Laboratory of Data Analysis
University of Jyväskylä

EUREDIT - WP4.5, WP5.5 Internal reports

A DLL-library for the TS-SOM Based Editing and Imputation Software.

Ismo Horppu - University of Jyväskylä
Pasi P. Koikkalainen - University of Jyväskylä

Contents

Summary

This document specifies a programming interface for a library version of the TS-SOM based editing and imputation system. The software is implemented with the Neural Data Analysis (NDA)[1] software platform, including a NDA library, example macros and a sample C-program.

The interface has been tested and used under several host programs, including Delph/Kylix, VisualBasic, Exel, Matlab, and X11/R4. This document explains how NDA routines can be used from the Microsoft Visual C programming language.

The document is organized as follows. Section 1 is a brief introduction to NDA. In section 2 the principles of data transfer between NDA and an external programs are described. Actual procedures for editing and imputation are explained in section 3, including definitions for data and algorithm specific parameters. System spefific files are described in the section 4, including installing instructions and NDA DLL interface on-line documentation. Finally an example of a Visual C project is described in the appendix 4.2.

1 Introduction to Neural Data Analysis DLL Library (NDA-DLL)

The NDA is a software that implements applications. Is consists of five layers:

A data layer, which provides memory management routines for the NDA data structures. It also implements several error prevention and grabage collection mechanisms.

Operation layer, which is a collecion of commands and algorithms. These are groupped under several modules that can be compiled and included into the NDA distribution separately.

Interface layer, which provides function entry points for NDA routines. It also allows one to access NDA data structures and to execute application level macros.

Application layer, which is a macro language intepreter for NDA commands. All the functionality of NDA applications is implemented this way.

End-user layer, which implements the user interface or host programs for the NDA application.

The NDA/DLL is a library that implements the first four layer of the NDA software. Thus it allows the execution of NDA macros and the manipulation of NDA data structures from external programs. Typical usage of the NDA/DLL starts with the uploading of data into NDA namespace, which is followed by the execution of NDA macros. Finally the results are loaded from the NDA to the host application. This is summarized in Fig. 1.

1. upload data (= input) to NDA's namespace,
2. run NDA macro command(s)
3. download data (= output) from NDA's namespace.

Figure 1: The steps of the usage of NDA/DLL from application programs.

The Windows compatible NDA/DLL provides a set of functions that operate with three types of data values: float (32-bit), signed integer (32-bit), and string (8-bit characters).

---

[1] ©NDA Software is under Copyright of the Laboratory of Data Analysis/Univeristy of Jyväskylä, Finland

2 Transfering data

The data uploading and loading routines for the NDA/DLL interface are introduced in this section.

2.1 Transfering data to the NDA namespace

In the NDA terminology `data` is a structure of one or more tt fields (variables). Since also `fields` are NDA structures the data transfer from an external program is done in two steps.

1. Before data can be transferred to the namespace, a data frame must be created using function `nda_create_new_dataframe`.
2. Next all variables must be inserted into the created data frame. The insertion can be done using functions
   `nda_insert_ifield`,
   `nda_insert_ffield`, or
   `nda_insert_sfield`.

The uploading functions correspond to 32-bit signed integer, 32-bit float and 8-bit string types, respectively. The functions make a copy of the given data, and thus the original data set can be deallocated after the call (if possible).

NOTES:

a) data frames and variables (= fields) names in the NDA's namespace may not contain special characters, like space or "/" etc, therefore it is best to use simple names.

b) old data frame named `"data"` can be deleted using
`run_nda_command("rm data");`

2.1.1 Uploading example: 32-bit float data transfer to the NDA namespace

Example 1 creates a new data frame named *data*, and inserts a 32-bit float variable into NDA's namespace.

```
int Status;                          /* function call errorcode */
float *myVar;                        /* pointer to data, which
                                        has 1000 entries)       */

/* allocate and fill myVar here */

Status = nda_create_new_frame("data"); /* create new data frame   */
if (Status != OPERATION_OK) return -1; /* failure? */

/* create variable "var1" into data frame "data" */
Status = nda_insert_ffield("data", "var1", 1000, myVar);
if (Status != OPERATION_OK) return -1; /* failure? */
```

2.2 Getting data from the namespace

The data transfer from the NDA namespace is done using function

`nda_get_field_ptr`.

The information is assingned to the position, provided by a given `(void **)` pointer. For integers and floats it expect `int*` and `float*` types, whereas for a string the data type is `char**`.

2.2.1 Downloading example from the NDA

The following retrieves pointer to 32-bit float variable which was created in the previous example

```
int Status;                          /* function call errorcode */
float *fvect;                        /* float data */
long records;                        /* amount of records */

/* retrieve first variable, whose index is 0, in data frame data */
Status = nda_get_field_ptr("data", 0, (void **)&fvect, &records);
if (Status != OPERATION_OK) return -1; /* failure? */

/* access data via fvect ... */
```

3 The execution of TS-SOM based editing and imputation routines

The NDA macros for editing and imputation include several options, which define the functionality of the procedures as well as the given output data sets.

For a given incomplete and erroneous data set one can do either:

a) Imputation, which gives as an output a completed data set

b) Editing (outlier detection), which gives are matrix of error probabilities.

c) Both editing and imputation, in which case bout a complete data and the probabilitys matrix is given as an output.

To make the use of the editing and imputation routines simple, two pre-programmed macros are provided with the NDA/DLL package:

1. `impute.cmd` is a NDA marco to impute missing values in a given data set.
2. `edit.cmd` is a NDA macro to edit (detect outliers) of a given data set. This routine can also do imputation of both the missing values as well as the found outliers.

The macros are executed using the `run_nda_command("macro options");` command.

Both macros expect similar type of input file with a predefined name (`data`) and they understand the same set of parameters, which are provided via sixteen (16) of commandline parameters (global for all variables) and a special parameter file (`sfr`) that specifies variable dependent options.

For example, the missingness is indicated via a special `missing value`, which is given when calling the macros (`edit.cmd` and `impute.cmd`). An example of this is included the attached Visual C program.

The NDA/DLL editing and imputation interface requires that the input data and variable parameters data are set before a call.

The input data, named as `data`, and the variable specific options, named as `sfr`, must be uploaded to the NDA namespace before calling editing or imputation routines.

After the operation is executed the new (clean) data will be named as `output` data frame in the NDA namespace. Similarly the error probabilities (if used) are named as a `errors` data frame. The output contains imputed and/or edited records only, and error probabilities data contain only probabilities for the erroneous records.

To summarize, the steps for the imputation, and editing operations through the NDA/DLL interface are:

Imputation :

1. create data `data` and attach variables to it,
2. build the variable spesific options data `sfr`,
3. call macro `impute.cmd` with commandline options, and
4. read result from a data frame `output`.

Editing :

1. create data `data` and attach variables to it,
2. build the variable spesific options data `sfr`,
3. call macro `edit.cmd` with commandline options, and
4. read results from a data frame `errors`.

Both editing and imputation steps :

1. create data `data` and attach variables to it,
2. build the variable spesific options data `sfr`,
3. call macro `edit.cmd` with commandline options, and
4. read results from data frames `output` and `errors`.

The implementation of various steps can be examined from the source code of included Visual C example project.

3.1 The sfr data: editing and imputing options for data variables

Before calling the editing and imputation macros, the special data frame `sfr` that includes variable specific optins must be set. The data consists of 4 item string variables which are:

1. row: (flag) the imputation action, which can be
   IMP_NONE (no imputation),
   IMP_MEAN (mean imputation),
   IMP_URAND (uniform distribution random imputation)
   IMP_NRAND (gaussian distribution random imputation).

2. row: (flag) the editing action, which can be
   EDIT_NONE (no edit),
   EDIT_CONTINUOUS (continuous variable edit)
   EDIT_CATEGORIAL (categorial variable edit).
3. row: (float) edit cut probability, which is between 0.0 and 1.0.
4. row: (float) the sigma1 parameter for robust training. In the case of continuous data it is the deviation related influence (factor) of the Huber estimator. Its default value is 3.0. In the case of categorial variables it is prior cut probability of the estimator, which must be between 0 and 1.

3.2 The commandline parameters

In addition to input data and variable specific options the macros require a set of commandline parameters, which are listed in detail in table 1. Parameters 13-15 are required only by the editing. Note also that an increase of parameter 14 decreases the error probabilities.

Parameter	Range	Description	default
0	>0	TS-SOM dimension	2
1	>0	TS-SOM layer	
2	0,1,2	TS-SOM topology: 0=lattice, 1=ring, 2=TS-VQ	0
3	integer	missing data value	
4	>0	lower limit for cluster record count when classifying incomplete records	5
5	>0.0	weighting of neighbours	0.5
6	epsilon	stopping criteria	0.001
7	>0	maximum number of iterations	20
8	>0	number of corrected layers	3
9	0,1	training rule: 0=VQ, 1=spread	0
10	0,1	use lookup table: 0=no,1=yes	1
11	0,1	use fullsearch: 0=no, 1=yes	0
12	0,1	use Huber estimator: 0=no,1=yes	
13	0,1	do outlier imputation: 0=no,1=yes	
14	>0.0	Sigma2, affects to continuous variables' error probabilities	1.0
15	>0.0	Sigma1, training robustness parameter	3.0

Table 1: The commandline parameters for the macros

4 NDA/DLL distribution CD.

This section is the contents of the NDA/DLL distribution. Table 2 is a list of files required to compile NDA DLL with other programs (or DLLs).

The DLL functions' return 0 (= OPERATION_OK, defined in errors.h) or -10000 (= IGNORE, defined in errors.h) on success. Any other return value is an error.

File	Description
nda.h	DLL functions' and helper functions declarations
ndadll.h	DLL function type declarations
ndadll.cpp	C interface for the DLL
errors.h	general errors
hdr.h	som errors
nda.dll	the NDA dll (binary)

Table 2: Required files

4.1 Helper functions in ndadll.cpp

A couple of helper functions have been defined in ndadll.cpp. Function `init_nda_dll` initializes the NDA/DLL interface, it returns 0 on success, and it must be called at first. Function `run_nda_command` can be called, as other NDA/DLL functions, after the NDA/DLL interface has been initialized. This function executes a NDA macro command (or a macro file). Function `free_nda_dll` releases NDA/DLL from memory, and it must be called when exiting from the program (but note that there are memory leaks in the current Beta level implemtation).

4.2 Visual C example project: ndaexample

The included example Visual C project, ndaexample, is a demonstration how NDA DLL can be used. The project is a 32-bit Windows console application. It includes main file `ndaexample.cpp`, files listed in table 2, and files `StdAfx.cpp` and `StdAfx.h`, which are both generated by Visual C. Two external

NDA macro files, `impute.cmd` and `edit.cmd`, are required. The files must be placed in the program's working directory.

The function `generate_data` shows how to create a new data frame into NDA namespace and how to attach variables to it. It also creates 5 missing data values and 5 obvious errors in the data. Function `display_output` retrieves changed (= edited or/and imputed) data from NDA's namespace. Edit error probabilities are read by function `display_error_probabilities`. Function `do_edit` does error detection, and it only creates error probabilities, whereas `do_imputation` does imputation and creates only imputed data. Edit and imputation are done by function `do_edit_and_imputation` which created edited data and error probabilities.

The actual editing is done by running NDA macro `edit.cmd`, whereas `impute.cmd` does imputing.

4.3 NDA DLL functions

Table 3 contains list of six necessary DLL functions for data creating, data reading, data writing, and executing NDA macros. The return value of DLL function is 0=`OPERATION_OK` or -10000=`IGNORE` on success, otherwise function call failed.

Function and description
`int nda_run_command(char *command, char *ret_space, long int len)` executes NDA macro `command` and returns output to `ret_space`.
`long nda_create_new_frame(char *framename)` creates a new data frame named `framename`.
`long nda_insert_ffield(char *frame,char *name, long len,float *vect)` inserts 32-bit float variable to frame `frame` and names the variable as `name`, variable's length is `len` and its data is pointed by pointer `vect`.
`long nda_insert_ifield(char *frame,char *name, long len,long *vect)` inserts 32-bit signed integer variable to frame `frame` and names the variable as `name`, variable's length is `len` and its data is pointed by pointer `vect`.
`long nda_insert_sfield(char *frame,char *name, long len,char **vect)` inserts 8-bit string variable to frame `frame` and names the variable as `name`, variable's length is `len` and its data is pointed by pointer `vect`.
`long nda_get_field_ptr(char *name, long index, void **ptr,long *nof)` gets data pointer (int*, float* or char**) for `index`:th variable in data frame `name` to pointer `ptr`. The length of data is stored to `nof`.

Table 3: Exported DLL functions