CMAT© Newsletter: December 2015

Wolfgang M. Hartmann

December 2015

Contents

1 General Remarks **2**
1.1 New Functions 2
1.2 Fixed Bugs 2

2 Modifications of Features **3**

3 Extensions to the Language **3**
3.1 Extending Data Types to Structs 3

4 Extensions to Various Functions **10**
4.1 Extensions of `rank()` Function 10
4.2 Extensions of `lp()` Function 12
4.3 Extensions of `svm()` Function 15
4.4 Extensions of `smsvm()` Function 16
4.5 Extensions of `boruta()` Function 17
4.6 Extensions of `split()` Function 19

5 New Developments **21**
5.1 The `rccount()` Function 21
5.2 The `sound()` Function 24

6 Illustrations **26**
6.1 Some Data Sets for `svm` 26
6.2 Comparing Computer Time for `lp()` Algorithms 27

7 The Bibliography **33**

1 General Remarks

Most of the time in 2015 I spent on testing some LP algorithms, especially interfaces to LPSolve, CLP by Forrest, de la Nuez, & Lougee-Helmer (2014), and the LPasL1 algorithm by Madsen, Nielsen, & Pinar (1996). Unfortunately, I was not able to solve a serious problem with the convergence of LPasL1.

Much more satisfying was my work on interfacing the SVM regression algorithm by Bi, Bennett, Embrechts, Breneman, & Song (2002) for variable selection with the CLP algorithm. In addition, a CLP interface was added as an alternative to the pcx algorithm for the multicategory SVM method in `smsvm()` by Lee, Lin, & Wahba (2003).

The last three months in 2015 I spent on including structs to the data types of the CMAT language. From my point of view that feature is completing the grammar of the CMAT language, and maybe aside from new functions, only some language components for KDTrees and some more preprocessor commands (like #define) must be added.

New (especially output) options were added to the `lrforw()`, `split()`, `scad()`, `ranfor()`, and `boruta()` functions. Most of the variable selection functions were tested with very large data sets.

Together with the work on data structs, the processing of data lists was tested and found that the code had some bugs especially where data lists where involved in user specified functions. Hopefully, this is running now.

1.1 New Functions

rccount() counts the number of entries that match a specified value. The syntax is similar to the `replace()` function.

sound() plays a beep of a specific frequency and duration.

1.2 Fixed Bugs

2 Modifications of Features

3 Extensions to the Language

3.1 Extending Data Types to Structs

With that extension CMAT can now deal with the following data types:

- scalars of different data types like
  - (long) integer
  - (double) real
  - (double) complex
  - string (of integer length)
- vectors and matrices with entries of all data types
- tensors (more than 2-dimensional arrays)
- lists which are one-dimensional arrays, where each entry can be scalar, vector, tensor, list, or struct
- structs where each entry can be scalar, vector, tensor, list, or struct
- KDTrees (used only internally)

The difference between lists and structs is that the entries of a list are accessed by an index, where the entries of the struct are accessed by compound names like `struct_name.entry_name`
You may use the `struct` keyword to specify a data struct. Here we assign a scalar `b` as an entry of struct `a`:

```
struct a;
a.b = 5;
print "[1] Struct a=", a;
```

```
[1] Struct a=

        *************************
        Structure a with 1 Entries
        *************************

     Struct a Entry[1]=a.b:       5
```

This is how we access the entry `b` of struct `a`:

```
b = a.b;
print "Entry b=",b;
a.b *= 2.;
print "Struct Entry 2 * a.b=",a.b;
```

```
Entry b= 5
Struct Entry 2 * a.b= 10.000
```

Make matrix `c` to another entry of struct `a`:

```
c = [ 2. 1.,
      1. 2.];
a.c = c;
print "[2] Struct a=", a;
```

```
[2] Struct a=
      [1] Struct a with 2 Entries
      **************************

         Struct a Entry[1]=a.c:
         **********************

        Dense Symmetric Matrix a.c

         S |           1          2
         --------------------------
         1 |   2.0000000
         2 |   1.0000000  2.0000000

      Struct a Entry[2]=a.b:  10.0000000
```

After assigning struct `d` to an entry, the struct `a` now has three entries:

```
struct d;
d.ent = [ 1 2 3 ];
a.d = d;
print "[3] Struct a=", a;
lstmem(1);
lststk(1);
```

```
[3] Struct a=
      [2] Struct a with 3 Entries
      **************************

     [1] Struct a.d with 1 Entries
     *****************************

      Struct a.d Entry[1]=a.d.ent:
      ****************************

         Dense Row Vector d.ent

        R |       1      2      3
                  1      2      3

         Struct a Entry[2]=a.c:
         **********************

        Dense Symmetric Matrix a.c

         S |           1          2
         --------------------------
         1 |   2.0000000
         2 |   1.0000000  2.0000000

      Struct a Entry[3]=a.b:  10.0000000
```

Here we assign a vector `a.d.f` as a second entry of substruct `a.d`:

```
a.d.f = [ 3 2 1 ];
print "[4] Struct a=", a;
```

```
[4] Struct a=
      [3] Struct a with 3 Entries
      ***************************

     [1] Struct a.d with 2 Entries
     ****************************

       Struct a.d Entry[1]=a.d.f:
       **************************

         Dense Row Vector a.d.f

        R |       1      2      3
                  3      2      1


     Struct a.d Entry[2]=a.d.ent:
     ****************************

        Dense Row Vector d.ent

        R |       1      2      3
                  1      2      3


        Struct a Entry[2]=a.c:
        **********************

      Dense Symmetric Matrix a.c

       S |           1          2
       --------------------------
       1 |   2.0000000
       2 |   1.0000000  2.0000000


   Struct a Entry[3]=a.b:  10.0000000
```

```
print "[5] Struct a=", a;
```

```
[5] Struct a=
      [3] Struct a with 4 Entries
      **************************

     [1] Struct a.d with 2 Entries
     ****************************

       Struct a.d Entry[1]=a.d.f:
       *************************

         Dense Row Vector a.d.f

        R |       1      2      3
                  3      2      1


      Struct a.d Entry[2]=a.d.ent:
      ****************************

         Dense Row Vector d.ent

        R |       1      2      3
                  1      2      3


        Struct a Entry[2]=a.cb:
        **********************

      Dense Symmetric Matrix a.cb

        S |           1          2
        --------------------------
        1 |   20.000000
        2 |   10.000000  20.000000


         Struct a Entry[3]=a.c:
         **********************

       Dense Symmetric Matrix a.c

        S |           1          2
        --------------------------
        1 |   2.0000000
        2 |   1.0000000  2.0000000


    Struct a Entry[4]=a.b:  10.0000000
```

The following show operations with struct entries:

```
cb =  a.c * a.b;
print "cb", cb;
a.cb = a.c * a.b;
print "Entry a.cb=", a.cb;
```

```
cb
 S |         1         2
-------------------------
 1 |    20.000
 2 |   10.0000    20.000

Entry a.cb=
 S |         1         2
-------------------------
 1 |    20.000
 2 |   10.0000    20.000
```

```
free a.d;
print "a after freeing a.d=", a;
```

```
a after freeing a.d=
      [4] Struct a with 3 Entries
      ***************************


        Struct a Entry[1]=a.cb:
        ***********************

      Dense Symmetric Matrix a.cb

        S |           1          2
        --------------------------
        1 |   20.000000
        2 |   10.000000  20.000000



         Struct a Entry[2]=a.c:
         **********************

       Dense Symmetric Matrix a.c

        S |           1          2
        --------------------------
        1 |   2.0000000
        2 |   1.0000000  2.0000000


    Struct a Entry[3]=a.b:  10.0000000
```

Lists and structs can be members of lists and structs:

```
struct str1;
list lst1;

str1.a = -15; str1.b = 27;
lst1[1] = str1.a; lst1[2] = str1.b;
lst1[1] += lst1[1]; lst1[2] += lst1[2];
print "List1=", lst1;
```

```
List1=

       *************************
       lst1 (List with 2 Entries)
       *************************

            lst1[1]:     -30
            lst1[2]:      54
```

```
struct str1;
list lst1;

lst1[1] = [ 1 2 3 ]; lst1[2] = [ 3 2 1 ];
str1.a = lst1[1]; str1.b = lst1[2];
print "Struct1=", str1;
```

```
Struct1=

       *************************
       Struct str1 with 2 Entries
       *************************

      Struct str1 Entry[1]=str1.b:
      ***************************

        Dense Row Vector str1.b

        R |       1      2      3
                  3      2      1


      Struct str1 Entry[2]=str1.a:
      ***************************

        Dense Row Vector str1.a

        R |       1      2      3
                  1      2      3
```

Struct names can be arguments of user specified functions:

```
print " User Function with struct argument";This should be 15:  15
 function tff2(str1) {
    a = str1.a; b = str1.b;
    k = (a < b) ? -a : -b;
    a = a + b;
    return(k);
 }

 struct str1;
 str1.a = -15; str1.b = 27;

 c = tff2(str1);
 print " This should be 15: ",c;
```

Struct names can be returned by user specified functions:

```
print "User Function returns with Struct";  This should be 15:  15
 function tff4(a,b) {
    struct str;
    k = (a < b) ? -a : -b;
    a = a + b;
    str.k = k; str.a = a; str.b = b;
    return(str);
 }

 str = tff4(-15,27);
 lstvar(1);
 lstmem(1);
 print "Struct result=", str;
 print " This should be 15: ",str.k;
```

4 Extensions to Various Functions

4.1 Extensions of `rank()` Function

Until recently, the rank of a matrix **A** was computed using the singular value decomposition of that matrix. This was a time consuming task, especially for very large and maybe very sparse matrices. For matrices, which have one very large and another very small dimension, i.e. for $\max(nrow, ncol) \gg \min(nrow, ncol)$, a QR decomposition designed especially for large sparse matrices was added as a method for estimating the rank of a matrix. The new method needs only memory of size $\min(nrow, ncol) \times \min(nrow, ncol)$.

```
srand(12345);
print "Generate nr by nc sparse matrix with nullity=nc/2";

nr = [ 10000 50000 100000 200000 ];
nc = 10; nc2 = nc / 2;
rank = cons(4,4);


for (ir = 1; ir <= 4; ir++) {
  mr = nr[ir];
  rmat = cons(mr,nc,0.); col = cons(mr,1);
  for (ic = 1; ic <= nc2; ic++) {
    ind = mr * rand(1000,1); ind = ceil(ind);
    /* attrib(ind); */
    val = 1000. * (rand(1000,1) - .5);
    col[ind] = val;
    rmat[,ic] = col;
    rmat[,ic+5] = 2. * col;
    if (ic > 1) rmat[,ic-1] += .5 * col;
  }

  t1 = time("clock");
  r1 = rank(rmat,"svd");
  t2 = time("clock");
  print " Time with SVD=", t2 - t1, " Rank=", r1;
  rank[ir,1] = r1;  rank[ir,2] = t2-t1;

  r2 = rank(rmat,"qrd");
  t3 = time("clock");
  print " Time with QRD=", t3 - t2, " Rank=", r2;
  rank[ir,3] = r2;  rank[ir,4] = t3-t2;
}


rnam = [" N10000 N50000 N100000 N200000 "];
```

```
cnam = [" RankSVD TimeSVD RankQRD TimeQRD "];
rank = rname(rank,rnam); rank = cname(rank,cnam);
print rank;
```

The following table shows the computer time for the Debug compiled version of CMAT (times would be much smaller for the Release compiled version):

```
Time with SVD= 0.01000 Rank= 5
Time with QRD= 0.01200 Rank= 5
Time with SVD= 0.04500 Rank= 5
Time with QRD= 0.02800 Rank= 5
Time with SVD= 0.08600 Rank= 5
Time with QRD= 0.04500 Rank= 5
Time with SVD= 0.2540  Rank= 5
Time with QRD= 0.08200 Rank= 5
```

	RankSVD	TimeSVD	RankQRD	TimeQRD
N10000	5.0000	0.00800	5.0000	0.01200
N50000	5.0000	0.04400	5.0000	0.02800
N100000	5.0000	0.08900	5.0000	0.04900
N200000	5.0000	0.25400	5.0000	0.08600

See the updated reference manual for a modified document of the `rank()` function.

4.2 Extensions of `lp()` Function

There are now two specifications of the `lp()` function available to the user:

< xr,lm,rp,duals,sens > = lp("meth",c,lau<,lubc<,optn<,xint>>>)

< xr,lm,rp,duals,sens > = lp("meth","path"<,optn<,xint>>)

1. The older type specifies the LP model using the matrix specification `(c,lau,lubc)` of the n vector `c` of coefficients of the cost function, the $m \times n + 2$ matrix `lau` of linear constraints including lower and upper ranges, and the $n \times 2$ matrix `lubc` of lower and upper bounds.
2. The newly added type specifies the LP model with a path (string) pointing to a valid .mps file.

The first input argument `"meth"` which is a string argument specifying the optimization method. There are now the following four methods available:

- `"pcx"`: The sparse interior point method by Stephen Wright (1997) which is also implemented with the `pcx` function (see Czyzyk, Mehrotra, Wagner, and Wright, 1997). Integer constraints (`xind`) cannot be specified. Three new new options were added
  - `"nolm"` do not compute Lagrange multipliers,
  - `"pmps"` for printing the MPS file,
  - `"prlp"` for printing the LP.
- `"clp"`: Provides an interface to the `Clp` code written by J. J. Forrest for the *COIN* and *Clp* packages. Both, the primal and dual simplex methods are very fast and very reliable methods. The packages also include code for the specification of integer constraints (Cbc) and the analysis of the influence of the optimal estimates on the objective function. For current runtime options see the table below.
- `"lps"`: The package `lpsolve` (Berkelaar et al., 2004), version 5.5, is being used. This is the only method which always needs dense storage of **A**. Integer constraints can be specified with `xind` as well as a sensitivity analysis w.r.t. the influence of the estimates. Two new new options were added
  - `"nolm"` do not compute Lagrange multipliers,
  - `"pmps"` for printing the MPS file.
- `"con"`: The continuation method by Madsen, Nielsen, & Pinar (1996) also called *LPasL1* is used. There are a Fortran and a C version available. The Fortran version always requires a dense matrix **A** of linear constraints, the C version decides either for dense or sparse storage depending on the number of nonzeros. Integer constraints (`xind`) cannot be specified. Two new new options were added

– "nolm" do not compute Lagrange multipliers,

– "pmps" for printing the MPS file.

This method has a flaw with many examples.

For the new method "clp" the options argument optn is recognizing the following settings:

Option Name	Column 2	Meaning
"lce"		required precision of constraint satisfaction
"max"		perform maximization (minimization is default)
"maxit"		maximum iterations
"nolm"		do not compute Lagrange multipliers
"nopr"		no printed output
"pmps"		print content of MPS file passed to Clp (not relevant for matrix specification)
"print"	int	amount of printed output
"vers"	char	specifies the optimization method used by Clp
	"sprim"	primal simplex (is the default for Clp)
	"sdual"	dual simplex method
	"barr"	Barrier method

The new "nolm" option has also been added to all other LP functions. The new "pmps" option has been added to all other LP methods but is effective only when an MPS input file is provided. For the old "print" option, the following output is consecutively added:

Int Value	Output Added
0	no printed output
1	very short result summary
2	iteration history (if available)
3	table of optimal estimates
4	table of constraints
> 4	maybe some debug output

Some specific preprocessing was added to each of the methods:

- The code for the "pcx" method included already a number of preprocessing steps. These steps were extended by the following two simple additional steps, which are also done with the other three LP methods: steps are not only:

  – linear constraints with only one nonzero coeffizient are reformulated into the bounds,

  – variables with equality boundary constraints ("masks") are removed from the optimization and replaced by postprocessing.

- The preprocessing code for "lps" now also provides "shifting" (of lower bounds to zero) and flipping the sign when there are lower bounds at $-\inf$. That means, LPsolve can now be applied to many more applications.

- Since `"clp"` has its own preprocessing methods, only a few simple preprozessing steps are performed before the model is feeded into the CLP code.
- The most extensive preprocessing was applied to `"con"`, even reformulating the model into the standard LP model,

$$\min c^T x \quad \text{s.}t. \quad \mathbf{A}x = b \quad and \quad 0 \leq x < u$$

  by including slack variables for all linear inequality constraints did not fix some serious problems with large applications, obviously due to problems with recognizing the correct rank of a set of general linear constraints. Until fixing this problem the algorithm cannot be recommended.

See also the `mpsread()` function for a number of additional preprocessing steps reducing the number of linear constraints. Especially the method of removing linearly dependent linear equality constraints is only in `mpsread()` but not in `lp()` available.

4.3 Extensions of `svm()` Function

For L1 sparse feature selection SVM for SVM ν regression (see Bi et.al, 2002), an LP algorithm is needed which for every grid point of (C, ν) solves an LP. Now, there are three algorithms implemented:

Method Name	**Algorithm**
"l1fs"	using PCx algorithm by Czyzyk et al. (1997)
"l1fs2"	using LP algorithm by Madsen, Nielsen, Pinar
"l1fs3"	using CLP algorithm by J. J. Forrest (2014)

For different values of a specified grid (C, ν) the constraints (bounds and general linear constraints) stay the same and only the values of the cost function changes. For some examples see `tsvm22.inp`, `tsvm23.inp`, and `tsvm24.inp` located in the `cmat/test` directory.

Example	**(m,n)**	**Grid**	**Algorithm**	**Time in sec.**
tsvm22.inp: NIR	(21,268)	4	PCX	0
tsvm22.inp: NIR	(21,268)	4	LPasL1	0
tsvm22.inp: NIR	(21,268)	4	CLP	0
tsvm22.inp: NIR	(21,268)	24	PCX	3
tsvm22.inp: NIR	(21,268)	24	LPasL1	0
tsvm22.inp: NIR	(21,268)	24	CLP	2
tsvm23.inp: Caco2	(27,714)	33	PCX	53
tsvm23.inp: Caco2	(27,714)	33	LPasL1	29095
tsvm23.inp: Caco2	(27,714)	33	CLP	5
tsvm23.inp: Housing	(506,14)	16	PCX	498
tsvm23.inp: Housing	(506,14)	16	LPasL1	big
tsvm23.inp: Housing	(506,14)	16	CLP	39

For large problems the LPasL1 algorithm does not perform well, and the simplex algorithm in CLP can be faster than the interior point algorithm PCX.
At the time, I'm still working on splitting off the SVM feature selection algorithms into two new functions `svmfsm()`:

- default Classification: L1 or L2 feature selection via Mangasarian's methods
- default Regression: using Bi and Bennett (2002) FLP algorithm

and `svmstw()`:

- stepwise classification: greedy feature selection method by Guyon et.al.; using methods like: NSVM, FQP, SMO, ASVM, PROX
- stepwise regression: greedy feature selection method by Guyon et.al.; using methods like: LSQ, FQP, SMO

See the next newsletter for a modified document of the `svm()`, `svmfsm()`, and `svmstw()` functions.

4.4 Extensions of `smsvm()` Function

Similar to the `svm()` function, a CLP interface was added to the `smsvm()` function for a multinomial response variable by Lee, Lin, & Wahba (2003). This method needs not only a fast sparse QP algorithm but in addition also some fast LP algorithm. Until now this function was working for the LP problem only with pcx, but with some range space and null space algorithm for QP.
Using the `meth` option one can now choose between two LP algorithms:

Method Name	Algorithm
"pcx"	using PCx algorithm by Czyzyk et al. (1997)
"clp"	using CLP algorithm by J. J. Forrest (2014)

Separate svm's are are computed for different values of a specified grid (λ, ν) using k-fold cross validation or test validation for selecting a best model. For some examples see `tsmsvm.inp`, `tsmsvm2.inp`, and `tsmsvm3.inp` located in the `cmat/test` directory.

Example	(m,n,lev)	Grid	Algorithm	Time in sec.
tsmsvm.inp: Heart	(210,14,3)	19	MSVM, SPLIT1, PCX, Range	0
tsmsvm.inp: Heart	(210,14,3)	19	MSVM, SPLIT1, CLP, Range	2
tsmsvm.inp: Heart	(210,14,3)	19	MSVM, SPLIT1, PCX, Null	1
tsmsvm.inp: Heart	(210,14,3)	19	MSVM, SPLIT1, CLP, Null	2
tsmsvm3.inp: Heart	(210,14,3)	19+11	SSVM, SPLIN1, PCX, Range	10
tsmsvm3.inp: Heart	(210,14,3)	19+11	SSVM, SPLIN1, CLP, Range	10
tsmsvm3.inp: NIR (7)	(21,268,2)	12	MSVM, SPLIN1, PCX, Null	1
tsmsvm3.inp: NIR (7)	(21,268,2)	12	MSVM, SPLIN1, CLP, Null	1
tsmsvm3.inp: NIR (7)	(21,268,2)	12+11	SSVM, SPLIN1, PCX, Range	14
tsmsvm3.inp: NIR (7)	(21,268,2)	12+11	SSVM, SPLIN1, CLP, Range	64
tsmsvm3.inp: Caco2	(27,714,2)	11	MSVM, SPLIT1, PCX, Range	1
tsmsvm3.inp: Caco2	(27,714,2)	11	MSVM, SPLIT1, CLP, Range	4
tsmsvm3.inp: Caco2	(27,714,2)	11+11	CSVM, SPLIT1, PCX, Range	35
tsmsvm3.inp: Caco2	(27,714,2)	11+11	CSVM, SPLIT1, CLP, Range	178
tsmsvm4.inp: LungCancer	(27,56,3)	9	MSVM, SPLIT1, PCX, Range	0
tsmsvm4.inp: LungCancer	(27,56,3)	9	MSVM, SPLIT1, CLP, Range	1
tsmsvm4.inp: LungCancer	(27,56,3)	9+11	CSVM, SPLIT1, PCX, Range	15
tsmsvm4.inp: LungCancer	(27,56,3)	9+11	CSVM, SPLIT1, CLP, Range	55
tsmsvm4.inp: Iris	(150,4,3)	9	MSVM, SPLIT1, PCX, Range	4
tsmsvm4.inp: Iris	(150,4,3)	9	MSVM, SPLIT1, CLP, Range	13
tsmsvm4.inp: Iris	(150,4,3)	9+11	CSVM, SPLIT1, PCX, Range	57
tsmsvm4.inp: Iris	(150,4,3)	9+11	CSVM, SPLIT1, CLP, Range	212

The integer in the "grid" column of the table shows the computed number of QP's and LP's. The reason for CLP being in general slower than pcx is the need to rewrite a new .mps file for the model input for each call of CLP.
See the updated reference manual for a modified document of the `smsvm()` function.

4.5 Extensions of `boruta()` Function

We have added some additional output (options) and two more return arguments to the `boruta()` function. That means the old call sequence

$$< \text{gof,stat,hist} > = \text{boruta(data,modl<,optn<,class<,cwgt> .. >)}$$

has been changed to the new sequence:

$$< \text{gof,bstv,stat,hist,hist2} > = \text{boruta(data,modl<,optn<,class<,cwgt> .. >)}$$

The following list describes the new output:

gof This is a vector of scalar results, containing

1. indicates failure in execution
2. total time in seconds
3. total number n_{run} random forest calls
4. number "tentative" variable asignments
5. number "rejected" variable asignments
6. number "confirmed" variable asignments

bstv This is a $nc \times 5$ matrix of the nc confirmed (corresponding to rows) variables sorted w.r.t. descending median of the importance statistics, containing in its columns:

1. the number of the variable
2. the median of the importance Z score
3. the mean of the Z score
4. the minimum of the Z score
5. the maximum of the Z score

stat This is a

- $p \times 8$ matrix if `"rough"` is specified or
- $p \times 6$ matrix if `"rough"` is not specified

with univariate statistics of the variable importance Z scores in its columns:

1. mean across all n_{run} random forest runs
2. median across all n_{run} random forest runs
3. minimum across all n_{run} random forest runs
4. maximum across all n_{run} random forest runs
5. Normalized (in $[0, 1]$) number of hits
6. coded final decision: 0: tentative, 1: rejected, 2: confirmed

7. if `"rough"` is specified: updated median estimate,
8. if `"rough"` is specified: updated coded final decision

hist this is a $n_{run} \times 4$ matrix which contains some information about the Z scores of each of the n_{run} random forest calls. The first three columns refer to the Z scores of the randomly permuted variables which are added in each run. The columns refer to:

1. the minimum of the Z scores of the randomly permuted columns
2. the mean of the Z scores of the randomly permuted columns
3. the maximum of the Z scores of the randomly permuted columns
4. the number of nonrejected variables at that time of stage

hist2 this is a $n_{run} \times p$ matrix which contains some information about the Z scores of each of the n_{run} random forest calls. The p columns contain the Z scores of the original variables at that stage; missing values indicate a variable which has been rejected earlier.

The following is the new list of options:

Option	Second Column	Meaning
"conf"	real	significance level of test (default=.999)
"maxrun"	int	maximum number of final runs
"phits"		print number of hits table
"prej"	int	print rejected vars, def=0
		=1: only print for final run
		=2: print for each run
"print"	int	amount of prointed output, def=2
"pzstat"		print Z values, variable importance
"rough"	string	across which Z values Median is computed
	"all"	all Z values are considered
	"final"	only Z values from final runs
		no Median, only the last run
—	—	following options refer to the random forest calls:
"mtry"	int	number of variables randomly selected at each node; default in R: $\max(floor(p/3), 1)$ for regression, $\max(floor(\sqrt(p)), 1)$ for classification
"ndsize"	int	no node with fewer cases will be split
		in R: def=5 for Regression, =1 for classification
"nrnod"	int	maximum number of nodes per tree
		def= 2*N+1 for vers. 5; 2*(N/ndsiz)+1 for vers. 4
"ntree"	int	number of trees to be produced
		here: def=10*N, but in [200,500]; in R: def=500
"seed"	int	seed for random generator (def=time of day)
"vers"	int	should be 0, 4, or 5 (default is 0):
	0	R version for the variable importance
	5	Fortran version 5 for the variable importance
	4	Fortran version 4 for the variable importance

See the updated reference manual for a modified document of the `boruta()` function.

4.6 Extensions of `split()` Function

The third input argument was specified as a vector with input options. Now it specifies a number of options in form of one 2-column matrix where the first column defines the option as string value (in quotes) and the second column can be used for a numeric or string specification of the option, see the following table:

Option Name	Second Column	Meaning
"bins"	int	maximum number of function bins for each interval variable; default=100
"chisq"	real	lower χ^2 threshold for accepting splits; default=0.
"cutoff"	real	cutoff probability for accuracy table; default=.5
"npass"	int	number of passes through data = number splits; default=12
"pobs"	int	printout observationwise stat: predicted values and residuals, def=0 no printout
"print"	int	the amount of printed output, default is 0, that means no printed output
"ptab"	int	printout acuracy tables, def=0 no printout, =1: print only the last table, ¿ 1: all tables

See the updated reference manual for a modified document of the `split()` function.

5 New Developments

5.1 The `rccount()` Function

< nrow,ncol > = rccount(a<,val<,rel>>)

Purpose: The function `rccount()` counts the number of occurences of one specified value in rows and columns of a matrix. (The function may be extended to tensors later.) The first argument, *a* specifies the name of a vector or matrix object, the second argument *val* specifies the scalar numeric or text value that should be looked for and which is being counted. There is an optional third argument *rel*, specifying a relationship between the object values and the value *val*. By default this relationship is $rel = 0$, which means equality, i.e. all object entries with value *equal to val* are replaced by the value *new*. The following relationships are possible:

rel	Replace all entries with
0	values == *val*
-1	values <= *val*
1	values >= *val*
-2	values < *val*
2	values > *val*

Input: **a** The first argument is a vector or matrix `a` of which some of its entries should be counted.

val The second argument specifies the numeric or string value which is looked for in `a` and is counted. The optional argument defaults to 0.

rel The third argument specifies one of the comparison relationships =, $\leq$, $\geq$, <, or >, see table above. The optional argument defaults to equality.

Output: The function returns two vectors:

1. nrow: the number of occurences of `val` in the rows of `a`.
2. nrow: the number of occurences of `val` in the columns of `a`.

Restrictions: 1. The (optional) third input argument must be an integer scalar with values -2, -1, 0, 1 or 2.

Relationships: replace()

Examples: 1. Count zeros in identity matrix:

```
id = ide(10);
< nrow,ncol > = rccount(id,0,0);
print "nrow=", nrow," ncol=", ncol;
```

```
 nrow=
   COL |   1
------------
   1  |   9
   2  |   9
   3  |   9
   4  |   9
   5  |   9
   6  |   9
   7  |   9
   8  |   9
   9  |   9
  10  |   9


 ncol=
  R |   1   2   3   4   5   6   7   8   9  10
-----------------------------------------------
  1 |   9   9   9   9   9   9   9   9   9   9
```

2. Count missing values:

```
a = [ 1  2  2  3,
      .  4  4  5,
      3  .  .  4 ];
< nrow,ncol > = rccount(a,.);
print "nrow=", nrow," ncol=", ncol;
```

```
nrow=
  COL |   1
-----------
    1 |   0
    2 |   1
    3 |   2


 ncol=
 R |   1   2   3   4
---------------------
 1 |   1   1   1   0
```

3. Count values geq 3:

```
a = [ 1  2  2  3,
      .  4  4  5,
      3  .  .  4 ];
< nrow,ncol > = rccount(a,3,1);
print "nrow=", nrow," ncol=", ncol;
```

```
nrow=
  COL |   1
------------
    1 |   1
    2 |   3
    3 |   2
```

```
ncol=
R |   1   2   3   4
---------------------
1 |   1   1   1   3
```

4. Illustrating single and multiple replacement on data from `assoc()` function:

```
ass0=[" lumber hammer nails , lumber hammer nails , lumber hammer   0   ,
       lumber hammer   0   , lumber   0    nails , lumber   0    nails ,
       lumber   0    nails , lumber   0    nails , lumber   0    nails ,
       lumber   0    nails ,   0    hammer nails ,   0    hammer nails ,
         0    hammer nails ,   0    hammer nails ,   0    hammer nails ,
         0    hammer nails ,   0    hammer nails ,   0    hammer nails ,
         0    hammer   0   ,   0    hammer   0   ,   0    hammer   0   ,
         0    hammer   0   ,   0    hammer   0   ,   0    hammer   0   ,
         0    hammer   0   ,   0    hammer   0   ,   0      0    nails ,
         0      0    nails ,   0      0    nails ,   0      0    nails ,
         0      0    nails ,   0      0    nails ,   0      0    nails ,
         0      0    nails ,   0      0    nails ,   0      0    nails ,
         0      0    nails ,   0      0    nails ,   0      0    nails ,
         0      0    nails "];
```

```
< nrow,ncol > = rccount(ass0,"lumber");
print "Lumber: nrow=", nrow," ncol=", ncol;
```

```
ncol=
R |    1    2    3
-------------------
1 |   40   20   10
```

```
< nrow,ncol > = rccount(ass0,"hammer");
print "Hammer: nrow=", nrow," ncol=", ncol;
```

```
ncol=
R |    1    2    3
```

```
------------------
 1 |   30   40   10


   < nrow,ncol > = rccount(ass0,"nails");
   print "Nails: nrow=", nrow," ncol=", ncol;


 ncol=
 R |    1    2    3
-------------------
 1 |   30   20   40
```

5.2 The `sound()` Function

r = sound(ifreq<,octav<,durat>>)

Purpose: Playing a sound of a specific frequency and duration at the computers speakers.

Input: **ifreq** The first input argument must be an integer scalar or a n vector of integers in $[0, 12]$ which is the index of the frequency in the following table corresponding to `octav=6`. For lower octaves the frequencies are step by step divided by 2, for higher octaves the are multiplied by 2. A value of 0 specifies a quiet period.

ifreq	Frequency	Sound
0	0	quiet
1	1046.502	C
2	1108.731	C# / Db
3	1174.659	D
4	1244.508	D# / Eb
5	1318.510	E
6	1396.913	F
7	1479.978	F# / Gb
8	1567.082	G
9	1661.219	G# / Ab
10	1760.000	A
11	1864.655	A# / Bb
12	1975.533	B

octav The second input argument must be an integer scalar or a n vector of integers in $[0, 10]$ specifying the octave where "`octav`" $= 6$ is the default.

durat The third input argument must be a nonnegative integer specifying the duration in milliseconds where "`durat`" $= 200 = 1/5sec$ is the default.

Output: Playing a sound at the computers speakers.

Restrictions:
1. If one of the three input arguments specifies a n vector of integers, the other arguments may be scalars or vectors of the same length n.
2. The first input argument `freq` cannot be missing.

Relationships:

Example:

```
/* freq[0]=0: no sound, quietness */
del = 500;
for (ioct = 8; ioct >= 0; ioct--) {
  freq = cons(24,1);
  for (j = k = 1; j <= 12; j++) {
    freq[k++] = j; freq[k++] = 0;
  }
  sound(freq,ioct,del);
}
```

6 Illustrations

6.1 Some Data Sets for `svm`

Response	Name	Nobs	Nvar	File
interval	CACO	27	714	tsvm7, tsvm9, tsvm23
	NIR	21 (7)	268	tsvm8, tsvm9, tsvm22
	mysvm	100	11	tsvm20, tsvm21
	BostonHousing	506	14	tsvm16, tsvm20, tsvm23
binary	Heart	270	13	tsvm2
	Heart	210 (60)	13	tsvm7
	Diabetes	768	8	tsvm5
	Australian	690	14	tsvm6, tsvm17, tsvm24
	Galaxy	4192	14	tsvm13
	BUPA Liver	345	6	tsvm13
	Ionosphere	351	34	tsvm13
	Pima Diabetes	768	8	tsvm13
	German	1000	20	tsvm13
	Myeloma Affym.	105	7009	tmicro\taffym0
	Myeloma Affym.	105	28033	tmicro\taffym0
	Ulrich Affym.	37	22283	tmicro\taffym2
multinomial	Iris (3 cat)	130	4	tsvm1,tsvm10,tsvm12
	LungCancer (3)	27	56	tsvm8, tsmsvm3
	Roche (18 cat)	719 (190)	29501	tmicro\roche
ordinal	BreastCancer	98	24189	tmicro\vanVeer

For `Nobs` the number in parenthesis is the number of observations in a test set.

6.2 Comparing Computer Time for `lp()` Algorithms

CMAT is compiled with MS Visual C/C++ in *Release* mode (and the Fortran code with Intel Parallel Studio) on a DELL Precision T1650.
The "CON" column in the following table refers to the LPasL1 algorithm by Madsen, Nielsen, & Pinar (1996). Whereas CLP and PCX where always run without output, the more volatile and still not completely bugfree code of "CON" (LPasL1) was run with the output of the iteration history in the `...lst` file. In addition, "CON" was run with a more time consuming preprocessor option checking the linear equality constraints for linear dependencies. That means, the computer time of "CON" can still be improved whenever the code is completely bugfree.
A missing value (dot) for "CON" indicates that the algorithm does not converge within of 3000 iterations, usually due to the oscillating property based on the implementation bug. A missing value for "LPS" usually indicates that the (dense) LPsolve method requires too much memory.

The following table shows the execution time in seconds for the Netlib LP examples:

Name	Nvar	Ncon	Nonz	CON	LPS	PCX
25fv47	1571	821	11127	37.378	0.5300	0.1250
80bau3b	9799	2262	29063	.	4.5390	.
AFIRO	32	27	88	0.0000	0.0000	0.0000
AFTST1	32	25	.	0.0150	.	0.0000
AFTST2	32	25	.	0.0000	.	0.0000
AGG	163	489	2541	.	0.0160	0.0310
AGG2	302	516	4515	.	0.0470	0.0310
AGG3	302	516	4531	.	0.0470	0.0470
BANDM	472	305	2659	0.4370	0.0470	0.0160
BEACONFD	262	173	3476	.	0.0310	0.0000
BLEND	83	74	521	0.0150	0.0000	0.0000
BNL1	1175	643	1175	13.8840	0.1560	0.320
BNL2	3489	2324	16124	.	3.0570	2.6050
BOEING1	384	351	3865	.	0.0780	.
BOEING2	143	166	1339	0.0620	0.0150	0.0000
BRANDY	249	220	2150	.	0.0310	0.0000

Name	Nvar	Ncon	Nonz	CLPprim	CLPdual
25fv47	1571	821	11127	0.2970	0.2960
80bau3b	9799	2262	29063	1.6070	1.5910
AFIRO	32	27	88	0.0000	0.0160
AFTST1	32	25	83	0.0000	0.0000
AFTST2	32	25	83	0.0000	0.0000
AGG	163	489	2541	0.0160	0.0310
AGG2	302	516	4515	0.0310	0.0310
AGG3	302	516	4531	0.0310	0.0150
BANDM	472	305	2659	0.0310	0.0150
BEACONFD	262	173	3476	0.0160	0.0160
BLEND	83	74	521	0.0160	0.0000
BNL1	1175	643	1175	0.1100	0.0930
BNL2	3489	2324	16124	2.6990	2.5740
BOEING1	384	351	3865	0.0150	0.0320
BOEING2	143	166	1339	0.0160	0.0160
BRANDY	249	220	2150	0.0160	0.0150

Name	Nvar	Ncon	Nonz	CON	LPS	PCX
CYCLE	2857	1903	21322	.	.	.
D2Q06C	5167	2171	35674	.	4.6330	.
D6CUBE	6184	415	43888	.	0.7170	0.0940
DEGEN2	534	444	4449	.	0.1250	.
DEGEN3	1818	1503	26230	.	.	0.5460
DFL001	12230	6071	41873	.	.	.
ETAMACRO	688	400	2489	2.3560	0.0780	0.3120
FINNIS	614	497	2714	1.8720	0.0930	0.0620
FIT1D	1026	24	14430	0.0620	0.1870	0.0470
FIT1P	1677	627	10894	.	0.2030	0.0310
FIT2D	10500	25	138018	4.0560	17.8150	0.3740
FIT2P	13525	3000	60784	.	.	0.3430
FORPLAN	421	161	4916	.	0.0630	0.0150
GANGES	1681	1309	70212	25.7550	0.5150	0.3280
GFRD_PNC	1092	616	3467	4.3210	0.1250	.
GROW7	301	140	2633	0.0160	0.0310	0.0160
GROW15	645	300	20	0.0930	0.0780	0.0310
GROW22	946	440	8318	0.2180	0.1560	0.0310

Name	Nvar	Ncon	Nonz	CLPprim	CLPdual
CYCLE	2857	1903	21322	0.6390	0.5930
D2Q06C	5167	2171	35674	2.9010	2.7770
D6CUBE	6184	415	43888	1.5440	0.4210
DEGEN2	534	444	4449	0.0780	0.0470
DEGEN3	1818	1503	26230	0.8110	0.6860
DFL001	12230	6071	41873	29.2970	24.6010
ETAMACRO	688	400	2489	0.0310	0.0310
FINNIS	614	497	2714	0.0630	0.0630
FIT1D	1026	24	14430	0.0940	0.0620
FIT1P	1677	627	10894	0.0930	0.0940
FIT2D	10500	25	138018	3.9620	1.4190
FIT2P	13525	3000	60784	2.1220	2.8700
FORPLAN	421	161	4916	0.0310	0.0160
GANGES	1681	1309	70212	0.3270	0.3280
GFRD_PNC	1092	616	3467	0.0620	0.0630
GROW7	301	140	2633	0.0150	0.0160
GROW15	645	300	20	0.0470	0.0470
GROW22	946	440	8318	0.0930	0.0780

Name	Nvar	Ncon	Nonz	CON	LPS	PCX
ISRAEL	142	175	2358	.	0.0150	0.0320
MAROS	1443	847	10006	.	0.3580	0.0780
MAROS-R7	9408	3137	151120	.	.	.
MODSZK1	1620	688	4158	.	.	.
NESM	2923	663	13988	.	1.2010	10.6240
PEROLD	1376	626	6026	.	.	20.0460
PILOT	3652	1442	43220	.	.	0.8430
PILOT4	1000	411	5145	.	.	5.6780
PILOT87	4883	2031	73804	.	.	977.3010
PILOTNOV	2172	976	13129	.	0.4210	0.1250
RECIPE	180	92	752	0.0000	0.0160	0.0000

Name	Nvar	Ncon	Nonz	CLPprim	CLPdual
ISRAEL	142	175	2358	0.0150	0.0160
MAROS	1443	847	10006	0.1250	0.1250
MAROS-R7	9408	3137	151120	15.8960	15.6780
MODSZK1	1620	688	4158	0.2810	0.2500
NESM	2923	663	13988	0.2650	0.2650
PEROLD	1376	626	6026	0.1560	0.1560
PILOT	3652	1442	43220	2.4020	1.9030
PILOT4	1000	411	5145	0.0780	0.0780
PILOT87	4883	2031	73804	5.9430	6.4430
PILOTNOV	2172	976	13129	0.2650	0.6080
RECIPE	180	92	752	0.0150	0.0000

Name	Nvar	Ncon	Nonz	CON	LPS	PCX
SC50A	48	51	131	0.0000	0.0000	0.0000
SC50B	48	51	119	0.0150	0.0000	0.0000
SC105	103	106	281	0.0000	0.0000	0.0000
SC205	206	203	552	0.0160	0.0150	0.0000
SCAGR7	140	130	553	0.0000	0.0160	0.0000
SCAGR25	500	472	2029	0.2650	0.0630	0.0000
SCFXM1	457	331	2612	0.3900	0.0470	0.0160
SCFXM2	914	661	5229	3.6510	0.1400	0.0470
SCFXM3	1371	991	7846	.	0.2960	0.0780
SCORPION	358	389	1708	.	0.0310	.
SCRS8	1169	491	4029	1.9660	0.1090	0.0780
SCSD1	760	78	3148	0.0320	0.0150	0.0160
SCSD6	1350	148	5666	.	0.0470	0.0310
SCSD8	2750	398	11334	1.9340	0.2810	0.1400
SCTAP1	480	301	2052	0.6710	0.0310	0.0160
SCTAP2	1880	1091	8124	54.0230	0.5460	0.4530
SCTAP3	2480	1481	10734	234.8270	2.1220	1.9500

Name	Nvar	Ncon	Nonz	CLPprim	CLPdual
SC50A	48	51	131	0.0160	0.0000
SC50B	48	51	119	0.0000	0.0000
SC105	103	106	281	0.0000	0.0000
SC205	206	203	552	0.0160	0.0150
SCAGR7	140	130	553	0.0160	0.0000
SCAGR25	500	472	2029	0.0320	0.0150
SCFXM1	457	331	2612	0.0150	0.0320
SCFXM2	914	661	5229	0.0620	0.0630
SCFXM3	1371	991	7846	0.1410	0.1240
SCORPION	358	389	1708	0.0150	0.0160
SCRS8	1169	491	4029	0.0780	0.0780
SCSD1	760	78	3148	0.0150	0.0160
SCSD6	1350	148	5666	0.0310	0.0470
SCSD8	2750	398	11334	0.2340	0.2030
SCTAP1	480	301	2052	0.0150	0.0310
SCTAP2	1880	1091	8124	0.4680	0.4680
SCTAP3	2480	1481	10734	1.9970	1.9810

Name	Nvar	Ncon	Nonz	CON	LPS	PCX
SEBA	1028	516	4874	1.7630	0.0780	0.0930
SHARE1B	225	118	1182	.	0.0310	0.0000
SHARE2B	79	97	730	0.0150	0.0160	0.0000
SHELL	1775	537	4900	.	0.1250	.
SHIP04L	2118	403	8450	0.5930	0.1560	0.0780
SHIP04S	1458	403	5810	0.2180	0.0940	0.0310
SHIP08L	4283	779	17085	2.5270	0.7960	0.5610
SHIP08S	2387	779	9501	0.4680	0.1720	0.0780
SHIP12L	5427	1152	21597	5.0230	1.3880	1.0610
SHIP12S	2763	1152	10941	0.5770	0.2180	.
SIERRA	2036	1228	9252	.	0.6400	.
STAIR	467	357	3857	0.5150	.	0.0310
STANDATA	1075	360	3038	0.4680	0.0630	0.0150
STANDGUB	1184	362	3147	0.4520	0.0620	0.0160
STANDMPS	1075	468	3686	.	0.0780	0.0320
STOCFOR1	111	118	474	0.0000	0.0150	0.0000
STOCFOR2	2031	2158	9492	.	1.6220	1.2640
TUFF	587	334	4523	0.7800	.	0.0310
WOOD1P	2594	245	70216	.	0.4210	.
WOODW	8405	1099	37478	.	1.4970	0.3440

Name	Nvar	Ncon	Nonz	CLPprim	CLPdual
SEBA	1028	516	4874	0.0320	0.0310
SHARE1B	225	118	1182	0.0000	0.0150
SHARE2B	79	97	730	0.0150	0.0000
SHELL	1775	537	4900	0.0630	0.0620
SHIP04L	2118	403	8450	0.1090	0.1250
SHIP04S	1458	403	5810	0.0470	0.0620
SHIP08L	4283	779	17085	0.6400	0.6390
SHIP08S	2387	779	9501	0.1100	0.1090
SHIP12L	5427	1152	21597	1.1860	1.1700
SHIP12S	2763	1152	10941	0.1560	0.1410
SIERRA	2036	1228	9252	0.3900	0.4060
STAIR	467	357	3857	0.0470	0.0310
STANDATA	1075	360	3038	0.0160	0.0160
STANDGUB	1184	362	3147	0.0150	0.0150
STANDMPS	1075	468	3686	0.0460	0.0470
STOCFOR1	111	118	474	0.0160	0.0000
STOCFOR2	2031	2158	9492	1.2940	1.2800
TUFF	587	334	4523	0.0310	0.0160
WOOD1P	2594	245	70216	0.4050	0.3750
WOODW	8405	1099	37478	0.6860	0.6710

7 The Bibliography

References

[1] Bi, J., Bennett, K.P., Embrechts, M., Breneman, C.M., & Song, M. (2002), "Dimensionality reduction via sparse support vector machines"; *Journal of Machine Learning*, **1**, 1-48.

[2] Czyzyk, J., Mehrotra, S., Wagner, M. & Wright, S.J. (1997) "PCx User Guide (Version 1.1)", Technical Report OTC 96/01, Office of Computational and Technology Research, US Dept. of Energy.

[3] Forrest, J. J. & Lougee-Helmer, R. (2014), *Cbc*, `jjforre@us.ibm.com`, `robinlh@us.ibm.com`.

[4] Forrest, J. J., de la Nuez, D., & Lougee-Helmer, R. (2014), *Clp*, `http://www.tsp.gatech.edu/concorde`.

[5] Guyon, I., & Elisseeff, A. (2003), "An introduction to variable and feature selection", *Journal of Machine Learning Research*, **3**, 1157-1182, 2003.

[6] Kursa, M.B. & Rudnicki, W.R. (2010), "Feature Selecion with the Boruta Package"; *JSS*, 2010.

[7] Lee, Y., Lin, Y., & Wahba, G. (2003), *Multicategory Support Vector Machines, Theory and Application to the Classification of Microarray Data and Satellite Radiance Data*, Technical Report 1064, Dep. of Statistics, Univ. of Wisconsin, 2003.

[8] "MPS file format", `http://lpsolve.sourceforge.net/5.5/mps-format.htm`

[9] Madsen, K., Nielsen, H.B., & Pinar, M.C. (1996), "A new finite continuation algorithm for linear programming", *SIAM Journal on Optimization*, **6**, 600-616.