Visual C++ .NET API for Contaminated Binormal Model Dynamic-Link Library (CBM DLL)
[version CBM190A.dll]

This document describes the API for calling the CBM DLL from a Visual C++ .NET program. The following table describes the argument list for calling the `CBM_SAS` subroutine that is exported by the DLL. The DLL was compiled with Lahey-Fujitsu Fortran 95 mixed-language options. These options correspond with the Microsoft Visual C++ .NET `extern "C"` DLL interface options (valid in Visual C++ for Visual Studio 2003) and the .NET `DllImport` statement with `CallingConvention=CallingConvention::Cdecl`. `CBM_SAS` is a subroutine, not a function, so it has a return type of `void` in C++. When calling the DLL subroutine `CBM_SAS` from C++ code, it is necessary to append an underscore to the name, so the subroutine is referred to as "`CBM_SAS_`" in C++ code.

Arg	Variable	Description	Fortran Type	C++ .NET Type
1	`noiseVector`	rating frequencies for normals (array)	`REAL*8(101)`	`double[101]`
2	`signalVector`	rating frequencies for signals (array)	`REAL*8(101)`	`double[101]`
3	`catNames`	integer labels for categories (array)	`INTEGER*4(101)`	`int[101]`
4	`numCat`	number of categories (scalar)	`INTEGER*4`	`int`
5	`numNoise`	sum of normal frequencies (scalar)	`REAL*8`	`double`
6	`numSignal`	sum of signal frequencies (scalar)	`REAL*8`	`double`
7	`anyInside`	any inside points (scalar)	`INTEGER*4`	`int`
8	`mu`	mu for cbm model (scalar)	`REAL*8`	`double`
9	`alpha`	alpha for cbm model (scalar)	`REAL*8`	`double`
10	`auc`	area under ROC curve (scalar)	`REAL*8`	`double`
11	`cutoffs`	cutoffs/cutpoints/thresholds (array)	`REAL*8(101)`	`double[101]`
12	`sens`	sensitivity (scalar)	`REAL*8`	`double`
13	`spec`	specificity (scalar)	`REAL*8`	`double`
14	`retcode`	return code from CBM DLL (scalar)	`INTEGER*4`	`int`
15	`debug`	turn on/off debugging (scalar)	`INTEGER*4`	`int`

Notes:

1. All arguments are <u>required</u>. There are no optional arguments.

2. You must initialize **<u>ALL</u>** variables passed as arguments to the DLL (even if you are only interested in returned values). For example, you must initialize `anyInside`, `mu`, `alpha`, `auc`, `cutoffs`, and `retcode` to some value such as 0 for `int` variables and 0.0 for `double` variables prior to the call. **See a separate note below for initializing sens and spec.** It is not enough to declare and allocate the variables—they must be initialized. If you fail to do this, the DLL may crash or exhibit unpredictable behavior.

3. The `sens` & `spec` variables are used to choose a sensitivity [p(TP)] at a given specificity [1-p(FP)] or specificity and a given sensitivity analysis. These variables also return the sensitivity or specificity. The following rules should be followed:

   If you want only an area analysis and do not care about sensitivity or specificity, then set both variables to -1:
   `sens` = -1.0
   `spec` = -1.0

If you want to compute sensitivity at a given specificity, then set `sens` to -1 and `spec` to the target specificity [1 – p(FP)]:
`sens` = -1.0
`spec` = target specificity value (range 0.0 < `spec` < 1.0)

If you want to compute sensitivity at a given specificity, then set `spec` to -1 and `sens` to the target sensitivity [p(TP)]
`sens` = target sensitivity value (range 0.0 < `sens` < 1.0)
`spec` = -1.0

Note that the above ranges specify "<", not "<="! Target values cannot equal 0.0 or 1.0.

If you set both sens and spec to values greater than -1, an error condition will result (`retcode` will be value greater than 0).

4. All arguments must be passed by reference.

5. If the DLL successfully computes values, the return code should be 0 (`retcode` = 0). Any value other than 0 indicates an error condition, so you should check `retcode` after each call to the DLL.

6. The debug variable is used to turn on logging for the DLL. Normally, you should set `debug` = 0. If you set `debug` = 1, the DLL will create a text log for each call. This log contains a listing of the input values to the DLL as well as any output values that were computed. This debug log will (usually) be created in the directory that contains the CBM DLL file.

7. You will need to add an import reference to `CBM190A.lib` in your linker options. The method for doing this varies by version of Visual Studio. In Visual Studio 2003, this was done by adding `CBM190.lib` to the list of additional dependencies under linker options. If you fail to do this, you will get an error message at compile and link time.

8. You will need to place `CBM190A.dll` in the same directory as the executable that calls it or in a directory that is specified in your system PATH variable. If you fail to do this, you will get an error message at run time, but not at compile or link time.

Last Edit:
Kevin M. Schartz, Ph.D., M.C.S.
April 23, 2008