WDS-JniPMML-XLL Documentation

General ReadMe

The primary objective of WDS-JniPMML-XLL is to provide model evaluators to Excel. In particular, access to the standard PMML evaluator is a starting point, both for use or for comparison. Later versions will be include other model specs and implement other evaluators.

See documentation articles for a brief introduction.

Through this version, WDS-JniPMML-XLL provides:

- A pair of Excel AddIns (XLLs) and VBA support for:
  - Evaluating PMML models
    - As an Excel function call
    - Using the *de facto* standard implementations
    - Using input data from an in-worksheet table
      Uses XmlMap'd exportable ListObjects, but provides tools to facilitate
    - Can evaluate one or multiple observations (rows) per call
    - Results returned as normal function outputs
    - With cachable models for efficiency
  - Additional data wrangling tools for
    - Importing/Exporting HDF5 compound datasets
    - Importing/Exporting flat files
  - Additional VBA module handling
- A Java wrapper of jpmml.evaluator
  - Callable from the XLL via jni
  - Testable as a standalone from the command line
    But, can be called through the Excel AddIn using the JVM.
  - Input and output data can be:
    - HDF5 compound datasets
    - Flat files
    - In memory (as when called through jni)
- Examples are included
  - A test workbook and launch .bat to run the AddIns without installing
  - A test set of the usual PMML cases

Prerequisites

- 64 bit Excel
  Although, if compiling, 32 bit could possibly be added.
- Access to the VBA project object model (if using the VBA module handlers
- HDF5 and HDFView
  - The HDF5 and HDFView libs are required if compiling, but the functionality could be removed.
  - The provided jars require at least HDFView be on the path or the path passed in as a command line option when starting Excel
- Java jdk-12
  Required when using the latest HDFView install.
- Compiling environment
  The github configurations are for Visual Studio Community Edition and Intellij Community edition.
- DocFx

DocFx is use for the documentation build, including the DocFxDoclet for on the JavaDoc side.

License Note

TODOs, version 0.5.0

Outstanding items and items for the next version:

- Additional documentation and expanded test suite.
- Date and DateTime datatypes are not fully implemented. They are preliminarily setup to pass as doubles, but wrangling of string values, detection via cell information, and testing needs to be performed.
- The object cache and Handle/Tag handlers could be written better.
- Additional HDF5 utilities, such as returning a layout and querying an element when an HDF5 is used like a memory mapped file.
- Expanded VBA library.

An interesting thought or wish list:

- External memory mapped files, using the DBB wrangler.
- A parallel assembly for calling from MSSQLServer.

Related projects to be published on WDataSci's github site:

- WDataSci XML Model Specification, documentation and tools, including xsl implementation transformations.
- WDataSci Systems Model, documentation and tools.

MIT License

Articles

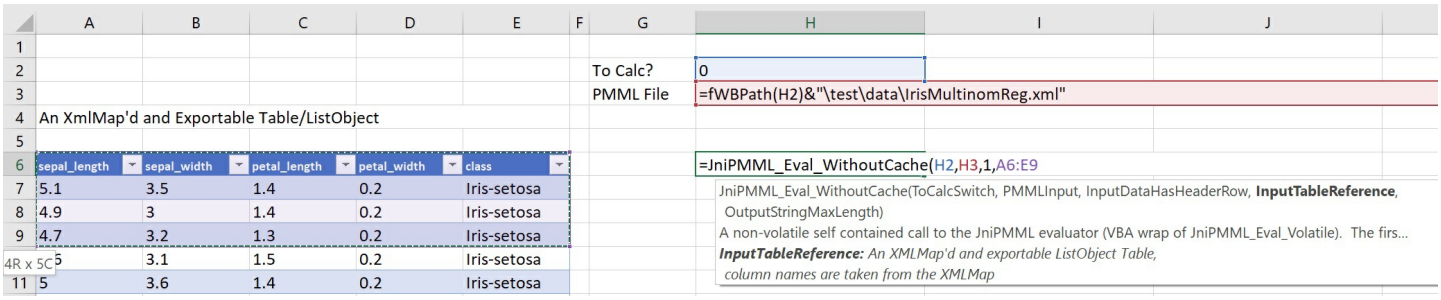
Brief Intro

The primary objective of WDS-JniPMML-XLL is to provide model evaluators to Excel. In particular, access to the standard PMML evaluators is a starting point, both for use or for comparison. Later versions will be include other model specs and implement other evaluators.

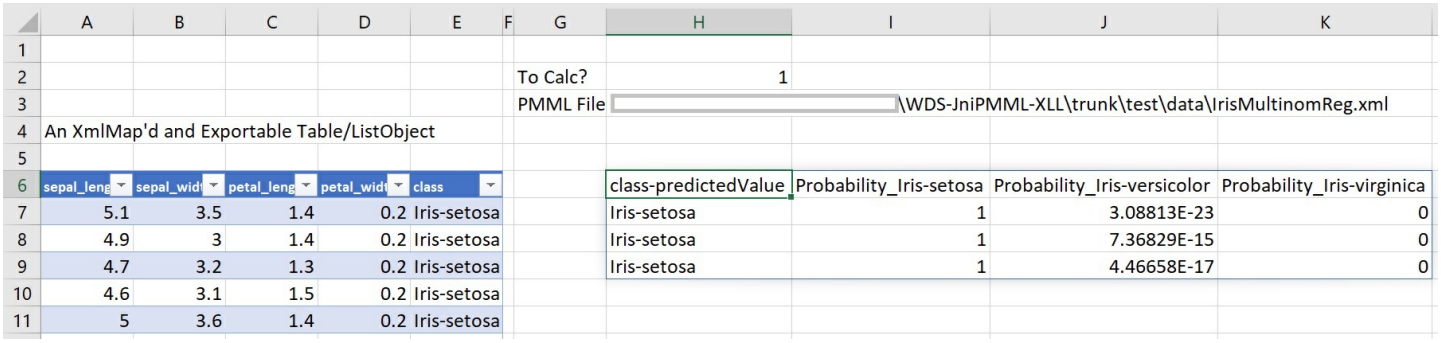
A quick easy way to evaluate PMML should be available to anyone, even those without access to the latest data science tools. In the finance industry, Excel is ubiquitous. To academic-type data scientists who might scoff at using Excel for anything, it is still a tool which can be used smartly or extremely poorly (which can certainly be said of Python, R, Java, C#, or anything else).

Simple Example

For one-time evaluations (less efficient, but simple), data is arranged in a table object. A macro is provided to assist in providing a technical requirement on the table. The JniPMML_Eval_WithoutCache function takes just a few arguments as below:



In older versions of Excel, results could be returned as an array-valued function, but in Excel 2016, the *spill* feature allows the function to return a dynamic number of rows and columns:



One thing that might not be obvious from the images above is these are function results. The inputs may even be randomized and (relatively) instantaneous evaluations returned.

Slightly More Complex

More efficient model evaluation involves caching the model and then repeated calls to the evaluator without having to do all of the parsing process of the model implementation spec for every calculation. There are several steps involved, but are simplified in the workbook:

- Pick a *Tag* for model
  If you are thinking this should be called a *Handle*, that would seem correct. Except, the Handles are provided and controlled by the Java side and Tags are used on the Excel/C# side.

  A Handle actually has two parts, *HandleMajor.HandleMinor*. The HandleMajor is unique to the Tag and the cached model on the Java side. The HandleMinor increments with subsequant configuraton changes, such as caching the input schema from the ListObject and the output schema. Why do this? It is an Excel-trick. When other ranges depend on the HandleMajor.HandleMinor value of some cell, and that cell recalculates, the correct cascade of recalculations occurs.

- Provide a PMML Model

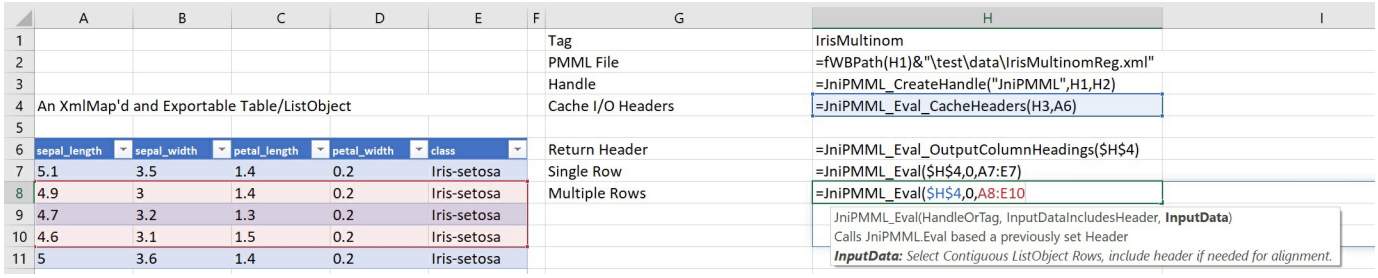
Here the model can be either a path to a PMML file or (more interestingly) a full PMML file as a string. Why take the string? One could build the file in the workbook. Perhaps one is testing some transformation structure or just wants to see what happens.

- Create a handle
  Cache the PMML model on the Java side and return the new handle to Excel.

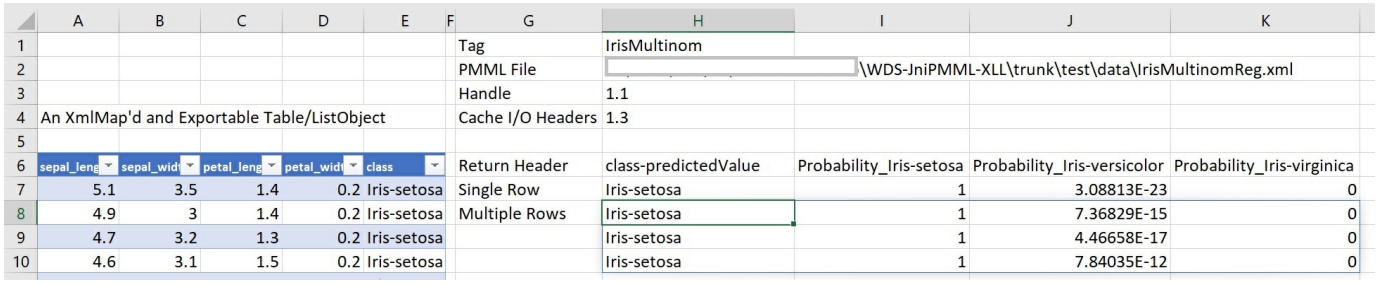
- Cache the input and output headers
  Based on the input XmlMap'd exportable table, update the HandleMinor. This also internally caches the model outputs which can be queried for column headings.
  Note: Some PMML models are harder to fully determine the output structures and results are returned as a dictionary-like structure. In the current version and for this case, there is a function, JniPMML_Expand_ComplexValue that can be used to return an expansion. See the WDS-JniPMML-XLL-Test.xlsm workbook for an example.

- Point to input and return the evaluation
  The input is an XmlMap'd exportable table. Before that raises any concerns, there is a macro available through the ribbon, "Add XmlMap to Selected ListObject", which will assign one to it through the following steps:
  - Select a cell in a table or the entire table
  - Hit the macro and you will be queried for one of the following
    - Point to an external XSD file
    - Point to an XSD as a string in a cell
    - Infer one from the table
    - Use a cached PMML dictionary, matching by column name, and infer where a column is not in the dictionary

  The slightly more complex example:



With result:



See the provided Excel test workbook for additional examples.

Additional Usage Notes

- Accessing Java via JNICode creates a COM AddIn
  Efforts have been made to make sure COM objects are clean up. However, should the process break for whatever reason, there may be an Excel process hanging around. In that case, look in the taskmgr's details. Or, use something like the powershell snippets in the scripts folder to find and stop.

Notes on JniPMML

Author: Christian Wypasek

Simple Motivation

My daughter, a college student, asked me to explain this project in one sentence and this was as close as I could get: Scientists build models. For even something as simple as linear regression, there is a formula that needs to evaluated. It might be for my own purposes, or it might be for a company I work for, but model implementation needs to be easily accessible. Even though data scientists might use special tools, everyone in financial services at least has Excel.

Slightly More Technical Motivation

Regardless of whether or not Excel might be highly regarded as a computational framework among academicians, it is ubiquitous in financial services (even if it might not be used well). Therefore, it makes sense that invoking an XML based evaluator from within Excel would be worthwhile. In particular, since Excel can enable rapid visualization, one should also be able to compare evaluator implementations and view model response to variable changes and/or model structure in a live manner.

XML/PMML

For someone like myself who works across the spectrum of big data projects (project management and business interface, data science, and data engineer) and works across multiple programming languages, consistency of treatments is a fundamental key to efficiency. After years of engineering databases, building complex statistical models for financial instruments, and incorporating these models into asset backed cash flow valuations, the greatest risks in this data science process are often operational. There is the most obvious question, "Is the data being used for forecasting sufficiently like the data the model was fit on?", but one also has to ask "Is the model being calculated correctly?".

From personal experience, hand coding something like a scoring model requires significant quality checks and carries the persistent risk that something was overlooked. It does not take too many hand coding events to make one believe there has got to be a better way, both for efficiency of process and the reduction of mistakes that come from mind numbing exercises. Starting back in 1998/1999, I started using markup styles to facilitate both the modeling process and facilitating the implementation for scoring and other types of regresson and non-parametric models. Since then, PMML (predictive modeling markup language) as become an industry standard.

The PMML standard has evolved and early versions were not sophisticated enough for my needs. For example, the Scorecard implementation was not added until the end of 2011, and transformations were not added until 2014. For all that it is, PMML is still a communication standard for model implementation and is often generated after a model has been fit. Continued diligence is required so the communicated model truly represents the intended relationship between the input data and the output results. A process oriented view of statistical model building starts with data preparation and can be exploited at every step of the process through to final implementation.

There may be more than one way to skin a cat, but very few which leave you with anything looks like a cat. My personal work has included using mathematical and statistical model specifications in XML with implementations in SAS, C++, Python, R, in-database (Vertica, MSSQL) UDTFs in C++/Java/R, and VBA (in Excel). After drilling into PMML implementation details, there is still much to be desired. An updated XML specification used by WDataSci for model fitting and alternate implementations will be released on the its github as a later project, but transformation (such as through XSLT) into PMML for delivery is reasonable given the industry standardization that PMML offers. Other model implementation specifications, such as pfa, will emerge, and Excel will remain a platform for either a model delivery or easy comparison.

WDS-JniPMML as a multi-language project

The JniPMML project combines several APIs, each for a specific purpose:

- Java
  The *de facto* implementation of PMML is jpmml.evaluator. JniPMML-Java wraps the implementation in a manner that creates a standalone jar that can also be called from C# via jni.
- C#
  Using the ExcelDna project to facilitate Excel functionality, the JniPMML-Cs assembly wraps the jni calls.
- VB
  Some odds and ends which I have traditionally done in VBA, but using ExcelDna .Net. In particular, wrangling of the VBA modules is done in VB.
- VBA
  Certain Excel functions created with ExcelDna through either C# or VB become *volatile* in that they recalculate at every calculation event (which can be a bad thing). However, good old fashioned VBA can do the same thing in a non-volatile manner.

Working in different languages for different aspects of a larger project is not unusual. For example, database work might be done in SQL, with processing either in database or written-out-processed-read-back-in, and final summaries might have an entirely different framework. When sub-projects have many parallel functions, the tendency of programmers to have a project on one side and then start from scratch on the other side, can lead to unexpected differences which the programmer then might struggle to balance. Complete one side, move to the other, discover some new or useful treatment, go back to the first side, restart loop. This project also started in that manner.

Passing data back and forth in-memory between Java and C# involves packing memory in a particular way, which also turned out to be the HDF5.PInvoke bulk writes a HDF5 compound dataset (such as R can export). Development of the project included consideration of in-memory HDF5s, which despite HDF5 docs, is not ready for prime-time. For testing purposes, HDF5 CompoundDS and flat file functionality is included in the JniPMML-Java project and the Excel AddIn.

Finally, the Excel AddIn also includes other tools representative of some extended functionality I have come to expect over the years, such as VBA component wrangling and other examples. Even if this project is not used extensively outside of WDataSci, this project also become an in-house reference for C#/Java differences and quirks, DocFx, Excel AddIns (quirks across C#, ExcelDna, VB, VBA, COM, non-COM), PMML (and jpmml quirks), HDF5 (and quirks across HDF.PInvoke, HDF-Java, HDF-Object), etc.

Notes on Java x Cs

Author: Christian Wypasek

The mirroring of the C# and Java code is meant not to be slick or cute. It is simply because both implementations are reading and writing the same formats. When handing off data in a ByteBuffer between C# and Java, in both directions, the formats must be *exactly* the same. (Note, not going down the AST route. It seems like if you are going to go that route, you should be all in.)

Some syntax differences are too big to bridge, such as in how enums are more flexible in Java than C#. With enums, just the values and methods (extensions in C#) are in common. The source codes will still be organized similarly, but this is also why enums are not otherwise in the files with their naturally associated classes.

Some syntax differences are not marked but obvious:

- Non-method properties, such as length/Length or boolean/Boolean, which are easy enough to fix in IDEs.
- To break String object references in Java where C# does not require it, a simple new_String() function in C# is a pass through and differs only with the "_".
- Method *throws* required in Java but not C# are on separate lines and commented out in C#.
- In switch-case statements on enum values where Java case statements do not require qualified names, there will be two lines one uncommented for Java, the other commented for C#, and visa-versa.

The syntax differences for many common methods amount only to the case of the leading letter, such as with Java's String.toString() vs C#'s String.ToString(). When this leading case issue is on a class method, they can be minimized through C#'s static extension methods, included in a static class, JavaLikeExtensions. Why not just let one letter differences ride, like in length/Length above? One line in one file and one less thing to highlight a difference in vimdiff. Other differences can be eliminated through specially named classes, mimicking names and methods used on the Java side, such as Map, PrintWriter, and ArrayList. Even though broken out in the documentation, on C#, they can all be included in the WDSXJava.cs, along with JavaLikeExtensions.

Syntax differences over lines or blocks are handled in two ways: First, when a one line change is required, a comment leading with //Java or //C# precedes the line. On the Java side, the //C# and subsequent line are collapsed, commenting out the C# syntax. The reverse treatment is used on the C# side.

For example, in C# version:

```
//C#
if ( !base.Equals(arg) ) return false;
//Java if ( !super.Equals(arg) ) return false;
```

And in the Java version:

```
//C# if ( !base.Equals(arg) ) return false;
//Java
if ( !super.Equals(arg) ) return false;
```

For larger blocks, we can exploit the behavior that an open-comment /* jumps over other open comments until the first closing */. Therefore, in the C# version (Note that the Java >>> comment is open):

```
/* C# >>> */
if ( !base.Equals(arg) ) return false;
/* <<< C# */
/* Java >>> *
if ( !super.Equals(arg) ) return false;
/* <<< Java */
```

And in the Java version (Note that the C# >>> comment is open):

```
/* C# >>> *
if ( !base.Equals(arg) ) return false;
/* <<< C# */
/* Java >>> */
if ( !super.Equals(arg) ) return false;
/* <<< Java */
```

There are multiple programming languages used in this project:

Note: This documentation bundle was created using DocFx, which was confusing documentation across APIs. Therefore, a separate PDF has been generated for each.

Java APIs

JniPMML-Java

The initial design of JniPMML-Java is to wrap jpmml into a single jar which can be called from Excel/C# via JNICode. However, it is a standalone that can be used with command line calls. It therefore has wranglers for text and HDF5 files in addition to the ByteBuffers for interacting with C#.

WDS-Java

General utilitities that independent of JniPMML code. To simplify assemblies and jars, this code is included in the larger projects, but is also compiled as a stand alone.

On the java side, there is a separate WDS-00.00.00.jar generated but it is pulled into a shaded jar so that only one WDS-JniPMML-00.00.00.jar needs to be used in practice.

C# APIs

JniPMML-Cs

The Java style com.WDataSci namespaces are specifically for C# code which mirrors the Java modules.

The JNI namespace originated externally, but with a few local completion and extensions.

WDS-Cs

General utilitities that independent of JniPMML code. To simplify assemblies and jars, this code is included in the larger projects, but is also compiled as a stand alone.

The Java style com.WDataSci namespaces are specifically for C# code which mirrors the Java modules.

The C# style namespaces are not specifically mirrored in the Java code.

VB APIs

JniPMML-VB

The JniPMML-VB code is primarily for some additional Excel manipulation functionality. In particular, the wrangling the Excel VBE components. The ExcelDna and Microsoft.Office.Interop.Excel libraries are generally mirrored in both C# and VB, however, ExcelDna UDF functions which take references as objects so that information about the caller can be determined at run-time become automatically volatile. For this reason, there are several function wrappers implemented in VBA which must be either in an another addin, or as a VBA module in the workbook. The JniPMML-VB (and supporting WDS-VB code which is pulled into the assembly) addin facilitates these wrapped functions by providing a wrangler for a WDSJniPMML.bas module.

WDS-VB

General utilitities that independent of JniPMML code. To simplify assemblies and jars, this code is included in the larger projects, but is also compiled as a stand alone.

VBA APIs

WDS-VBA

The WDS-VBA code is a collection of VBA macros that can be included in Excel workbooks and there are C#/VB macros accessibly through the ribbon to wrangle them in and out of workbooks as needed.

There is a necessity for at least the WDSCore macro for adding ExcelDna Intellisense capabilities.

JniPMML-VB

The JniPMML-VB code is primarily for some additional Excel manipulation functionality. In particular, the wrangling the Excel VBE components. The ExcelDna and Microsoft.Office.Interop.Excel libraries are generally mirrored in both C# and VB, however, ExcelDna UDF functions which take references as objects so that information about the caller can be determined at run-time become automatically volatile. For this reason, there are several function wrappers implemented in VBA which must be either in an another addin, or as a VBA module in the workbook. The JniPMML-VB (and supporting WDS-VB code which is pulled into the assembly) addin facilitates these wrapped functions by providing a wrangler for a WDSJniPMML.bas module.

Namespace com.WDataSci.JniPMML

Classes

AddInExcelDna

AddInExcelDna.WDSJniPMMLVBAddIn

JniPMMLUtil

JniPMMLVBAConstructor

Util

VBAUtil

WDSCoreVBAConstructor

Class AddInExcelDna

Inheritance

AddInExcelDna

Namespace: com.WDataSci.JniPMML

Assembly: JniPMML-VB.dll

Syntax

```
public class AddInExcelDna
```

Class AddInExcelDna.WDSJniPMMLVBAddIn

Inheritance

System.Object
AddInExcelDna.WDSJniPMMLVBAddIn

Implements

ExcelDna.Integration.IExcelAddIn

Inherited Members

System.Object.ToString()
System.Object.Equals(System.Object)
System.Object.Equals(System.Object, System.Object)
System.Object.ReferenceEquals(System.Object, System.Object)
System.Object.GetHashCode()
System.Object.GetType()
System.Object.Finalize()
System.Object.MemberwiseClone()

Namespace: com.WDataSci.JniPMML

Assembly: JniPMML-VB.dll

Syntax

```
public class WDSJniPMMLVBAddIn : IExcelAddIn
```

Explicit Interface Implementations

IExcelAddIn.AutoClose()

Declaration

```
public void IExcelAddIn.AutoClose()
```

IExcelAddIn.AutoOpen()

Declaration

```
public void IExcelAddIn.AutoOpen()
```

Implements

ExcelDna.Integration.IExcelAddIn

Class JniPMMLUtil

Inheritance

JniPMMLUtil

Namespace: com.WDataSci.JniPMML

Assembly: JniPMML-VB.dll

Syntax

```
public class JniPMMLUtil
```

Methods

JniPMML_ExpandComplexValue(Object)

Declaration

```
[ExcelFunction(Name = "JniPMML_ExpandComplexValue", Category = "WDS.JniPMML", Description = "Takes a complex
PMML evaluator target object and expands.", ExplicitRegistration = true, IsVolatile = false)]
public static object JniPMML_ExpandComplexValue([ExcelArgument(Name = "Value", Description = "A jpmml-like
complex value {k1=v1, k2=[sk1=sv1, sk2=sv2],...}")] object Value)
```

Parameters

TYPE	NAME	DESCRIPTION
System.Object	Value	

Returns

TYPE	DESCRIPTION
System.Object	

Class JniPMMLVBAConstructor

Inheritance

JniPMMLVBAConstructor

Namespace: com.WDataSci.JniPMML

Assembly: JniPMML-VB.dll

Syntax

```
public class JniPMMLVBAConstructor
```

Fields

WDSJniPMMLModuleName

Declaration

```
public const string WDSJniPMMLModuleName = "WDSJniPMML"
```

Field Value

TYPE	DESCRIPTION
System.String	

Methods

JniPMMLVBACheck()

Declaration

```
public static void JniPMMLVBACheck()
```

WDSVBAComponentAdd_WDSJniPMML()

Declaration

```
public static void WDSVBAComponentAdd_WDSJniPMML()
```

WDSVBAComponentRemove_WDSJniPMML()

Declaration

```
public static void WDSVBAComponentRemove_WDSJniPMML()
```

Class Util

Inheritance

Util

Namespace: com.WDataSci.JniPMML

Assembly: JniPMML-VB.dll

Syntax

```
public class Util
```

Methods

bIn(String, Object[])

Declaration

```
[ExcelFunction(Name = "bIn", Category = "WDS", Description = "Returns true if first argument value is any of
the optional arguments", ExplicitRegistration = true)]
public static bool bIn([ExcelArgument(Name = "PrimarySubject", Description = "Compares string value against
all other arguments", AllowReference = false)] string arg, [ExcelArgument(Name = "CompareValues", Description
= "Compares each against the first agument", AllowReference = false)] params object[] arglist)
```

Parameters

TYPE	NAME	DESCRIPTION
System.String	arg	
System.Object[]	arglist	

Returns

TYPE	DESCRIPTION
System.Boolean	

Concatenate_Range(Object[,], String)

Declaration

```
[ExcelFunction(Name = "Concatenate_Dlm", Category = "WDS", Description = "Concatenates the string values of a
range of cells into a delimited string, empty values are not included", ExplicitRegistration = true,
IsVolatile = false)]
public static string Concatenate_Range([ExcelArgument(Name = "Values", Description = "A contiguous range to
concatenate")] object[, ] Values, [ExcelArgument(Name = "dlm", Description = "A delimiter")] string dlm)
```

Parameters

TYPE	NAME	DESCRIPTION
System.Object[,]	Values	
System.String	dlm	

Returns

TYPE	DESCRIPTION
System.String	

Class VBAUtil

Inheritance

VBAUtil

Namespace: com.WDataSci.JniPMML

Assembly: JniPMML-VB.dll

Syntax

```
public class VBAUtil
```

Fields

WDSVBAModuleCheckSheetName

Declaration

```
public const string WDSVBAModuleCheckSheetName = "VBA_Modules_Check"
```

Field Value

TYPE	DESCRIPTION
System.String	

Methods

WDSRemoveVBACheckSheet()

Declaration

```
public static void WDSRemoveVBACheckSheet()
```

WDSVBADeleteSelected()

Declaration

```
public static void WDSVBADeleteSelected()
```

WDSVBAExportSelected()

Declaration

```
public static void WDSVBAExportSelected()
```

WDSVBAImportExport_Guts()

Declaration

```
[ExcelCommand(IsHidden = true, ExplicitRegistration = true)]
public static void WDSVBAImportExport_Guts()
```

WDSVBAImportSelected()

Declaration

```
public static void WDSVBAImportSelected()
```

WDSVBAModuleReview()

Declaration

```
public static void WDSVBAModuleReview()
```

WDSVBAModuleReviewRefresh()

Declaration

```
public static void WDSVBAModuleReviewRefresh()
```

Class WDSCoreVBAConstructor

Inheritance

WDSCoreVBAConstructor

Namespace: com.WDataSci.JniPMML

Assembly: JniPMML-VB.dll

Syntax

```
public class WDSCoreVBAConstructor
```

Fields

WDSCoreVBAModuleName

Declaration

```
public const string WDSCoreVBAModuleName = "WDSCore"
```

Field Value

TYPE	DESCRIPTION
System.String	

Methods

WDSCoreVBACheck()

Declaration

```
public static void WDSCoreVBACheck()
```

WDSVBAComponentAdd_WDSCore()

Declaration

```
public static void WDSVBAComponentAdd_WDSCore()
```

WDSVBAComponentRemove_WDSCore()

Declaration

```
public static void WDSVBAComponentRemove_WDSCore()
```

WDS-VB

General utilitities that independent of JniPMML code. To simplify assemblies and jars, this code is included in the larger projects, but is also compiled as a stand alone.

Namespace com.WDataSci.WDS

Classes

AddInExcelDna

Util

VBAUtil

WDSCoreVBAConstructor

Class AddInExcelDna

Inheritance

System.Object
AddInExcelDna

Implements

ExcelDna.Integration.IExcelAddIn

Inherited Members

System.Object.ToString()
System.Object.Equals(System.Object)
System.Object.Equals(System.Object, System.Object)
System.Object.ReferenceEquals(System.Object, System.Object)
System.Object.GetHashCode()
System.Object.GetType()
System.Object.Finalize()
System.Object.MemberwiseClone()

Namespace: com.WDataSci.WDS

Assembly: WDS-VB.dll

Syntax

```
public class AddInExcelDna : IExcelAddIn
```

Explicit Interface Implementations

IExcelAddIn.AutoClose()

Declaration

```
public void IExcelAddIn.AutoClose()
```

IExcelAddIn.AutoOpen()

Declaration

```
public void IExcelAddIn.AutoOpen()
```

Implements

ExcelDna.Integration.IExcelAddIn

Class Util

Inheritance

Util

Namespace: com.WDataSci.WDS

Assembly: WDS-VB.dll

Syntax

```
public class Util
```

Methods

bIn(String, Object[])

Declaration

```
[ExcelFunction(Name = "bIn", Category = "WDS", Description = "Returns true if first argument value is any of
the optional arguments", ExplicitRegistration = true)]
public static bool bIn([ExcelArgument(Name = "PrimarySubject", Description = "Compares string value against
all other arguments", AllowReference = false)] string arg, [ExcelArgument(Name = "CompareValues", Description
= "Compares each against the first agument", AllowReference = false)] params object[] arglist)
```

Parameters

TYPE	NAME	DESCRIPTION
System.String	arg	
System.Object[]	arglist	

Returns

TYPE	DESCRIPTION
System.Boolean	

Concatenate_Range(Object[,], String)

Declaration

```
[ExcelFunction(Name = "Concatenate_Dlm", Category = "WDS", Description = "Concatenates the string values of a
range of cells into a delimited string, empty values are not included", ExplicitRegistration = true,
IsVolatile = false)]
public static string Concatenate_Range([ExcelArgument(Name = "Values", Description = "A contiguous range to
concatenate")] object[, ] Values, [ExcelArgument(Name = "dlm", Description = "A delimiter")] string dlm)
```

Parameters

TYPE	NAME	DESCRIPTION
System.Object[,]	Values	
System.String	dlm	

Returns

TYPE	DESCRIPTION
System.String	

Class VBAUtil

Inheritance

VBAUtil

Namespace: com.WDataSci.WDS

Assembly: WDS-VB.dll

Syntax

```
public class VBAUtil
```

Fields

WDSVBAModuleCheckSheetName

Declaration

```
public const string WDSVBAModuleCheckSheetName = "VBA_Modules_Check"
```

Field Value

TYPE	DESCRIPTION
System.String	

Methods

WDSRemoveVBACheckSheet()

Declaration

```
public static void WDSRemoveVBACheckSheet()
```

WDSVBADeleteSelected()

Declaration

```
public static void WDSVBADeleteSelected()
```

WDSVBAExportSelected()

Declaration

```
public static void WDSVBAExportSelected()
```

WDSVBAImportExport_Guts()

Declaration

```
[ExcelCommand(IsHidden = true, ExplicitRegistration = true)]
public static void WDSVBAImportExport_Guts()
```

WDSVBAImportSelected()

Declaration

```
public static void WDSVBAImportSelected()
```

WDSVBAModuleReview()

Declaration

```
public static void WDSVBAModuleReview()
```

WDSVBAModuleReviewRefresh()

Declaration

```
public static void WDSVBAModuleReviewRefresh()
```

Class WDSCoreVBAConstructor

Inheritance

WDSCoreVBAConstructor

Namespace: com.WDataSci.WDS

Assembly: WDS-VB.dll

Syntax

```
public class WDSCoreVBAConstructor
```

Fields

WDSCoreVBAModuleName

Declaration

```
public const string WDSCoreVBAModuleName = "WDSCore"
```

Field Value

TYPE	DESCRIPTION
System.String	

Methods

WDSCoreVBACheck()

Declaration

```
public static void WDSCoreVBACheck()
```

WDSVBAComponentAdd_WDSCore()

Declaration

```
public static void WDSVBAComponentAdd_WDSCore()
```

WDSVBAComponentRemove_WDSCore()

Declaration

```
public static void WDSVBAComponentRemove_WDSCore()
```