

The following query does a BLAST search and alignment of the given query sequence against the human genes and returns the `publication_date`, `organism`, and the alignment attributes of the matching sequences that score > 25 and where more than 50% of the sequence is conserved in the match:

```
SELECT t.t_seq_id, t.alignment_length, t.pct_identity,
       t.q_seq_start, t.q_seq_end, t.t_seq_start, t.t_seq_end,
       t.score, t.expect, g.publication_date, g.organism
FROM GENE_DB g, TABLE (
    BLASTN_ALIGN (
       (SELECT sequence FROM query_db WHERE sequence_id = '1'),
       CURSOR (SELECT seq_id, sequence FROM GENE_DB
        WHERE publication_date > '01-JAN-2000'),
       1,
       -1,
       0,
       0,
       10,
       0,
       0,
       0,
       0,
       11,
       0,
       0)
) t WHERE t.t_seq_id = g.seq_id AND t.score > 25
    AND t.pct_identity > 50;
```

You can use BLASTP_ALIGN and TBLAST_ALIGN in a similar way.

Output of BLAST Queries

The output of a BLAST query is a table; the output table is described as the output table for the specific query.

Here are two examples of queries and the resulting output tables.

Query 1 is as follows:

```
select T_SEQ_ID AS seq_id, score, EXPECT as evalue
  from TABLE(
       BLASTP_MATCH (
         (select sequence from query_db),
         CURSOR(SELECT seq_id, seq_data
                FROM swissprot
                WHERE organism = 'Homo sapiens (Human)'),
         1,
         -1,
         0,
         0,
         'BLOSUM62',
         10,
         0,
         0,
         0,
         0,
         0)
       );
```

The output for query 1 is as follows:

```
SEQ_ID        SCORE     EVALUE
-------- ----------     ----------
P31946           205     5.8977E-18
Q04917           198     3.8228E-17
P31947           169     8.8130E-14
P27348           198     3.8228E-17
P58107            49     7.24297332
```

Query 2 is as follows:

```
select T_SEQ_ID AS seq_id, ALIGNMENT_LENGTH as len,
       Q_SEQ_START as q_strt, Q_SEQ_END as q_end, Q_FRAME, T_SEQ_START as t_strt,
       T_SEQ_END as t_end, T_FRAME, score, EXPECT as evalue
  from TABLE(
       BLASTP_ALIGN (
         (select sequence from query_db),
         CURSOR(SELECT seq_id, seq_data
                FROM swissprot
                WHERE organism = 'Homo sapiens (Human)' AND
                      creation_date > '01-Jan-90'),
         1,
         -1,
         0,
         0,
         'BLOSUM62',
```

```
        10,
        0,
        0,
        0,
        0,
        0)
      );
```

The output for Query 2 is as follows:

```
SEQ_ID    LEN Q_STRT Q_END Q_FRAME T_STRT T_END T_FRAME   SCORE  EVALUE
-------- ---- ------ ----- ------- ------ ----- ------- ------- ----------
P31946     50      0    50       0     13    63       0     205  5.1694E-18
Q04917     50      0    50       0     12    62       0     198  3.3507E-17
P31947     50      0    50       0     12    62       0     169  7.7247E-14
P27348     50      0    50       0     12    62       0     198  3.3507E-17
P58107     21     30    51       0    792   813       0      49  6.34857645
```

Using BLASTN_COMPRESS to Improve Search Performance

If you perform frequent BLAST searches on nucleotide sequences, performance improves significantly when the data set of sequences is transformed into a compressed binary format, and the compressed data is used in the searches. The BLASTN_COMPRESS() function transforms a nucleotide data set represented as CLOBs into compressed binary format represented as BLOBs.

Compress Sequences

Suppose that the table GENE_DB contains DNA sequences upon which you will perform frequent searches. Suppose that GENE_DB has attributes `(seq_id, publication date, modification date, organism, sequence)` among other attributes. The following query stores all human DNA sequences in compressed binary format, in the table COMPRESSED_HUMAN_GENES.

```
create table COMPRESSED_HUMAN_GENES as
select seq_id, seq_data
from Table(BLASTN_COMPRESS (
 from GENE_DB
 where organism = 'human')))
```

The portion of the database to be compressed can be specified using SQL. The full power of SQL can be used to perform more sophisticated selections involving joins.

Passing a Compressed Sequence to a BLAST Function

The compressed sequences can be directly passed to BLAST match and align functions as shown in the following example.

```
select t.t_seq_id, t.alignment_length, t.pct_identity, t.q_start, t.q_end, t.s_
start, t.s_end, t.score, t.expect, g.publication_date, g.organism
from GENE_DB g, Table(BLASTN_ALIGN (
select sequence from QUERY_SEQ where id = '1'),
seqdb_cursor => cursor(select seq_id, seq_data
from Table(BLASTN_COMPRESS (
cursor(select seq_id, sequence
           from GENE_DB
           where organism = 'human')))),
expect_value => 5,
word_size => 12)) t
```

```
where t.t_seq_id = g.identifier
AND t.score > 25
AND t.pct_identity > 50;
```

Sample Data for BLAST

We provide a few sample data sets and queries that can be used to verify that the BLAST functions work correctly after ODM is installed.

The DM_USER schema contains the following sequence data tables:

- SWISSPROT
- PROT_DB
- ECOLI10

SWISSPROT Table

The SWISSPROT table contains the sequences in Release 40 of the SwissProt database. This table has the sequence identifier, creation_date, organism, and sequence_data attributes. It has 101,602 protein sequences.

```
SQL> describe SWISSPROT;
Name                                      Null?    Type
----------------------------------------- -------  -------------
SEQ_ID                                             VARCHAR2(32)
CREATION_DATE                                      DATE
ORGANISM                                           VARCHAR2(256)
SEQ_DATA                                           CLOB
```

PROT_DB Table

The PROT_DB table consists of 19 protein sequences from Release 40 of the SwissProt data set.

```
SQL> describe prot_db;
Name                                      Null?    Type
----------------------------------------- -------  -------------
SEQ_ID                                             VARCHAR2(32)
SEQ_DATA                                           CLOB
```

ECOLI10 Table

The ECOLI10 table contains 10 nucleotide sequences from the Escherichia coli data set.

```
SQL> describe ECOLI10;
Name                                      Null?    Type
----------------------------------------- -------- ---------------
SEQ_ID                                             VARCHAR2(32)
SEQ_DATA                                           CLOB
```

Genetic Codes and Names

Table 9–1 lists genetic codes and associated names.

Table 9–1 Table of Genetic Codes

Genetic Code	Name
1	Standard
2	Vertebrate Mitochondrial
3	Yeast Mitochondrial
4	Mold Mitochondrial, Protozoan Mitochondrial, Coelenterate Mitochondrial, Mycoplasma, Spiroplasm
5	Invertebrate Mitochondrial
6	Ciliate Nuclear, Dasycladacean Nuclear, Hexamita Nuclear
9	Echinoderm Mitochondrial
10	Euplotid Nuclear
11	Bacterial and Plant Plastid
12	Alternative Yeast Nuclear
13	Ascidian Mitochondrial
14	Flatworm Mitochondrial
15	Blepharisma Macronuclear
16	Chlorophycean Mitochondrial
21	Trematode Mitochondrial
22	Scenedesmus Obliquus Mitochondrial
23	Thraustochytrium Mitochondrial Code

Sequence Databases

There are several public domain sequence databases available. One of them is the SwissProt database, which is a highly curated collection of protein sequences. SwissProt has recently been combined with other databases to create UniProt. The last release of the SwissProt database can be downloaded from

```
ftp://ftp.ebi.ac.uk/pub/databases/swissprot/release/sprot45.dat
```

In addition to the raw sequence data, the SwissProt database contains several other attributes of the sequence including organism, date published, date modified, published literature references, annotations, and so on. BLAST requires only the sequence identifier and the sequence data to be stored to perform searches.

Depending on the needs of your specific application, different sets of these attributes may be important. Therefore, the database schema required to store the data needs to be appropriately designed. You can use a scripting language to parse the required fields from the SwissProt data and format the fields so that they can be loaded into an Oracle database.

The following Perl script outputs the sequence identifier, creation_date, organism, and sequence data in the required format for SQL*Loader. (SQL*Loader is the utility that loads data into an Oracle database; it is described in detail in *Oracle Database Utilities*.)

```
#!/bin/perl
#swissprot.pl < input > output
#Input: protein db as provided by SWISSPROT
#
my $string = "";
my $indicator = "";
```

```
$sq = 0;
$ac = 0;

while(<>)
{
    #chop;
    if ( /^\/\// ) {
      print "\n";
      $sq = 0;
      $ac = 0;
      next;
    }
    if ($sq == 1) {
        @words = split;
        foreach $word (@words) {
          print "$word";
        }
        next;
    }
    if( /^AC(\s+)(\w+);/ ) {
      if ($ac == 0) {
        $indicator = $2;
        print "$indicator|";
        $sq = 0;
        $dt = 0;
        $ac = 1;
        next;
      }
    }
    if ( /^OS(\s+)(.*)\./ ) {
        $organism = $2;
        print "$organism|";
        next;
    }
    if ( /^DT(\s+)(\S+)/ ) {
        if ($dt == 0) {
           print "$2|";
           $dt = 1;
        }
    }
    if ( /^SQ(\s+)/ ) {
        $sq = "1";
        next;
    }
}
```

Loading Sequences into an Oracle Database

Follow these steps to download, parse, and save the SwissProt data in an Oracle database:

1. Download SwisProt data to the file `sprot45.dat`.
2. Save the perl script in a file named `swissprot.pl`, type the command

   ```
   swissprot.pl sprot45.dat > sprot_formatted.txt
   ```

   This command will read the SwissProt data stored in `sprot45.dat`, format it, and write it out to `sprot_formatted.txt`.

3. In order to load the data using SQL*Loader, you must create a table to hold the data and a control file. Create the table `swissprot` using the following SQL statement:

```
create table swissprot (SEQ_ID VARCHAR2(32), CREATION_DATE DATE,
ORGANISM VARCHAR2(256), SEQ_DATA CLOB);
```

4. Create a control file named `sprot.ctl` with the following contents:

```
LOAD DATA
INFILE sprot40_formatted.txt
INTO TABLE swissprot
REPLACE
FIELDS TERMINATED BY '|'
TRAILING NULLCOLS
(
seq_id,
creation_date,
organism,
seq_data char(100000)
)
```

5. Finally, load the data:

```
sqlldr userid=<user_name>/<passwd> control=sprot.ctl log=sprot.log
direct=TRUE data=sprot40_formatted.txt
```

The SwisProt data is now stored in the Oracle table `swissprot`.

Summary of BLAST Table Functions

The BLAST functionality is available as table functions; these table functions can be used in the FROM clause of a SQL query.

Table 9–2 BLAST Table Functions

Table Function	Description
BLASTN_COMPRESS Table Function	Compress nucleotide sequence data to improve performance of sequence searches.
BLASTN_MATCH Table Function	Perform a search of the given nucleotide sequence against the selected portion of the nucleotide database
BLASTP_MATCH Table Function on page 9-15	Perform a search of the given amino acid sequence against the selected portion of the protein database
TBLAST_MATCH Table Function on page 9-17	Perform a search involving translations of either the query sequence or the database of sequences
BLASTN_ALIGN Table Function on page 9-19	Perform an alignment of the given nucleotide sequence against the selected portion of the nucleotide database
BLASTP_ALIGN Table Function on page 9-22	Perform an alignment of the given amino acid sequence against the selected portion of the protein database
TBLAST_ALIGN Table Function on page 9-25	Perform alignments involving translations of either the query sequence or the database of sequences

BLASTN_COMPRESS Table Function

This table function compresses nucleotide sequence data. It takes as input a cursor of sequence identifier and sequence data represented as a CLOB and returns the sequence identifier and a BLOB representing the sequence data in compressed binary format. The result of BLASTN_COMPRESS can be either materialized in a table for future use or passed into the BLAST search functions that accept nucleotide sequence data

Syntax

```
function BLASTN_COMPRESS (
  sequence_cursor REF CURSOR)
  return table of row (seq_id VARCHAR2, seq_data BLOB);
```

Parameters

Table 9–3 describes the input parameters for BLASTN_COMPRESS; Table 9–4, the output parameters.

Table 9–3 Input Parameters for BLASTN_COMPRESS Table Function

Parameter	Description
`sequence_cursor`	The cursor of the sequences to be compressed. The cursor has two columns the sequence identifier and the sequence string.

Table 9–4 Output Parameters for BLASTN_MATCH Table Function

Attribute	Description
`seq_id`	The sequence identifier of the sequence. The value returned is the same as the sequence identifier in the input cursor.
`seq_data`	The compressed sequence represented as a BLOB.

BLASTN_MATCH Table Function

This table function performs a BLASTN search of the given nucleotide sequence against the selected portion of the nucleotide database. The database can be selected using a standard SQL select and passed into the function as a cursor. It accepts the standard BLAST parameters that are listed in the following section. The match returns the identifier of the matched (target) sequence (`t_seq_id`) (for example, the NCBI accession number), the score of the match, and the expect value.

Syntax

```
function BLASTN_MATCH (
  query_seq CLOB,
  seqdb_cursor REF CURSOR,
  subsequence_from NUMBER default 1,
  subsequence_to NUMBER default -1,
  filter_low_complexity BOOLEAN default false,
  mask_lower_case BOOLEAN default false,
  expect_value NUMBER default 10,
  open_gap_cost NUMBER default 5,
  extend_gap_cost NUMBER default 2,
  mismatch_cost NUMBER default -3,
  match_reward NUMBER default 1,
  word_size NUMBER default 11,
  xdropoff NUMBER default 30,
  final_x_dropoff NUMBER default 50)
  return table of row (t_seq_id VARCHAR2, score NUMBER, expect NUMBER);
```

Parameters

Table 9–5 describes the input parameters for BLASTN_MATCH; Table 9–6, the output parameters.

Table 9–5 Input Parameters for BLASTN_MATCH Table Function

Parameter	Description
`query_seq`	The query sequence to search. This version of ODM BLAST accepts bare sequences only. A *bare sequence* is just lines of sequence data. Blank lines are not allowed in the middle of bare sequence input.
`seqdb_cursor`	The cursor parameter supplied by the user when calling the function. It should return two columns in its returning row, the sequence identifier and the sequence string.
`subsequence_from`	Start position of a region of the query sequence to be used for the search. The default is 1.
`subsequence_to`	End position of a region of the query sequence to be used for the search. If -1 is specified, the sequence length is taken as `subsequence_to`. The default is -1.
`filter_low_complexity`	`TRUE` or `FALSE`. If `TRUE`, the search masks off segments of the query sequence that have low compositional complexity. Filtering can eliminate statistically significant but biologically uninteresting regions, leaving the more biologically interesting regions of the query sequence available for specific matching against database sequences. Filtering is only applied to the query sequence. The default is `FALSE`.

Table 9–5 (Cont.) Input Parameters for BLASTN_MATCH Table Function

Parameter	Description
mask_lower_case	TRUE or FALSE. If TRUE, you can specify a sequence in upper case characters as the query sequence and denote areas to be filtered out with lower case. This customizes what is filtered from the sequence. The default is FALSE.
expect_value	The statistical significance threshold for reporting matches against database sequences. The default value is 10. Specifying 0 invokes default behavior.
open_gap_cost	The cost of opening a gap. The default value is 5. Specifying 0 invokes default behavior.
extend_gap_cost	The cost of extending a gap. The default value is 2. Specifying 0 invokes default behavior.
mismatch_cost	The penalty for nucleotide mismatch. The default value is -3. Specifying 0 invokes default behavior.
match_reward	The reward for a nucleotide match. The default value is 1. Specifying 0 invokes default behavior.
word_size	The word size used for dividing the query sequence into subsequences during the search. The default value is 11. Specifying 0 invokes default behavior.
xdropoff	Dropoff for BLAST extensions in bits. The default value is 30. Specifying 0 invokes default behavior.
final_x_dropoff	The final X dropoff value for gapped alignments in bits. The default value is 50. Specifying 0 invokes default behavior.

Table 9–6 Output Parameters for BLASTN_MATCH Table Function

Attribute	Description
t_seq_id	The sequence identifier of the returned match.
score	The score of the returned match.
expect	The expect value of the returned match.

BLASTP_MATCH Table Function

This table function performs a BLASTP search of the given amino acid sequence against the portion of the selected protein database. The database can be selected using a standard SQL select and passed into the function as a cursor. We also accept the standard BLAST parameters that are listed in the following section. The match returns the identifier of the matched (target) sequence (`t_seq_id`) (for example, the NCBI accession number), the score of the match, and the expect value.

Syntax

```
function BLASTP_MATCH (
  query_seq CLOB,
  seqdb_cursor REF CURSOR,
  subsequence_from NUMBER default 1,
  subsequence_to NUMBER default -1,
  filter_low_complexity BOOLEAN default false,
  mask_lower_case BOOLEAN default false,
  sub_matrix VARCHAR2 default 'BLOSUM62',
  expect_value NUMBER default 10,
  open_gap_cost NUMBER default 11,
  extend_gap_cost NUMBER default 1,
  word_size NUMBER default 3,
  x_dropoff NUMBER default 15,
  final_x_dropoff NUMBER default 25)
  return table of row (t_seq_id VARCHAR2, score NUMBER, expect NUMBER);
```

Parameters

Table 9–7 describes the input parameters for BLASTN_MATCH; Table 9–8, the output parameters.

Table 9–7 Input Parameters for BLASTP_MATCH Table Function

Parameter	Description
`query_seq`	The query sequence to search. This version of ODM BLAST accepts bare sequences only. A *bare sequence* is just lines of sequence data. Blank lines are not allowed in the middle of bare sequence input.
`seqdb_cursor`	The cursor parameter supplied by the user when calling the function. It should return two columns in its returning row, the sequence identifier and the sequence string.
`subsequence_from`	Start position of a region of the query sequence to be used for the search. The default is 1.
`subsequence_to`	End position of a region of the query sequence to be used for the search. If -1 is specified, the sequence length is taken as `subsequence_to`. The default is -1.
`filter_low_complexity`	`TRUE` or `FALSE`. If `TRUE`, the search masks off segments of the query sequence that have low compositional complexity. Filtering can eliminate statistically significant but biologically uninteresting regions, leaving the more biologically interesting regions of the query sequence available for specific matching against database sequences. Filtering is only applied to the query sequence. The default value is `FALSE`.

Table 9–7 (Cont.) Input Parameters for BLASTP_MATCH Table Function

Parameter	Description
mask_lower_case	TRUE or FALSE. If TRUE, you can specify a sequence in upper case characters as the query sequence and denote areas to be filtered out with lower case. This customizes what is filtered from the sequence. The default value is FALSE.
sub_matrix	Specifies the substitution matrix used to assign a score for aligning any possible pair of residues. The different options are PAM30, PAM70, BLOSUM80, BLOSUM62, and BLOSUM45. The default is BLOSUM62. See Table 9–9 for supported values of (open_gap_cost, extend_gap_cost) for each matrix.
expect_value	The statistical significance threshold for reporting matches against database sequences. The default value is 10. Specifying 0 invokes default behavior.
open_gap_cost	The cost of opening a gap. The default value is 11. Specifying 0 invokes default behavior.
extend_gap_cost	The cost of extending a gap. The default value is 1. Specifying 0 invokes default behavior.
word_size	The word size used for dividing the query sequence into subsequences during the search. The default value is 3. Specifying 0 invokes default behavior.
x_dropoff	Dropoff for BLAST extensions in bits. The default value is 15. Specifying 0 invokes default behavior.
final_x_dropoff	The final X dropoff value for gapped alignments in bits. The default value is 25. Specifying 0 invokes default behavior.

Table 9–8 Output Parameters for BLASTP_MATCH Table Function

Attribute	Description
t_seq_id	The sequence identifier of the returned match.
score	The score of the returned match.
expect	The expect value of the returned match.

For each substitution matrix (sub_matrix), only certain combinations of (open_gap_cost, extend_gap_cost) values are supported. Table 9–9 shows the supported combinations of values for each substitution matrix.

Table 9–9 Supported Combinations of (open_gap_cost, extend_gap cost)

Substitution Matrix Name	Supported (open_gap_cost, extend_gap_cost) Values
BLOSUM45	(13,3), (12,3), (11,3), (10,3), (16,2), (15,2), (14,2), (13,2), (12,2), (19,1), (18,1), (17,1), (16,1)
BLOSUM62	(11,2), (10,2), (9,2), (8,2), (7,2), (6,2), (13,1), (12,1), (11,1), (10,1), (9,1)
BLOSUM80	(25,2), (13,2), (9,2), (8,2), (7,2), (6,2), (11,1),(10,1), (9,1)
PAM30	(7,2), (6,2), (5,2), (10,1), (9,1), (8,1)
PAM70	(8,2), (7,2), (6,2), (11,1), (10,1), (9,1)

TBLAST_MATCH Table Function

This table function performs BLAST searches involving translations of either the query sequence or the database of sequences. The available options are:

- BLASTX: The query nucleotide sequence is translated and compared against a protein database.
- TBLASTN: The query amino acid sequence is compared against a translated nucleotide database.
- TBLASTX: The query nucleotide sequence is translated and compared against a translated nucleotide database.

The database can be selected using a standard SQL select and passed into the function as a cursor. We also accept the standard BLAST parameters that are listed in the following section. The match returns the identifier of the matched (target) sequence (`t_seq_id`) (for example, the NCBI accession number), the score of the match, and the expect value.

Syntax

```
function TBLAST_MATCH (
  query_seq CLOB,
  seqdb_cursor REF CURSOR,
  subsequence_from NUMBER default 1,
  subsequence_to NUMBER default -1,
  translation_type VARCHAR2 default 'BLASTX',
  genetic_code NUMBER default 1,
  filter_low_complexity BOOLEAN default false,
  mask_lower_case BOOLEAN default false,
  sub_matrix VARCHAR2 default 'BLOSUM62',
  expect_value NUMBER default 10,
  open_gap_cost NUMBER default 11,
  extend_gap_cost NUMBER default 1,
  word_size NUMBER default 3,
  x_dropoff NUMBER default 15,
  final_x_dropoff NUMBER default 25)
  return table of row (t_seq_id VARCHAR2, score NUMBER, expect NUMBER);
```

Parameters

Table 9–10 describes the input parameters for TBLAST_MATCH; Table 9–11, the output parameters.

Table 9–10 Input Parameters for TBLAST_MATCH Table Function

Parameter	Description
`query_seq`	The query sequence to search. This version of ODM BKLAST accepts bare sequences only. A *bare sequence* is just lines of sequence data. Blank lines are not allowed in the middle of bare sequence input.
`seqdb_cursor`	The cursor parameter supplied by the user when calling the function. It should return two columns in its returning row, the sequence identifier and the sequence string.
`subsequence_from`	Start position of a region of the query sequence to be used for the search. The default is 1.

Table 9–10 (Cont.) Input Parameters for TBLAST_MATCH Table Function

Parameter	Description
subsequence_to	End position of a region of the query sequence to be used for the search. If -1 is specified, the sequence length is taken as subsequence_to. The default is -1.
translation_type	Type of the translation involved. The options are BLASTX, TBLASTN, and TBLASTX. The default is BLASTX.
genetic_code	Used for translating nucleotide sequences to amino acid sequences. genetic_code is sort of like a mapping table. NCBI supports 17 different genetic codes. The supported genetic codes and their names are given in Table 9–1. The default genetic code is 1.
filter_low_ complexity	TRUE or FALSE. If TRUE, the search masks off segments of the query sequence that have low compositional complexity. Filtering can eliminate statistically significant but biologically uninteresting regions, leaving the more biologically interesting regions of the query sequence available for specific matching against database sequences. Filtering is only applied to the query sequence. The default is FALSE.
mask_lower_case	TRUE or FALSE. If TRUE, you can specify a sequence in upper case characters as the query sequence and denote areas to be filtered out with lower case. This customizes what is filtered from the sequence. The default is FALSE.
sub_matrix	Specifies the substitution matrix used to assign a score for aligning any possible pair of residues. The different options are PAM30, PAM70, BLOSUM80, BLOSUM62, and BLOSUM45. The default is BLOSUM62. See Table 9–9 for supported values of (open_gap_cost, extend_gap_cost) for each matrix.
expect_value	The statistical significance threshold for reporting matches against database sequences. The default value is 10. Specifying 0 invokes default behavior.
open_gap_cost	The cost of opening a gap. The default value is 11. Specifying 0 invokes default behavior.
extend_gap_cost	The cost of extending a gap. The default value is 1. Specifying 0 invokes default behavior.
word_size	The word size used for dividing the query sequence into subsequences during the search. The default value is 3. Specifying 0 invokes default behavior.
x_dropoff	Dropoff for BLAST extensions in bits. The default value is 15. Specifying 0 invokes default behavior.
final_x_dropoff	The final X dropoff value for gapped alignments in bits. The default value is 25. Specifying 0 invokes default behavior.

Table 9–11 Output Parameters for TBLAST_MATCH Table Function

Attribute	Description
t_seq_id	The sequence identifier of the returned match.
score	The score of the returned match.
expect	The expect value of the returned match.

BLASTN_ALIGN Table Function

This table function performs a BLASTN alignment of the given nucleotide sequence against the selected portion of the nucleotide database. The database can be selected using a standard SQL select and passed into the function as a cursor. It accepts the standard BLAST parameters that are listed in the following section.

BLASTN_MATCH returns only the score and expect value of the match. It does not return information about the alignment. BLASTN_MATCH is typically used when a BLAST search will be followed up with h a more compute intensive alignment, such as the Smith-Waterman alignment.

BLASTN_ALIGN does the BLAST alignment and returns the information about the alignment.

Syntax

```
function BLASTN_ALIGN (
  query_seq CLOB,
  seqdb_cursor REF CURSOR,
  subsequence_from NUMBER default 1,
  subsequence_to NUMBER default -1,
  filter_low_complexity BOOLEAN default false,
  mask_lower_case BOOLEAN default false,
  expect_value NUMBER default 10,
  open_gap_cost NUMBER default 5,
  extend_gap_cost NUMBER default 2,
  mismatch_cost NUMBER default -3,
  match_reward NUMBER default 1,
  word_size NUMBER default 11,
  xdropoff NUMBER default 30,
  final_x_dropoff NUMBER default 50)
  return table of row (
    t_seq_id VARCHAR2,
    pct_identity NUMBER,
    alignment_length NUMBER,
    mismatches NUMBER,
    positives NUMBER,
    gap_openings NUMBER,
    gap_list [Table of NUMBER],
    q_seq_start NUMBER,
    q_frame NUMBER,
    q_seq_end NUMBER,
    t_seq_start NUMBER,
    t_seq_end NUMBER,
    t_frame NUMBER,
    score NUMBER,
    expect NUMBER);
```

Parameters

Table 9–12 describes the input parameters for BLASTN_ALIGN; Table 9–13, the output parameters.

Table 9–12 Input Parameters for BLASTN_ALIGN Table Function

Parameter	Description
`query_seq`	The query sequence to search. This version of ODM BLAST accepts bare sequences only. A *bare sequence* is just lines of sequence data. Blank lines are not allowed in the middle of bare sequence input.
`seqdb_cursor`	The cursor parameter supplied by the user when calling the function. It should return two columns in its returning row, the sequence identifier and the sequence string.
`subsequence_from`	Start position of a region of the query sequence to be used for the search. The default is 1.
`subsequence_to`	End position of a region of the query sequence to be used for the search. If -1 is specified, the sequence length is taken as `subsequence_to`. The default is -1.
`filter_low_ complexity`	`TRUE` or `FALSE`. If `TRUE`, the search masks off segments of the query sequence that have low compositional complexity. Filtering can eliminate statistically significant but biologically uninteresting regions, leaving the more biologically interesting regions of the query sequence available for specific matching against database sequences. Filtering is only applied to the query sequence.
`mask_lower_case`	`TRUE` or `FALSE`. If `TRUE`, you can specify a sequence in upper case characters as the query sequence and denote areas to be filtered out with lower case. This customizes what is filtered from the sequence. The default is `FALSE`.
`expect_value`	The statistical significance threshold for reporting matches against database sequences. The default value is 10. Specifying 0 invokes default behavior.
`open_gap_cost`	The cost of opening a gap. The default value is 5. Specifying 0 invokes default behavior.
`extend_gap_cost`	The cost of extending a gap. The default value is 2. Specifying 0 invokes default behavior.
`mismatch_cost`	The penalty for nucleotide mismatch. The default value is -3. Specifying 0 invokes default behavior.
`match_reward`	The reward for a nucleotide match. The default value is 1. Specifying 0 invokes default behavior.
`word_size`	The word size used for dividing the query sequence into subsequences during the search. The default value is 11. Specifying 0 invokes default behavior.
`xdropoff`	Dropoff for BLAST extensions in bits. The default value is 30. Specifying 0 invokes default behavior.
`final_x_dropoff`	The final X dropoff value for gapped alignments in bits. The default value is 50. Specifying 0 invokes default behavior.

Table 9–13 Output Parameters for BLASTN_ALIGN Table Function

Parameter	Description
`t_seq_id`	Identifier (for example, the NCBI accession number) of the matched (target) sequence
`pct_identity`	Percentage of the query sequence that identically matches with the database sequence.
`alignment_length`	Length of the alignment.

Table 9–13 (Cont.) Output Parameters for BLASTN_ALIGN Table Function

Parameter	Description
`mismatches`	Number of base-pair mismatches between the query and the database sequence.
`positives`	Number of base-pairs with a positive match score between the query and the database sequence.
`gap_openings`	Number of gaps opened in gapped alignment.
`gap_list`	List of offsets where a gap is opened.
`q_seq_start, q_seq_end`	The indexes of the portion of the query sequence that is aligned
`q_frame`	Translation frame number of the query.
`t_seq_start, t_seq_end`	The indexes of the portion of the target sequence that is aligned.
`t_frame`	Translation frame number of the target sequence.
`expect`	Expect value of the alignment.
`score`	Score corresponding to the alignment.

BLASTP_ALIGN Table Function

This table function performs a BLASTP alignment of the given amino acid sequences against the selected portion of the protein database. The database can be selected using a standard SQL select and passed into the function as a cursor. You can also use the standard BLAST parameters that are listed in the following section.

BLASTP_MATCH function returns only the score and expect value of the match. It does not return information about the alignment. The BLASTP_MATCH is typically used when a BLAST search will be followed up with h a more compute intensive alignment, such as the Smith-Waterman alignment or a full FASTA alignment.

The BLASTP_ALIGN function does the BLAST alignment and returns the information about the alignment. The schema of the returned alignment is the same as that of BLASTN_ALIGN.

Syntax

```
function SYS_BLASTP_ALIGN (
  query_seq CLOB,
  seqdb_cursor REF CURSOR,
  subsequence_from NUMBER default 1,
  subsequence_to NUMBER default -1,
  filter_low_complexity BOOLEAN default false,
  mask_lower_case BOOLEAN default false,
  sub_matrix VARCHAR2 default 'BLOSUM62',
  expect_value NUMBER default 10,
  open_gap_cost NUMBER default 11,
  extend_gap_cost NUMBER default 1,
  word_size NUMBER default 3,
  x_dropoff NUMBER default 15,
  final_x_dropoff NUMBER default 25)
 return table of row (
    t_seq_id VARCHAR2,
    pct_identity NUMBER,
    alignment_length NUMBER,
    mismatches NUMBER,
    positives NUMBER,
    gap_openings NUMBER,
    gap_list [Table of NUMBER],
    q_seq_start NUMBER,
    q_frame NUMBER,
    q_seq_end NUMBER,
    t_seq_start NUMBER,
    t_seq_end NUMBER,
    t_frame NUMBER,
    score NUMBER,
    expect NUMBER);
```

Parameters

Table 9–14 describes the input parameters for BLASTP_ALIGN; Table 9–15, the output parameters.

Table 9–14 Input Parameters for BLASTP_ALIGN Table Function

Parameter	Description
query_seq	The query sequence to search. This version of ODM BKLAST accepts bare sequences only. A *bare sequence* is just lines of sequence data. Blank lines are not allowed in the middle of bare sequence input.
seqdb_cursor	The cursor parameter supplied by the user when calling the function. It should return two columns in its returning row, the sequence identifier and the sequence string.
subsequence_from	Start position of a region of the query sequence to be used for the search. The default is 1.
subsequence_to	End position of a region of the query sequence to be used for the search. If -1 is specified, the sequence length is taken as subsequence_to. The default is -1.
filter_low_ complexity	TRUE or FALSE. If TRUE, the search masks off segments of the query sequence that have low compositional complexity. Filtering can eliminate statistically significant but biologically uninteresting regions, leaving the more biologically interesting regions of the query sequence available for specific matching against database sequences. Filtering is only applied to the query sequence. The default is FALSE.
mask_lower_case	TRUE or FALSE. If TRUE, you can specify a sequence in upper case characters as the query sequence and denote areas to be filtered out with lower case. This customizes what is filtered from the sequence. The default is FALSE.
sub_matrix	Specifies the substitution matrix used to assign a score for aligning any possible pair of residues. The different options are PAM30, PAM70, BLOSUM80, BLOSUM62, and BLOSUM45. The default is BLOSUM62. See Table 9–9 for supported values of (open_gap_cost, extend_gap_cost) for each matrix.
expect_value	The statistical significance threshold for reporting matches against database sequences. The default value is 10. Specifying 0 invokes default behavior.
open_gap_cost	The cost of opening a gap. The default value is 11. Specifying 0 invokes default behavior.
extend_gap_cost	The cost of extending a gap. The default value is 1. Specifying 0 invokes default behavior.
word_size	The word size used for dividing the query sequence into subsequences during the search. The default value is 3. Specifying 0 invokes default behavior.
x_dropoff	X-dropoff for BLAST extensions in bits. The default value is 15. Specifying 0 invokes default behavior.
final_x_dropoff	The final X dropoff value for gapped alignments in bits. The default value is 25 Specifying 0 invokes default behavior.

Table 9–15 Output Parameters for BLASTP_ALIGN Table Function

Parameter	Description
t_seq_id	Identifier (for example, the NCBI accession number) of the matched (target) sequence
pct_identity	Percentage of the query sequence that identically matches with the database sequence.
alignment_length	Length of the alignment.

Table 9–15 (Cont.) Output Parameters for BLASTP_ALIGN Table Function

Parameter	Description
mismatches	Number of base-pair mismatches between the query and the database sequence.
positives	Number of base-pairs with a positive match score between the query and the database sequence.
gap_openings	Number of gaps opened in gapped alignment.
gap_list	List of offsets where a gap is opened.
q_seq_start, q_seq_end	The indexes of the portion of the query sequence that is aligned.
q_frame	Translation frame number of the query.
t_seq_start, t_seq_end	The indexes of the portion of the target sequence that is aligned.
t_frame	Translation frame number of the target sequence.
score	Score corresponding to the alignment.

TBLAST_ALIGN Table Function

This table function performs BLAST alignments involving translations of either the query sequence or the database of sequences or both the query sequence and the database of sequences. The available translation options are BLASTX, TBLASTN, and TBLASTX. The schema of the returned alignment is the same as that of BLASTN_ALIGN and BLASTP_ALIGN.

Syntax

```
function TBLAST_ALIGN (
  query_seq CLOB,
  seqdb_cursor REF CURSOR,
  subsequence_from NUMBER default 1,
  subsequence_to NUMBER default 0,
  translation_type VARCHAR2 default 'BLASTX',
  genetic_code NUMBER default 1,
  filter_low_complexity BOOLEAN default false,
  mask_lower_case BOOLEAN default false,
  sub_matrix VARCHAR2 default 'BLOSUM62',
  expect_value NUMBER default 10,
  open_gap_cost NUMBER default 11,
  extend_gap_cost NUMBER default 1,
  word_size NUMBER default 3,
  x_dropoff NUMBER default 15,
  final_x_dropoff NUMBER default 25)
 return table of row (
    t_seq_id VARCHAR2,
    pct_identity NUMBER,
    alignment_length NUMBER,
    mismatches NUMBER,
    positives NUMBER,
    gap_openings NUMBER,
    gap_list [Table of NUMBER],
    q_seq_start NUMBER,
    q_frame NUMBER,
    q_seq_end NUMBER,
    t_seq_start NUMBER,
    t_seq_end NUMBER,
    t_frame NUMBER,
    score NUMBER,
    expect NUMBER);
```

Parameters

Table 9–16 describes the input parameters for TBLAST_ALIGN; Table 9–17, the output parameters.

Table 9–16 Input Parameters for TBLAST_ALIGN Table Function

Parameter	Description
`query_seq`	The query sequence to search. This version of ODM BKLAST accepts bare sequences only. A *bare sequence* is just lines of sequence data. Blank lines are not allowed in the middle of bare sequence input.
`seqdb_cursor`	The cursor parameter supplied by the user when calling the function. It should return two columns in its returning row, the sequence identifier and the sequence string.

Table 9–16 (Cont.) Input Parameters for TBLAST_ALIGN Table Function

Parameter	Description
subsequence_from	Start position of a region of the query sequence to be used for the search. The default is 1.
subsequence_to	End position of a region of the query sequence to be used for the search. If -1 is specified, the sequence length is taken as subsequence_to. The default is -1.
translation_type	Type of the translation involved. The options are BLASTX, TBLASTN, and TBLASTX. The default is BLASTX.
genetic_code	Used for translating nucleotide sequences to amino acid sequences. genetic_code is sort of like a mapping table. NCBI supports 17 different genetic codes. The supported genetic codes and their names are given in Table 9–1. The default genetic code is 1.
filter_low_ complexity	TRUE or FALSE. If TRUE, the search masks off segments of the query sequence that have low compositional complexity. Filtering can eliminate statistically significant but biologically uninteresting regions, leaving the more biologically interesting regions of the query sequence available for specific matching against database sequences. Filtering is only applied to the query sequence.The default is FALSE.
mask_lower_case	TRUE or FALSE. If TRUE, you can specify a sequence in upper case characters as the query sequence and denote areas to be filtered out with lower case. This customizes what is filtered from the sequence.The default is FALSE.
sub_matrix	Specifies the substitution matrix used to assign a score for aligning any possible pair of residues. The different options are PAM30, PAM70, BLOSUM80, BLOSUM62, and BLOSUM45. The default is BLOSUM62. See Table 9–9 for supported values of (open_gap_cost, extend_gap_cost) for each matrix.
expect_value	The statistical significance threshold for reporting matches against database sequences. The default value is 10. Specifying 0 invokes default behavior.
open_gap_cost	The cost of opening a gap. The default value is 11. Specifying 0 invokes default behavior.
extend_gap_cost	The cost of extending a gap. The default value is 1. Specifying 0 invokes default behavior.
word_size	The word size used for dividing the query sequence into subsequences during the search. The default value is 3. Specifying 0 invokes default behavior.
x_dropoff	Dropoff for BLAST extensions in bits. The default value is 15. Specifying 0 invokes default behavior.
final_x_dropoff	The final X dropoff value for gapped alignments in bits. The default value is 25. Specifying 0 invokes default behavior.

Table 9–17 Output Parameters for TBLAST_ALIGN Table Function

Parameter	Description
t_seq_id	Identifier (for example, the NCBI accession number) of the matched (target) sequence
pct_identity	Percentage of the query sequence that identically matches with the database sequence.
alignment_length	Length of the alignment.

Table 9–17 (Cont.) Output Parameters for TBLAST_ALIGN Table Function

Parameter	Description
`mismatches`	Number of base-pair mismatches between the query and the database sequence.
`positives`	Number of base-pairs with a positive match score between the query and the database sequence.
`gap_openings`	Number of gaps opened in gapped alignment.
`gap_list`	List of offsets where a gap is opened.
`q_seq_start, q_seq_end`	The indexes of the portion of the query sequence that is aligned.
`q_frame`	Translation frame number of the query.
`t_seq_start, t_seq_end`	The indexes of the portion of the target sequence that is aligned.
`t_frame`	Translation frame number of the target sequence.
`score`	Score corresponding to the alignment .
`expect`	Expect value of the alignment.

Index

A

B

C

D

E

F

G

I

J

K

M

N

O

P

R

S

T

U