

# Innehåll

<b>RCA</b>	<b>3</b>
ProtocolQ . . . . .	5
AddFactor . . . . .	7
AddVariable . . . . .	8
RemoveFactor . . . . .	8
RemoveVariable . . . . .	8
RenameVariable . . . . .	8
RenameFactor . . . . .	8
RenameModalities . . . . .	9
RepositionFactors . . . . .	9
CombineFactors . . . . .	10
Factor2Variable . . . . .	10
Variable2Factor . . . . .	10
Recode . . . . .	11
SubProtocol . . . . .	12
Group . . . . .	13
Factors . . . . .	14
Variables . . . . .	14
Modalities . . . . .	14
Info . . . . .	15
ListData . . . . .	15
GetNumberOfWords . . . . .	15
GetModalities . . . . .	15
ModalityIds . . . . .	16
GetVariableValues . . . . .	16
GetText . . . . .	16
Name2Index . . . . .	16
Index2Name . . . . .	17
CheckModality . . . . .	18
CheckVariable . . . . .	18
ProtocolQMess . . . . .	18
ArgumentQ . . . . .	18
CheckIndexBounds . . . . .	19
<b>Linguistics</b>	<b>20</b>
GetStopWordList . . . . .	22
AddHitCountVariable . . . . .	23
AddCollocationVariables . . . . .	23
AddPOSVariables . . . . .	23
TagProtocol . . . . .	23
StemHitCountVariables . . . . .	24
AddTermDiscriminationValues . . . . .	24
SplitText . . . . .	24
CleanProtocol . . . . .	26
LSA . . . . .	26
StemProtocol . . . . .	28
WordFrequencies . . . . .	28
ZScore . . . . .	28
TScore . . . . .	28
SpanFrequency . . . . .	28
RemoveTags . . . . .	29
TagWords . . . . .	29
GroupAndTag . . . . .	30
WordStemmer . . . . .	30
WordList . . . . .	30

NumberOfWordsVariable . . . . .	31
NumberOfSentencesVariable . . . . .	31
AverageWordLengthVariable . . . . .	31
AverageSentenceLengthVariable . . . . .	31
WordVariationVariable . . . . .	31
TermDiscriminationValues . . . . .	31
GetFilteredWords . . . . .	32
SmoothPartition . . . . .	32
PairsOfList . . . . .	32
PairsOfLists . . . . .	34
CA2 . . . . .	35
<b>Transformation</b>	<b>35</b>
ShowIterations . . . . .	38
IndicatorMatrix . . . . .	38
MCA . . . . .	39
GetSingularValues . . . . .	41
GetCoordinateStatus . . . . .	41
GetPrincipalAxes . . . . .	41
ModalityCoordinates . . . . .	41
CrossTabulate . . . . .	42
CA . . . . .	43
FactorContributions . . . . .	44
ModalityContributions . . . . .	45
FactorAssociationGraph . . . . .	46
ModalityAssociationGraph . . . . .	46
PCA . . . . .	46
GetEigenvalues . . . . .	48
FirstStep . . . . .	49
NLPCAInner . . . . .	50
InnerIter . . . . .	51
ToProtocol . . . . .	52
<b>GUI</b>	<b>53</b>
CreateSymbol . . . . .	54
Cloud . . . . .	55
Interact . . . . .	57
GetCoordinates . . . . .	59

# RCA

```

BeginPackage["RCA`"]

Unprotect[METADATA, TEXTDATA, FACTORS, VARIABLES];
5 METADATA=1;
TEXTDATA=2;
FACTORS=3;
VARIABLES=4;

10 Protect[METADATA, TEXTDATA, FACTORS, VARIABLES];

Unprotect[
Protocol,
ProtocolQ,
15 GetNumberOfWords,
CreateProtocol,
AddFactor,
AddVariable,
Factors,
20 CombineFactors,
Factor2Variable,
Variable2Factor,
FactorIndex2Name,
FactorName2Index,
25 VariableIndex2Name,
VariableName2Index,
ModalityIndex2Name,
ModalityName2Index,
Index2Name,
30 Name2Index,
CheckModality,
CheckVariable,
CheckIndexBounds,
GetModalityValues,
35 GetModalityIds,
GetVariableValues,
GetText,
RemoveFactor,
RemoveVariable,
40 Group,
Variables,
Modalities,
Info,
ListData,
45 SubProtocol,
Recode,
RenameVariable,
RenameFactor,
RenameModalities,
50 ArgumentQ,
CheckIndexBounds,
RepositionFactors,
Cloud,
GetCoordinates,
55 CreateSymbol,
Interact
];

60 (* ::Section:: *)
(*Usage:*)

Protocol::usage="Head of protocols.";
65 Format[Protocol[l_?List]] := Info[Protocol[l]];

ProtocolQ::usage="ProtocolQ[expr] gives True if expr is a Protocol, and False otherwise.";
TagError::usage="TagError is an option for ProtocolQ specifying if you want to return the
error message with the function as a tuple ex: {False, \"NameConflictError\"}.";
70 GetNumberOfWords::usage="GetNumberOfWords[p] gives the list of the number of words of the texts in
protocol p. a \"word\" is defined as a string containing non-space characters.
GetNumberOfWords[p, sep] the words are separated by sep.
GetNumberOfWords[p, list] gives a list of number of words in the texts specified in \"list\".
GetNumberOfWords[p, list, sep] the words are separated by sep.";
75 CreateProtocol::usage="CreateProtocol[txts] creates a Protocol based on the list of texts in txts.
"
AddFactor::usage="AddFactor[p,fn,d] adds a factor with name fn and data d to protocol p."

```

```

80 AddVariable::usage="AddVariable[p,vn,d] adds a variable with name vn and data d to protocol p."
Factors::usage="Factors[p] lists the factors in protocol p. See Options[Factors] for a list of options."
Variables::usage="Variables[p] lists the variables in protocol p. See Options[Variables] for a list of options."
85 Modalities::usage="Modalities[p,f] lists the modalities in factor f in protocol p with some information about frequencies."
Info::usage="Info[p] prints information about protocol p. Since Format[Protocol[p_List]]=Info[p] this is also what is show when a protocol is converted on its own to output."
CombineFactors::usage="CombineFactors[p,t,fc] combines the factors in the list fc and adds it to the protocol p with new title t."
90 Factor2Variable::usage="Factor2Variable[p, facname] converts factor facname into a variable from protocol p."
Variable2Factor::usage="Variable2Factor[p, fac] converts variable varname/varnumber into a factor from protocol p and group the variable values into modalities. See Options[Variable2Factors] for options."
95 Group::usage="Group[p,f] merges individuals with the same modalities for a factor f, and returns the corresponding protocol. To keep additional factors or variables, specify this with options \\"KeepFactors\\" and \\"KeepVariables\\" respectively. For example, \\"KeepVariables\\" -> {{Mean,{1,2}}, {Median,{3,4}}} will carry over variables 1 and 2 by taking the mean of grouped individuals, while variables 3 and 4 will carry over using the median."
ListData::usage="ListData[p] lists the data in p. See Options[ListData] for options."
Recode::usage =
100 "Recode[p,fact,newname,repl,mods] takes a factor fact in protocol p and list of lists repl. Each list in repl is a list of modalities and list number j in repl will be recoded to modality j in mods. The new factor with the new modalities will get the name newname.";
RenameVariable::usage="RenameVariable[p,{{v,nn}...}] takes a list of one or more pairs {v,nn} where variable v gets the new name nn
RenameVariable[p,1] renames all variables in protocol p with the names in the list 1"
105 RenameFactor::usage="RenameFactor[p,{{f,nn}...}] takes a list of one or more pairs {f,nn} where variable f gets the new name nn
RenameFactor[p,1] renames all factors in protocol p with the names in the list 1"
RenameModalities::usage="RenameModalities[p,f,{{m,nn}...}] takes a list of one or more pairs {m,nn} where modality m (in factor f) renames to nn
RenameModalities[p,f,1] renames all modalities in factor f with the names in the list 1"
110 FactorName2Index::usage="FactorName2Index[p,n] returns the index of factor name n in p.";
VariableName2Index::usage="VariableName2Index[p,n] returns the index of factor name n in p.";
ModalityName2Index::usage="ModalityName2Index[p,n] returns the index of factor name n ({factor, modality}) in p.";
FactorIndex2Name::usage="FactorIndex2Name[p,i] returns the factor name of factor at index i in p."
115 VariableIndex2Name::usage="FactorIndex2Name[p,i] returns the variable name of variable at index i in p.";
ModalityIndex2name::usage="FactorIndex2Name[p,i] returns the modality name of factor at f and modality m in p given in argument 2 as a two element list {f,m}.";
Index2Name::usage="Index2Name[p, i, type] returns the name of the type pointed at by index. type is string specifying the type of data. Example: Index2Name[p,2,\\"Factor\\"].";
Name2Index::usage="name2Index[p, n, type] returns the index to the type with name n. Example: Name2Index[p,3,\\"Variable\\"]."
120 SubProtocol::usage="SubProtocol[p, f] Returns the sub protocol with the individuals in which the function f is true (#1 = factor, #2= variable, #3 = text, #4 = text index).
SubProtocol[p, list] picks out all texts at indices in list (list of Integers).
SubProtocol[p, var, f] uses the function f on variable var (Integer or String).
SubProtocol[p, fc, mod] with factor fc (Integer or String) and modality index mod (Integer).
125 SubProtocol[p, fc, mods] with factor fc (Integer or String) and modality indices mods (Integers). See Options[SubProtocol] for options.";
GetText::usage="GetText[p] retrieves all the texts in protocol p.
GetText[p, n] retrieves the nth text from protocol p.
130 GetText[p, list] retrieves texts specified by a list of integers in list."
ListData::usage="ListData[p] lists the data in p. See Options[ListData] for options.";
RepositionFactors::usage="RepositionFactors[p, rules] rearranges the factors of protocol p using the index rules in rules. The positioning of the factors is inserting, moving the factor from lhs to rhs in turn.";
```

135 **DetailedError**::usage="DetailedError is an option for ProtocolQ that specifies if the detail of the error message is detailed or not.";

```

CheckVariable::usage="CheckVariables[p, var] Aborts evaluation if the variable var does not exist.
"
CheckModality::usage="CheckModality[p, fc, mod] Aborts evaluation if the modality mod does not
exist."
140
(* ::Section:: *)
(* Options:: *)

145 Options[ProtocolQ]={DetailedError->True, TagError->False};
TagError::usage="TagError is an option for ProtocolQ used by VersionProtocol to assess the
correctness of ProtocolQ by specifying which error was returned.";
Options[Factors]={TableForm->True, Header->{"Id","Factor","Modalities"}};

150 Header::usage="Header is an option for listing functions like Factors, Variables, Modalities and
Info that specifies which headers to display."
TableForm::usage="TableForm is an option for listing functions like Factors, Variables, Modalities
and Info that specifies (Boolean) if the data will be displayed in table form."
Options[Variables]={TableForm->True, Header->{"Id","Variable","Min","Max","Median","Mean","SD"}};

155 SortByFrequencies::usage="SortByFrequencies is an option for Modalities which specifies the order
of the modality listing."; (*check*)
TruncateNames::usage="TruncateNames is an option for Modalities which specifies if the names
should be truncated or not.";
Options[Modalities]={TableForm->True, Header->{"Id","Modality","F","RF"}, SortByFrequencies->False
, TruncateNames->True};

Options[Info]={TableForm->True, Header->{"Rows","Factors","Variables"}};

160 Options[CombineFactors] = {IncludeAll -> True};
IncludeAll::usage="IncludeAll is an option for CombineFactors specifying whether to include all
factors or not.";

Option[Factor2Variable]={KeepOriginal->True;(*riight? -Matz*)}
165 KeepOriginal::usage="KeepOriginal is an option for Factor2Variable specifying if the factors
after conversion should be kept or not./";

Options[Group] = {KeepFactors->{{First,{}}},KeepVariables -> {{Mean,{}}}};
Options[ListData]={TableForm -> True, TruncateNames -> True};

170 Options[Recode] = {OverwriteFactor -> False};
OverwriteFactor::usage="OverwriteFactor is an option for Recode which specifies whether to
overwrite factor of not.";(*rephrase?*)

Options[SubProtocol] = {KeepEmptyModalities->False};
KeepEmptyModalities::usage = "An option for SubProtocol which specifies if you want to keep empty
modalities or not.";

175 Options[ListData] = {TableForm-> True, TruncateNames-> True, SortBy->None};
TruncateNames::usage = "An option for ListData which specifies if you want to truncate the names
or not";

Options[Variable2Factor]={KeepVariable->True, Method->Automatic, DistanceFunction->Automatic,
FactorNames->Automatic};

180 KeepVariable::usage = "An option for Variable2Factor which specifies whether to keep variable
after conversion.";
FactorNames::usage="";

185 Begin[``Private`"]

(* ::Section:: *)
(* Basic functionality :*)

190 (** CreateProtocol, by Sverker 2010-01-19 **)
CreateProtocol[txts_List]:=Protocol[ {
(* Metadata *)
{"ProtocolTitle"->""},
(* Texts *)

$$\text{txts},$$

(* Factors *)
{} ,
(* Variables *)
{} }];
```



```

fIndices = p[[FACTORS, All, 3]];
vIndices = p[[VARIABLES, All, 2]];

275 maxInds = Max/@ fIndices; (*the used maximum indices*)
maxMods = Length /@ mods; (*maximum number of modalities, for each factor*)
minInds = Min/@ fIndices;
numTexts = Length@textsAll;
lenVInds = Length /@ vIndices;
lenFInds = Length /@ fIndices;
280 (*Test that each factor has 3 elements, factorname, modality and indices to the modalities*)

(* LENGTH TESTS *)
errorIndices=Position[({==numTexts&/@ lenVInds), False] //Flatten;
If[Length[errorIndices]!=0, If[mess, Message[ProtocolQ::ValuesLengthError, lenVInds,
errorIndices, numTexts]]; Return@ProtocolQMess["ValuesLengthError", tag]];

285 If[!( And @@ ({==3 & /@ (Length[p[[FACTORS, #]]& /@ Range[Length[facAll]]])), If[mess,
Message[ProtocolQ::FactorLengthError]]; Return@ProtocolQMess["FactorLengthError", tag]]];

errorIndices=Position[({==numTexts&/@ lenFInds), False]; (*The number of indices (at modalities)
are not the same as the number of texts*)
If[Length[errorIndices]!=0, If[mess, Message[ProtocolQ::ModalityLengthError, p[[FACTORS,
errorIndices//Flatten, 1]], numerus[numTexts], numTexts ]]; Return@ProtocolQMess["ModalityLengthError", tag]];

290 (* If [Depth[p[[VARIABLES]]]==4 ,(*make sure we have variables*)
errorIndices=Position[(Length[#]<=numTexts&/@ vIndices), False]]; (*where are the indexing
outofbounds?*)*)

errorIndices=Position[(1<=#[[1]]<=#[[2]]<=#[[3]] &/@ Transpose[{minInds, maxInds, maxMods}]), False] //Flatten; (*outofbounds test*)
295 If[Length[errorIndices]!=0, If[mess, Message[ProtocolQ::ModalityOutOfBoundsError, errorIndices,
numerus[Length@maxMods[[errorIndices]]], maxMods[[errorIndices]]]]; Return@ProtocolQMess["ModalityOutOfBoundsError", tag]];

(* TYPE TESTS *)
errorIndices=Position[ListStringsQ[textsAll], False];
If[Length[errorIndices]!=0, If[mess, Message[ProtocolQ::TextTypeError, errorIndices]];
Return@ProtocolQMess["TextTypeError", tag]];

300 errorIndices=Position[ListStringsQ[varsAll[[All, 1]]], False];
If[Length[errorIndices]!=0, If[mess, Message[ProtocolQ::VariableTypeError, errorIndices]];
Return@ProtocolQMess["VariableTypeError", tag]];

errorIndices=Position[ListsIntegersQ[facAll[[All, 3]]], False];
305 If[Length[errorIndices]!=0, If[mess, Message[ProtocolQ::ModalityIndicesTypeError, errorIndices]];
Return@ProtocolQMess["ModalityIndicesTypeError", tag]];

errorIndices=Position[ListsStringsQ[facAll[[All, 2]]], False];
If[Length[errorIndices]!=0, If[mess, Message[ProtocolQ::ModalityTypeError, errorIndices]];
Return@ProtocolQMess["ModalityTypeError", tag]];

310 errorIndices=Position[ListStringsQ[facAll[[All, 1]]], False];
If[Length[errorIndices]!=0, If[mess, Message[ProtocolQ::FactorTypeError, errorIndices]];
Return@ProtocolQMess["FactorTypeError", tag]];

errorIndices=Position[ListsNumbersQ[varsAll[[All, 2]]], False]; (*Integer or real etc.*)
If[Length[errorIndices]!=0, If[mess, Message[ProtocolQ::ValuesTypeError, errorIndices]];
Return@ProtocolQMess["ValuesTypeError", tag]];

315 (* DUPLICATE TESTS *)
(*Duplicates, ignores case according to "Protocol Structure" (see also Name2Index)*)
errorIndices = Select[Tally[facAll[[All, 1]], ToUpperCase@#1==ToUpperCase@#2&], #[[2]]> 1 & ];
If[Length@errorIndices>0, If[mess, Message[ProtocolQ::NameConflictError, "factor", (ToString@
#[[1]]~~" occurred ~~ToString@#[[2]]~~" times" &) /@ errorIndices]]; Return@ProtocolQMess["NameConflictError", tag]];

320 errorIndices = Select[Tally[varsAll[[All, 1]], ToUpperCase@#1==ToUpperCase@#2&], #[[2]]> 1 & ];
If[Length@errorIndices>0, If[mess, Message[ProtocolQ::NameConflictError, "variable", (ToString@
#[[1]]~~" occurred ~~ToString@#[[2]]~~" times" &) /@ errorIndices]]; Return@ProtocolQMess["NameConflictError", tag]];

325 If[tag, Return[{True, ""}], Return[True]];

```

---

```

(** AddFactor, by Sverker 2010-01-19 **)
330 AddFactor::lengtherror="The number of rows in the new factor does not match the number of rows in
the Protocol.";
AddFactor[Protocol][pd_List], facname_String, data_List]:=Module[{newpd, mods, reprules, ids},
(* Check so that length of data is same as nr of rows in protocol *)

```

```

335 If [Length [pd [[TEXTDATA]]] != Length [data], Message [AddFactor :: lengtherror]; Return []];
      newpd=pd;
      mods=Union [data]; (* Unique modalities *)
      reprules=Array [mods[[#1]]->#1&, Length [mods]]; (* List of replacements from data to modality-id,
          ie. {"Name one" ->1, "Name two" ->2,... *})
      ids=data/.reprules; (* Do the replacements *)
      340 newpd [[FACTORS]]=Append [newpd [[FACTORS]], {facname, mods, ids}]; (* Add the new factor*)
      Return [Protocol [newpd]];

```

---

```

345 (** AddVariable, by Sverker 2010-01-19 **)
AddVariable :: lengtherror="The number of rows in the new variable does not match the number of rows
in the Protocol";
AddVariable [Protocol [pd_List], varname_String, data_List]:=Module [{newpd},
      newpd=pd;
      newpd [[VARIABLES]]=Append [newpd [[VARIABLES]], {varname, data}];
      350 Return [Protocol [newpd]];

```

---

```

355 (** RemoveFactor, by Tobias Raski 2010-01-26 **)
RemoveFactor [Protocol [pd_List], fac_]:=Module [{newpd, facname},
      If [fac // IntegerQ, facname = FactorIndex2Name [Protocol [pd], fac], facname = fac];
      newpd = pd;
      newpd [[FACTORS]] = Select [newpd [[FACTORS]], First [#] != facname &];
      360 Return [Protocol [newpd]];

```

---

```

365 (** RemoveVariable, by Tobias Raski 2010-01-26 **)
RemoveVariable [Protocol [pd_List], var_]:=Module [{newpd, varname},
      If [var // IntegerQ, varname = VariableIndex2Name [Protocol [pd], var], varname = var];
      newpd = pd;
      newpd [[VARIABLES]] = Select [newpd [[VARIABLES]], First [#] != varname &];
      Return [Protocol [newpd]];
      370

```

---

```

      (**RenameVariable, by Anna 2010-03-06**)
RenameVariable :: lengtherror="The number of new names does not equal the number of variables.";
RenameVariable [Protocol [pd_List], pairs:{-_Integer|_String, _String}...]:=Module [{newpd, ids},
      375 newpd=pd;
      ids={};
      For [i=1, i<=Length [pairs], i++,
          If [StringQ [pairs [[i, 1]]], ids=Append [ids, Position [newpd [[VARIABLES]], pairs [[i, 1]]][[1, 1]]], ids=
              Append [ids, pairs [[i, 1]]]];
      ];
      380 For [i=1, i<=Length [ids], i++, newpd [[VARIABLES, ids [[i]], 1]]=pairs [[i, 2]]];
      Return [Protocol [newpd]];
      ]

```

---

```

385 RenameVariable [Protocol [pd_List], newnames_List]:=Module [{newpd},
      (*Checks that the the number of new names agrees with the number of variables in the protocol*)
      If [Length [pd [[VARIABLES]]] != Length [newnames], Message [RenameVariable :: lengtherror]; Return []];
      newpd=pd;
      (*Changing names*)
      For [i=1, i<=Length [newnames], i++, newpd [[VARIABLES, i, 1]]=newnames [[i]]];
      Return [Protocol [newpd]];
      ]

```

---

```

395 (**RenameFactor, by Anna 2010-03-06**)
RenameFactor :: lengtherror="The number of new names does not equal the number of factors.";
RenameFactor [Protocol [pd_List], pairs:{-_Integer|_String, _String}...]:=Module [{newpd, ids},
      400 newpd=pd;
      ids={};
      For [i=1, i<=Length [pairs], i++,
          If [StringQ [pairs [[i, 1]]], ids=Append [ids, Position [newpd [[FACTORS]], pairs [[i, 1]]][[1, 1]]], ids=
              Append [ids, pairs [[i, 1]]]];
      ];
      405 For [i=1, i<=Length [ids], i++, newpd [[FACTORS, ids [[i]], 1]]=pairs [[i, 2]]];
      Return [Protocol [newpd]]];

```

---

```

410 RenameFactor[Protocol[pd_List], newnames_List]:=Module[{newpd},
  (*Checks that the number of new names agrees with the number of variables in the protocol*)
  If[Length[pd[[FACTORS]]]!=Length[newnames], Message[RenameFactor::lengtherror];Return[]];
  newpd=pd;
  (*Changing names*)
  For[i=1,i<=Length[newnames], i++, newpd[[FACTORS, i, 1]]=newnames[[i]]];
  Return[Protocol[newpd]]

```

420

---

```

(**RenameModalities, by Anna 2010-03-06, last updated 6/5**)
RenameModalities::lengtherror="The number of new names does not equal the number of modalities.";
RenameModalities[Protocol[pd_List], id_, pairs:{-_Integer|_String,_String}...]:=Module[{newpd,ids,
  factorid},newpd=pd];
425  ids={};
  If[StringQ[id],
    factorid=Position[pd[[FACTORS]], id][[1,1]],
    factorid=id
  ];
430  For[i=1, i<=Length[pairs], i++,
  If[StringQ[pairs[[i,1]]],
    ids = Append[ids, Position[newpd[[FACTORS, factorid, 2]], pairs[[i,1]]][[1,1]]],
    ids = Append[ids, pairs[[i,1]]]
  ];
435  ];
  For[i=1, i<=Length[ids], i++,
    newpd[[FACTORS, factorid, 2]][[ids[[i]]]] = pairs[[i,2]]
  ];
440  Return[Protocol[newpd]]

```

  

```

RenameModalities[Protocol[pd_List], id_, newnames_List];VectorQ[newnames, StringQ]:=Module[{newpd,
  factorid},(*Checks that the number of new names agrees with the number of modalities in
  the factor*)If[StringQ[id], factorid=Position[pd[[FACTORS]], id][[1,1]], factorid=id];
  If[Length[pd[[FACTORS, factorid, 2]]]!=Length[newnames], Message[RenameModalities::lengtherror];
    Return[]];
445  newpd=pd;
  (*Changing names*)newpd[[FACTORS, factorid, 2]]=newnames;
  Return[Protocol[newpd]]
]

```

450

---

```

(**RepositionFactors, by Matz 3/5~10 version 0.2, fixade en bugg, anv\[ADoubleDot]nder numera
  checkindexbounds**)
(*RepositionFactors, by Matz 1/5~10 version 0.1*)
RepositionFactors::OutOfBoundsError="` is out of bounds, the index should be between 1 and the
  number of individuals ('') inclusive";
455 RepositionFactors[Protocol[pd_List], switch_List]:=Module[{newpd=Protocol[pd], indices, len, to, from
  , range, switchlen, tmp},
  switchlen = Length@switch;
  len = Length@pd[[FACTORS, All, 1]];
  {from, to} = {switch[[All, 1]], switch[[All, 2]]};
  range = Join[from, to];(*test indices range*)
  CheckIndexBounds[Protocol[pd], range, FACTORS];
  indices = Range@len;

  (*We insert the new element hence need to know if the old element is shifted, insert also "
   prepends", we compensate for this also*)
  Do[
465    If[from[[i]] <= to[[i]],
      tmp = Insert[indices, indices[[from[[i]]]], to[[i]]+1];
      indices = Delete[tmp, from[[i]]];
      ,
      tmp = Insert[indices, indices[[from[[i]]]], to[[i]]];
      indices = Delete[tmp, from[[i]]+1];
    ];
    , {i, 1, switchlen}];
  newpd[[1, FACTORS]] = newpd[[1, FACTORS, indices]]; (*finalize changes*)
475  Return[newpd]
]

```

  

```

480 (* ::Section:: *)
  (*Converting - functions which potentially makes significant changes to a Protocol:*)

```

---

```

485 (** CombineFactors, by Sandra 2010-02-09 **)
CombineFactors::factorerror="One (or more) of the factors does not exist.";
CombineFactors[Protocol[pd_List], title_String, factors_List, OptionsPattern[CombineFactors]]:=
Module[{f, rules, factorsind, ia, a, mods, modsused, ids, newpd, newfactor},
(*Finds the indexes of the factors*)
490 f = Factors[Protocol[pd], TableForm->False][[All, 1;;2]];
rules = Array[f[[#1, 2]]->f[[#1, 1]]&, Length[f]];
factorsind = factors/.rules;
(*Checks that the factors exists*)
If[!ArrayQ[factorsind], _, IntegerQ[Positive[factorsind], False], Message[
  CombineFactors::factorerror]; Return[]];
495 If[Max[factorsind]>Length[pd[[FACTORS]]], Message[CombineFactors::factorerror]; Return[]];

(* Combines the factors*)
ia = OptionValue[IncludeAll];

500 If[ia,
  (*Default option, combine all the modalities of the given factors*)
  a = Prepend[Array[pd[[FACTORS, factorsind[[#]], 2]]&, Length[factorsind]], StringJoin@@Riffle
    {[##}, ", ", "]&];
  mods = Flatten[Outer@a]; (*All possible combinations of the modalities*)

505 (*The combinations of modalities used by the texts in the protocol*)
  modsused = StringJoin@@Riffle[#, ", ", "]& /@ Array[pd[[FACTORS, factorsind[[#2]], 2, pd[[FACTORS,
    factorsind[[#2]], 3, #1]]]]&, {Length[pd[[TEXTDATA]]], Length[factorsind]}];
  ids = Flatten[Position[mods, #]&/@modsused]; (*The indexes of the used modalities*)

510 newpd = pd;
newpd[[FACTORS]] = Append[pd[[FACTORS]], {title, mods, ids}];
Return[Protocol[newpd]];
;
(*If option "IncludeAll" is false, combine only used modalities*)
515 newfactor = StringJoin@@Riffle[#, ", ", "]& /@ Array[pd[[FACTORS, factorsind[[#2]], 2, pd[[FACTORS,
  factorsind[[#2]], 3, #1]]]]&, {Length[pd[[TEXTDATA]]], Length[factorsind]}];
newpd = AddFactor[Protocol[pd], title, newfactor];
Return[newpd]
];
520 ]

```

---

```

525 (** Factor2Variable, by Tobias Raski 2010-01-26 **)
Factor2Variable::conversionerror="Factor can't be converted into a variable. On or more modality
values can't be converted to Integer";

Factor2Variable[Protocol[pd_List], fac_, OptionsPattern[Factor2Variable]] := Module[{facname, mods
  , newpd, ids, values, reprules, del},
  If[fac // IntegerQ,
    facname = FactorIndex2Name[Protocol[pd], fac],
    facname = fac];
530
  mods = ToExpression[#] & /@ GetModalityValues[Protocol[pd], fac];
  If[(And @@ (IntegerQ /@ mods)) // Not,
    Message[Factor2Variable::conversionerror]; Return[]];
  ids = GetModalityIds[Protocol[pd], fac];
  reprules = Array[# -> mods[[#]] &, Length[mods]];
  values = ids /. reprules;
540 newpd = AddVariable[Protocol[pd], facname, values];
  del = OptionValue[KeepOriginal];
  If[del // Not,
    newpd = RemoveFactor[newpd, facname], (* delete factor facname if option KeepOriginal -> False
      was set *)
545   newpd];
  Return[newpd]
]


```

---

```

550 (**Variable2Factor, by Tobias Raski 2010-02-27,updated 03-30**)
(*Variable2Factor, Sandra lade till extra funktionalitet 2010-05-04*)
Variable2Factor::imagedlengtherror="the length of the given image to the grouping function is to
  small.";
Variable2Factor[Protocol[pd_List], var_, image_List, fn_, OptionsPattern[Variable2Factor]] := Module
  [{varname, values, newpd, ivals},

```

```

555  If [ var//IntegerQ ,
      varname=VariableIndex2Name[Protocol[pd], var] ,
      varname=var
    ];
560  values = GetVariableValues[Protocol[pd], varname];
      ival = fn /@ values;
      If[Select[ival, Not[0<#<=Length[image]]&] != {},
        Message[Variable2Factor::imageLengthError]; Abort[]
      ];
565  ival = ival /. Array[#->image[[#]]&, Length[image]];
      newpd = AddFactor[Protocol[pd], varname, ival];
570  If[OptionValue[KeepVariable]//Not,
      newpd = RemoveVariable[newpd, varname],
      newpd
    ]; (* delete variable varname if option KeepOriginal->False was set *)
      Return[newpd]
575 ];
580 Variable2Factor[Protocol[pd_List], var_, OptionsPattern[Variable2Factor]]:=Module[{},
Return[Variable2Factor[Protocol[pd], var, 0, KeepVariable->OptionValue[KeepVariable], Method->
  OptionValue[Method], DistanceFunction->OptionValue[DistanceFunction], FactorNames->OptionValue[FactorNames]]]];
585 Variable2Factor::grouperror="Too few factor names given";
Variable2Factor[Protocol[pd_List], var_, n_Integer, OptionsPattern[Variable2Factor]] := Module[{values, varname, newpd, tempvals, factorvals, intervals, factornames, reps, i, j, fun, ind},
  If[var//IntegerQ, varname=VariableIndex2Name[Protocol[pd], var], varname=var];
  values = GetVariableValues[Protocol[pd], var];
  tempvals = If[n!=0, FindClusters[values, n, Method->OptionValue[Method], DistanceFunction->
    OptionValue[DistanceFunction]], FindClusters[values, Method->OptionValue[Method], DistanceFunction->OptionValue[DistanceFunction]]];
  ]; (* Sandra *)
  intervals = {Min[#], Max[#]}& /@ tempvals;
590 If[OptionValue[FactorNames]//ListQ//Not,
  factornames="Cluster ["<>ToString[#[[1]]]<>", "<>ToString[#[[2]]]<>"] "&/@intervals;
  fun[x_, y_]:= (ToString[0]<>ToString[#])& /@ Join[x, Range[10^(y-1), 10^y-1]]; (* Sandra *)
  ind=Join[Fold[fun, {}, # // N // Log10 // Floor // Range], ToString /@ Range[10^(# // N //
    Log10 // Floor), #]&[n]]; (* Sandra *)
  factornames = MapThread[StringJoin[ind[[#1]], #2]&, {Ordering[Ordering[intervals]], factornames}], 
  factornames }];
595 If[(OptionValue[FactorNames]//Length) < Length[tempvals],
  Message[Variable2Factor::grouperror];
  Abort[],
  factornames = OptionValue[FactorNames]
];
600 factornames = OptionValue[FactorNames];
reps={};
605 Do[
  reps = Append[reps, tempvals[[i,j]]->factornames[[i]]],
  {i, 1, Length[tempvals]}, {j, 1, Length[tempvals[[i]]]}];
610 factornames = values/.reps;
newpd = AddFactor[Protocol[pd], varname, factornames];
615 If[OptionValue[KeepVariable]//Not,
  newpd = RemoveVariable[newpd, varname], (* delete variable varname if option KeepOriginal->False
  was set *)
  newpd (*not necessary -Matz*)
];
620 ];
625 (** Recode, by Joakim 2010-03-29 Version 2 **)
Recode::Factor = "Factor error";
Recode::ReplacementList = "Error in replacement list";
Recode[Protocol[pd_List], facnum_Integer, name_String, repl_List, mods_List, OptionsPattern[Recode]

```

```

    ]] :=

630  Module[{ newpd, ComplementList, overwritefactor },
        (*Check if the factor number given as input is a valid number for a factor, else abort!*)
        If[facnum > Length[pd[[FACTORS]]], Message[Recode::Factor]; Abort[]];

635  (*Ensures that the replacement list is given on the appropriate form*)
ComplementList = Length[Complement[Range[1, Max[Flatten[repl]]], Flatten[repl]]];

If[ComplementList > 0 || Length[Union[Flatten[repl]]] != Length[pd[[FACTORS, facnum, 2]]],
   Message[Recode::ReplacementList]; Abort[]];

640  overwritefactor = OptionValue[OverwriteFactor];
newpd = pd;

If[overwritefactor,
  newpd[[FACTORS]][[facnum]] = {name, mods, Position[repl, #][[1, 1]] & /@ newpd[[FACTORS,
    facnum, 3]]];
  Return[Protocol[newpd]];
  , newpd[[FACTORS]] = Append[newpd[[FACTORS]], {name, mods, Position[repl, #][[1, 1]] & /@
    newpd[[FACTORS, facnum, 3]]}];
  Return[Protocol[newpd]];
]
];

650

```

---

```

(** SubProtocol, by Matz 2010-05-01 ver 0.75 **)
(* Sverker added text as third argument 2010-03-07 ver 0.55 *)
655 (* SubProtocol, by Matz 2010-04-08 ver 0.70 *)
(*News
 * fixa s\[ARing] den kan ta ett heltalet ist \[ADoubleDot] llet f\[ODoubleDot]r en lista i Subprotocol
 [p, fac, mod] [31/5]
 *)
(* Todo/Issues
660 * Fix indexing tests for functions [29/3]
*)
SubProtocol::emptyresulterror="Protocol is empty.";
SubProtocol::novarsref="There are no variables in the protocol but are referenced in the function
argument ('').";
SubProtocol::factornameerror="The factor name is of the wrong type, should be a String or Integer.
";
665 SubProtocol::variablenameerror="The variable name is of the wrong type, should be a String or
Integer.";
SubProtocol::indexError="Indexing is out of bounds.";
SubProtocol::modstypeError="The list of modality indices should be a list of Integers.";

(*fixa bugg, den tar inte tv\[ARing] heltalet som argument*)
670 SubProtocol[Protocol[pd_List], fac_, mods_List, OptionsPattern[SubProtocol]]:= Module[{ finDEX },
  Which[StringQ[fac], finDEX=FactorName2Index[Protocol[pd], fac],
  IntegerQ[fac], finDEX=fac; FactorIndex2Name[Protocol[pd], fac],
  True, Message[SubProtocol::factornameerror]; Abort[]];
];
675
If[!ListIntegersQ[mods], Message[SubProtocol::modstypeError]; Abort[]];
Return[SubProtocol[Protocol[pd], MemberQ[mods, #1[[finDEX]]]&, KeepEmptyModalities->OptionValue[
  KeepEmptyModalities]]];
];

680 (*must check if variables are present*)
SubProtocol[Protocol[pd_List], var_, fun_Function, OptionsPattern[SubProtocol]]:= Module[{ vINDEX },
  Which[StringQ[var], vINDEX=VariableName2Index[Protocol[pd], var],
  IntegerQ[var], vINDEX=var; VariableIndex2Name[Protocol[pd], var],
  True, Message[SubProtocol::variablenameerror]; Abort[]];
];
685
Return[SubProtocol[Protocol[pd], fun /@ (#2[[vINDEX]] & ), KeepEmptyModalities->OptionValue[
  KeepEmptyModalities]]];
];

690 SubProtocol[Protocol[pd_List], indices_List, OptionsPattern[SubProtocol]]:= Module[{}],
  Return[SubProtocol[Protocol[pd], MemberQ[indices, #4]&, KeepEmptyModalities->OptionValue[
  KeepEmptyModalities]]]; (*rewrote*)
];
695 SubProtocol[Protocol[pd_List], fun_Function, OptionsPattern[SubProtocol]]:=
Module[{ fl, vl, tl, d, indices, keepmods, data, len, numfacS, i, newpd, mods, ids, reprules,
  novars=False, vars},
  newpd = Protocol[pd];
  vars = pd[[VARIABLES, All, 2]];
  novars = vars=={};(*If no variables then we create a dummy and later won't copy the variable

```

```

    data*)

If[ novars && StringPosition[Tostring[fun], "#2"]!={} , Message[SubProtocol:: novarsref , fun]; Abort
    []]; (*catch if no variables are present and still referenced in function*)
    fl= Transpose[pd[[FACTORS, All, 3]]];
    len=Length[fl];
705
(*vl is either a dummy or the real values from VARIABLES*)
If[ novars, vl=ConstantArray[0,len],
    vl=Transpose[ pd[[VARIABLES, All, 2]] ]
];
710
t1= pd[[TEXTDATA]];
numfac = Length[pd[[FACTORS]]];
d = {fl[[#]], vl[[#]], t1[[#]], #} & /@ Range[len];
indices = Select[d, fun @@ #[[1;;4]] &][[All, 4]];
715
If[ indices=={}, Message[SubProtocol:: emptyresulterror]; Abort[] ]; (*necessary grounds to
abort evaluation?*)

newpd[[1, TEXTDATA]] = pd[[TEXTDATA, indices]];
newpd[[1, FACTORS]] = {#[[1]], #[[2]], #[[3, indices]]} & /@ pd[[FACTORS]];
(*if no variables in original, then don't copy*)
720
If[Not[novars], newpd[[1, VARIABLES]] = {#[[1]], #[[2, indices]]} & /@ pd[[VARIABLES]] ];
keepmods = OptionValue[KeepEmptyModalities];

725 If[ Not[keepmods],
    Do[
        data = newpd[[1, FACTORS, i, 3]];
        mods = Union[data];
        reprules = Array[mods[[#]] -> # &, Length[mods]]; (*Replacing the indices*)
        ids = data /. reprules;
        newpd[[1, FACTORS, i, 2]] = pd[[FACTORS, i, 2]][[mods]];
        newpd[[1, FACTORS, i, 3]] = ids; (*The new indices to the new modalities*)
        ,{i, numfac}];
];
735
Return[newpd];
];

740


---


(** Group, by Oskar version 1/5~10 **)
(* Group, by Oskar version 2010-04-08 *)
Group[Protocol[pd_List], f_Integer, OptionsPattern[Group]]:= Module[{txts, txtids, newtxts, plength,
    flength, fname, fmods, newp, kf, kv, funcs, keepfac, keepvars, modlist, varlist, newvarlist, newmodlist, g
    },
745 fmods = GetModalityValues[Protocol[pd], f];
plength = Length[pd[[TEXTDATA]]];
flength = Length[fmods];
fname = pd[[FACTORS, f, 1]];
txts = GetText[Protocol[pd], #]& /@ Range[plength]; (* Det r\[ADoubleDot]cker att skriva <<txts =
    GetText[ Protocol[pMotioner]];>> -Matz [24/5]*)
750 txtids = GetModalityIds[Protocol[pd], f];
txtids = Array[Position[txtids, #]&, flength];
txtids = Flatten/@txtids;
newtxts = txts[[#]]& /@ txtids;

(*Construct grouped protocol*)
755 newp = CreateProtocol[newtxts];
newp = AddFactor[newp, fname, fmods];

(*Add factors*)
760 kf = OptionValue[KeepFactors];
g = GetModalityValues[Protocol[pd], #1][[GetModalityIds[Protocol[pd], #1]]]&;
funcs = kf[[#,1]]& /@ Range[Length[kf]];
keepfac= kf[[#,2]]& /@ Range[Length[kf]];
765 modlist=Map[GetModalityValues[Protocol[pd], #][[GetModalityIds[Protocol[pd], #]]]&, keepfac, {2}];
modlist =Table[Table[g[[keepfac[[j,i]]]]][[#]]& /@ txtids, {i, Length[keepfac[[j]]]}], {j, Length[keepfac]}];
newmodlist=Array[Map[funcs[[#]], modlist[[#]], {2}]&, Length[funcs]];
keepfac = Flatten[keepfac];
newmodlist = Flatten[newmodlist];
newmodlist = Partition[newmodlist, flength];
770 newp = Fold[AddFactor[#1, pd[[FACTORS, keepfac[[#2]], 1]], newmodlist[[#2]]]&, newp, Range[Length[keepfac]]];

(*Add variables*)
kv = OptionValue[KeepVariables];
g = GetVariableValues[Protocol[pd], #1]&;
775 funcs = kv[[#,1]]& /@ Range[Length[kv]];

```

```

keepvars=kv[[#,2]]&/@Range[Length[kv]];
varlist=Map[GetVariableValues[Protocol[pd],#]&,keepvars,{2}];
varlist =Table[Table[g[keepvars[[j,i]]][[#]]& /@ txtids,{i,Length[keepvars[[j]]]}],{j,Length[keepvars]}];
780 newvarlist=Array[Map[func[[#]],varlist[[#]],{2}]&,Length[func]];
keepvars = Flatten[keepvars];
newvarlist = Flatten[newvarlist];
newvarlist = Partition[newvarlist,flength];
newp = Fold[AddVariable[#1,pd[[VARIABLES,keepvars[[#2]],1]],newvarlist[[#2]]]&,newp,Range[Length[keepvars]]];
newp[[1,TEXTDATA]]= First /@ newp[[1,TEXTDATA]];(*buggfix av Tobias [24/5]*)
785 Return[newp];
];

790 (* ::Section:: *)
(*Listing - Misc. functions used to display data:*)
795



---


800 (** Factors, by Sverker 2010-01-19 **)
Factors[Protocol[pd_List],OptionsPattern[Factors]]:=Module[{h,tf,a},
h=OptionValue[Header];(* Construct the header*)
800 tf=OptionValue[TableForm];(* TableForm? *)
a=Array[{#1, pd[[FACTORS, #1, 1]], Length[pd[[FACTORS, #1, 2]]]}&, Length[pd[[FACTORS]]]];
If[tf,
805 Prepend[a, Style[#, Bold]& /@ h]//TableForm, (* tf was true *)
a];
(* tf was false *)
];

810 (** Variables, by Sverker 2010-01-19 **)
Variables[Protocol[pd_List],OptionsPattern[Variables]]:=Module[{h,tf,a},
h=OptionValue[Header];
810 tf=OptionValue[TableForm];
a=Array[
{
#1, (* id *)
pd[[VARIABLES, #1, 1]], (* names *)
Min[pd[[VARIABLES, #1, 2]]], (* min *)
Max[pd[[VARIABLES, #1, 2]]], (* max *)
Median[pd[[VARIABLES, #1, 2]]]/N[#, 2]&, (* median *)
Mean[pd[[VARIABLES, #1, 2]]]/N[#, 2]&, (* mean of data *)
815 StandardDeviation[pd[[VARIABLES, #1, 2]]]/N[#, 2]& (* standard deviation of data *)
}&,
Length[pd[[VARIABLES]]];
];

820 (* Return with or without header and formatting *)
If[tf,
825 Prepend[a,Style[#,Bold]&/@h]//TableForm, (* tf was true *)
a];
(* tf was false *)
];

830 Protect[Variables];
835



---


840 (** Modalities, by Sverker 2010-01-19 **)
Modalities[Protocol[pd_List], f_Integer, OptionsPattern[Modalities]] := Module[{h, tf, a, s, trunc,
name, mods, data},
h=OptionValue[Header];
840 tf=OptionValue[TableForm];
s=OptionValue[SortByFrequencies];
trunc=OptionValue[TruncateNames];
{name, mods, data}=pd[[FACTORS, f]];

(* Truncate names of modalities if trunc is true *)
850 If[trunc, mods=StringTake[#, Min[60, StringLength[#]]]&/@mods];
(* Construct the array with information*)
a=Array[

```

```

855 {#1,
  mods[[#1]],
  Length[Cases[data, #1]],
  (Length[Cases[data, #1]]/Length[data])//N[#, 2]&
}&, Length[mods]];
860 (* Sort by frequencies if s is true *)
If[s, a=Sort[a, #1[[3]]>#2[[3]]&]];
If[tf, Prepend[a, Style[#, Bold]&/@h]//TableForm,
a]
865 ];


---


870 (** Info, by Sverker 2010-01-19 **)
Info[Protocol[pd_List], OptionsPattern[Info]] := Module[{h, tf, a},
h = OptionValue[Header];
tf = OptionValue[TableForm];
a = {{Length[pd[[TEXTDATA]]], Length[pd[[FACTORS]]], Length[pd[[VARIABLES]]]}};
875 If[tf, Prepend[a, Style[#, Bold]&/@h]//TableForm,
a]
];
880


---


885 (** ListData, by Sverker 2010-01-19 **)
ListData[Protocol[pd_List], fs_, vs_, OptionsPattern[ListData]]:=Module[{tf, trunc, facdata, vardata,
facheadings, varheadings, a, sb},
tf = OptionValue[TableForm];
trunc = OptionValue[TruncateNames];
sb = OptionValue[SortBy];
facdata = pd[[FACTORS, #, 2]][[pd[[FACTORS, #, 3]]]]&/@fs;
If[trunc, facdata = (StringTake#[, Min[30, StringLength#[]]] & /@#) & /@facdata];
facheadings = pd[[FACTORS, fs, 1]];
vardata = pd[[VARIABLES, #, 2]]&/@vs;
varheadings = pd[[VARIABLES, vs, 1]];
a = Transpose[Join[{Range[1, Length[pd[[TEXTDATA]]]]}, facdata, vardata]];
If[IntegerQ[sb],
If[sb>0,
  a = SortBy[a, -#[[1 + Length[fs] + sb]]&],
  a = SortBy[a, #[[1 + Length[fs] - sb]]&]
]
];
895
If[tf,
  Prepend[a, Style[#, Bold] & /@Join[{ "Id"}, facheadings, varheadings]]//TableForm,
a]
];
900
905 (* ::Section:: *)
(* Retrieval – in comparison to listing function is not primarily used to display data *)
910


---


915 (** GetNumberOfWords, by Matz **)
GetNumberOfWords[Protocol[pd_List]] := GetNumberOfWords[Protocol[pd], Range@Length@pd[[TEXTDATA]],
" "];
GetNumberOfWords[Protocol[pd_List], list_] := GetNumberOfWords[Protocol[pd], list, " "];
915 GetNumberOfWords[Protocol[pd_List], sep_String] := GetNumberOfWords[Protocol[pd], Range@Length@pd[[TEXTDATA]], sep];
GetNumberOfWords[Protocol[pd_List], list_, sep_String] := Module[{list2=list},
If[ Not@ListQ[list], list2 = {list} ];(*integer to list*)
920 CheckIndexBounds[Protocol[pd], list2, TEXTDATA];
Return@ If[sep==",", StringLength /@ pd[[TEXTDATA, list2]], (*special case, on average ~140 times
faster*)
Length[StringSplit#[, sep]]& /@ pd[[TEXTDATA, list2]]];
];
925


---


930 (** GetModalities, by Tobias Raski 2010-01-26 **)
GetModalityValues[Protocol[pd_List], fac_] := Module[{facindex},

```

```

930 If[ fac // IntegerQ // Not, facindex = FactorName2Index[Protocol[pd], fac], facindex = fac ];
Return[pd[[FACTORS, facindex, 2]]];



---


935 (** ModalityIds, by Tobias Raski 2010-01-26 **)
GetModalityIds[Protocol[pd_List], fac_] := Module[{facindex},
If[ fac // IntegerQ // Not, facindex = FactorName2Index[Protocol[pd], fac], facindex = fac];
Return[pd[[FACTORS, facindex, 3]]];

940

(* GetVariableValues, by Tobias Raski 2010-01-26 *)
(*Matz added checking of index and type with messages BETA, 2010-03-12*)
(*Matz added function call to CheckIndexBounds BETA, 2010-05-24*)
945 GetVariableValues::usage="GetVariable[p,k] returns data for variable k (Name or Index).";
GetVariableValues::outofbounderror="Variable index `` is out of bounds, the number of variables
in the protocol is `.`;
GetVariableValues::typeerror="Second argument is of the wrong type (should be String or Integer)."

GetVariableValues[Protocol[p_List], n_]:= Module[{maxInd, nInt=n},
950 ArgumentQ[{Protocol[p], n}, {ProtocolQ, {IntegerQ, StringQ}}];
If[StringQ@n, nInt = VariableName2Index[Protocol[p], n]];
CheckIndexBounds[Protocol[p], {nInt}, VARIABLES];
Return[ p[[VARIABLES, nInt, 2]] ];
955 ]
(*
If[1<= nInt<= maxInd,
Return[ p[[VARIABLES, nInt, 2]] ],(* if Integer*)
Message[GetVariableValues::outofbounderror, nInt, maxInd]; Abort[]
960 ];
*)



---


965 (** GetText, by Sverker **)
(*GetText 30/4, by Matz [resurrected and updated with range tests and alternative list argument]*)
(*GetText 17/5, added so GetText can fetch all texts*)
GetText::OutOfBoundsError="` is out of bounds, the index should be between 1 and the number of
individuals (`), inclusive.";
GetText::TypeError="` of the wrong type, it should be an Integer or a list of Integers.";
970 GetText[Protocol[pd_List]] := GetText[Protocol[pd], Range@Length@pd[[TEXTDATA]]];
GetText[Protocol[pd_List], n_]:= Module[{len},
len=Length@pd[[TEXTDATA]];
975 If[Not[IntegerQ@n || And@@ListIntegersQ@n], Message[GetText::TypeError,"The second argument"];
Abort[]];
If[IntegerQ@n, (*replace with CheckIndexRange*)
If[1<= n<= len,
Return@pd[[TEXTDATA, n]],
Message[GetText::OutOfBoundsError,"The index", len]; Abort[]
980 ];
];
If[ListQ@n,
If[And@@ListIntegersQ@n,
If[1<= Min@n<=Max@n<=len,
Return@pd[[TEXTDATA, n]],
Message[GetText::OutOfBoundsError, "At least one of the indices in the list n", len];
Abort[]
985 ];
];
];
990 ];
];
];



---


995

(** Name2Index, by Matz 23/5, last updated 6/5**)
Name2Index::NameError="The `` was not found.";
Name2Index::NotSetError="Several `` matched.";
Name2Index::OptionFromError="The third argument is `, use any of \"Modality\", \"Factor\" or \
Variable\".>(* fixa s\[ARing] felmeddelandet inte h\[ADoubleDot]nvisar till "from")"
1000 (*should take both factor and modality, assumes each factor and modality is uniquely named*)
Name2Index[Protocol[pd_List], name_String, fr_String]:=Module[{fac, vars, mods, data, index, lenIndex
, s2n, dataindex, plurals, from=ToLowerCase@fr, fvm},

```

```

fvm={"factor", "variable", "modality"};

1005 If[from=="", Message[Name2Index::OptionMethodError]; Abort[]];
If[And@@(#!=from&!/fvm), Message[Name2Index::OptionFromError, from]; Abort[]];
fac = pd[[FACTORS, All, 1]];
vars = pd[[VARIABLES, All, 1]];
mods = pd[[FACTORS, All, 2]];
1010 data = from /. Thread@Rule[fvm, {fac, vars, mods}];
s2n = Thread@Rule[fvm, Range@3]; (*string2name*)
plurals = {"factors", "variables", "modalities"}; (*used in messages*)
dataindex = from /. s2n;
index = Flatten[Position[ToLowerCase@data, ToLowerCase@name]]; (*case insensitive [2/5^10]*)
lenIndex = Length[index];
(*take care of modality which always give 2 elements if found*)

1015 Which[lenIndex==0, Message[Name2Index::NameError, from, name]; Abort[],
           dataindex==3 && lenIndex!=2, Message[Name2Index::NotSetError, plurals[[dataindex]]], (*can be
               many modalities, fix*)
1020           dataindex==1 && lenIndex !=1, Message[Name2Index::NotSetError, plurals[[dataindex]]]
];
If[dataindex==3, Return@index]; (*we want a 2-list*)

1025 Return@index [[1]];
]



---


1030
(*Index2Name, by Matz 23/5, last updated 6/5*)
Index2Name::OutOfBoundsError="Index `` is outside the bounds of number of `` (` `).";
Index2Name::NoDataError="There are no `` in the protocol.";
Index2Name::LengthError="The `` need `` argument(s) {f,m}, f or v for factor modality and variable
respectively.";

1035 Index2Name::TargetError= "The target does not exist, choose Factor, Variable or Modality.";

Index2Name[Protocol[pd_List], i_, t_String] := Module[{fac = {}, vars = {}, mods = {}, data, dim, s2n,
           dataindex, data2, index, plurals, to=ToLowerCase@t},
           (*to convert to modality name we need index for factor and modality*)

1040 s2n={"factor"->1,"variable"->2,"modality"->3,->0}; (*string2name*)
plurals={"factors","variables", "modalities"}; (*used for messages*)
dataindex=to /. s2n;
If[Not[1<= dataindex <=3],Message[Index2Name::TargetError];Abort[]];
1045 index=If[Not[ListQ[i]],{i},i];

Which[ (dataindex==1 || dataindex==2) && Length[index]!=1,Message[Index2Name::LengthError,
           plurals[[dataindex]],1];Abort[],
           dataindex==3 && Length[index]!= 2,Message[Index2Name::LengthError, plurals[[dataindex]],2];Abort
];
1050
If[dataindex==1 && Length[pd[[FACTORS]]]==0 || dataindex==2 && Length[pd[[VARIABLES]]]==0 ||
           dataindex==3 && Length[pd[[FACTORS]]]==0,
           Message[Index2Name::NoDataError, plurals[[dataindex]]]; Abort[]
];
1055 Which[ dataindex==1||dataindex==3,
           mods=pd[[FACTORS, All, 2]];fac= pd[[FACTORS, All, 1]],
           dataindex==2,
           vars= pd[[VARIABLES, All, 1]]
];
1060 data = {fac, vars, mods};(*e.g. if no variables, hence all are initialized with {}*)
data2 = data[[dataindex]]; (*the data, facs, vars or mods*)
1065 (*two indices for modality, hence pick the first one, otherwise do nothing*)
If[ dataindex==3, data2 = Extract[data2, index[[1]]]];
dim = Length[data2];(*first element for modalities*)
(*check factor out of bounds also*)

1070 If[dim<index[[-1]] || index[[-1]]<1, Message[Index2Name::OutOfBoundsError, index[[-1]], plurals
           [[dataindex]], dim]; Abort[]];
Return[Extract[data2, index[[-1]]]];
]

1075 FactorName2Index[Protocol[pd_List, name_String]] := Name2Index[Protocol[pd], name, "Factor"];
VariableName2Index[Protocol[pd_List, name_String]] := Name2Index[Protocol[pd], name, "Variable"
];

```

```

ModalityName2Index[Protocol[pd_List], name_String] := Name2Index[Protocol[pd], name, "Modality"]
];

1080 FactorIndex2Name[Protocol[pd_List], index_Integer] := Index2Name[Protocol[pd], index, "Factor"];
VariableIndex2Name[Protocol[pd_List], index_Integer] := Index2Name[Protocol[pd], index, "Variable"]
];
ModalityIndex2name[Protocol[pd_List], index_List] := Index2Name[Protocol[pd], index, "Modality"]

1085 (* ::Section:: *)
(* Deprecated (will be replaced/removed):*)



---


1090 (** CheckModality, and variable functions, by Matz 2010-02-28 **)
(*Will probably be replaced with a generic check function later [1/3]*)  

CheckModality::usage="CheckModality[p, fc, mod] Aborts evaluation if the modality does not exist."  

CheckModality::outofbounderror="Modality index `` is outside the bounds of factor ` `."  

CheckModality[Protocol[pd_List], fac_Integer, mod_Integer]:= Module[{facname, dim},
1095 facname = FactorIndex2Name[Protocol[pd], fac]; (*catch error here first*)
dim = Length[pd[[FACTORS, fac, 2]]];
Which[mod<1 || mod>dim, Message[CheckModality::outofbounderror, mod, facname]; Abort[]]
];

1100 (* ::Section:: *)
(*Hidden functions:*)



---


1105 (**CheckVariable, by Matz**)
(*To catch the empty variables error in SubProtocol, amongst other uses*)
CheckVariable::outofbounderror="Variable `` is outside the bounds of p."
CheckVariable::novars="p contains no variables."  

CheckVariable[Protocol[pd_List], var_Integer]:= Module[{varname, dim},
dim = Length[pd[[VARIABLES]]];
If[dim==0, Message[CheckVariable::novars]; Abort[]];
varname = VariableIndex2Name[Protocol[pd], var]; (*using Index2Variable to get error message*)
];
1110

(* ::Section:: *)
(*Hidden functions:*)



---


1115 (*By Matz, used by ProtocolQ and GetText, updated with ListNumbersQ and reverted from ArrayQ 23/5
*)
ListIntegersQ[list_List] := IntegerQ /@ list;
ListsIntegersQ[list_List] := ListIntegersQ /@ list;
ListStringsQ[list_List] := StringQ /@ list;
1120 ListsStringsQ[list_List] := ListStringsQ /@ list;
ListNumbersQ[list_List] := NumberQ /@ list;
ListsNumbersQ[list_List] := ListNumbersQ /@ list;



---


1125 (**ProtocolQMess, by Matz, used by VersionProtocolQ to abbreviate the messaging**)
ProtocolQMess[error_String, tagged_]:= Module[{}],
If[tagged, Return[{False, error}], Return[False]];
1130 Abort[];
]



---


1135 (**ArgumentQ, by Matz version 0.1, 24/5 ~10**)
(*
This function is used internally to check that the arguments types are correct.
The function takes two arguments, the function call list of arguments argc and the "pattern" to
match (the Q-function), argp.

1140 If a function is defined as taking an integer and an integer OR and integer and a string, then the
pattern will be:
{IntegerQ, {IntegerQ, StringQ}} where the alternative types are enclosed in a list.

You cannot define functions in the list so a solution for more complex type patterns is to define
your own:

1145 ArrayOfStringsQ[list_List]:= ArrayQ[list ,_, StringQ];
to be used temporarily and then call: {ArrayOfStringsQ}.

This function could be used with ProtocolQ where ProtocolQ will also return a detailed message of
the error, ArgumentQ

```

1150 will also display a message of which argument that was wrong.

For the function to work best don't use \_List etc. just let the function catch the error. This way of programming is however not possible for overshadowed functions in where you have to catch the error in another way...

```
1155 *)
(* Options[ArgumentQ]:= {SelfCall->1}; (*to allow recursive calls*)*)

ArgumentQ::ArgumentError="The argument `` `` `` of the wrong type.";
ArgumentQ[argc_, argp_, OptionsPattern[ArgumentQ]] := Module[{flags={}},
1160 (*
(*check list of available Q-functions are in the list , depth should not be >2 etc.*)
If[OptionValue[RecursiveDepth]>0,(*checking arguments of the function itself*)
  ArgumentQ[{argc ,argp },{ListQ ,ListQ },RecursiveDepth->Optionvalue[RecursiveDepth]-1];
];*)
1165 (*the types of this function is NOT checked, as an internal function this should be no problem*)
Do[
  flags = {flags , If[Length@argp [[ i]]>0,Or@@(#@argc [[ i]]&/@argp [[ i]]) ,argp [[ i]] @argc [[ i]] ]};
  ,{i, 1, Length@argc }];
1170 errorp = Position[flags//Flatten ,False]//Flatten;
numeris[a_Integer]:= If[a==1, "is", "are"];
pluralis[a_Integer]:= If[a==1, "", "s"];
If[Length@errorp >0, Message[ArgumentQ::ArgumentError, pluralis[Length@errorp], errorp, numeris[Length@errorp]]; Abort[]];
1175 ]
(* argument type message...
Do[
  argp [[ i]]=StringDrop[ If[Length@argp [[ i]]==0,ToString@argp [[ i]], ToString /@ argp [[ i]]], -1];
  ,{i,1,Length@argp }];*)
1180
```

---

```
1185 (**CheckIndexBounds, by Matz 2/5**)

1190 The function tests the range of a type of data, such as FACTORS, VARIABLES, MODALITIES or TEXTDATA
, it aborts execution and takes care of
the errormessage.

Example:
1195 Protocol p has 10 individuals and we want to index {1,3,11} in the texts.
IndicesInRangeQ[p, {1,3,11}, TEXTDATA]
*)

CheckIndexBounds::usage="CheckIndexBounds[p, list, i] checks if the indices given in list is
within range of type i. Available types are VARIABLES, FACTORS and TEXTDATA.";
1200 CheckIndexBounds::OutOfBoundsError="Tried to index outside the range of ``(``).";
CheckIndexBounds::IndexTypeError="The indices in the second argument is not of the type (Integer).
";
CheckIndexBounds::NoElementsError="There are no `` in the protocol.";

CheckIndexBounds[Protocol[pd_List], ids_List, type_Integer]:= Module[{min,max,len,typename},
1205 (*ArrayOfIntegersQ[a_List]:=ArrayQ[a, _, IntegerQ];
ArgumentQ[{_,ids, type}, {_,,ArrayOfIntegersQ, IntegerQ}];*) (*the arguments should be correct*)

len=Length@pd[[ type]];
typename = {FACTORS->"factors", VARIABLES->"variables", TEXTDATA->"texts"};
1210 If[ len == 0, Message[CheckIndexBounds::NoElementsError, type/.typename]; Abort[] ];
If[Not@And@@ListIntegersQ@ids,
  Message[CheckIndexBounds::IndexTypeError]; Abort[]
];
1215 (*No tests of the type of the indices since this function is only used inside of this package...
*)
{min, max} = #@@ids& /@ {Min,Max}; (*filter the integers and pick min and max to test the range
*)
1220 If[Not[1<= min <= max <= len],(*early "hit test")
  Message[CheckIndexBounds::OutOfBoundsError, type /. typename, len]; Abort[]
];
]
(*Flatten@MapIndexed[ If[Not[1 <= # <= len], #2, {}]&, ids]*)
```

1225

End[]

EndPackage[]

## Linguistics

```
(* :: Package:: *)  
5  
(*  
Matz fixed some errors: addid and viewprogress is no longer used  
Usage of splittext was updated (removed boxes)  
Usage of LSA was updated  
10 added tags for "kodbilaga"  
    [7/6^10]  
*)  
  
BeginPackage["Linguistics`", {"RCA`"}]  
15 (* Exports from package*)  
  
TransformTexts::usage =  
"TransformTexts[p,fun] applies the text-transforming  
20 function fun on the textdata in protocol p.";  
RemoveTags::usage = "";  
DocumentVectors::usage = "";  
GetFilteredWords::usage = "";  
  
25 AddHitCountVariable::usage =  
"AddHitCountVariable[p,pat] adds a variable based  
on StringCount[txt,pat] to p.";  
  
AddCollocationVariables::usage =  
30 "AddCollocationVariables[pd, word/regex] adds words  
co-occurring with word/regex as hit count variables.  
See Options[AddCollocationVariables] for options."  
  
AddPOSVariables::usage =  
35 "AddPOSVariables[p] counts occurrences of parts-of-speech  
in p according to Options[AddPOSVariables]. AddPOSVariables  
uses the external program hunpos which must be installed for  
this function to work."  
  
40 TagProtocol::usage = "TagProtocol[pd, model] tags all the individual  
texts in the protocol pd with the HunPos tagger by means of the of  
the given model. See Options[TagProtocol] for options.  
See http://code.google.com/p/hunpos/ for information about HunPos."  
  
45 StemProtocol::usage =  
"StemsProtocol[pd] takes a protocol and stems the individuals in  
it using predefined rules for swedish."  
  
AddStandardVariables::usage =  
50 "AddStandardVariable[pd] adds predefined standard variables  
to the given protocol pd. See Options[AddStandardVariables]"  
  
AddTermDiscriminantVariables::usage =  
"AddTermDiscriminantVariables[pd] adds discriminating terms from  
55 the protocol as hitcount variables."  
  
StemHitCountVariables::usage =  
"StemHitCountVariables[pd] Stems the patterns used as hitcount variables in the given protocol pd."  
  
60 SplitText::usage =  
"SplitText[p, mean] returns a protocol with the splitted texts in p as new individuals.  
The text lengths (counted in strings separated with whitespace) is calculated to all be as close  
to mean (Integer) as possible.  
According to the Central Limit Theorem the statistical mean of the splitted texts will converge to  
the given mean provided a sufficient number of texts was splitted.  
A Factor \"Split ID\" is appended by default, it contains the indices to the original individuals.  
See Options[SplitText] for options."  
65 KeepFactors::usage = "An Option for SplitText. Lets you specify which factors you want to keep  
after the splitting, e.g. \"All\", \"None\" or a list of indices";  
KeepVariables::usage = "An Option for SplitText. Lets you specify which variables you want to keep  
after the splitting, e.g. \"All\", \"None\" or a list of indices";
```

```

SplitSeparator::usage = "An Option for SplitText. The string which is separating the elements in
an individual, for words \" \\" (Default).";
SplitID::usage = "An Option for SplitText. If the option is set to True it adds a factor \"Split
ID\" of integers where each split points to the pre-split ID.";
70 TextID::usage = "An Option for SplitText. If the options is set to True it adds a factor \"Text ID
\" of integers where each text ID points to the part of a text it has split.";
(*TextID might need a better explanation*)

75 CleanProtocol::usage =
"CleanProtocol[p] removes nonletter character from protocol text.
See Options[CleanProtocol] for options."
LSA::usage = "LSA[p] computes the \"Latent Semantic Analysis\" of protocol p and returns protocols
{p,wp} containing the document protocol and word protocol respectively.
80 See Options[LSA] for options.";

Name::usage =
"An option for AddHitCountVariable. Used if you want to name your variable
85 in another way then the predefined."

RelativeCount::usage =
"An option for AddHitCountVariable. Used if you don't want your hitcounts
expressed as absolute values."
90 RelativeCountFactor::usage =
"An option for AddHitCountVariable. By default a relative count factor of
1000 is multiplied to relative hitcount given when the option 'RelativeCount'
is set to true. To change the size of this factor use this option."
95 MinimumWindowSize::usage =
"An option for AddCollocationVariables. By default a word will be considered
uninteresting if it's less then 3 characters long. To lower or higher this use
this cut off use this option."
100 MinimumFrequency::usage =
"An option for AddCollocationVariables. By default a word will be considered
uninteresting if it's got a total number of occurrences in the whole protocol
less than 5. To change this use this option."
105 SlidingWindowSize::usage =
"An option for AddCollocationVariables. By default a word will be counted as
within the 'collocation' of another word if it's no more then five words away
from it. To change this value use this option."
110 Model::usage =
"An option for either AddPOSVariables or TagProtocol. By default these
functions uses a swedish model file when creating/counting part of speech tags.
To change the model use this option."
115 Tags::usage =
"An option for AddPOSVariables. The Linguistics package uses a swedish model file,
named model-swedish-suc2, when communicating with HunPos. Because of this the default
tags for AddPOSVariables are patterns matching a subset of the tags available in
120 model-swedish-suc2. If other model files are used this option should be used
to get correct behaviour."
WordGroupingNumber::usage =
"An option for TagProtocol. Depending on the available stack size for the external
125 program HunPos the program can handle a different number of words at a time.

If you have a small stack size you might need to lower the WordGroupingNumber
for the function to work. If you on the other hand have a large available stack size
you can higher it to get a faster performing function."
130 AddNumberOfWordsVariable::usage =
"An option for AddStandardVariables. Adds a variable measuring the number of words."
AddNumberOfSentencesVariable::usage =
135 "An option for AddStandardVariables. Adds a variable measuring number of sentences."
AddAverageWordLengthVariable::usage =
"An option for AddStandardVariables. Adds a variable measuring average word length."
140 AddAverageSentenceLengthVariable::usage =
"An option for AddStandardVariables. Adds a variable measuring average sentence length."
AddWordVariationVariable::usage =
"An option for AddStandardVariables. Adds a variable measuring word variation."
145

```

```

NumberOfVariables::usage =
"An option for AddTermDiscriminationVariables. Simply defines how many of the
150 highest ranked terms to add as hitcount variables."

RemoveDigits::usage =
"An option for CleanProtocol. Defines if you want digits to be kept in the
texts or not."
155 RemovePunctuation::usage =
"An option for CleanProtocol. Defines if you want punctuation characters ('.' and ',')
to be kept in the texts or not."

160 KeepFactors::usage =
"An Option for SplitText. Lets you specify which factors you want to
keep after the splitting, e.g. \\"All\\", or a list of indices";

KeepVariables::usage =
165 "An Option for SplitText. Lets you specify which variables you want to
keep after the splitting, e.g. \\"All\\", \\"None\\" or a list of indices";

SplitSeparator::usage =
170 "An Option for SplitText. The string which is separating the elements in an
individual, for words \" \\\" (Default).";

MinimumFrequency::usage =
"An Option for LSA. The minimum frequency of words used in the analysis.";
175 MinimumWords::usage =
"An Option for LSA. The minimum number of words used in the analysis.';

MaximumWords::usage =
180 "An Option for LSA. The maximum number of words used in the analysis.';

MaximumPairs::usage =
"An Option for LSA. The maximum number of pair of words used in the analysis.';

185 MinimumPairFrequency::usage =
"An Option for LSA. The minium frequency of pair of words used in the analysis.';

LSAVariables::usage =
"An Option for LSA. The number of variables returned.";
190 UseStopWords::usage =
"An option for CleanProtocol. Specifies a list of words to be removed.';

UseImportStopList::usage = "Option for AddTermDiscriminationVariable";
195 WordFrequencyThreshold::usage = "Option for AddTermDiscriminationVariable";
TfIdfWeighting::usage = "An option for AddTermDiscriminationValues";

200 Begin[\"'Private '\"] (* Begin Private Context *)

```

---

(\*

205 THE FOLLOWING FUNCTIONS CAN BE USED WHEN IMPORTING THE PACKAGE

\*)

210

---

```

(** GetStopWordList, **)
GetStopWordList[list_] :=
Module[{tries = 0, words},
215   If[FindFile[ToLowerCase[list]] == $Failed,
      Print["No stopword list found"];
      Abort[]];
   ];
   While[(words = Import[ToLowerCase[list]]) == $Failed || tries == 5, tries++];
220   words = words // Flatten // Union;
   Return[words]
]

(** TransformTexts, by Sverker 2010-01-19 **)
225 TransformTexts[Protocol[pd_], fun_Function] :=
Module[{newpd},
  newpd = pd;
  newpd[[TEXTDATA]] = fun/@newpd[[TEXTDATA]];
  Return[Protocol[newpd]]
230 ];

```

---

(\* Updated by Sverker 2010-03-19 \*)

---

```

235 (** AddHitCountVariable, by Sverker 2010-01-19 **)
Options[AddHitCountVariable] = {Name->Automatic, RelativeCount->False, RelativeCountFactor ->1000};
AddHitCountVariable[Protocol[pd_], patt_String, OptionsPattern[AddHitCountVariable]] :=
Module[{hc, name},
name = Switch[OptionValue[Name], Automatic, "HC: `~~`patt, _, OptionValue[Name]];
240 Switch[OptionValue[RelativeCount],
False, hc = StringCount[#, patt, IgnoreCase -> True]&/@pd[[TEXTDATA]],
True, hc = (N[StringCount[#, patt, IgnoreCase -> True]* OptionValue[RelativeCountFactor]/
StringLength[#]])&/@pd[[TEXTDATA]]];
Return[AddVariable[Protocol[pd], name, hc]]
];
245

```

---

```

250 (** AddCollocationVariables, by Tobias **)
Options[AddCollocationVariables] = {MinimumWordSize -> 3,
MinimumFrequency -> 5, SlidingWindowSize -> 5};
255 AddCollocationVariables[Protocol[pd_], word_, opts___?OptionQ] :=
Module[{newfactors, newpd, i},
newpd = Protocol[pd];
newfactors = {};
newfactors = SpanFrequency[newpd[[1,TEXTDATA]], word, opts];
For[i = 2, i <= Length[newfactors], i++,
newpd = AddHitCountVariable[newpd, " " <> newfactors[[i,1]] <> " "]];
Return[newpd];
];
260

```

---

```

265 (** AddPOSVariables, by Sverker **)
Options[AddPOSVariables] = Join[{Model->"model-swedish-suc2", Tags->
{{"_SPS", "Preposition"}, {"_AF", "Particip"}, {"_AQ", "Adjektiv"}, {"_NC", "Substantiv"}, {"_P", "Pronomen"}, {"_RG", "Adverb"}, {"_V", "Verb"}}, Options[AddHitCountVariable]];
270 AddPOSVariables[p-Protocol, OptionsPattern[AddPOSVariables]] :=
Module[{ptmp, txts, ftext},
If[(FilterRules[p[[1,METADATA]], "Tagged"] // Length) > 0,
275 ftext = RemoveTags/@p[[1,TEXTDATA]],
ftext = p[[1,TEXTDATA]]];
];
txts = ftext;
ptmp = TagProtocol[p, Model->OptionValue[Model]];
280 ptmp = Fold[AddHitCountVariable[#1, #2[[1]], Name -> #2[[2]], RelativeCount -> OptionValue[RelativeCount],
RelativeCountFactor -> OptionValue[RelativeCountFactor]]&, ptmp, OptionValue[Tags]];
ptmp[[1,TEXTDATA]] = txts;
Return[ptmp];
];
285

```

---

```

290 (** TagProtocol, by Tobias *)
Options[TagProtocol] = {WordGroupingNumber -> 75000, Model -> "model-swedish-suc2" };
TagProtocol[Protocol[pd_], OptionsPattern[TagProtocol]] :=
Module[{newpd, ftext},
If[(FilterRules[pd[[METADATA]], "Tagged"] // Length) > 0,
295 ftext = RemoveTags/@pd[[TEXTDATA]],
ftext = pd[[TEXTDATA]]];
];
newpd = pd;
Print["Please note that the HunPos tagger will be loaded and run several times depending on
you protocol size."];
newpd[[TEXTDATA]] = GroupAndTag[ftext, OptionValue[Model],
WordGroupingNumber -> OptionValue[WordGroupingNumber]];
300 newpd[[METADATA]] = Append[newpd[[METADATA]], "Tagged" -> True];
Return[Protocol[newpd]];
];

```

---

305 (\*\* AddStandardVariables, by Tobias \*\*)

Options[AddStandardVariables] = {

```

AddNumberOfWordsVariable -> True,
AddNumberOfSentencesVariable -> True,
310 AddAverageWordLengthVariable -> True,
AddAverageSentenceLengthVariable -> True,
AddWordVariationVariable -> True;
AddStandardVariables[Protocol[pd_], OptionsPattern[AddStandardVariables]] := 
Module[ {newpd_, ftext},
If[ (FilterRules[pd[[METADATA]], "Tagged"] // Length) > 0,
      ftext = RemoveTags /@ pd[[TEXTDATA]],
      ftext = pd[[TEXTDATA]]
];
newpd = Protocol[pd];
newpd[[1,TEXTDATA]] = ftext;
If[ OptionValue[AddNumberOfWordsVariable],
      newpd = newpd // NumberOfWordsVariable
];
If[ OptionValue[AddNumberOfSentencesVariable],
      newpd = newpd // NumberOfSentencesVariable
];
If[ OptionValue[AddAverageWordLengthVariable],
      newpd = newpd // AverageWordLengthVariable
];
If[ OptionValue[AddAverageSentenceLengthVariable],
      newpd = newpd // AverageSentenceLengthVariable
];
If[ OptionValue[AddWordVariationVariable],
      newpd = newpd // WordVariationVariable
];
335 newpd[[1,TEXTDATA]] = pd[[TEXTDATA]];
Return[newpd]
];

340 -----
(* StemHitCountVariables , by Tobias *)
StemHitCountVariables[Protocol[pd_]] :=
Module[ {newpd_, variables},
newpd = pd;
newpd[[VARIABLES]] = {};
variables = pd[[VARIABLES]];
newpd = Fold[ (* re-add all variables from pd into newpd*)
  If[ StringMatchQ[#2[[1]], "HC:" ~~ __], (* check if it is a hitcount variable
  *)
    AddHitCountVariable[#1, WordStemmer[StringReplace[#2[[1]], "HC: " -> ""]],
    (* yes, stem and add *)
    AddVariable[#1, #2[[1]], #2[[2]]];
  ] &, Protocol[newpd], variables]; (* no, add as normal variable *)
Return[newpd];
];

355 -----
(* AddTermDiscriminationValues , by Tobias *)
Options[AddTermDiscrimantionVariables] :=
{StemWords -> False, NumberOfVariables -> 10, TfIdfWeighting -> False,
UseImportStopList -> {False, ""}, UseStopWords -> {},
360 WordFrequencyThreshold -> {0.1, 0.02}};
AddTermDiscrimantionVariables[Protocol[pd_], OptionsPattern[AddTermDiscrimantionVariables]] :=
Module[ {newpd_, terms_, words_, ftext},
If[ (FilterRules[pd[[METADATA]], "Tagged"] // Length) > 0,
      ftext = RemoveTags/pd[[TEXTDATA]],
      ftext = pd[[TEXTDATA]]
];
words = GetFilteredWords[ftext, UseImportStopList -> OptionValue[UseImportStopList],
  UseStopWords -> OptionValue[UseStopWords],
  WordFrequencyThreshold -> OptionValue[WordFrequencyThreshold]];
370 If[ Length[words] == 0,
      Print["No words to consider with current options. Change WordFrequencyThreshold or use
fewer stopwords."];
      Return[Protocol[pd]]
];
terms = TermDiscriminationValues[ftext, words,
  StemWords -> OptionValue[StemWords], TfIdfWeighting -> OptionValue[TfIdfWeighting]];
terms = Reverse[SortBy[terms, #[[2]] &]];
terms = terms[[1 ;; Min[OptionValue[NumberOfVariables], Length[terms]], 1]];
newpd = Fold[AddHitCountVariable, Protocol[pd], terms];
380 Return[newpd]

(* last updated 2010-05-24 version 0.7*)

385 (** SplitText , by Matz last updated 2010-04-10 **)

```

```

(*
News/
 * Added an Option called "SplitSeparator"(used by XML parser)
 * Added "TextID" and changed the code around abit [24/5]
390  * First use of function ArgumentQ [24/5]
*)

SplitText::keeperror = "KeepFactors or KeepVariables error, requires a list of Indices or keywords
 \\"All\\" or \\\"None\\";
SplitText::sepError = "Separator `` is not a String.";
395 Options[SplitText] = { KeepFactors->"All",
                           KeepVariables->"All",
                           SplitID->True,
                           TextID->False,
                           SplitSeparator-> " "};

400 SplitText[p_, mean_, OptionsPattern[SplitText]] :=
  Module[ {len, nparts, split, splits = {}}, words, i = 1, mods, modsNew, vals, valsNew,
         idxrepeats, newpd,
         facsLen, varsLen, keepFactors, keepVariables, fIndices = {}, vIndices = {}, nSplits,
         splitID, textID, sep, pd}];

405 ArgumentQ[{p, mean}, {ProtocolQ, IntegerQ}]; (*Since we let ArgumentQ take care of type
                                                 checking we only pass the variable names (without the type-wrapping "Protocol") and
                                                 let ProtocolQ take care of any inconsistencies *)
pd=p[[1]];

410 len = Length@pd[[TEXTDATA]];
If[ Length@pd[[VARIABLES]]>0,
      varsLen = Length@pd[[VARIABLES, All, 1]],
      varsLen = 0
    ];
415 If[ Length@pd[[FACTORS]]>0,
      facsLen = Length@pd[[FACTORS, All, 1]],
      facsLen = 0
    ];
(*should test index range of factors and variables*)
fIndices = Range@facsLen;(*we add "split ID")
420 vIndices = Range@varsLen;
splitID = OptionValue[SplitID];
sep = OptionValue[SplitSeparator];(*"SplitSeparator"/.Flatten[{opts}]/.Options[
  SplitText];*)
textID = OptionValue[TextID];
If[ !StringQ@sep,
425   Message[SplitText::sepError, sep];
   Abort[]
 ];
keepFactors = OptionValue[KeepFactors];
fIndices = keepFactors /. {"All"->fIndices, "None"->{}};
keepVariables = OptionValue[KeepVariables];
vIndices = keepVariables /. {"All"->vIndices, "None"->{}};
If[ !(ListQ@fIndices && ListQ@vIndices),
      Message[SplitText::keeperror, Head@mean];
      Abort[]
 ];
435 ];
newpd = CreateProtocol[{}];
newpd[[1, TEXTDATA]] = pd[[TEXTDATA]];
newpd[[1, FACTORS]] = pd[[FACTORS, fIndices]];
newpd[[1, VARIABLES]] = pd[[VARIABLES, vIndices]];

440 (*if textid OR splitid then do this, after we use splitID to get textid, if we don't want
       splitid then we just overwrite splitid*)
If[ splitID || textID,
      newpd = AddFactor[newpd, "Split ID", Tostring/@Range[len]]
 ];
445 (*in next version we should allocate before*)
Do[
  words = StringSplit[ newpd[[1, TEXTDATA, i]], sep];
  nparts = Round[Length@words/mean];
450  (*This is the main arithmetic which enforce a local and global mean value splitting*)
  split = StringJoin@Riffle[#, " "] & /@ SmoothPartition[words, nparts];
  splits = Append[splits, split];
  ,{i, 1, len}
];
455 nSplits = Length@splits;
idxrepeats = Length /@ splits;

(*for each modality, insert repetitions to accomodate the additional indices*)
If[ textID || splitID || fIndices>0,(*added textID*)
      mods = newpd[[1, FACTORS, All, 3]];
      modsNew = Table[ Flatten[ConstantArray[mods[[i, #]], idxrepeats[[#]]] &/@
        Range@nSplits], {i, 1, Length@mods}];(*repeat each position -number of splits-
      times*)
];

```

```

        newpd[[1, FACTORS, All, 3]] = modsNew;
    ];

465   (*add the variable values, if any*)
  If[ varsLen>0,
    vals = newpd[[1, VARIABLES, All, 2]]; (*the values matrix*)
    valsNew = Table[ Flatten[ConstantArray[ vals[[i, #]], idxrepeats[[#]] ] &/@
      Range@nSplits], {i, 1, Length@vals} ];
    newpd[[1, VARIABLES, All, 2]] = valsNew;
  ];
  newpd[[1, TEXTDATA]] = Flatten@splits;

  If[Not[splitID] && Not[textID], Return[newpd]]; (*neither splitID nor TextID is added*)

475   (*splitinformation extraction for textID and splitID*)
  splitfactor = Name2Index[newpd, "Split ID", "Factor"];
  splitidmods = newpd[[1, FACTORS, splitfactor, 3]];
  pack = Gather[splitidmods, #1==#2&];
  textidmods = ToString/@ Flatten@Range@Length /@ pack;
(*we know that either splitid or textid is going to be used*)
  If[ textID && !splitID, (*replace splitID*)
    newpd[[1, FACTORS, splitfactor, 3]] = ToExpression@textidmods;
    newpd[[1, FACTORS, splitfactor, 2]] = ToString/@Range@Max@ToExpression@textidmods; (*
      the new modalities*)
    newpd[[1, FACTORS, splitfactor, 1]] = "Text ID";
  ];
(*erase splitID*)
  If[ textID && splitID,(*add the textID factor*)
    newpd = AddFactor[newpd, "Text ID", textidmods];
    splitfactor = Name2Index[newpd, "text id", "Factor"];
(*replace modalities with manual ones (in order)*)
    newpd[[1, FACTORS, splitfactor, 3]] = ToExpression@textidmods;
    newpd[[1, FACTORS, splitfactor, 2]] = ToString/@Range@Max@ToExpression@textidmods;
(*the new modalities*)
  ];

495   (*for the case (!textID && splitID) we are already done*)
(*perhaps add a factor with the lonely splits, for visualisation and filtering (using
  subprotocol)*)
  Return[newpd];
];

500

```

---

```

(**CleanProtocol, by Tobias**)
505 Options[CleanProtocol] = {RemoveDigits -> True,
  RemovePunctuation -> False, ToLowerCase -> True, UseStopWords -> {}};
  CleanProtocol[Protocol[pd_], OptionsPattern[CleanProtocol]] :=
  Module[{texts, newpd},
    newpd = pd;
    Switch[OptionValue[RemoveDigits], False,
      texts = StringReplace[StringTrim /@ pd[[TEXTDATA]],
        Except["." | "," | LetterCharacter | WhitespaceCharacter |
          DigitCharacter] -> ""], True,
      texts = StringReplace[StringTrim /@ pd[[TEXTDATA]],
        Except["." | "," | LetterCharacter | WhitespaceCharacter] ->
        ""]];
    Switch[OptionValue[RemovePunctuation], False, texts = texts, True,
      texts = StringReplace[texts,
        Except[LetterCharacter | WhitespaceCharacter |
          DigitCharacter] -> ""]];
    Switch[OptionValue[ToLowerCase], False, texts = texts, True,
      texts = ToLowerCase /@ texts];
    If[ Length@OptionValue[UseStopWords] > 0,
      texts = texts = StringReplace[texts, (# -> "") & /@ OptionValue[UseStopWords]]
    ];
    newpd[[TEXTDATA]] = texts;
    Return[Protocol[newpd]]
  ];

```

530

---

```

(** LSA, by Matz 2010-04-10 version 0.5 **)
(*
535 * Preprocess corpus by splitting it in sentences with splittext and splitseparator ". " and use
  * LSA with ws=50 (catching long sentences) to gain a better, more coherent analysis
  * Option tf\[Dash]idf
  * Option "MinimumRank"
  * Choose if a word protocol is returned
  *)

```

```

540 LSA::minworderror = "Option \"MinimumWords\" is set too high.";
LSA::maxworderror = "Option \"MaximumWords\" is set too low.";
LSA::minfreqerror = "Option \"MinimumFrequency\" is set too high, analysis generated no words.";
Options[LSA] = {MinimumFrequency -> 5,
               MinimumWords -> 10,
545             MaximumWords -> 700,
               MinimumPairFrequency -> 5,
               MaximumPairs -> 30000,
               SlidingWindowSize -> 10,
               LSAVariables -> 5};

550 LSA[Protocol[pd_List], OptionsPattern[LSA]] :=
Module[{minwords, maxwords, minfreq, minpairfreq, maxpairs, ws, numLSAvars, fitxts, redxts,
      pss, retained,
      rangeRetained, pstally, rs, M, indcoords, indcoordsT, CAM, strv, wordcoords, strwc, pWord,
      pInd, txts,
      words, tals, ts, word2Index, index2Word, itxts, paletteWindow, progcent},
555   Print["Warning: Depending on the input data this might take a long time to complete."];
   pInd = CleanProtocol[Protocol[pd], RemovePunctuation -> True];

   Monitor[
560     progcent = 0;
     minwords = OptionValue[MinimumWords];
     maxwords = OptionValue[MaximumWords];
     minfreq = OptionValue[MinimumFrequency];
     minpairfreq = OptionValue[MinimumPairFrequency];
565     maxpairs = OptionValue[MaximumPairs];
     ws = OptionValue[SlidingWindowSize];
     numLSAvars = OptionValue[LSAVariables];
     txts = pInd[[1, TEXTDATA]];
     words = StringSplit@StringJoin@Riffle[txts, " "];
570     (* pick out and riffle texts and pad with space*)
     tals = Tally[words];
     progcent = 20;
     ts = SortBy[tals, -#[[2]] &]; (*reverse sort [max to min]*)]
575     ts = Select[ts, #[[2]] > minfreq &];
     (*fitxts: after shifting back, used with index conversion*)
     If[ fitxts == {},
         Message[LSA::minfreqerror];
         Abort[]];
580     ];
     progcent = 40;
     (*Convert between index and word*)
     word2Index = Dispatch@Thread@Rule[ts[[All, 1]], Range@Length@ts];
     index2Word = Dispatch@Thread@Rule[Range@Length@ts, ts[[All, 1]]];
585     itxts = (StringSplit/@txts)/.word2Index; (*indexed version of the texts*)
     fitxts = Select[#, minwords <= maxwords &] &/@itxts; (*filter ranks between [min, max], *)
     (*filtered index texts. notice the scope of #
We filter the word index between [min,max], since we tally and sort reversed, the rank is
also the same as the indices (adding rank later)*)
590     If[ Length[ts] < minwords,
         Message[LSA::minworderror];
         Abort[]];
     ];
     (*If[Length[ts]>maxwords, Message[LSA::maxworderror]; Abort[]]; *)
     redxts = fitxts - minwords;
595     (*reduced texts, used in CA, we shift it to avoid block of zeros, starting index on 1*)
     progcent = 50;
     pss = PairsOfLists[redxts, ws]; (*ws - SlidingWindowSize*)(*create pairs of lists of
the words fetched using the "scanning window" method.*)
     progcent = 60;
     retained = Union@Flatten@pss[[1;;Min[Length@pss, maxpairs]]];
600     rangeRetained = Range@Length@retained;
     progcent = 70;
     pstally = SortBy[Tally[pss], -#[[2]] &]; (*tally of pairs*)
     rs = Thread@Rule[pstally[[All, 1]], pstally[[All, 2]]]; (*{{a,b},n} ==> {{a,b}->n}*)
605     M = SparseArray[rs];
     M = 0.5*(M+Transpose[M]); (*use the symmetric part of M, so M has no rows*)
     CAM = CA2[M];
     progcent = 85;
     (*Append LSA variables :*)
     indcoords = Take[#, numLSAvars] & /@ (Total@CAM[[#]] & /@ redxts); (*number of LSA vars*)
     indcoordsT = Transpose@indcoords; (*strv = {strings, variable vectors}*)
     strv = {"LSA var. " <> ToString@# <> " (text)", indcoordsT[[#]]} & /@
     Range@Length@indcoordsT;
610     pInd = Fold[AddVariable[#1, #2[[1]], #2[[2]] &, pInd, strv];
     progcent = 90;
     (*LSA word protocol:*)
     wordcoords = CAM[[1;;numLSAvars]];
     pWord = CreateProtocol[Range@Length@CAM+minwords /. index2Word]; (*convert back to the
original word indexing*)
     strwc = {"LSA var. " <> ToString@# <> " (word)", wordcoords[[#]]} & /@

```

```

Range@Length@wordcoords;(*{string , word coordinates}*)
pWord = Fold[AddVariable[#1 , #2[[1]] , #2[[2]]]& , pWord, strwc];
pWord = AddFactor[pWord, "Word" , pWord[[1 , TEXTDATA, #]] & /@ Range@Length@pWord[[1 ,
TEXTDATA]]];
620 progcent = 100;
, ProgressIndicator[Dynamic[progcent] , {0, 100}]];
Return[{pInd,pWord}];
];

```

625

---

```

(** StemProtocol , by Tobias **)
Options[StemProtocol] = {Language -> "Swedish"};
StemProtocol[Protocol(pd_ , OptionsPattern[StemProtocol])] :=
630 Module[ {newpd, words, offsets, max, ftext},
If[ (FilterRules[pd[[METADATA]], "Tagged"] // Length) > 0,
ftext = RemoveTags/@pd[[TEXTDATA]];
Print["Tagging will be removed"];
ftext = pd[[TEXTDATA]]
];
newpd = pd;
words = StringSplit /@ ftext;
offsets = Length /@ words;
max = Length[words // Flatten];
640 offsets = {FoldList[#1 + #2 &, 1, offsets] // Most, max} // Flatten;
words = Flatten[words];
newpd[[TEXTDATA]] =
WordStemmer[words , ToLowerCase[OptionValue[Language]] , offsets];
645 Return[newpd // Protocol];
];

```

```

650 (*
THE FOLLOWING FUNCTIONS ARE USED WITHIN THE PACKAGE
AND CAN'T BE USED WHEN IMPORTING THE PACKAGE
*)
655

```

---

```

(** WordFrequencies , by Tobias **)
Options[WordFrequencies] = {MinimumWordSize -> 3, MinimumFrequency -> 5, StemWords -> False};
660 WordFrequencies[txts_List , opts___ ? OptionQ] :=
Module[ {ftxt , words, minword, minfreq, stem},
minfreq = MinimumFrequency /. Flatten[{opts}] /. Options[WordFrequencies];
minword = MinimumWordSize /. Flatten[{opts}] /. Options[WordFrequencies];
stem = WordStemmer /. Flatten[{opts}] /. Options[WordFrequencies];
665 ftxt = StringReplace[#, RegularExpression["[^a-\[ODoubleDot]A-\[CapitalODoubleDot]\]|\[RightGuillemet]" ] -> " "] & /@ txts;
words = StringSplit[#] & /@ ftxt // Flatten;
words = Select[words, StringLength[#] >= minword &];
words = ToLowerCase[#] & /@ words;
If[ stem,
670 words = WordStemmer[words]
];
words = Tally[words];
Return[SortBy[Select[words, #[[2]] >= minfreq &], #[[2]] &] // Reverse]
];
675

```

---

```

680 (** ZScore , by Tobias **)
ZScore[obs_ , expt_ , nrtokens_ , prob_] :=
Return[N[(obs-expt)/Sqrt[nrtokens (prob (1-prob))]]];

```

---

```

685 (** TScore , by Tobias **)
TScore[obs_ , expt_] :=
Return[N[(obs-expt)/Sqrt[obs]]];

```

---

```

690 (** SpanFrequency , by Tobias **)
Options[SpanFrequency] = {MinimumWordSize -> 3, MinimumFrequency -> 5,
SlidingWindowSize -> 5, StemWords -> False};
SpanFrequency[txts_List , word_ , opts___ ? OptionQ] :=

```

```

Module[ {minfreq , minword , stem , freqs , pos , ftxt , words , neighbours , win , scores , length ,
ssize} ,
695   win = SlidingWindowSize /. Flatten[{opts}] /. Options[SpanFrequency];
minfreq = MinimumFrequency /. Flatten[{opts}] /. Options[SpanFrequency];
minword = MinimumWordSize /. Flatten[{opts}] /. Options[SpanFrequency];
stem = WordStemmer /. Flatten[{opts}] /. Options[SpanFrequency];
freqs = WordFrequencies[txts , MinimumWordSize -> minword , MinimumFrequency -> 1];
ftxt = StringReplace[#, RegularExpression["^a-\\[ODoubleDot]A-\\[CapitalODoubleDot] "]\|\[
RightGuillemet]" -> " "] & /@ txts;
700   words = (StringSplit[#] & /@ ftxt) // Flatten;
words = ToLowerCase[#] & /@ words;
length = Length[words];
scores[posi_ , obs_ , sampsize_] :=
  {{freqs[[posi , 2]]/ length)* sampsize // N,
ZScore[obs , (freqs[[posi , 2]]/ length)* sampsize , sampsize , (freqs[[posi , 2]]/ length)],
TScore[obs , (freqs[[posi , 2]]/ length)* sampsize]};
pos = StringReplace[#, RegularExpression[ToLowerCase[word]] -> "\\[RightGuillemet]" ] & /@
words;
705   pos = Position[pos , "\\[RightGuillemet]" ] // Flatten;
neighbours = Array[
  {
    Take[words , {pos[[#]] - win , pos[[#]] - 1}] ,
    Take[words , {pos[[#]] + 1 , pos[[#]] + win}]
  }
  &, Length[pos]
] // Flatten;
710   ssize = Length[neighbours];
neighbours = Tally[neighbours];
neighbours = Select[neighbours , StringLength[#[[1]]] >= minword &];
neighbours = Select[neighbours , #[[2]] >= minfreq &];
neighbours = Array[ ({neighbours[[#]] ,
  scores[Position[freqs , {neighbours[[#]][[1]] , -}]] // Flatten , neighbours
[[#]][[2]] , ssize]} // Flatten)
  &, Length[neighbours]];
715   neighbours = SortBy[neighbours , #[[4]] &] // Reverse;
neighbours = Prepend[neighbours , Style[#, Bold] & /@ {"Words" , "# observed \nco-
occurrences" ,
  "# expected \nco-occurrences" , "Berry-
Rogghe's \nZ-score" , "T-score"}];
720   Return[neighbours]
];

```

730

---

```

(** RemoveTags, by Tobias **)
RemoveTags[text_String] :=
735   StringReplace[StringTrim[StringReplace[text ,
Shortest["_ ~~ ___ ~~ WhitespaceCharacter] -> " "]] , {" ." -> ".," , ",." -> ","}];

```

```

(**TagWords, by Tobias **)
TagWords::tagerror = "A tagging error occurred. Try raising the systems stack size, lower Option 'WordGroupingNumber', or split up the larger individuals.
740   There might also be issues with the submitted model.";
TagWords::modelnotfounderror = "Couldnt load model file '1'";
TagWords::hunposnotfounderror = "Couldnt find hunpos-tag.exe.";
TagWords[words_List , model_] :=
Module[ {ftext , ord , strings , tags , test , temp , tries = 0},
745   ftext = words;
temp = words;
ftext = Riffle[ftext , "\n"];
ftext = StringJoin[{ftext , "\n"} // Flatten];
If[ FindFile[model] == $Failed ,
  Print[Message[TagWords::modelnotfounderror , model]];
  Abort[]
];
750   If[ FindFile["hunpos-tag.exe"] == $Failed ,
  Print[Message[TagWords::hunposnotfounderror]];
  Abort[]
];
While[Export["input" , ftext , "String" , CharacterEncoding -> "UTF-8"] == $Failed || tries
== 5 , tries++];
755   If[ Run["hunpos-tag.exe <> model <> " < input > output.txt" ] < 0 ,
  Print[Message[TagWords::tagerror]];
  Print[Length[temp]];
  Return[-1]
];
tries = 0;
760   While[test = Import["output.txt"]; == $Failed || tries == 5 , tries++];
strings = StringSplit[test , "\n"];
strings = Flatten[StringSplit /@ Select[strings , Length[StringSplit[#]] >= 2 &]];
ord = Take[strings , {1,Length[strings] , 2}];

```

```

tags = Take[strings, {2, Length[strings], 2}];
DeleteFile["output.txt"];
DeleteFile["input"];
(*ftext = Inner[#1 <> "_" <> #2 <> " " &, Take[ord, #], Take[tags, #], StringJoin] & /@
ls ; *)
ftext = StringSplit[Inner[#1 <> "_" <> #2 <> " " &, ord, tags, StringJoin], Shortest[--BREAKPOINT--~~---~WhitespaceCharacter]];
Return[ftext];

```

---

```

(**GroupAndTag, by Tobias **)
Options[GroupAndTag] = {WordGroupingNumber -> 75000};
GroupAndTag[text_List, model_, opts___? OptionQ] :=
770 Module[{nr, totwords, ftext, i, group, oldgroup, tagged = {}},
nr = WordGroupingNumber /. Flatten[{opts}] /. Options[GroupAndTag];
ftext = StringReplace[text, {"." -> ". ", "?" -> "? ", "," -> " , ", "!" -> " ! "}];
ftext = StringSplit[ftext];
If[Max[Length /@ ftext] > nr,
Print["Warning: There are individual texts in the protocol larger than the
WordGroupingNumber"];
];
If[Min[Length /@ ftext] == 0,
Print["Warning: Protocol consist of an empty string and can not be tagged"];
Abort[];
];
totwords = Total[Length /@ ftext];
group = {};
n = 0;
Monitor[
795 Do[
oldgroup = group;
group = Append[group, {"--BREAKPOINT--", ftext[[i]]}] // Flatten;

(*ls = Append[ls, {Last[ls][[1]]+1, Last[ls][[2]]+Length[ftext[[i]]]}]; *)
n = i;
If[(Length[group] >= nr) || (i == Length[ftext]),
Module[{check = {}},
If[Length[group] > nr + 5000,
check = TagWords[ftext[[i]], model];
group = oldgroup;
];
check = Prepend[check, TagWords[group, model]];
tagged = Append[tagged, check] // Flatten;
totwords = totwords - Length[group];
group = {};
];
805 ];
];
];
, {i, Length[ftext]}], ProgressIndicator[Dynamic[n], {0, Length[text]}]];
810 Return[tagged]
];
815 ];

```

---

```

(**WordStemmer, by Tobias**)
WordStemmer::stemmernotfound = "The stemmer Lib.exe couldnt be found.";
820 WordStemmer[words_, lang_, offsets_] :=
Module[{wordslist, stemmedwords, import, tries = 0},
wordslist = StringJoin[{Riffle[words, "\n"], "\n"} // Flatten];
If[FindFile["Lib.exe"] == \$Failed,
Print[Message[WordStemmer::stemmernotfounderror]];
Abort[];
];
825 While[Export["input", wordslist, "String"] == \$Failed ||
tries == 5, tries++];
tries = 0;
Run["Lib.exe -l " <> lang <> "< input > output"];
While[(import = Import["output", "String"]) == \$Failed ||
tries == 5, tries++];
stemmedwords = Flatten[StringSplit[import]];
830 stemmedwords =
StringJoin[
Riffle[Take[
stemmedwords, {offsets[[#]], offsets[[# + 1]] - 1}], " "] & /@
Range[Length[offsets] - 1];
];
835 Return[stemmedwords]
];
840

```

---

```

845 (** WordList, by Sverker 2010-01-19 **)
WordList[Protocol[pd_List]] :=
```

```
SortBy [ Tally [ Flatten [ StringSplit /@ pd [ [ TEXTDATA ] ] ] ] , -#[ [ 2 ] ] & ];
```

---

```
850 (**NumberOfWordsVariable, by Tobias**)
NumberOfWordsVariable[ Protocol[ pd_ ] ] :=
Module[ { nrwords },
  nrwords = Length /@ ( StringSplit[ # ] & /@ pd [ [ TEXTDATA ] ] );
  Return[ AddVariable[ Protocol[ pd ] , "NumberOfWords" , nrwords ] ]
];

```

```
860 (**NumberOfSentencesVariable, by Tobias**)
NumberOfSentencesVariable[ Protocol[ pd_ ] ] :=
Module[ { nrsentences },
  nrsentences = Length /@ ( StringSplit[ #, ". " ] & /@ pd [ [ TEXTDATA ] ] );
  Return[ AddVariable[ Protocol[ pd ] , "NumberOfSentences" , nrsentences ] ]
];

```

```
865 (**AverageWordLengthVariable, by Tobias**)
AverageWordLengthVariable[ Protocol[ pd_ ] ] :=
Module[ { textsplits , means = { } },
  textsplits = StringSplit[ # ] & /@ pd [ [ TEXTDATA ] ];
  Do[ means =
    Append[ means , N[ Mean[ StringLength /@ textsplits [ [ i ] ] ] ] ] , { i , 1 ,
      Length[ textsplits ] } ];
  Return[ AddVariable[ Protocol[ pd ] , "AvgWordLength" , means ] ]
];

```

```
875 (**AverageSentenceLengthVariable, by Tobias**)
AverageSentenceLengthVariable[ Protocol[ pd_ ] ] :=
Module[ { means = { } , sentlengths },
  sentlengths = StringSplit[ #, ". " ] & /@ pd [ [ TEXTDATA ] ];
  Do[ means =
    Append[ means ,
      N[ Mean[ Length /@ ( StringSplit[ # ] & /@ sentlengths [ [ i ] ] ) ] ] , { i ,
      1 , Length[ sentlengths ] } ];
  Return[ AddVariable[ Protocol[ pd ] , "AvgSentenceLength" , means ] ]
];

```

```
880 (**WordVariationVariable, by Tobias**)
WordVariationVariable[ Protocol[ pd_ ] ] :=
Module[ { dls , tls , vs },
  tls = Length /@ ( StringSplit /@ pd [ [ TEXTDATA ] ] );
  dls = Length /@ ( Union /@ ( StringSplit /@ pd [ [ TEXTDATA ] ] ) );
  vs = N[ dls / tls ];
  Return[ AddVariable[ Protocol[ pd ] , "WordVariation" , vs ] ]
];
Options[ DocumentVectors ] = { InputWordsAreStemmed -> False , TfIdfWeighting -> True };
DocumentVectors[ texts_List , words_ , OptionsPattern[ DocumentVectors ] ] :=
Module[ { selwords , occs = { } , reps , allwords , numwords , numtexts , compiledCount },
  allwords = StringSplit[ # ] & /@ ToLowerCase /@ texts ;
  selwords = words ;
  If[ OptionValue[ InputWordsAreStemmed ] ,
    (*True*)
    allwords = WordStemmer /@ allwords ;
  ];
  reps = Array[ selwords [ [ # ] ] -> # & , Length[ selwords ] ];
  numwords = reps [ [ All , 2 ] ];
  numtexts = allwords /. Dispatch[ reps ];
  numtexts = numtexts /. Dispatch[ x_String -> 0 ];
  compiledCount = Compile[ { { list , _Integer , 1 } , { w , _Integer , 1 } } , Module[ { },
    Return[ Count[ list , # & /@ w ]
  ] ];
  Do[ occs = Append[ occs , compiledCount[ Normal[ numtexts [ [ i ] ] ] , Normal[ numwords ] ] ] , { i , 1 ,
    Length[ texts ] } ];
  Return[ occs ]
];

```

```
920 (**TermDiscriminationValues, by Tobias**)
Options[ TermDiscriminationValues ] :=
```

```

{StemWords -> False, TfIdfWeighting -> False};

TermDiscriminationValues[texts_List, words_List, OptionsPattern[TermDiscriminationValues]] :=
925   Module[{docvectors, centroid, avgdistance, avgdistancek = {}, termdiscrim, selwords},
    selwords = words;
    If[ OptionValue[StemWords],
      selwords = WordStemmer[selwords] // Union
    ];
930   If[ OptionValue[TfIdfWeighting],
      docvectors = Normalize[N[#]] & /@ Tfidf[texts, selwords],
      docvectors = Normalize[N[#]] & /@ DocumentVectors[texts, selwords]
    ];
    centroid = Normalize[N[Mean[docvectors]]];
    avgdistance = Total[N[centroid.#] & /@ docvectors];
    (*compiledDot = Compile[{{cent, _Real, 1}, {docvec, _Real, 1}, {ind, _Integer}},
      Return[N[Dot[Normalize[Delete[cent, ind]], Normalize[Delete[docvec, ind]]]]]]; *)
    n = 0;
    Monitor[
940     Do[
      n++;
      avgdistancek =
        {avgdistancek, Total[Normalize[Delete[centroid, i]] . Normalize[Delete[#, i]]] & /@
          docvectors}], {i, 1, Length[selwords]}];
      , ProgressIndicator[Dynamic[n], {0, Length[selwords]}]];
    avgdistancek = avgdistancek // Flatten;
    termdiscrim = Inner[#, #2] &, SparseArray[selwords], SparseArray[avgdistancek -
      avgdistance], List];
    Return[termdiscrim]
  ];

```

950

---

```

(** GetFilteredWords, by Tobias **)
Options[GetFilteredWords] :=
  {UseImportStopList -> {False, ""}, UseStopWords -> {}},
955  WordFrequencyThreshold -> {0.1, 0.005};
GetFilteredWords[txt_, OptionsPattern[GetFilteredWords]] :=
  Module[{l, words, freqs, stops},
    freqs = WordFrequencies[{txt} // Flatten, MinimumFrequency -> 1, MinimumWordSize -> 4];
    l = Length[freqs];
    stops = OptionValue[UseStopWords];
960    If[ OptionValue[UseImportStopList][[1]],
      stops = {stops, GetStopWordList[OptionValue[UseImportStopList][[2]]]} // Flatten
    ];
    words = Select[freqs, N[#[[2]]/1] <= OptionValue[WordFrequencyThreshold][[1]] &&
965      N[#[[2]]/1] >= OptionValue[WordFrequencyThreshold][[2]] &&
      Not[MemberQ[stops, #[[1]]]] &][[All, 1]];
    Return[words]
  ];

```

970

---

```

(**SmoothPartition, by Matz 12/4**)
(* Future optimization: Use modulo arithmetic to calculate the intervals (two parts) of the
 partitions.*)
SmoothPartition[list_List, parts_Integer] :=
  Module[{lists, partsVar, lenVar, stride, i = 1, roof},
975    If[ parts == 0,
      Return[{list}]
    ];
    partsVar = parts; (*variable to determine how many parts are left to deal with*)
    lenVar = Length[list];
980    (*variable which tells us how many elements are left in the list, better than using Length
     for each loop*)
    roof = Ceiling[lenVar/Max[parts, 1]]; (*first element size, larger than rest*)
    lists = Table[Null, {i, parts}, {j, roof}]; (*allocate memory for lists *)
    Do[stride = Ceiling[lenVar/partsVar];
      (*compensate for each partition, distributing the rest over the sublists*)
      lists[[j]] = list[[i; i+stride-1]]; (*list*)
      lenVar = lenVar - stride; (*we collapse list and change the next stepsize*)
      i = i+stride;
      partsVar = partsVar - 1; (*decreasing the counter of how many parts are left*)
      , {j, 1, Min[parts, lenVar]}
    ];
    Return[lists];
  ];

```

(\*aka "SlidingWindow", Optimized version of PairsOfList based on Sverkers transpose idea, by Matz
version 0.95 25/4 updated 6/6\*)

995

---

```

(**PairsOfList, by Matz**)
PairsOfList[a_List, ws_Integer] :=

```



```

[[11;; -1]]} ', {a[;; -12]], a[[12;; -1]]} ', {a[;; -13]], a[[13;; -1]]} ', {a[;; -14]], a
[[14;; -1]]} ', {a[;; -15]], a[[15;; -1]]} ', {a[;; -16]], a[[16;; -1]]} ', {a[;; -17]], a
[[17;; -1]]} ']
1050 ];
If [ ws==17,
  Return[Join[{a[;; -2]], a[[2;; -1]]} ', {a[;; -3]], a[[3;; -1]]} ', {a[;; -4]], a[[4;; -1]]} ', {a
  [[;; -5]], a[[5;; -1]]} ', {a[;; -6]], a[[6;; -1]]} ', {a[;; -7]], a[[7;; -1]]} ', {a[;; -8]], a
  [[8;; -1]]} ', {a[;; -9]], a[[9;; -1]]} ', {a[;; -10]], a[[10;; -1]]} ', {a[;; -11]], a
  [[11;; -1]]} ', {a[;; -12]], a[[12;; -1]]} ', {a[;; -13]], a[[13;; -1]]} ', {a[;; -14]], a
  [[14;; -1]]} ', {a[;; -15]], a[[15;; -1]]} ', {a[;; -16]], a[[16;; -1]]} ', {a[;; -17]], a
  [[17;; -1]]} ', {a[;; -18]], a[[18;; -1]]} ']
];
1055 If [ ws==18,
  Return[Join[{a[;; -2]], a[[2;; -1]]} ', {a[;; -3]], a[[3;; -1]]} ', {a[;; -4]], a[[4;; -1]]} ', {a
  [[;; -5]], a[[5;; -1]]} ', {a[;; -6]], a[[6;; -1]]} ', {a[;; -7]], a[[7;; -1]]} ', {a[;; -8]], a
  [[8;; -1]]} ', {a[;; -9]], a[[9;; -1]]} ', {a[;; -10]], a[[10;; -1]]} ', {a[;; -11]], a
  [[11;; -1]]} ', {a[;; -12]], a[[12;; -1]]} ', {a[;; -13]], a[[13;; -1]]} ', {a[;; -14]], a
  [[14;; -1]]} ', {a[;; -15]], a[[15;; -1]]} ', {a[;; -16]], a[[16;; -1]]} ', {a[;; -17]], a
  [[17;; -1]]} ', {a[;; -18]], a[[18;; -1]]} ', {a[;; -19]], a[[19;; -1]]} ']
];
1060 If [ ws==19,
  Return[Join[{a[;; -2]], a[[2;; -1]]} ', {a[;; -3]], a[[3;; -1]]} ', {a[;; -4]], a[[4;; -1]]} ', {a
  [[;; -5]], a[[5;; -1]]} ', {a[;; -6]], a[[6;; -1]]} ', {a[;; -7]], a[[7;; -1]]} ', {a[;; -8]], a
  [[8;; -1]]} ', {a[;; -9]], a[[9;; -1]]} ', {a[;; -10]], a[[10;; -1]]} ', {a[;; -11]], a
  [[11;; -1]]} ', {a[;; -12]], a[[12;; -1]]} ', {a[;; -13]], a[[13;; -1]]} ', {a[;; -14]], a
  [[14;; -1]]} ', {a[;; -15]], a[[15;; -1]]} ', {a[;; -16]], a[[16;; -1]]} ', {a[;; -17]], a
  [[17;; -1]]} ', {a[;; -18]], a[[18;; -1]]} ', {a[;; -19]], a[[19;; -1]]} ', {a[;; -20]], a
  [[20;; -1]]} ']
];
(*Adaptive threshold engaged, below is a "slower" version (approximately twice as slow)*)
1065 take = Max[0, Binomial[Min[Length@a, ws], 2] + ws*Max[0, len-ws]];
(*This version creates the rolling in ten-word chunks and returns the relevant elements
after the fact, slower but makes subsequent rolling more practical*)
tmp = Join[
{a[;; -1]], a[[1;; -1]]} ', {a[;; -2]], a[[2;; -1]]} ', {a[;; -3]], a[[3;; -1]]} ', {a[;; -4]], a
[[4;; -1]]} ', {a[;; -5]], a[[5;; -1]]} ', {a[;; -6]], a[[6;; -1]]} ', {a[;; -7]], a[[7;; -1]]} ', {a
[[;; -8]], a[[8;; -1]]} ', {a[;; -9]], a[[9;; -1]]} ', {a[;; -10]], a[[10;; -1]]} ', {a[;; -11]], a
[[11;; -1]]} ', {a[;; -12]], a[[12;; -1]]} ', {a[;; -13]], a[[13;; -1]]} ', {a[;; -14]], a
[[14;; -1]]} ', {a[;; -15]], a[[15;; -1]]} ', {a[;; -16]], a[[16;; -1]]} ', {a[;; -17]], a
[[17;; -1]]} ', {a[;; -18]], a[[18;; -1]]} ', {a[;; -19]], a[[19;; -1]]} ', {a[;; -20]], a
[[20;; -1]]} ', {a[;; -21]], a[[21;; -1]]} '];
If [ ws<30,
  Return[tmp[[;; take]]]
];
1070 tmp = Join[tmp, {a[;; -31]], a[[31;; -1]]} ', {a[;; -32]], a[[32;; -1]]} ', {a[;; -33]], a
[[33;; -1]]} ', {a[;; -34]], a[[34;; -1]]} ', {a[;; -35]], a[[35;; -1]]} ', {a[;; -36]], a
[[36;; -1]]} ', {a[;; -37]], a[[37;; -1]]} ', {a[;; -38]], a[[38;; -1]]} ', {a[;; -39]], a
[[39;; -1]]} ', {a[;; -40]], a[[40;; -1]]} '];
If [ ws<40,
  Return[tmp[[;; take]]]
];
1075 tmp = Join[tmp, {a[;; -41]], a[[41;; -1]]} ', {a[;; -42]], a[[42;; -1]]} ', {a[;; -43]], a
[[43;; -1]]} ', {a[;; -44]], a[[44;; -1]]} ', {a[;; -45]], a[[45;; -1]]} ', {a[;; -46]], a
[[46;; -1]]} ', {a[;; -47]], a[[47;; -1]]} ', {a[;; -48]], a[[48;; -1]]} ', {a[;; -49]], a
[[49;; -1]]} ', {a[;; -50]], a[[50;; -1]]} '];
If [ ws<50,
  Return[tmp[[;; take]]]
];
(*The window size above 50 is deemed unlikely hence uses a slow automatic algorithm, about
30-40 times slower than the above*)
1080 (*if needed should use the already calculated list "tmp")
initSubsets = Subsets[a[[1;; Min[len, ws]]], {2}]; (*if ws<len*)
restSubsets = Table[{#, a[[i+ws]]}& /@ a[[i;; i+ws-1]], {i, len-ws}];
Return[Join[initSubsets, Flatten[restSubsets, 1] ]]
];
1085


---


(*PairsOfLists, by Matz*)
PairsOfLists[lists_List, ws_Integer] :=
Flatten[PairsOfList [#, ws] &/@ lists, 1];

```

---

```

(*>CA2, by Sverker*)
CA2[F_] := 
Module[ {n,P,rowsums,colsums,tmp,A,u,g,v,U,coords},
105   n = Plus@@Plus@@F;
   P = F/n;
   rowsums = Normal@N[Plus@@Transpose[F]/n]; (*opt*)
   colsums = Normal@N[Plus@@F/n];
110   tmp = Transpose[(1/Sqrt[rowsums])*(POuter[Times,rowsums,colsums])];
   A = Transpose[ Table[(1/Sqrt[colsums[[i]]])*tmp[[i]],{i,1,Length[colsums]}]];
   {u,g,v} = SingularValueDecomposition[A]; (*numerical*)
   U = Conjugate[Transpose[u]];
   coords = Transpose[Transpose[((1/Sqrt[rowsums])*U)].g];
   Return[coords];
115 ];
];

(**TfIdfs, by Tobias**)
TfIdfs[texts_List, words_] :=
Module[ {selwords, occs = {}, Ns, docfreq, reps, allwords, numwords,
  numtexts, compiledCount},
  allwords = StringSplit[#] & /@ ToLowerCase /@ texts;
  selwords = words;
  Ns = Length[texts];
115  reps = Array[selwords[[#]] -> # &, Length[selwords]];
  numwords = reps[[All, 2]];
  numtexts = allwords /. Dispatch[reps];
  numtexts = numtexts /. Dispatch[x_String -> 0];
  compiledCount =
    Compile[{{list, _Integer, 1}, {w, _Integer, 1}},
      Module[ {},
        Return[Count[list, #] & /@ w]
      ];
    ];
  Do[occs =
    Append[occs,
      compiledCount[Normal[numtexts[[i]]], Normal[numwords]], {i, 1,
      Length[texts]}];
    docfreq = Log[Ns/(Total[occs /. x_ /; x > 0 -> 1] + 1)] // N;
    Inner[#1*#2 &, #, docfreq, List] & /@ occs
1130 ];
];

End[] (* End Private Context *)
135 EndPackage[]

```

## Transformation

```

BeginPackage["Transformations`", {"RCA`, "Combinatorica`"}]

Unprotect[
IndicatorMatrix,
5 MCA
MultipleCorrespondenceAnalysis,
GetSingularValues,
GetCoordinateStatus,
GetPrincipalAxes,
10 ModalityCoordinates,
CrossTabulate,
CA,
CorrespondenceAnalysis,
FactorContributions,
15 ModalityContributions,
PCA,
NLPCA]

IndicatorMatrix::usage="IndicatorMatrix[p,f] takes a list f of one or more factors from protocol p
and makes an indicator matrix"
20 MCA::usage="MCA[p,{f1,f2,...}] takes a protcol p and a list of factors f1,f2,... and performs
multiple correspondence analysis with respect to these.
Returns a protcol with a new variable for each axis."
Options[MCA]={Dimensions->2,AxesName->"Coordinates",StandardCoordinates->False,PassiveModalities
->{},MakePassiveThreshold->0,PrintPassiveModalities->False};
Options[MultipleCorrespondenceAnalysis]={Dimensions->2,AxesName->"Coordinates",StandardCoordinates
->True,PassiveModalities ->{},MakePassiveThreshold ->0,PrintPassiveModalities ->False};
25 GetSingularValues::usage="GetSingularValues[p] returns the singular values for the latest MCA
runned on protocol p"

```

```

GetCoordinateStatus::usage="GetCoordinateStatus[p] returns True if the protocol p contains
standard coordinates and False if it contains principal coordinates."
30 ModalityCoordinates::usage="ModalityCoordinates[p,f,v] takes a protocol p, a list f of factors and
a list v defining which variables should be used as axes.
Returns a protocol with coordinates for every modality in f and a size variable indicating how many
individuals each modality represent.
The factor and modality names in f will be stored as factors in the new protocol."
Options[ModalityCoordinates]={Rescale->True,StandardCoordinates->False,MatrixForm->False};

35 CrossTabulate::usage = "Crosstabulate[p,{f1,f2}] creates a contingency table of the two factors f1
and f2."
Options[CrossTabulate] = {AddTitles -> False, TableForm -> False, Statistic -> Count,
ReturnAssociation -> False, ReturnSorted -> False, ReturnList -> False, SignificanceLevel ->
.05};
Statistic::usage="Statistic is an option for CrossTabulate indicating what should be put in the
cells.";
RowPercentage::usage="RowPct is an option for Crosstabulation.";
ColumnPercentage::usage="ColPct is an option for Crosstabulation.";
40 DependencyPValue::usage="An options for Crosstabulation.";
SignificanceDifference::usage="An options for Crosstabulation.";
SignificanceLevel::usage="An options for Crosstabulation.";
StandardResidual::usage="An options for Crosstabulation.";
ChiSquare::usage="An options for Crosstabulation.";
45 MeanChiSquare::usage="An options for Crosstabulation.";
ReturnList::usage="An options for Crosstabulation.";
PercentageDeviation::usage="An options for Crosstabulation.";
ReturnSorted::usage="An option for InertiaDifference.";
ReturnAssociation::usage="An option for Crosstabulation.";

50 CA::usage="CA[p,f1,f2] creates a contingency table based on the modalities of factors f1 and f2,
and applies simple correspondence analysis to it. Two protocols, grouped over f1 and f2
respectively, are returned, with the corresponding principal coordinates added as variables.
The dimension of the projections may be specified with option \"Dimensions\". The variable
names may be specified with \"AxesName\". Singular values may also be returned by specifying
so with the \"ReturnSingularValues\" option.

CA[M] applies simple correspondence analysis to a matrix M and returns a matrix with the principal
coordinates of the rows of M."
Options[CA]={AxesName -> "x", Dimensions -> All, ReturnSingularValues->False};

55 FactorContributions::usage="FactorContributions[p,v,f] assumes MCA has been done on p with respect
to factors f (where f is a list of factor indices). A table showing the contributions of
factors to the principal axes is returned. Specify the variables defining the axes with the \
Variables\ option."
Options[FactorContributions] = {RelativeContributions -> True};

ModalityContributions::usage="ModalityContributions[p,v,f] assumes MCA has been done on p with
respect to factors f (where f is a list of factor indices). A table of relevant modalities,
and a percentage indicating how well they summarize the axis, is returned."
60 Options[ModalityContributions]={RelativeContributions -> True, ReturnMean -> False, PrinterFriendly
->False, Concise->True, Threshold->0};

FactorAssociationGraph ::usage="FactorAssociationGraph[p,fs,s,t] returns a graph with edges between
\
factors f_1 and f_2 in fs if the statistic s for the crosstabulation between f_1 and f_2 > t.";
Options[FactorAssociationGraph] = {ShowIndex -> False, Method -> "SpringEmbedding"};
65 ShowIndex::usage="An option for FactorAssociationGraph.";

ModalityAssociationGraph ::usage="ModalityAssociationGraph[p,fs,s,t] returns a graph with \
edges between modalities m_1 and m_2 in fs if the statistic s for (m_1,m_2) in the crosstabulation
between \
f_1 and f_2 > t. ";
70 Options[ModalityAssociationGraph]={ShowIndex->False,Method->"SpringEmbedding"};

PCA::usage="PCA[p,dim,{v1,v2,...,vm}] performs a weighted principal component analysis on the
variables {v1, v2, ..., vm} of the protocol p. It returns a list of protocols {p0,p1}. p1 is p
with the coordinates of the modalities of the variables in dim dimensions added as new
variables. p0 contains the component loadings of the variables.";
Options[PCA]={MaxIterations->100,Convergence->10.^-5,StartX->{},Stat->False,DiscriminationInfo->
False};
MaxIterations::usage="MaxIterations is an option that specifies the maximum number of iterations
that should be tried in the Alternating Least Squares-algorithm of PCA.";
75 Convergence::usage="Convergence is an option for PCA that specifies the convergence limit, that is,
at what difference between consecutive values of the coordinates for the individuals the
algorithm is to be considered to have converged.";
DiscriminationInfo::usage="DiscriminationInfo is an option for NLPCA and PCA. If set to true,
NLPCA and PCA returns an extra protocol with discrimination measures.";

NLPCA::usage="NLPCA[p,dim,{f1,f2,...,fn},{v1,v2,...,vm}] performs a nonlinear principal component
analysis on the factors {f1, f2, ..., fn} and the variables {v1, v2, ..., vm} from the
protocol p. It is not necessary to supply any variables, but it is necessary to supply some
factors. The function returns a list of protocols {p1,p2} if there are no variables or
constrained factors, otherwise it returns the list of protocols {p0,p1,p2}. The protocol p0

```

contains the component loadings of the variables and constrained factors.  $p_1$  is the protocol  $p$  with the coordinates of the individuals and of the modalities of the variables  $\{v_1, v_2, \dots, v_m\}$  added as new variables. The protocol  $p_2$  contains the coordinates of the modalities of  $\{f_1, f_2, \dots, f_n\}$ . See Options[NLPCA] for a list of options.";  
**Options[NLPCA]**=  
 {MaxIterationsOuter->200,MaxIterationsInner->50,ConvergenceInner->10.^-3,  
 ConvergenceOuter->10.^-3,StartX->{},Single->{},Ordinal->{},OrdinalFunction->1,Stat->**False**,  
 DiscriminationInfo->**False**};  
**80** MaxIterationsOuter::usage="MaxIterationsOuter is an option that specifies the maximum number of iterations that should be tried in the outer Alternating Least Squares-algorithm of NLPCA.";  
 MaxIterationsInner::usage="MaxIterationsInner is an option that specifies the maximum number of iterations that should be tried in the inner Alternating Least Squares-algorithm of NLPCA.";  
 ConvergenceOuter::usage="ConvergenceOuter is an option for NLPCA that specifies the convergence limit for the outer Alternating Least Squares-algorithm. That is, at what difference between consecutive values of the coordinates for the individuals the outer algorithm is to be considered to have converged.";  
 ConvergenceInner::usage="ConvergenceInner is an option for NLPCA that specifies the limit for convergence for the inner Alternating Least Squares-algorithm. That is at what difference between consecutive values of the coordinates for the individuals the inner algorithm is to be considered to have converged.";  
 StartX::usage="StartX is an option for PCA and NLPCA that lets the user specify what matrix of coordinates for the individuals that should be used as the starting matrix of the algorithm. If not specified a random starting matrix is used in PCA and NLPCA.";  
**85** Stat::usage="Stat is an option for PCA and NLPCA that specifies whether or not to return extra statistics, concerning the analysis performed, in addition to the protocol/protocols. If set to true the output of both PCA and NLPCA be a list containing two lists. The protocols that are usually the output of PCA and NLPCA are now contained in the first list and the statistics is contained in the second list. Use the functions ShowStat and ShowIterations to visualize the statistics.";  
 Single::usage="Single is an option for NLPCA that specifies which, if any, of the factors should be treated as single. The factors must be supplied in a list with the indexes of the factors to be treated as single. Treating a factor as single forces the coordinates of the modalities to lie on a straight line through the origin.";  
 Ordinal::usage="Ordinal is an option for NLPCA that specifies which, if any, of the factors should be treated as ordinal. The factors must be supplied in a list with the indexes of the factors to be treated as ordinal. Treating a factor as single forces the coordinates of the modalities to lie on a straight line through the origin, with the modalities ordered.";  
 OrdinalFunction::usage="OrdinalFunction is an option for NLPCA that specifies which minimizing function to use when treating a factor as ordinal.";  
 DiscriminationInfo::usage="DiscriminationInfo is an option for NLPCA and PCA. If set to true, NLPCA and PCA returns an extra protocol with discrimination measures.";  
**90** ShowStat::usage="ShowStat[statlist] takes statistics from a principal component analysis or a nonlinear principal component analysis and prints it in different forms. The statistics needed is contained in statlist, which is the output {protocollist,statlist}=PCA[Protocol[pd],dim,{v1,...,vm},Stat->True] or {protocollist,statlist}=NLPCA[Protocol[pd],dim,{f1,...,fn},{v1,...,vm},Stat->True]. That is the protocols that are usually the output of PCA and NLPCA are now contained in the first list in the output when Stat is set to true, and the statistics is contained in the second list. See Options>ShowStat for a list of options.";  
**Options>ShowStat**=  
 {StatFormat->**Plot**,ListLinePlotOptions->{**AxesOrigin**->{1,0},**PlotRange**->**All**},  
 ListPlotOptions->{**AxesOrigin**->{1,0},**PlotRange**->**All**} };  
 StatFormat::usage="StatFormat is an option for ShowStat that specifies in what format the statistics is to be presented.";  
 ListLinePlotOptions::usage="ListLinePlotOptions is an option for ShowStat that lets the user specify options for the ListLinePlot. ListLinePlot is only used in ShowStat when the option StatFormat is set to Plot. Any option of ListLinePlot can be used.";  
**95** ListPlotOptions::usage="ListPlotOptions is an option for ShowStat that lets the user specify options for ListPlot. ListPlot is only used in ShowStat when the option StatFormat is set to Plot, and when ShowStat presents statistics from a nonlinear principal component analysis with some factor treated as either single or ordinal. Any option of ListLinePlot can be used.";  
 ShowIterations::usage="ShowIterations[list] takes statistics from a PCA or a NLPCA and returns an interactive plot that shows the coordinates of the different iterations of the analysis performed. The statistics needed is contained in statlist, which is the output {protocollist, statlist}=PCA[Protocol[pd],dim,{v1,...,vm},Stat->True] or {protocollist,statlist}=NLPCA[Protocol[pd],dim,{f1,...,fn},{v1,...,vm},Stat->True]. That is the protocols that are usually the output of PCA and NLPCA are now contained in the first list in the output when Stat is set to true, and the statistics is contained in the second list. See Options>ShowIterations for a list of options.";  
**Options>ShowIterations**=  
 {Include->**All**,ListPlotOptions->{**ImageSize**->{300,200},**PlotRange**->**All**} };  
 Include::usage="Include is an option for ShowIterations that specifies what coordinates to plot. Possible values are All, Individuals and Modalities.";  
**100** Individuals::usage="Individuals is a possible value of the option Include for the function ShowIterations. " ;  
 GetStartX::usage="GetStartX[p1,{v1,v2}] takes a protocol  $p_1$ , returned from a principal component analysis or a nonlinear principal component analysis on a protocol  $p$ . The coordinates for the individuals from the first analysis must be contained in the variables with index  $\{v_1, v_2\}$  in  $p_1$ . It returns a list with coordinates for individuals to start a new analysis on  $p$  with. See Options[GetStartX] for a list of options.";  
**Options[GetStartX]**=**{Random->False}**;  
**105** GetEigenvalues::usage="GetEigenvalues[p] takes a protocol from a NLPCA or PCA, which contains the coordinates of the individuals from the analysis and hence also the eigenvalues, and displays information about the eigenvalues. See Options[GetEigenvalues] for a list of options.";

```

Options[ GetEigenvalues]={List->False};



---


110 (**ShowIterations , by Sandra**)
ShowIterations::error="The input to ShowIterations is not in a correct format.";
ShowIterations[ stat_List , OptionsPattern[]]:=Module[{XPlot , YPlot , n},
If [Length[stat[[3]]]==3&&Length[stat[[3]]]==4&&Length[stat[[3]]]==5 , Message[ShowStat::error]];
{XPlot , YPlot}=stat[[1;;2]];
115 n=stat[[3,1]];
Switch[OptionValue[Include],
All,
Return[ Manipulate[Show[ ListPlot[XPlot[[k]], OptionValue[ ListPlotOptions]], ListPlot[YPlot[[k]], OptionValue[ ListPlotOptions]]], {k,1,n,1}]]]
];
120 Individuals ,
Return[ Manipulate[ ListPlot[XPlot[[k]], OptionValue[ ListPlotOptions]], {k,1,n,1}]]
;
Modalities ,
Return[ Manipulate[ ListPlot[YPlot[[k]], OptionValue[ ListPlotOptions]], {k,1,n,1}]]
125 ];
];

Begin["`Private`"]
130



---


(*IndicatorMatrix , by Anna 2010-03-06*)
IndicatorMatrix[Protocol[pd_List], f_List] :=
135 Module[ {ff , indicatormatrix , rules , fnum , factorconstant },
ff = {};
For[i = 1, i<=Length[f], i++,
If[ StringQ[f[[i]]],
ff = Append[ff , Position[pd[[FACTORS]] , f[[i]]][[1,1]]],
140 ff = Append[ff , f[[i]]];
];
fnum = Plus@@Table[Length[pd[[FACTORS, ff [[i]], 2]]], {i,1,Length[ff]}];
factorconstant = {0};
145 For[i = 1, i<=Length[ff]-1, i++,
If[ Length[factorconstant]==1,
factorconstant = Append[factorconstant , Length[pd[[FACTORS, ff [[i]], 2]]]],
factorconstant = Append[factorconstant , Length[pd[[FACTORS, ff [[i]], 2]]]+
150 factorconstant[[i]]];
];
rules = Table[{i , factorconstant [[j]]+pd[[FACTORS, ff [[j]], 3]][[i]]}>1,{i,1,Length[pd[[FACTORS,1,3]]]}, {j,1,Length[f]}];
indicatormatrix = Normal[SparseArray[Flatten[ rules ],{Length[pd[[FACTORS,1,3]]],fnum},0]];
Return[indicatormatrix];
155 ]
];

(*2nd Indicatormatrix Indi , by Sandra & Oskar*)
Indi[Z_List , factorsL_List] :=
Module[ {factorsind , G},
160 factorsind = FoldList[Plus , 0 , factorsL];
G = Array[Z[[All , factorsind[[#]] + 1 ; factorsind[[# + 1]]]] &,
Length[factorsL]];
Return[G];
]

165 Indi[Protocol[pd_List], f_List] :=
Module[ {G, factorsL , tmp},
G = ConstantArray[0 , Length[f]];
Do[
170 tmp = Array[#, pd[[FACTORS, f[[i]], 3, #]]]>1 &,
Length[pd[[TEXTDATA]]];
G[[i]] = SparseArray[tmp],
{i, Length[f]};
];
factorsL = Array[Length[G[[#]]][[2]]]&, Length[G]];
175 Return[{factorsL , G}];
]

Indi[Protocol[pd_List], f_List , v_List] :=
Module[ {G, factorsL , tmp, variables},
180 G = ConstantArray[0 , Length[f]];
Do[
tmp = Array[#, pd[[FACTORS, f[[i]], 3, #]]]>1 &,
Length[pd[[TEXTDATA]]];
G[[i]] = SparseArray[tmp],

```

```

185      {i , Length[f]}  
];  
factorsL = Array[Length[G[[#]][[2]]] &, Length[G]];  
variables =  
Table[GetVariableValues[Protocol[pd], v[[i]]], {i, Length[v]}];  
190  Return[{factorsL, variables, G}];  
]  
  



---


195 (* *MCA, by GruppA/Anna*)  
MCA:=pmerror = "For one (or several) individuals, all modalities are set to passive";  
MCA:=ferror = "One (or several) of the modalities you have chosen as passive is not a part of the  
analysis";  
MCA:=rankerror = "Matrix rank is lower than the option value of Dimensions. Try to set Dimensions  
to a smaller value.";  
MCA:=axeserror = "One (or several) of the variables is not a coordinate axis.";  
MultipleCorrespondenceAnalysis[Protocol[pd>List,f_List,OptionsPattern[]]:=MCA[Protocol[pd],f,  
Dimensions->OptionValue[Dimensions],AxesName->OptionValue[AxesName],StandardCoordinates->  
OptionValue[StandardCoordinates],PassiveModalities->OptionValue[PassiveModalities],  
MakePassiveThreshold->OptionValue[MakePassiveThreshold],PrintPassiveModalities->OptionValue[  
PrintPassiveModalities]]  
200 MCA[Protocol[pd>List,f_List,OptionsPattern[]]:=  
Module[ {varname,newpd,ff,Z,B,rowtot,coltot,tot,R,M,Dr,Dw,Centroid,Dc,T,Bb,U,W,V,Nn,Mm,GS,RZ,  
FZ,dimensionnr,stcoordinates,tol,temp,temp1,temp2,tosmallmods,factorconstant,  
passivemodalities,memberlist,numvar},  
varname=OptionValue[AxesName];  
newpd=pd;  
ff={};  
205 For[i=1,i<=Length[f],i++,  
If[ StringQ[f[[i]]],  
ff=Append[ff,Position[newpd[[FACTORS]],f[[i]]][[1,1]]],  
ff=Append[ff,f[[i]]]  
];  
210 ];  
ff=Sort[ff]; (*Sorts all factors, important for passive modalities*)  
Z=IndicatorMatrix[Protocol[newpd],ff];  
Z=SparseArray[Z];  
B=Transpose[Z].Z;  
215 (*Correspondence Analysis*)  
rowtot=Total[B,{2}];  
coltot=Total[B];  
tot=Total[coltot];  
220 R=SparseArray[B/rowtot]; (*The Matrix of the row profiles*)  
M=SparseArray[rowtot/tot];  
Dr=DiagonalMatrix[M]; (*The Diagonal Matrix of the Masses*)  
Centroid=SparseArray[coltot/tot];  
Dw=SparseArray[DiagonalMatrix[1/Normal[Centroid]]]; (*The Diagonal Matrix of the  
Dimension Weights*)  
225 Dc=DiagonalMatrix[Centroid]; (*The Diagonal Matrix of the Centroid*)  
T=R-SparseArray[ConstantArray[Centroid,Dimensions[B][[1]]]]; (*The Profiles' deviations  
from the Centroid*);  
;  
(*Single Value Decomposition*)  
230 Dr=N[Dr];  
Dw=N[Dw];  
T=N[T];  
  

235 passivemodalities=OptionValue[PassiveModalities];  
If[OptionValue[MakePassiveThreshold]>=1,  
tol=OptionValue[MakePassiveThreshold];  
temp=Array[Position[Modalities[Protocol[pd],ff[[#1]],TableForm->False]][[All,3]],Range[1,  
tol][[#2]]]&,{Length[ff],tol}];  
tosmallmods={};  
240 Do[AppendTo[tosmallmods,{ff[[i]],#}&/@Flatten[temp[[i,All]]]],{i,Length[ff]}];  
tosmallmods=Partition[Flatten[tosmallmods],2];  
passivemodalities=Union[Join[passivemodalities,tosmallmods]];  
,  
tol=OptionValue[MakePassiveThreshold];  
temp1=Array[Select[Modalities[Protocol[pd],ff[[#1]],TableForm->False]][[All,4]],#<=tol  
&]&,{Length[ff]}];  
temp1=Union[Flatten[temp1]];  
temp2=Array[Position[Modalities[Protocol[pd],ff[[#1]],TableForm->False]][[All,4]],temp1  
[[#2]]]&,{Length[ff],Length[temp1]}];  
tosmallmods={};  
245 Do[AppendTo[tosmallmods,{ff[[i]],#}&/@Flatten[temp2[[i,All]]]],{i,Length[ff]}];  
tosmallmods=Partition[Flatten[tosmallmods],2];  
passivemodalities=Union[Join[passivemodalities,tosmallmods]];  
];  
If[OptionValue[PrintPassiveModalities],Print[passivemodalities]];];

```

```

255
If [ Length[ passivemodalities]===0,
dimensionnr = OptionValue[Dimensions];
Bb = Sqrt[Dr].T.Sqrt[Dw];
If [ MatrixRank[Bb]<dimensionnr, Message[MCA:: rankerror]; Abort[]];
{U,W,V} = SingularValueDecomposition[Bb,dimensionnr];
260 U = SparseArray[U];
W = SparseArray[W];
V = SparseArray[V];
Nn = Inverse[Sqrt[Dr]].U;
Mm = Inverse[Sqrt[Dw]].V;
(*F=Nn.W; *) (*Coordinates for the rows*)
(*G=Inverse[Dc].Mm.W; (*Coordinates for the Columns*)*)
GS = Dw.Mm; (*To Standard Coordinates for the columns*)
RZ = SparseArray[Z/Total[Z,{2}]];
stcoordinates = OptionValue[StandardCoordinates];
270 If [ stcoordinates ,
FZ = RZ.GS.Inverse[Sqrt[W]]; (*Transforms to Standard Coordinates for the rows of
the indicator matrix*)
AppendTo[newpd[[METADATA]], "StandardCoordinates"→True]; (*Stores that the protocol
contains Standard Coordinates*),
FZ = RZ.GS;(*Transforms to Principal Coordinates for the rows of the indicator
matrix*)
AppendTo[newpd[[METADATA]], "StandardCoordinates"→False]; (*Stores coordinate type
*)
];
275 (* Stores which variables that are principal axes*)
numvar=Length[pd[[VARIABLES]]];
AppendTo[newpd[[METADATA]], "PrincipalAxes"→Range[numvar+1,numvar+dimensionnr ]],
,
memberlist = {};
280 Do[ memberlist = Append[ memberlist ,MemberQ[ ff , passivemodalities [[ i ,1]]], { i ,Length[
passivemodalities }]];
If [ MemberQ[memberlist ,False] ,
Message[MCA:: ferror ];
Abort[]];
];
285 passivemodalities = Reverse[Sort[ passivemodalities ]];
factorconstant = {0};
Do[ factorconstant = Append[ factorconstant , factorconstant [[ i ]]+Length[pd[[FACTORS, ff [[ i
]],2]]], { i ,Length[ ff ]}];
290 Bb = Sqrt[Dr].T.Sqrt[Dw];
(
*Drop rows and columns corresponding to passive modalities*
Do[Bb = Drop[Bb, { factorconstant [[ Position[ff , passivemodalities [[ i ,1]]][[1 ,1]]]]+
passivemodalities [[ i ,2]]} ],
{ factorconstant [[ Position[ff , passivemodalities [[ i ,1]]][[1 ,1]]]]+ passivemodalities [[ i
,2]]}, { i ,Length[ passivemodalities }]];
Do[Dr = Drop[Dr, { factorconstant [[ Position[ff , passivemodalities [[ i ,1]]][[1 ,1]]]]+
passivemodalities [[ i ,2]]} ],
{ factorconstant [[ Position[ff , passivemodalities [[ i ,1]]][[1 ,1]]]]+ passivemodalities [[ i
,2]]}, { i ,Length[ passivemodalities }]];
300 Do[Dw = Drop[Dw, { factorconstant [[ Position[ff , passivemodalities [[ i ,1]]][[1 ,1]]]]+
passivemodalities [[ i ,2]]} ],
{ factorconstant [[ Position[ff , passivemodalities [[ i ,1]]][[1 ,1]]]]+ passivemodalities [[ i
,2]]}, { i ,Length[ passivemodalities }]];
(
*Drop columns corresponding to passive modalities in the indicator profile matrix*
RZ = SparseArray[Z/Total[Z,{2}]]; (*Create row profile matrix*)
Do[RZ = Drop[RZ,None, { factorconstant [[ Position[ff , passivemodalities [[ i ,1]]][[1 ,1]]]]+
passivemodalities [[ i ,2]]}], { i ,Length[ passivemodalities }]];
305 If [ MemberQ[Total[RZ,{2}],0],
Message[MCA:: pmerror ];
Abort[]];
];
310 (
*SVD*)
dimensionnr = OptionValue[Dimensions];
If [ MatrixRank[Bb]<dimensionnr, Message[MCA:: rankerror]; Abort[]];
{U,W,V} = SingularValueDecomposition[Bb,dimensionnr];
U = SparseArray[U];
W = SparseArray[W];
V = SparseArray[V];
Nn = Inverse[Sqrt[Dr]].U;
Mm = Inverse[Sqrt[Dw]].V;
315 (*F=Nn.W; *) (*Coordinates for the rows*)
(*G=Inverse[Dc].Mm.W; *) (*Coordinates for the Columns*)
GS = Dw.Mm; (*To Standard Coordinates for the columns*)
stcoordinates = OptionValue[StandardCoordinates];
If [ stcoordinates ,

```

```

FZ = (RZ.GS-Sqrt[ConstantArray[Dr.ConstantArray[1,Length[Dr]],Length[RZ]]].V) .
    Inverse[Sqrt[W]]; (*Transforms to Standard Coordinates for the rows of the
    indicator matrix*)
newpd[[METADATA]] = Append[newpd[[METADATA]], "StandardCoordinates"→True];
FZ = RZ.GS-Sqrt[ConstantArray[Dr.ConstantArray[1,Length[Dr]],Length[RZ]]].V;(*
    Transforms to Principal Coordinates for the rows of the indicator matrix*)
newpd[[METADATA]] = Append[newpd[[METADATA]], "StandardCoordinates"→False];
330 ];
(* Stores which variables that are principal axes*)
numvar=Length[pd[[VARIABLES]]];
AppendTo[newpd[[METADATA]], "PrincipalAxes"→Range[numvar+1,numvar+dimensionnr]];
335 ];
(* Stores singular values for the indicator matrix*)
newpd[[METADATA]] = Append[newpd[[METADATA]], "SingularValues"→ConstantArray[1,Length[W]].
    Sqrt[W]];
FZ = Normal[FZ];
340 newpd = Protocol[newpd];
For[i = 1,i<=dimensionnr,i++,
    newpd = AddVariable[newpd, varname<>" Axis "<> ToString[i], FZ[[All, i]]];
Return[newpd];
345 ]

```

---

```

(**GetSingularValues , by Anna**)
GetSingularValues[Protocol[pd_List]] :=
Module[{singval},
350     singval = "SingularValues"/.pd[[METADATA]];
     Return[singval];
]

```

---

```

355 (**GetCoordinateStatus , by Anna**)
GetCoordinateStatus[Protocol[pd_List]] :=
Module[{cstatus},
    cstatus = "StandardCoordinates"/.pd[[METADATA]];
360     Return[cstatus];
]

```

---

```

(**GetPrincipalAxes , by Anna**)
GetPrincipalAxes[Protocol[pd_List]] :=
Module[{principalaxes},
365     principalaxes="PrincipalAxes"/.pd[[METADATA]];
     Return[principalaxes];
]

```

---

```

(**ModalityCoordinates , by Anna**)
370 ModalityCoordinates[Protocol[pd>List],f_Integer, axis_List, OptionsPattern[]]:=ModalityCoordinates[
    Protocol[pd],{f},axis,Rescale→OptionValue[Rescale],StandardCoordinates→OptionValue[
        StandardCoordinates],MatrixForm→OptionValue[MatrixForm]]
ModalityCoordinates[Protocol[pd_List],f_List, axis_List, OptionsPattern[]]:=Module[{ff,aa,singval,Z,FS,CC,G,sc,cstatus,positions,principalaxes,actualaxes,newstatus,mf,
    modnum, facdata,moddata,modsink,newpd},
375 (* If some of the inputs are strings , this converts to id*)
    ff = {};
    aa = {};
    Do[If[StringQ[f[[i]]],
        ff = Append[ff,Position[pd[[FACTORS]],f[[i]]][[1,1]]],
        ff = Append[ff,f[[i]]]],
380     ],
    {i,Length[f]}];
    Do[If[StringQ[axis[[i]]],
        aa = Append[aa,Position[pd[[VARIABLES]],axis[[i]]][[1,1]]],
        aa = Append[aa,axis[[i]]]],
385     ],
    {i,Length[axis]}];
    Z = IndicatorMatrix[Protocol[pd],ff];(*Creates an indicatormatrix for given factors*)
    CC = Transpose[Z]/Total[Z]; (*Column profiles (at rows!) *)
    FS = Transpose[pd[[VARIABLES,aa,2]]]; (*Variable values for axis*)
    If[OptionValue[Rescale],
        singval = GetSingularValues[Protocol[pd]];
        sc = OptionValue[StandardCoordinates];
        cstatus = GetCoordinateStatus[Protocol[pd]];
390
        principalaxes=GetPrincipalAxes[Protocol[pd]];
        positions=Position[principalaxes,aa[[#]]]&/@Range[Length[aa]];
        If[Length[aa]≠Length[Flatten[positions]],Message[MCA::axeserror];Abort[]];
395
        ];
    ];

```

```

actualaxes=Flatten[positions];
400  If[ sc,
      If[ cstatus ,
          G = CC.FS.DiagonalMatrix[1/singval [[ actualaxes ]]];
          newstatus = True; (*to standard coord. if FS is standard coord.*),
          G = CC.FS.DiagonalMatrix[1/singval [[ actualaxes ]]].DiagonalMatrix[1/singval [[
              actualaxes ]]];
          newstatus = True;
        ];(*to standard coord. if FS is prncipal coord.*),
      If[ cstatus ,
          G = CC.FS;
          newstatus = False; (*to principal coord. if FS i standard coord.*),
          G = CC.FS.DiagonalMatrix[1/singval [[ actualaxes ]]];
          newstatus = False;
        (*to principal coord. if FS is principal coord.*)
      ];
    ];
415  G=CC.FS;
];
mf = OptionValue[MatrixForm];
If[ mf,
  (*Returns matrix if MatrixForm is true*)
  Return[G];
  modnum = Array[Length[pd[[FACTORS, ff[[#]],2]]]& ,Length[ff]]; (*number of modalities in
  each factor*)
  facdata = Array[ConstantArray[pd[[FACTORS, ff[[#]],1]],Length[pd[[FACTORS, ff
  [[#]],2]]]]& ,Length[ff]];
  moddata = Array[pd[[FACTORS, ff[[#]],2]]& ,Length[ff]];
  modsink = Total[Z];
  newpd = CreateProtocol[ConstantArray[{},{Total[modnum]}];
 420  If[OptionValue[Rescale],newpd[[METADATA,1]] = Append[newpd[[METADATA,1]],"
    StandardCoordinates"→newstatus]; (*[[METADATA,1]] required since newpd is on
    Protocol[]-form*)
  Do[newpd = AddVariable[newpd,"Modality " <> pd[[VARIABLES, aa[[i]],1]],G[[All,i]]],{i,
    Dimensions[G][[2]]];
  newpd = AddFactor[newpd,"Factors",Flatten[facdata]];
  newpd = AddFactor[newpd,"Modalities",Flatten[moddata]];
  newpd = AddVariable[newpd,"Size",modsink];
  Return[newpd];
  (*Returns the coordinates as a protocol if MatrixForm is false*)
];
435
];

```

```

(** CrossTabulate, by Sandra and Sverker**)
440 CrossTabulate::nerror = "You have not given two correct factors";
DependencePVal[v_Real, dist_] :=
  Module[{x}, x = CDF[dist, v]; Return[N[If[x > 0.5, x, 1 - x]]]];
DependencePVal[v$-List, dist_] := DependencePVal[#, dist] & /@ vs;

445 CrossTabulate[Protocol[pd_List], factors_List, OptionsPattern[CrossTabulate] ] :=
  Module[{f, rules, factorsind, allmods, usedmods, ct, at, tf, rows, cols, ctcolumntitled, cttitled, rc, cc,
    relrc, relcc, indeptab, relct, dist, mods1, mods2},(*Finds the indexes of the factors*)
  f = Factors[Protocol[pd],TableForm→False][[All,1;;2]];
  rules = Array[f[[#1,2]]→f[[#1,1]]& ,Length[f]];
  factorsind = factors /. rules;
 450  (*Checks that the factors exist and that exactly two factors was supplied to the function
  *)
  If[ Length[factors]!=2,
    Message[CrossTabulate::nerror];
    Return["he"]
  ];
 455  If[ !ArrayQ[factorsind,_,IntegerQ]||MemberQ[Positive[factorsind],False],
    Message[CrossTabulate::nerror];
    Return["hi"]
  ];
 460  If[ Max[factorsind]>Length[pd[[FACTORS]]],
    Message[CrossTabulate::nerror];
    Return[]
  ];
  (*Creates the contingency table*)
  allmods = Outer@@Prepend[{pd[[FACTORS, factorsind[[1]],2]],pd[[FACTORS, factorsind
  [[2]],2]]}],{##}&;
  usedmods = Array[pd[[FACTORS, factorsind[[#2]],2],pd[[FACTORS, factorsind[[#2]],3,#1]]]]&,{Length[pd[[TEXTDATA]]],Length[factorsind]}];
  ct = Array[Count[usedmods[[#1]][[#2]]]&,{Length[pd[[FACTORS, factorsind[[1]],2]]],Length[pd[[FACTORS, factorsind[[2]],2]]]}];
  (* Compute statistic according to OptionValue[ Statistic]*)
  n = Plus@@Plus@@ct;
  rc = Plus@@ct; (*row counts*)
  cc = Plus@@Transpose[ct]; (*column counts*)
470

```

```

relrc = rc/n; (* relative row counts*)
relcc = cc/n; (* relative column counts*)
indeptab = Outer[Times, relcc, relrc]//N;
475    relct = ct/n;
dist = NormalDistribution[0,1];
Switch[OptionValue[Statistic],
      Count, ct = ct,
      RowPercentage, ct = N[#/((Plus@@#)&/@ct),
      ColumnPercentage, ct = N[Transpose[(#/Plus@@#)&/@Transpose[ct]]],*
      StandardResidual, ct = (relct*n-indeptab*n)/Sqrt[indeptab*n],
      ChiSquare, ct = (relct*n-indeptab*n)^2/(indeptab*n),
      MeanChiSquare, ct = (relct*n-indeptab*n)^2/(indeptab*n*Length[ct]*Length[ct
      [[1]]]),
      ExpectedValue, ct = indeptab*n,
      DependencyPValue, ct = DependencePVal[#, dist]&/@((relct*n-indeptab*n)/ Sqrt[
      indeptab*n]),
      PercentageDeviation, ct = 100*(relct-indeptab)/indeptab];
500    mods1 = pd[[FACTORS, factorsind[[1]], 2]];
    mods2 = pd[[FACTORS, factorsind[[2]], 2]];
    If[ OptionValue[ReturnList],
        Return[Table[{{mods1[[i]], factorsind[[1]]}, {mods2[[j]], factorsind[[2]]}, ct[[i,j]]}, {i,
        Length[mods1]}, {j, Length[mods2]}]]
    ];
    If[ OptionValue[ReturnAssociation],
        Return[Plus@@Flatten[ct]]
    ];
(* Returns the table as a matrix. Depending on the option AddTitles; with the titles of the rows
   and columns added to the matrix, or in separate lists. Depending on the option "TableForm
   "; in TableForm, or not in TableForm*)
495    at = OptionValue[AddTitles];
    tf = OptionValue[TableForm];
    If[ tf,
        rows = Style[StringInsert[#, " ", {-1}], Bold]&/@pd[[FACTORS, factorsind[[1]], 2]]; (*
        Longer titles , for TableForm true*)
        cols = Style[StringInsert[#, " ", {-1}], Bold]&/@pd[[FACTORS, factorsind[[2]], 2]],
        rows = Style[#, Bold]&/@pd[[FACTORS, factorsind[[1]], 2]]; (*Just the titles , for
        TablesForm false*)
        cols = Style[#, Bold]&/@pd[[FACTORS, factorsind[[2]], 2]]
    ];
    If[ at,(*Adding the titles to the contingency table*)
        ctcolumntitled = Prepend[ct, cols];
        cttitled = Prepend[Transpose[ctcolumntitled], Prepend[rows, ""]]/>Transpose;
        If[ tf,
            Return[cttitled/>TableForm],
            Return[cttitled]
        ],(*Contingency table and titles in separate lists*)
        If[ tf,
            Return[TableForm[ct, TableHeadings->{rows, cols}]],
            Return[{ct, rows, cols}]
        ]
    ]
515    ];
];

```

```

(**CA, by grupp A/Oskar**)
520 CA[Protocol[pd_List], f1_Integer, f2_Integer, opts___?OptionQ] :=
Module[ {dim, newpd, p, p1, p2, data, profiles, deviations, n, rowsums, colsums, centroid, axes, inertia, Dw
,Dq, P, B, U, D, V, F, G},
dim = Dimensions/.Flatten[{opts}]/.Options[CA];
axes = AxesName/.Flatten[{opts}]/.Options[CA];
data = CrossTabulate[Protocol[pd], {f1, f2}][[1]];
525    P = data/n;
n = Total @ Flatten[data];
rowsums = Total @ Transpose[data];
colsums = Total @ data;
profiles = Divide[data, rowsums];
Dw = DiagonalMatrix[rowsums/n];
Dq = DiagonalMatrix[n/colsums];
centroid = Total @ Dw.profiles;
deviations = profiles - ConstantArray[centroid, Length[profiles]];
inertia = Total[Total[deviations[[#]]*(1/centroid)*deviations[[#]]]&/@Range[Length[
deviations]]*Diagonal[Dw]]//N;
535    B = Sqrt[Dw].deviations.Sqrt[Dq];
If[ dim == All,
    dim = MatrixRank[B]
];
{U,D,V} = SingularValueDecomposition[N[B], dim];
F = Inverse[Sqrt[Dw]].U.D;
G = DiagonalMatrix[1/centroid].Transpose[P].F.Inverse[D];
newpd = pd;
p = Protocol[newpd];
p1 = Group[p, f1];

```

```

545 p2 = Group[p,f2];
p1 = Fold[AddVariable[#1,StringJoin[axes,ToString@#2],Transpose[F][[#2]]]&, p1, Range[dim
    ];
p2 = Fold[AddVariable[#1,StringJoin[axes,ToString@#2],Transpose[G][[#2]]]&, p2, Range[dim
    ];
p1[[1,1]] = Append[p1[[1,1]], "SingularValues" -> Diagonal[D]];
p1[[1,1]] = Append[p1[[1,1]], "Inertia" -> inertia];
p2[[1,1]] = Append[p2[[1,1]], "SingularValues" -> Diagonal[D]];
p2[[1,1]] = Append[p2[[1,1]], "Inertia" -> inertia];
Return[{p1,p2}];
];

555 CA[data_List, opts___?OptionQ] :=
Module[{sv,dim,profiles,deviations,n,rowsums,colsums,centroid,Dw,Dq,P,B,U,D,V,F},
    dim = Dimensions /. Flatten[{opts}] /. Options[CA];
    sv = ReturnSingularValues /. Flatten[{opts}] /. Options[CA];
560 P = data/n;
n = Total@Flatten[data];
rowsums = Total@Transpose[data];
colsums = Total@data;
profiles = Divide[data,rowsums];
565 Dw = DiagonalMatrix[rowsums/n];
Dq = DiagonalMatrix[n/colsums];
centroid = Total@Dw.profiles;
deviations = profiles - ConstantArray[centroid,Length[profiles]];
B = Sqrt[Dw].deviations.Sqrt[Dq];
570 If[ dim == All,
    dim = MatrixRank[B]
];
{U,D,V} = SingularValueDecomposition[N[B],dim];
F = Inverse[Sqrt[Dw]].U.D;
If[ sv,
    Return[{F,D}],
    Return[F]
];
];
580 
```

---

```

(*FactorContributions , by Oskar*)
585 FactorContributions[Protocol[pd_List],vars_List,facs_List,opts___?OptionQ] :=
Module[{tmp,flength,vlength,fnames,rel,axesnames,modcoords,modweights,modconts,fconts},
    rel = RelativeContributions/.Flatten[{opts}]/.Options[FactorContributions];
    modcoords = {};
    modweights = {};
    flength = Length[facs];
    vlength = Length[vars];
590     (* Get coordinates for modalities by calculating modality mean points *)
    Do[
        tmp = Group[Protocol[pd],facs[[i]], KeepVariables -> {{Mean,vars}}];
        modcoords = Append @@ {modcoords, Array[GetVariableValues[tmp,#]&,vlength]}, {i,flength}
    ];
595     (* Get modality weights *)
    Do[
        tmp = Array[Count[GetModalityIds[Protocol[pd],facs[[i]]],#]&,Length[GetModalityValues[
            Protocol[pd],facs[[i]]]]];
        600 tmp = tmp/Total[tmp];
        modweights = Append[modweights,tmp],
        {i,flength}
    ];
    modweights = modweights/flength;
605     (* Get modality contributions *)
    tmp = modcoords^2;
    modconts = Array[ConstantArray[modweights[[#]],vlength]*tmp[[#]]&,flength];
610     (* Get factor contributions *)
    fconts = Array[Total /@ modconts[[#]]&,flength];
    fconts = Round[fconts,0.001];
    If[ rel,
        fconts = Transpose[Transpose[fconts]/Total[fconts]];
615        fconts = Round[fconts,0.01];
    ];
620     (* Construct table *)
    fnames = Array[pd[[FACTORS,facs[[#]],1]]&,flength];
    axesnames = Array[StringJoin["Axis ",ToString[#]]&,vlength];
    Return @ TableForm[fconts,TableHeadings -> {fnames,axesnames}, TableAlignments -> Center];
]; 
```

---

```

625 (**ModalityContributions, by Oskar**)
ModalityContributions[Protocol[pd_List], v_Integer, facs_List, opts___?OptionQ] :=
Module[ {tmp, flength, rel, fnames, modnames, modcoords, modweights, modconts, meancont, tables,
returnmean, printerfriendly, concise, left, right, cleft, cright, ctotals, accuracy, threshold},
rel = RelativeContributions/.Flatten[{opts}]/.Options[ModalityContributions];
returnmean = ReturnMean/.Flatten[{opts}]/.Options[ModalityContributions];
630 printerfriendly = PrinterFriendly/.Flatten[{opts}]/.Options[ModalityContributions];
concise = Concise/.Flatten[{opts}]/.Options[ModalityContributions];
threshold = Threshold/.Flatten[{opts}]/.Options[ModalityContributions];
modcoords = {};
modweights = {};
modconts = {};
flength = Length[facs];
(* Calculate modality mean points *)
635 Do[
tmp = Group[Protocol[pd], facs[[i]], KeepVariables -> {{Mean, {v}}}]},
modcoords = Append @@{modcoords, GetVariableValues[tmp, 1]}, {i, flength}
];
(* Calculate modality weights *)
640 Do[
tmp = Array[Count[GetModalityIds[Protocol[pd], facs[[i]]], #]&, Length[GetModalityValues[
Protocol[pd], facs[[i]]]]];
tmp = tmp/Total[tmp];
modweights = Append[modweights, tmp], {i, flength}
];
645 modweights = modweights/flength//N;
modweights = Round[modweights, 0.001];
(* Calculate modality contributions *)
650 If[ rel,
modconts = modconts/Total[Flatten[modconts]];
];
655 If[ threshold==0,
meancont = Total[Flatten[modconts]]/Length[Flatten[modconts]],
meancont = threshold
];
660 If[ returnmean,
Return[meancont]
];
665 ctotals = Total/@modconts;
modconts = Round[modconts, 0.001];
(* Construct tables *)
670 fnames = Array[pd[[FACTORS, facs[[#]], 1]]&, flength];
modnames = Array[GetModalityValues[Protocol[pd], facs[[#]]]&, flength];
675 If[ !concrete,
If[ printerfriendly,
tmp = Position[modconts, x_ /;x>meancont];
(modconts[[#, #2]] = Subscript[modconts[[#, #2]], "*"])& @@ tmp;
];
If[ !printerfriendly,
tmp = Position[modconts, x_ /;x>meancont];
(modconts[[#, #2]] = Style[modconts[[#, #2]], FontColor->Red])& @@ tmp;
];
680 tables = Array[
TableForm[Transpose[{modnames[[#]], modweights[[#]], Round[modcoords[[#]], 0.001],
modconts[[#]]}], TableHeadings->{None, {fnames[[#]], "Weight", "Coordinate",
"Contribution"} }]&,
flength
];
685 Return[tables];
];
If[ concise,
left = Flatten/@Array[Intersection[Position[modconts[[#]], x_ /;x>meancont], Position[
modcoords[[#]], x_ /;x<0]]&, flength];
right = Flatten/@Array[Intersection[Position[modconts[[#]], x_ /;x>meancont], Position[
modcoords[[#]], x_ /;x>0]]&, flength];
690 cleft = modconts[[#, left[[#]]]]&/@Range[flength];
cleft = Flatten[Total/@cleft];
cright = modconts[[#, right[[#]]]]&/@Range[flength];
cright = Flatten[Total/@cright];
accuracy = Round[100*(cleft+cright)/ctotals];
tmp = StringJoin[fnames[[#]], " (", ToString[accuracy[[#]]], "%)"]&/@Range[flength];
left = Array[modnames[[#, left[[#]]]]&, flength];
right = Array[modnames[[#, right[[#]]]]&, flength];
695 Return@Row[{TableForm[left, TableHeadings->{Style[#, Bold]&/@tmp, {Style["Left", Italic
]}}, TableForm[right, TableHeadings->{None, {Style["Right", Italic]} }]}, " ", Frame
->True];

```

```

700      ];
];



---


705 (**FactorAssociationGraph, by Sverker**)
FactorAssociationGraph[Protocol[p_], fs_List, s_, t_, OptionsPattern[FactorAssociationGraph]] :=
Module[ {fps ,selfps ,labpairs },
fps = Subsets[fs ,{2}];
selfps = Select[fps ,CrossTabulate[Protocol[p],#,Statistic->s ,ReturnAssociation->True]>t &];
If[ OptionValue[ShowIndex],
710   labpairs = selfps /. {a_ ,b_}:>(ToString[a]~~"-~~p[[FACTORS,a,1]]->ToString[b]~~"-~~p
[[FACTORS,b,1]]),
labpairs = selfps /. {a_ ,b_}:>(p[[FACTORS,a,1]]->p[[FACTORS,b,1]])
];
GraphPlot[ labpairs ,Method->OptionValue[Method] ,VertexRenderingFunction ->(Text[#2,#1,
Background->White]&)]
];
715



---


720 (**ModalityAssociationGraph, by Sverker**)
ModalityAssociationGraph[Protocol[p_], fs_List ,s_ ,t_?NumberQ, OptionsPattern [
ModalityAssociationGraph]] :=
Module[ {pairs ,deps ,labpairs },
pairs = Subsets[fs ,{2}];
deps = SortBy[Select[Flatten[CrossTabulate[Protocol[p],#,Statistic->s ,ReturnList->True]&/
@pairs ,2],#[[3]]>t &], -#[[3]]&];
If[ OptionValue[ShowIndex],
725   labpairs = deps /. {{a_ ,b_},{c_ ,d_},e_}:>(ToString[b]~~"-~~a->ToString[d]~~"-~~c) ,
labpairs = deps /. {{a_ ,b_},{c_ ,d_},e_}:>(a->c)
];
GraphPlot[ labpairs ,Method->OptionValue[Method] ,VertexRenderingFunction ->(Text[#2,#1,
Background->White]&)]
];
725



---


730 (**PCA, by Sandra**)
PCA::errorX="The matrix you supplied with coordinates for the individuals does not have the
correct dimensions";
PCA::variableerror="One (or more) of the variables does not exist.";
PCA[Protocol[pd_List],dim_Integer,v_List,OptionsPattern[]]:=Module[{variables,NI,r,X,W,maxiter,
XPlot,YPlot,error,Xprim,Yprim,Q,B,H,Loss,n,DMsquared,eigenvalues},
(*Check the input to PCA*)
735 If[!ArrayQ[v,_,IntegerQ]||MemberQ[Positive[v],False],Message[PCA::variableerror];Return[]];
If[Max[v]>Length[pd[[VARIABLES]]],Message[PCA::variableerror];Return[]];
(*Get the variable values from the protocol*)
variables=Table[GetVariableValues[Protocol[pd],v[[i]]],{i,Length[v]}];
735

740 NI=Dimensions[variables][[2]];r=Length[v];
(*Get the startmatrix with coordinates for the individuals*)
If[Length[OptionValue[StartX]]==0,
X=RandomReal[{-10,10},{NI,dim}],
If[Dimensions[OptionValue[StartX]]!= {NI,dim},Message[PCA::errorX];Return[]];
745 X=OptionValue[StartX];
];
W=X-ConstantArray[Total@X/NI,NI];X=Sqrt[NI]*Orthogonalize[Transpose[W]]//Transpose;
(* initialize some variables*)
maxiter=OptionValue[MaxIterations];
750 {XPlot,YPlot,error,Loss}=ConstantArray[ConstantArray[0,maxiter+1],4];XPlot[[1]]=X;
Yprim=ConstantArray[0,r];Q=variables//N;Q=Sqrt[NI]*Normalize/@Array[Q[[#]]-Total@Q[[#]]/Length[Q
[[#]]]&,r];
n=1;

(* Alternating least squares algorithm to get the coordinates*)
755 While[n<maxiter,
n++;
(*Get new coordinates for the modalities*)
Yprim=ConstantArray[X,r];
B=Flatten[Array[{Transpose[Yprim[[#]]].Q[[#]]}/(Q[[#]].Transpose[{Q[[#]]}])&,r],1];
760 Yprim=Array[Transpose[{Q[[#]]}].{B[[#]]}&,r];
(*Get new coordinates for the individuals*)
Xprim=(Total@Yprim)/r;
W=Xprim-ConstantArray[Total@Xprim/NI,NI];
X=Sqrt[NI]*Orthogonalize[Transpose[W]]//Transpose;
765 (*Saving the coordinates in each step of the iteration*)
XPlot[[n]]=X;YPlot[[n]]=Yprim;

(* Calculate the loss*)
H=Array[X-Yprim[[#]]&,r]^2;
770 Loss[[n]]=Total@Total@Total@H/r;
(*Check convergence criterion*)
error[[n]]=Max[Abs[(XPlot[[n-1]]-XPlot[[n]])]];

```

```

If [error [[n]]<=OptionValue[Convergence],Break[]];
];
775 (*Calculate the squared discrimination measures and the eigenvalues of the solution*)
DMsquared=Array[Diagonal[Transpose[Yprim[[#]]].Yprim[[#]]]&,r]/NI;
eigenvalues=Total@(B^2)/Length[B];

780 (*Return protocols with the coordinates and component loadings and possibly discrimination
measures. If option Stat is True return extra statistics about the analysis*)
If[OptionValue[Stat],
Return[{ToProtocol[Protocol[pd],dim,{},{},v,B,XPlot[[n]],YPlot[[n]],DMsquared,{eigenvalues,{}},
eigenvalues},OptionValue[DiscriminationInfo]],
{XPlot[[2;;n]],YPlot[[2;;n]],[{n-1,error[[2;;n]],Loss[[2;;n]]}]},],
Return[ToProtocol[Protocol[pd],dim,{},{},v,B,XPlot[[n]],YPlot[[n]],DMsquared,{eigenvalues,{}},
eigenvalues},OptionValue[DiscriminationInfo]]];
];
785 (*NLP PCA by Sandra*)
NLP CA::inputerror="You must supply at least one factor";
NLP CA::factorerror="One (or more) of the factors does not exist.";
NLP CA::variableerror="One (or more) of the variables does not exist.";
NLP CA::restrictedfactorserror1="One (or more) of the list of ordinal or single factors you
supplied were not a part of the list of factors.";
NLP CA::restrictedfactorserror2="The lists with ordinal and single factors overlap.";
NLP CA[Protocol[pd_List],dim_Integer,f_List,v_List,OptionsPattern[]]:=Module[{factorsL,variables,G,
XPlot,YPlot,factorssingle,factorsordinal,n,iterYL,error,Loss,LossY,Q,B,DMsquared,eigenvalues,
eigenvaluesf,eigenvaluesv,singlefactors,fm},
795 (*Check the input to NLP CA*)
If[!ArrayQ[f,-,IntegerQ]||MemberQ[Positive[f],False],Message[NLP CA::factorerror];Return[]];
If[Max[f]>Length[pd[[FACTORS]]],Message[NLP CA::factorerror];Return[]];
If[Length[f]==0,
Message[NLP CA::inputerror];Return[]];
800 If[MemberQ[MemberQ[f,#]&/@Union[OptionValue[Single],OptionValue[Ordinal]],False],Message[NLP CA:::
restrictedfactorserror1];Return[]];
If[Intersection[OptionValue[Ordinal],OptionValue[Single]]!={},Message[NLP CA:::
restrictedfactorserror2];Return[]];
(*Get the indicatormatrix , number of modalities in each factor and possibly the variables from the
protocol*)
805 If[Length[v]!=0,
If[!ArrayQ[v,-,IntegerQ]||MemberQ[Positive[v],False],Message[NLP CA::variableerror];Return[]];
If[Max[v]>Length[pd[[VARIABLES]]],Message[NLP CA::variableerror];Return[]];
{factorsL,variables,G}=Indi[Protocol[pd],f,v],
{factorsL,G}=Indi[Protocol[pd],f];variables={};
];
810 (*Get the indexes of the constrained factors*)
factorssingle=Flatten[Position[f,#]&/@OptionValue[Single],2];factorsordinal=Flatten[Position[f
,#]&/@OptionValue[Ordinal],2];
factorssingle=Union[factorssingle,factorsordinal];
815 (*Perform the analysis*)
{XPlot,YPlot,Q,B,DMsquared,{n,iterYL,error,Loss,LossY}}=NLP CAInner[G,dim,factorsL,factorssingle,
variables,MaxIterationsOuter->OptionValue[MaxIterationsOuter],MaxIterationsInner->OptionValue[
MaxIterationsInner],ConvergenceOuter->OptionValue[ConvergenceOuter],ConvergenceInner->
OptionValue[ConvergenceInner],StartX->OptionValue[StartX],Ordinal->factorsordinal,
OrdinalFunction->OptionValue[OrdinalFunction]];
(*Calculate the eigenvalues of the solution*)
fm=Complement[Range[Length[factorsL]],factorssingle];
820 singlefactors=Union[OptionValue[Single],OptionValue[Ordinal]];
If[f!=singlefactors,
eigenvaluesf=Total@DMsquared[[fm]]/Length[DMsquared]];
];
If[v!={}||singlefactors!={},
825 eigenvaluesv=Total@(B^2)/Length[DMsquared];
];
If[Length[eigenvaluesv]==Length[eigenvaluesf],
eigenvalues=eigenvaluesv+eigenvaluesf,
If[Length[eigenvaluesv]>Length[eigenvaluesf],
830 eigenvalues=eigenvaluesv,
eigenvalues=eigenvaluesf
];
];
835 (*Return protocols with the coordinates , possibly component loadings and possibly discrimination
measures. If option Stat is true also return extra statistics about the analysis*)
If[OptionValue[Stat],
If[Length[factorssingle]==0,
If[Length[v]==0,
Return[{ToProtocol[Protocol[pd],dim,f,singlefactors,v,B,XPlot[[n]],YPlot[[n]],DMsquared,
eigenvalues,OptionValue[DiscriminationInfo]],[{XPlot,YPlot,{n,error,Loss}}}],
840 Return[{ToProtocol[Protocol[pd],dim,f,singlefactors,v,B,XPlot[[n]],YPlot[[n]],DMsquared,{


```

```

eigenvalues , eigenvaluesf , eigenvaluesv } , OptionValue[ DiscriminationInfo ] ] , { XPlot , YPlot , { n , error ,
Loss , LossY } } } ]
],
Return[ { ToProtocol[ Protocol[ pd ] , dim , f , singlefactors , v , B , XPlot[ [ n ] ] , YPlot[ [ n ] ] , DMsquared , {
eigenvalues , eigenvaluesf , eigenvaluesv } , OptionValue[ DiscriminationInfo ] ] , { XPlot , YPlot , { n , iterYL ,
error , Loss , LossY } } } ]
],
Return[ ToProtocol[ Protocol[ pd ] , dim , f , singlefactors , v , B , XPlot[ [ n ] ] , YPlot[ [ n ] ] , DMsquared , {
eigenvalues , eigenvaluesf , eigenvaluesv } , OptionValue[ DiscriminationInfo ] ] ]
845 ]
];

NPCA[ Protocol[ pd_List ] , dim_Integer , f_List , OptionsPattern[] ]:=NPCA[ Protocol[ pd ] , dim , f , { } ,
MaxIterationsOuter:>OptionValue[ MaxIterationsOuter ] , MaxIterationsInner:>OptionValue[ MaxIterationsInner ] ,
ConvergenceOuter:>OptionValue[ ConvergenceOuter ] , ConvergenceInner:>OptionValue[ ConvergenceInner ] ,
StartX:>OptionValue[ StartX ] , Single:>OptionValue[ Single ] , Ordinal:>OptionValue[ Ordinal ] ,
OrdinalFunction:>OptionValue[ OrdinalFunction ] , Stat:>OptionValue[ Stat ] ,
DiscriminationInfo:>OptionValue[ DiscriminationInfo ] ];

850 (*ShowStat by Sandra*)
ShowStat::error="The input to ShowStat is not in a correct format.";
ShowStat[ stat_List , OptionsPattern[] ]:=Module[{ n , iterYL , error , Loss , LossY , labelinner , labelouter },
If[ Length[ stat[[3]] ]!=3 && Length[ stat[[3]] ]!=4 && Length[ stat[[3]] ]!=5 , Message[ ShowStat::error ] ];
Switch[ Length[ stat[[3]] ] ,
855 3 ,
{ n , error , Loss }=stat[[3]];
Switch[ OptionValue[ StatFormat ] ,
Table ,
Return[ TableForm[ { { Range[n] , error , Loss } } , TableHeadings->{ None , { " Iteration " , " Error " , " Loss " } } ] ] ,
860 Plot ,
Return[ { ListLinePlot[ { Range[n] , error } // Transpose , OptionValue[ ListLinePlotOptions ] , PlotLabel->" Error " ] , ListLinePlot[ { Range[n] , Loss } // Transpose , OptionValue[ ListLinePlotOptions ] ,
PlotRangeClipping->False , PlotLabel->" Loss " ] } // TableForm ] ,
List ,
Return[ { n , error , Loss , LossY } ]
],
865 4 ,
{ n , error , Loss , LossY }=stat[[3]];
Switch[ OptionValue[ StatFormat ] ,
Table ,
Return[ TableForm[ { { Range[n] , error , Loss , LossY } } , TableHeadings->{ None , { " Iteration " , " Error " ,
" LossTotal " , " LossSingle " } } ] ],
870 Plot ,
labelinner=Graphics[ Text[ StyleForm[ " LossSingle " , FontSize ->13 ] , { 1 , LossY[[1]] } , { -1.2 , -.7 } ] ];
labelouter=Graphics[ Text[ StyleForm[ " LossTotal " , FontSize ->13 ] , { 1 , Loss[[1]] } , { -1.2 , -.7 } ] ];
Return[ { ListLinePlot[ { Range[n] , error } // Transpose , OptionValue[ ListLinePlotOptions ] , PlotLabel->" Error " ] , Show[ ListLinePlot[ { Range[n] , LossY } // Transpose , OptionValue[ ListLinePlotOptions ] ,
PlotRangeClipping->False ] , labelinner , ListLinePlot[ { Range[n] , Loss } // Transpose , OptionValue[ ListLinePlotOptions ] , PlotRangeClipping->False ] , labelouter ] } // TableForm ] ,
List ,
Return[ { n , error , Loss , LossY } ]
],
875 5 ,
{ n , iterYL , error , Loss , LossY }=stat[[3]];
Switch[ OptionValue[ StatFormat ] ,
Table ,
Return[ TableForm[ { { Range[n] , iterYL , error , Loss , LossY } } , TableHeadings->{ None , { " Iteration " ,
NrOfIterationsInner , " Error " , " LossTotal " , " LossSingle " } } ] ],
880 Plot ,
labelinner=Graphics[ Text[ StyleForm[ " LossSingle " , FontSize ->13 ] , { 1 , LossY[[1]] } , { -1.2 , 1 } ] ];
labelouter=Graphics[ Text[ StyleForm[ " LossTotal " , FontSize ->13 ] , { 1 , Loss[[1]] } , { -1.2 , 1 } ] ];
Return[ { ListLinePlot[ { Range[n] , error } // Transpose , OptionValue[ ListLinePlotOptions ] , PlotLabel->" Error " ] , ListPlot[ { Range[n] , iterYL } // Transpose , OptionValue[ ListPlotOptions ] , PlotLabel->" NrOfIterationsInner " , Filling ->Axis ] , Show[ ListLinePlot[ { Range[n] , LossY } // Transpose , OptionValue[ ListLinePlotOptions ] , PlotRangeClipping->False ] , labelinner , ListLinePlot[ { Range[n] , Loss } // Transpose , OptionValue[ ListLinePlotOptions ] , PlotRangeClipping->False ] , labelouter ] } // TableForm ] ,
List ,
Return[ { n , iterYL , error , Loss , LossY } ]
],
885 ];
];
];

890 (*GetStartX by Sandra*)
GetStartX::error="The input to GetStartX is not in a correct format.";
GetStartX[ Protocol[ pd_List ] , v_List , OptionsPattern[] ]:=Module[{ X },
X={ GetVariableValues[ Protocol[ pd ] , v[[1]] ] , GetVariableValues[ Protocol[ pd ] , v[[2]] ] } // Transpose ;
If[ OptionValue[ Random ] ,
895 X=RandomReal[ { -1 , 1 } , Dimensions[ X ] ]
];
Return[ X ]
];
];

900

```

```

(**GetEigenvalues , by Sandra**)
GetEigenvalues[Protocol[pd_List], OptionsPattern[]]:=Module[{eig},
eig="Eigenvalues"/.pd[[METADATA]];
If[OptionValue[List]==True,
905 Return[eig];
];

If[eig[[1]]==eig[[2]],
Print[StringJoin["The eigenvalues are ",ToString[Round[eig[[1]],.00001]]]];
910 Print["There are no variables or restricted factors in this analysis so this is equivalent to a
MCA. Therefore only unrestricted factors contribute to the eigenvalues which means that the
eigenvalues are the sum of the discrimination measures."];
Return[];
];

If[eig[[1]]==eig[[3]],
915 Print[StringJoin["The eigenvalues are ",ToString[Round[eig[[1]],.00001]]]];
Print["There are no unrestricted factors in this analysis. Therefore only variables and/or
restricted factors contribute to the eigenvalues which means that the eigenvalues are the sum
of the squared component loadings."];
Return[];
];

920 Print[StringJoin["The eigenvalues are ",ToString[Round[eig[[1]],.00001]]]];
Print[StringJoin["The contributions from the unrestricted factors (the sum of their discrimination
measures) are ",ToString[Round[eig[[2]],.00001]]]];
Print[StringJoin["The contributions from the variables and/or the restricted factors (the sum of
their component loadings) are ",ToString[Round[eig[[3]],.00001]]]];
Return[];
]
925

```

---

```

(**FirstStep , by Sandra (inner function for NLPICA)**)
FirstStep::errorQ="A vector in Q is zero";
930 FirstStep[factorsL_List,fs_List,fm_List,X_List,Yprim_List,G_List,D_List,Dinv_List,Qin_List,
variables_List,r1_Integer,r2_Integer,NI_Integer,J_Integer,Lfs_Integer,OrdIn_List,StartX_List,
OptionsPattern[NLPICA]]:=Module[{Q,Qnew,maxiter,Yprimnew,B,Xprim,Xnew,W,j,ind,m,ps,q,var,cond,
Dp,rules},
maxiter=OptionValue[MaxIterationsInner];m=1;{Qnew,Q}=ConstantArray[Qin,2];
Yprimnew=Yprim;
j=Total@{Lfs,r2};B=ConstantArray[0,j];
(* Normalize and center Q*)
935 Q=Array[Q[[#]]-Total@Q[[#]]/Length[Q[[#]]]&,j];
Q[[;;Lfs]]=Table[Normalize[Sqrt[D[[i]]].Q[[i]],{i,Lfs}]*Sqrt[NI];
Q[[Lfs+1;;]]=Sqrt[NI]*Normalize/@Array[Q[[#+Lfs]]-Total@Q[[#+Lfs]]/Length[Q[[#+Lfs]]]&,r2];
(* If there are any variables*)
940 If[r2!=0,
B[[Lfs+1;;]]=Flatten[Array[{Transpose[Yprimnew[[#+r1]]].Q[[#+Lfs]]}/(Q[[#+Lfs]].Transpose[{Q[[#+
Lfs]]}])&,r2],1];
Yprimnew[[r1+1;;]]=Array[Transpose[{Q[[Lfs#+#]]}].{B[[Lfs#+#]]}&,r2]
];
945 (* If there are any constrained factors (single or ordinal)*)
If[Lfs!=0,
Qnew=Q;ind=Join[fs,r1+Range[r2]];
Do[If[Cases[Q[[i]],Except[0.]]=={},Q[[i]]=RandomReal[{-1,1},Length[Q[[i]]]];Q[[i]]=Normalize[Sqrt[
D[[i]]].Q[[i]]];Message[FirstStep::errorQ],{i,Lfs}];
(* Inner alternating leastsquares-algorithm (ALS-algorithm)*)
950 While[m<maxiter,
B[[;;Lfs]]=Flatten[Array[{Transpose[Yprimnew[[ind[[#]]]].D[[#]].Q[[#]]}/(Q[[#]].D[[#]].Transpose[
{Q[[#]]})]&,Lfs],1];
Qnew[[;;Lfs]]=Flatten[Array[{Yprimnew[[ind[[#]]]].B[[#]]}/(B[[#]].Transpose[{B[[#]]})&,Lfs],1];
If[Max[Abs[Q-Qnew]]<OptionValue[ConvergenceInner],Break[]];
Q=Qnew;
955 m++;
];
Q=Qnew;
Yprimnew[[fs]]=Array[Transpose[{Q[[#]]}].{B[[#]]}&,Lfs];
];
960 (* If there are any ordinal factors*)
If[Length[OrdIn]!=0,
(* find the ordered columns of Qnew that are "as close" to Qnew as possible*)
Switch[OptionValue[OrdinalFunction],
965 1,
(* Using FindMinimum with startvectors in q*)
{pos,var,cond,Dp}=OrdIn;
q=Qnew[[#]]&/@pos;
rules=Table[FindMinimum[{Dp[[i]].((q[[i]]-var[[i]])^2),cond[[i]]},var[[i]]][[2]],{i,Length[q]}];
970 Qnew[[pos]]=Array[var[[#]]/.rules[[#]]&,Length[q]],
2,

```

```

(* Using FindMinimum without startvectors *)
{pos, var, cond, Dp}=OrdIn;
q=Qnew[[#]]&/@pos;
975 rules=Table[FindMinimum[{Dp[[i]].((q[[i]]-var[[i]])^2),cond[[i]]},var[[i]]][[2]],{i,Length[q]}];
Qnew[[pos]]=Array[var[[#]]/.rules[[#]]&,Length[q]],3,
(* Using NMinimize *)
{pos, var, cond, Dp}=OrdIn;
980 q=Qnew[[#]]&/@pos;
rules=Array[NMinimize[{Dp[[#]].((q[[#]]-var[[#]])^2),cond[[#]]},var[[#]]][[2]]&,Length[q]];
Qnew[[pos]]=Array[var[[#]]/.rules[[#]]&,Length[q]];
];
];
985 (* Calculate, center and orthogonalize X*)
Xprim=(Total@Array[G[[#]].Yprimnew[[#]]&,r1])/J+(Total@Yprimnew[[r1+1;;]])/J;(*Yprim or Yprimnew
???*)
W=Xprim-ConstantArray[Total@Xprim/NI,NI];
Xnew=Sqrt[NI]*Orthogonalize[Transpose[W]]//Transpose;
990 Return[{Xnew,Yprimnew,Q,B,m}];
];



---


995 (**NLPCAInner, by Sandra (inner function for NLPCA)**)
NLPCAInner::errorQ="One of the ordinal factors is not given as a single factor";
NLPCAInner::errorX="The matrix with coordinates for the individuals you supplied does not have the
correct dimensions";
NLPCAInner[G>List,dim_Integer,factorsL_List,factorssingle_List,variables_List,OptionsPattern[NLPCA
]]:=Module[{NI,r1,r2,fn,fs,Lfs,J,D,Dinv,X,n,maxiterX,XPlot,YPlot,Yprim,Yprimtemp,Q,iterYL,
error,B,Xprim,W,H,Loss,LossY,j,ind,fo, pos,q,var,g,cond,Dp,OrdIn,DMsquared,Dnrm,Dfs},
(* initialize some variables*)
1000 NI=Dimensions[G[[1]]][[1]];
{r1,r2}={Length[factorsL],Length[variables]};{fm,fs}={Complement[Range[Length[factorsL]],
factorssingle],factorssingle};Lfs=Length[fs];J=Total@{r1,r2};maxiterX=OptionValue[
MaxIterationsOuter];{XPlot,YPlot,iterYL,error,Loss,LossY}=ConstantArray[ConstantArray[0,
maxiterX+1],6];
D=SparseArray/@(Transpose[#].#&/@G);
Dfs=D[[fs]];
Dinv=SparseArray/@Inverse/@D;
1005 Q=Range[factorsL[[fs]]]/.N;Q=Fold[Append,Q,variables]/.N;
n=0;Yprim=ConstantArray[0,J];
(*Get the startmatrix with coordinates for the individuals*)
If[Length[OptionValue[StartX]]==0,
X=RandomReal[{-10,10},{NI,dim}],
1010 If[Dimensions[OptionValue[StartX]]!= {NI,dim},Message[NLPCAInner::errorX];Return[]];
X=OptionValue[StartX];
];
W=X-ConstantArray[Total@X/NI,NI];X=Sqrt[NI]*Orthogonalize[Transpose[W]]//Transpose;XPlot[[1]]=X;

1015 (*The first step of the alternating least squares-algorithm (ALS-algorithm) if there are any
variables or constrained factors*)
If[r2!=0||Lfs!=0,
n++;
j=Total@{Lfs,r2};ind=Join[fs,r1+Range[r2]];
(* Initialize some variables for the inner iteration if there are any ordinal factors*)
1020 If[Lfs!=0,
If[Length[OptionValue[Ordinal]]!=0,
fo=OptionValue[Ordinal];
If[Length[Complement[fo,Intersection[fs,fo]]]!=0,Message[NLPCAInner::errorQ]];
pos=Flatten[Position[fs,#]&/@fo,2];
1025 q=Q[[#]]&/@pos;
var=Table[ToExpression["q"]<>ToString[#]&/@Range[Length[q[[i]]]],{i,Length[q]}];
g=ToExpression[ToString[#1]<>"<=">ToString[#2]]&;
cond=Array[Fold[g, var[[#,1]], var[[#,2;;Length[q[[#]]]]]]&,Length[q]];
Dp=Array[Diagonal[D[[fo[[#]]]]]&,Length[fo]];
];
1030 OrdIn={pos,var,cond,Dp},
OrdIn={},
];
OrdIn={};
];
1035 (*First step of the ALS-algorithm*)
(*Get the unconstrained coordinates for the modalities*)
Yprim=Join[Array[(Dinv[[#]].Transpose[G[[#]]]).X,&,r1],ConstantArray[X,r2]];
Yprimtemp=Yprim;
(*Get the constrained coordinates for the modalities and the coordinates for the individuals*)
1040 {X,Yprim,Q,B,iterYL[[n]]}=FirstStep[factorsL,fs,fn,X,Yprim,G,Dfs,Dinv,Q,variables,r1,r2,NI,J,Lfs,
OrdIn,OptionValue[StartX],MaxIterationsInner->OptionValue[MaxIterationsInner],ConvergenceInner
->OptionValue[ConvergenceInner]];
(*Calculate the single loss*)
H=Array[Transpose[{Q[[#]]}].{B[[#]]}-Yprimtemp[[ind[[#]]]]&,j];
LossY[[n]]=(Total@Array[Transpose[H[[#]]].Dfs[[#]].H[[#]]&,Lfs]+Total@Array[Transpose[H[[#+Lfs]]].
H[[#+Lfs]]&,r2])/J//Tr;

```

```

(* Calculate the error and the total loss *)
1045 H=Join[Array[X-G[[#]].Yprim[[#]]&,r1]^2,Array[X-Yprim[[#+r1]]&,r2]^2];
Loss [[n]]=Total@Total@Total@H/J;
error [[n]]=Max[Abs[(XPlot [[n]]-X )]];
(*Saving the coordinates in each step of the ALS-algorithm*)
XPlot [[n]]=X; YPlot [[n]]=Yprim;
1050 ];

(* Alternating least squares algorithm to get the coordinate-matrices X and Y*)
While[n<maxiterX ,
n++;
1055 (*Get new unconstrained coordinates for the modalities*)
Yprim=Join[Array[(Dinv [[#]].Transpose[G[[#]]]).X&,r1],ConstantArray[X,r2]];
Yprimtemp=Yprim;
(*Get the constrained coordinates for the modalities of the variables , if there are any*)
If[r2!=0,
1060 B[[Lfs+1;;]]=Flatten[Array[{Transpose[Yprim[[#+r1]]].Q[[#+Lfs]]}/(Q[[#+Lfs]].Transpose[{Q[[#+Lfs
]]}])&,r2],1];
Yprim [[r1+1;;]]=Array[Transpose[{Q[[Lfs+#]]}].{B[[Lfs+#]]}&,r2]
];
(*Get the constrained coordinates for the modalities of the constrained factors , if there are any
constraints on any of them*)
If[Lfs!=0,
1065 {Q,B,iterYL [[n]]}=InnerIter [Q,Yprim,Dfs,{Lfs,j,ind,r2,NI},B,OrdIn,MaxIterationsInner->OptionValue[
MaxIterationsInner],ConvergenceInner->OptionValue[ConvergenceInner],OrdinalFunction->
OptionValue[OrdinalFunction]];
Yprim [[ fs]]=Array[Transpose[{Q[[#]]}].{B[[#]]}&,Lfs];
];
(*Get new coordinates for the individuals*)
Xprim=(Total@Array[G[[#]].Yprim[[#]]&,r1])/J+(Total@Yprim [[ r1+1;;]])/J;
1070 W=Xprim-ConstantArray[Total@Xprim/NI,NI];
X=Sqrt[NI]*Orthogonalize[Transpose[W]]//Transpose;
(*Saving the coordinates in each step of the ALS-algorithm*)
XPlot [[n]]=X; YPlot [[n]]=Yprim;
(*Calculate the single loss if there is any*)
1075 If[Lfs!=0||r2!=0,
H=Array[Transpose[{Q[[#]]}].{B[[#]]}-Yprimtemp [[ ind[[#]]]]&,j];
LossY [[n]]=(Total@Array[Transpose[H[[#]]].Dfs[[#]]&,Lfs]+Total@Array[Transpose[H[[#+Lfs]]].
H[[#+Lfs]]&,r2])/J//Tr;
];
(*Calculate the total loss and the error*)
1080 H=Join[Array[X-G[[#]].Yprim[[#]]&,r1]^2,Array[X-Yprim[[#+r1]]&,r2]^2];
Loss [[n]]=Total@Total@Total@H/J;
error [[n]]=Abs[Max[(XPlot [[n-1]]-XPlot [[n]] )]];
(*Check the convergence criterion*)
If[error [[n]]<=OptionValue[ConvergenceOuter],Break[]];
1085 ];

(*Calculate the squared discrimination measures of the solution*)
DMsquared=Join[Array[Diagonal[Transpose[Yprimtemp[[#]].D[[#]].Yprimtemp[[#]]]&,r1],Array[Diagonal
[Transpose[Yprimtemp[[#+r1]]].Yprimtemp[[#+r1]]]&,r2]]/NI;
(*Return the coordinates and some other useful information*)
1090 If[r2!=0||Lfs!=0,
(*Normalize Q for identification*)
Dnrm=Array[Sqrt[Q[[#]].Dfs[[#]].Transpose[{Q[[#]]}]]&,Lfs]//Flatten;Q [[ ; Lfs]]=Array[Q[[#]]/Dnrm
[[#]]&,Lfs]*Sqrt[NI];B [[ ; Lfs]]=Array[B[[#]]*Dnrm[[#]]&,Lfs]/Sqrt[NI];
Return[{XPlot [[1;;n]],YPlot [[1;;n]],Q,B,DMsquared,{n,iterYL [[1;;n]],error [[1;;n]],Loss [[1;;n]],
LossY [[1;;n]]}],Return[{XPlot [[2;;n]],YPlot [[2;;n]},{},{},DMsquared,{n-1,iterYL [[2;;n]],error [[2;;n]],Loss [[2;;n
]],LossY [[2;;n]]}]];
1095 ];

```

---

```

(**InnerIter , by Sandra (inner function to NLPCAInner)**)
1100 InnerIter::errorQ="A vector in Q is zero";
InnerIter[Q_List,Yprim_List,D_List,{Lfs_Integer,j_Integer,ind_List,r2_Integer,NI_Integer},
LastB_List,OrdIn_List,OptionsPattern[NLPCA]]:=Module[{m,Qnew,Qtemp,B,pos,q,qstart,var,cond,Dp,
rules},
m=1;B=LastB;Qnew=Q;Qtemp=Q;
Do[If[Cases[Qnew[[ i ]],Except[0.]]=={},Qnew[[ i ]]=RandomReal[{-1,1},Length[Q[[ i ]]]];Qnew[[ i ]]=
Normalize[Sqrt[D[[ i ]]].Qnew[[ i ]]]*Sqrt[NI];Message[InnerIter::errorQ],{ i ,Length[Lfs]}];
1105 (*Inner alternating leastsquares-algorithm (ALS-algorithm) if there are any constrained factors (
single or ordinal)*)
While[m<OptionValue[MaxIterationsInner],
B[[ ; Lfs]]=Flatten[Array[{Transpose[Yprim [[ ind[[#]]]].D[[#]].Qnew[[#]]}/(Qnew[[#]].D[[#]]].
Transpose[{Qnew[[#]]}]&,Lfs ],1];
Qtemp [[ ; Lfs]]=Flatten[Array[{Yprim [[ ind[[#]]]].B[[#]]}/(B[[#]].Transpose[{B[[#]]}])&,Lfs ],1];
If[Max[Abs[Qtemp-Qnew]]<OptionValue[ConvergenceInner],Break[]];
1110 Qnew=Qtemp;
m++;
Qnew=Qtemp;

```

```

(*For any ordinal factors, find the ordered columns of Qnew that are "as close" to Qnew as
possible*)
1115 If[Length[OrdIn]!=0,
Switch[OptionValue[OrdinalFunction],
1,
(*Using FindMinimum with startvectors in q*)
{pos,var,cond,Dp}=OrdIn;
1120 q=Qnew[[#]]&/@pos;
qstart=Q[[#]]&/@pos;
rules=Table[FindMinimum[{Dp[[i]].((q[[i]]-var[[i]])^2),cond[[i]]},MapThread[{{#1,#2}&,{var[[i]],
qstart[[i]]}}][[2]],{i,Length[q]]}];Qnew[[pos]]=Array[var[[#]]/.rules[[#]]&,Length[q]];
];
1125 2,
(*Using FindMinimum without startvectors*)
{pos,var,cond,Dp}=OrdIn;
q=Qnew[[#]]&/@pos;
rules=Table[FindMinimum[{Dp[[i]].((q[[i]]-var[[i]])^2),cond[[i]]},var[[i]]][[2]],{i,Length[q]]};
1130 Qnew[[pos]]=Array[var[[#]]/.rules[[#]]&,Length[q]],
3,
(*Using NMinimize*)
{pos,var,cond,Dp}=OrdIn;
q=Qnew[[#]]&/@pos;
1135 rules=Array[NMinimize[{Dp[[#]].((q[[#]]-var[[#]])^2),cond[[#]]},var[[#]]][[2]]&,Length[q]];
Qnew[[pos]]=Array[var[[#]]/.rules[[#]]&,Length[q]];
];
];
1140 Return[{Qnew,B,m}]
];



---


(*ToProtocol, by Sandra & Oskar (inner function for NLPCA and PCA)**)
1145 ToProtocol[Protocol[pd_List],dim_Integer,f_List,singlefactors_List,v_List,B_List,indcoords_List,
modvarcoord_List,DMsquared_List,eigenvalues_List,DiscriminationInfo_Symbol]:=Module[{newpd,p00
,p0,p1,p2,facdata,moddata,DM},
newpd=pd;
(*Add the eigenvalues to the protocol on which the analysis is being performed.*)
newpd[[METADATA]]=Append[newpd[[METADATA]],"Eigenvalues"→eigenvalues];
newpd[[METADATA]]=Append[newpd[[METADATA]],"Total nr of factors and variables"→Length[DMsquared
]];
];
1150 (*Add the coordinate variables of the individuals to the protocol on which the analysis is being
performed*)
p1=Fold[AddVariable[#1,StringJoin["Coordinates of individuals dim",ToString[#2]],indcoords[[All
,#2]]]&,<span style="color:blue;">Protocol[newpd],Range[dim]];
(*Add the coordinates of the modalities of the variables as new variables to the same protocol.*)
1155 If[v!={},
Do[
p1=Fold[AddVariable[#1,StringJoin[newpd[[VARIABLES,v[[i]],1]]," coordinates dim",ToString[#2]],
modvarcoord[[Length[f]+i,All,#2]]]&,p1,Range[dim]],
{i,Length[v]}
];
];
1160 (*Create protocol for the coordinates of the modalities*)
If[f!={},
p2=CreateProtocol[Flatten[GetModalityValues[Protocol[newpd],#]&/@f]];
Do[
1165 p2=AddVariable[p2,StringJoin["Coordinates of modalities dim",ToString[i]],Flatten[modvarcoord[[;
Length[f],All,i]]],
{i,dim}];
facdata = Array[ConstantArray[newpd[[FACTORS,f[[#],1]],Length[newpd[[FACTORS,f[[#],2]]]]&,Length
[f]]];
moddata = Array[newpd[[FACTORS,f[[#,2]]]&,Length[f]];
p2[[METADATA,1]]=Append[p2[[METADATA,1]],"StandardCoordinates"→False];
p2 = AddFactor[p2,"Factors",Flatten[facdata]];
p2 = AddFactor[p2,"Modalities",Flatten[moddata]];
];
(*Create protocol p0 with the component loadings for the variables and/or constrained factors (if
there are any).*)
1175 If[v!={}||singlefactors!={},p0=CreateProtocol[Join[newpd[[FACTORS,#,1]]&/@singlefactors,newpd[[VARIABLES,#,1]]&/@v]];
p0=Fold[AddVariable[#1,StringJoin["Component loadings dim",ToString[#2]],B[[All,#2]]]&,p0,Range[
dim]];
p0=AddFactor[p0,"Restricted factornames and/or variablenames",Join[StringJoin["F ",newpd[[FACTORS
,#,1]]]&/@singlefactors,StringJoin["V ",newpd[[VARIABLES,#,1]]]&/@v]];
];
1180 (*Create protocol for (optional) info on discrimination measures*)
If[DiscriminationInfo,

```

```

p00=CreateProtocol[Join[newpd[[FACTORS,#,1]]&/@f,newpd[[VARIABLES,#,1]]&/@v]];
DM=Sqrt/@DMsquared;
1185 p00=Fold[AddVariable[#1, StringJoin["Discrimination measures dim",ToString[#2]],DM[[All,#2]]]]&,p00,
    Range[dim]];
p00=AddFactor[p00,"Factornames and/or variablenames",Join[StringJoin["F ",newpd[[FACTORS,#,1]]]&/
    @f, StringJoin["V ",newpd[[VARIABLES,#,1]]]&/@v]];
];
(*Return the new protocols*)
1190 If[singlefactors=={}&&v=={},  

If[DiscriminationInfo,  

Return[{p00,p1,p2}],  

Return[{p1,p2}]]  

];
1195 ];
If[DiscriminationInfo,  

If[f!={}],  

If[v!={}||singlefactors!={}],  

Return[{p00,p0,p1,p2}],  

1200 Return[{p00,p1,p2}]]  

],  

Return[{p00,p0,p1}]]  

],  

If[f!={}],  

1205 If[v!={}||singlefactors!={}],  

Return[{p0,p1,p2}],  

Return[{p1,p2}]]  

],  

Return[{p0,p1}]]  

1210 ];
];
];
];
End[]  

1215 EndPackage[]

```

## GUI

```

BeginPackage["GUI`", {"RCA`", "Transformations`"}]

Unprotect[
Cloud,
5 GetCoordinates,
CreateSymbol,
Interact
]

10 Cloud::usage = "Cloud[p] draw a cloud of points using data in p and given options."
SizeVariable::usage = "SizeData is an option for Cloud specifying a variable index to use for
sizes of symbols. Default option value is None."
Variables::usage = "An option for Cloud. Specifying a list with variable indices in a protocol to
use for plotting. Default option value is {1,2}"
Axes::usage = "An option for Cloud. Should be either True or False for displaying or not
displaying axes respectively. The default option value is true."
ColorFactor::usage = "An option for Cloud. Specify a factor index to use for colors of symbols.
Then each modality in the factor gets a distinct color. Default option value is None."
15 Color::usage = "An option for Cloud. Specify the color to use for the symbols. You can enter any
RGBColor. Colors that have been given names by Mathematica is also accepted."
ShapeFactor::usage = "ShapeData is an option for Cloud specifying a factor index to use for shapes
of symbols. Default option value is None."
ImageSize::usage = "An option for Cloud. Determine the image size. Default option value is 400"
Shape::usage = "Shape is an option for Cloud. Shape->n gives symbols shape n."
TooltipFactor::usage = "An option for Cloud. Specify a factor index to use for tooltip labels. The
modalities of the given factor will be shown when you move the mouse over a symbol. The
default option value is None."
20 LabelFactor::usage = "An option for Cloud specifying a factor index to use for labeling. The
modalities of the given factor will be the labels. Default option value is None."
FontSize::usage = "An option for Cloud specifying the font size for the labels."
FontColor::usage = "An option for Cloud. Specify a font color (RGB) to use when labeling the symbols
."
LabelOrientation::usage = "An option for Cloud specifying the orientation of the label relative
the symbol it's representing. Default option value is {-1,-1}"
LabelPlacementOffset::usage = "An option for Cloud specifying the distance between a symbol and
its label."
25 RandomOffset::usage = "An option for Cloud specifying the bounds for a random offset which is
applied to the coordinates of the symbols. Default option value is 0"
Legend::usage = "Set Legend->True if you want a legend"
LegendSize::usage = "Specify the size of the Legend. Default option value is Unspecified and then
the width will be set to 200 and the height will be automatically adjusted to fit the image size

```

```

    "
Opacity::usage = "An option for Cloud. Specify the opacity for the symbols. Default option value
is 1."
SizeScale::usage = "An option for Cloud. Determine a scale factor between the data in a variable
given by SizeData and the size of symbols. Default option value is {1,1}"
30 SymbolSize::usage = "An option for Cloud. Specify the symbol size. You can change the size
uniformly by just entering a real positive number. Default option value is {0.05,0.05}."
TabPanel::usage = "An option for Cloud."
Arrows::usage = "An option for Cloud."
ArrowThickness::usage = "Determine the thickness of the arrows used in Shape->Arrows. Default
option value is 0.003"
ArrowHeadSize::usage = "Determine the size of the arrowheads used in Shape->Arrows. Default option
value is 0.01"
35 CloudMarkings::usage = "An option for Cloud which specifies which symbols to use in the cloud.";
CycleMarkings::usage = "An option for Cloud which specifies if the markings (symbols of imported
graphics) should be cycled or not.";

40 South::usage = "An option for Cloud"
North::usage = "An option for Cloud"
East::usage = "An option for Cloud"
West::usage = "An option for Cloud"

45

50 Begin[ "‘Private’"]

55 Options[CreateSymbol] = {CloudMarkings->{}, CycleMarkings->False};



---


(*CreateSymbol, by Matz version 0.35**)
CreateSymbol[symbType_, {x_, y_}, {sx_, sy_}, OptionsPattern[CreateSymbol]] := Module[{tmp, sindex
=symbType},
If[Length@OptionValue[CloudMarkings]==0, Print["ERROR: Cloudmarkings is empty!"]; Abort[]]; (*
detta skall inte h\ADoubleDot\nda -Matz*)
60 If[OptionValue[CycleMarkings], sindex = Mod[symbType-1, Length@OptionValue[CloudMarkings]]+1];

Return[ OptionValue[CloudMarkings][[sindex]]@@{x,y,sx,sy}](*apply the coordinates to the function
of the "symbol")
]

65
Cloud::colfac = "Color factor is out of bounds. Using factor 1.";
Cloud::shapefac = "Shape factor is out of bounds. Using symbol 1 for all objects.";
Cloud::labelfac = "Label factor is out of bounds.";
Cloud::sizevar = "Size variable out of bounds. Using size 1 for all objects.";
70 Cloud::novars= "Couldn't find 2 variables in the protocol. Aborting.";
Cloud::nocolor= "Not a valid color. Using Black for all symbols.";
Cloud::nodispvars= "Need 2 variables. Using variables 1 and 2.";

75 Options[Cloud]={Variables->{1,2},
SizeVariable->None,
SizeScale->{1,1},
Axes->True,
ColorFactor->None,
Color->None,
80 Shape->None,
ShapeFactor->None,
SymbolSize ->{0.05,0.05},
LabelFactor->None,
FontSize->10,
FontColor->Black,
85 LabelOrientation->South,
LabelPlacementOffset ->0.1,
RandomOffset->0,
TooltipFactor->None,
Opacity->1,
90 ImageSize->400,
Legend->False,
LegendSize->Unspecified,
TabPanel->False,
95 AspectRatio->Automatic,
ArrowThickness ->0.003,
ArrowHeadSize ->0.01,
CloudMarkings->{},
CycleMarkings->True
100 };

```

---

```

(**Cloud, by Joakim**)
Cloud[Protocol[pd_List], OptionsPattern[Cloud]]:=Module[{newpd, gfx, coord, pts, n, colors, symbtypes,
  sizes, tooltiplabel, labels, names, coltab, legend, legendsize, labelorientation, orientation},
105  variables = OptionValue[Variables];
  axes = OptionValue[Axes];
  color = OptionValue[Color];
  colorfactor = OptionValue[ColorFactor];
110  shape = OptionValue[Shape];
  shapefactor = OptionValue[ShapeFactor];
  symbolsize = OptionValue[SymbolSize];
  sizevariable = OptionValue[SizeVariable];
  sizescale = OptionValue[SizeScale];
115  tooltipfactor = OptionValue[TooltipFactor];
  imagesize = OptionValue[ImageSize];
  opacity = OptionValue[Opacity];
  labelfactor = OptionValue[LabelFactor];
  labelorientation = OptionValue[LabelOrientation];
120  labeloffset = OptionValue[LabelPlacementOffset];
  fontsize = OptionValue[FontSize];
  fontcolor = OptionValue[FontColor];
  randomoffset = OptionValue[RandomOffset];
  legend = OptionValue[Legend];
125  legendsize = OptionValue[LegendSize];
  aspectratio = OptionValue[AspectRatio];
  arrowthickness = OptionValue[ArrowThickness];
  arrowheadsize = OptionValue[ArrowHeadSize];

130  If[legendsize==Unspecified,
  legendsize = {200,imagesize};
];
cyc = OptionValue[CycleMarkings]; (*we want to cycle between markers (default for icons = false)
 *)
135  marks = OptionValue[CloudMarkings];
newpd=pd;

140  (*convert icons/markers to Functions for this session*)
symbols={ Disk[{x,y},{sx,sy}],
  Polygon[{{x-sx, y-sy},{x+sx, y-sy},{x+sx, y+sy},{x-sx, y+sy}}], (* Rectangle[{x-sx, y-sy}, {x+sx, y+sy}], (*convert to polygon...*)*)
145  Polygon[{{x-sx, y-sy},{x, y+sy},{x+sx, y-sy},{x-sx, y-sy}}], 
  Polygon[{{x+sx, y+sy},{x, y-sy},{x-sx, y+sy},{x+sx, y+sy}}], 
  Polygon[{{x-sx, y-sy},{x+sx, y},{x-sx, y+sy},{x-sx, y-sy}}], 
  Polygon[{{x+sx, y-sy},{x-sx, y},{x+sx, y+sy},{x+sx, y-sy}}], 
  {Line[{{x, y-sy},{x, y+sy}}], Line[{{x-sx, y},{x+sx, y}}]}, (*straight cross*)
150  {Line[{{x-sx, y-sy},{x+sx, y+sy}}], Line[{{x-sx, y+sy},{x+sx, y-sy}}]}, (*rotated cross*)
  Polygon[{{x-sx, y},{x, y+sy},{x+sx, y},{x, y-sy},{x-sx, y}}], 
  Circle[{x,y},{sx,sy}]]; (*unable to optimize*)

155  If[Length@marks!=0, symbols = Inset[#, {x, y}, {0, 0}, {sx, sy}] &/@ marks];
symbolsF = Function[{x, y, sx, sy}, #] & /@ symbols; (*make the symbols Functions*)

160  If[Length[newpd[[VARIABLES]]]<2, Message[Cloud::novars]; Abort[]];
If[ListQ[variables] && Length[variables]==2,
  coord = GetCoordinates[Protocol[newpd], variables],
  Message[Cloud::nodispvars];
  coord = GetCoordinates[Protocol[newpd], {1, 2}];
];
165  ];
n = Length[coord];
coltab=Array[ColorData[1][#]&, n];
170  (*Random offset*)
If[randomoffset>0,
  coord = coord + Table[{RandomReal[{-randomoffset, randomoffset}], RandomReal[{-randomoffset, randomoffset}], {n}}, {n}],
];
175  (* Hur skall symblerna fargas? *)
If[colorfactor==None, (* dvs vi har ingen faktor *)
  If[color==None, (* och ingen fixerad farg *)
```

```

180      colors=Table[Black,{n}], (* da gor vi symbolerna svarta! *)
181      If[Head[color]==RGBColor,
182          colors=Table[color,{n}], (* annars sa far de den fixa fargen col *)
183          Message[Cloud::nocolor];
184          colors=Table[Black,{n}];
185      ],
186  ];
187
188      If[ colorfactor<=Length[newpd[[FACTORS]]],
189          colors=coltab[[#]]&/@newpd[[FACTORS,colorfactor,3]],
190          Message[Cloud::colfac];
191          colors=coltab[[#]]&/@newpd[[FACTORS,1,3]]];
192  ];
193
194      (* Hur skall symbolerna fa sina former? Analogt med fargerna ovan. *)
195      If[shapefactor==None,
196          If[shape==None,
197              symbtypes=Table[1,{n}],
198              If[shape==Arrows,
199                  symbtypes={},
200                  symbtypes=Table[shape,{n}]],
201                  If[ListQ[shapefactor],shapefactor=shapefactor[[1]]];
202                  If[shapefactor<=Length[newpd[[FACTORS]]],
203                      symbtypes = newpd[[FACTORS,shapefactor,3]],
204                      Message[Cloud::shapefac];
205                      symbtypes=Table[1,{n}]];
206      ];
207
208      (* Symbolernas storlek*)
209      sizes = Which[sizevariable==None,
210          If[symbolsize==Automatic,
211              X=Max[coord[[All, 1]]]-Min[coord[[All, 1]]];
212              Y=Max[coord[[All, 2]]]-Min[coord[[All, 2]]];
213              ratio=X/Y;
214              Table[(Y/100)*{ratio,1},{n}],
215              If[ListQ[symbolsize], Table[symbolsize,{n}], Table[{symbolsize,symbolsize},{n}]];
216
217          !ListQ[sizevariable]&&sizevariable<=Length[newpd[[VARIABLES]]],
218          Abs[{sizescale[[1]]*#[[1]],sizescale[[2]]*#[[2]]}&/@Transpose[newpd[[VARIABLES,{sizevariable,sizevariable},2]]]],
219
220          sizevariable[[1]] < Length[newpd[[4]]]&&sizevariable[[2]]<=Length[newpd[[4]]],
221          Abs[{sizescale[[1]]*#[[1]],sizescale[[2]]*#[[2]]}&/@Transpose[newpd[[VARIABLES,{sizevariable,2}]]],
222
223          True,
224          Message[Cloud::sizevar];
225          Table[sizevariable,{n}]];
226
227      If[labelfactor==None,
228          labels = {},
229          If[labelfactor<=Length[newpd[[FACTORS]]],
230              names = ToString[newpd[[FACTORS,labelfactor,2]][[#]]]&/@newpd[[FACTORS,labelfactor,3]];
231
232          Which[
233              labelorientation==North, orientation = {0,1},
234              labelorientation==East, orientation = {1,0},
235              labelorientation==South, orientation = {0,-1},
236              labelorientation==West, orientation = {-1,0},
237              ListQ[labelorientation]==True, orientation = labelorientation
238          ];
239
240          labels = Table[Text[StyleForm[names[[k]], FontSize->fontsize, fontcolor],coord[[k]]+
241              orientation*(sizes[[k]]+labeloffset),-orientation], {k,1,n}],
242
243          labels = {};
244          Message[Cloud::labelfac]
245      ];
246  ];
247
248      If[shape==Arrows,
249          (*The points without or with tooltip*)
250      Which[tooltipfactor==None,
251          pts = Table[{Opacity->opacity,colors[[k]],Arrowheads[arrowheadsize],Thickness[arrowthickness],
252          },Arrow[{{0,0},coord[[k]]}]],{k,1,n}],
253          tooltipfactor!=None,
254          tooltiplabel = newpd[[FACTORS,tooltipfactor,2]][[#]]&/@newpd[[FACTORS,tooltipfactor,3]];

```

```

pts = Table[{Opacity->opacity , colors[[k]] , Arrowheads[arrowheadsize] , Thickness[arrowthickness
260 ] , Tooltip[Arrow[{{0,0},coord[[k]]}], tooltiplabel[[k]]], {k,1,n}};

(*The points without or with tooltip*)
Which[tooltipfactor==None,
265 pts = Table[{Opacity->opacity , colors[[k]] , CreateSymbol[symbtypes[[k]] , coord[[k]] , sizes[[k]] ,
CloudMarkings->symbolsF , CycleMarkings->cyc]}, {k,1,n}];
, tooltipfactor!=None,
, tooltiplabel = newpd[[FACTORS, tooltipfactor, 2]][[#]]&/@ newpd[[FACTORS, tooltipfactor, 3]];
270 pts = Table[{Opacity->opacity , colors[[k]] , Tooltip[CreateSymbol[symbtypes[[k]] , coord[[k]] ,
sizes[[k]] , CloudMarkings->symbolsF , CycleMarkings->cyc], tooltiplabel[[k]]], {k,1,n}}];
];

(*ColorLegend*)
If[legend==True,
275 Module[{L, pos, col},
If[colorfactor!=None,
L=Length[newpd[[FACTORS, colorfactor, 2]]];
pos = Position[newpd[[FACTORS, colorfactor, 3]], #, 1, 1]&/@Range[L];
col = colors[[#]]&/@Flatten[pos];

legendcols = Grid[Table[{Graphics[{col[[k]] , Rectangle[RoundingRadius->0.3]} , ImageSize->20],
280 newpd[[FACTORS, colorfactor, 2]][[k]] , {k,1,L}] , Spacings->{1,1}, Alignment->Left];
colorlegend=Pane[legendcols , legendsize , Scrollbars->{False, True} , ImageMargins->{{0, 0}, {0,
0}}, AppearanceElements -> None],
colorlegend = Null;
];
];
];
, colorlegend = Null
285 ];
];

(*SymbolLegend*)
290 If[legend==True,
Module[{L, pos, symb},
If[shapefactor!=None,
L=Length[newpd[[FACTORS, shapefactor, 2]]];
pos = Position[newpd[[FACTORS, shapefactor, 3]], #, 1, 1]&/@Range[L];
symb = symbtypes[[#]]&/@Flatten[pos];

legendsyms = Grid[Table[{Graphics[{Black, CreateSymbol[symb[[k]] , {0,0},{1,1} , CloudMarkings
->symbolsF , CycleMarkings->cyc}] , ImageSize->20], newpd[[FACTORS, shapefactor, 2]][[k]] , {k
,1,L}] , Spacings->{1,1}, Alignment->Left];
295 symbollegend=Pane[legendsyms , legendsize , Scrollbars->{False, True} , ImageMargins->{{0, 0}, {0,
0}}, AppearanceElements -> None],
symbollegend = Null;
];
];
];
, symbollegend = Null
300 ];
];

If[legend==True,
305 gfx = Grid[{{Graphics[{labels , pts} , Axes->axes , ImageSize->imagesize] , TabView[{"Color Legend"
->colorlegend , "Symbol Legend"->symbollegend}]}], Spacings ->{{0,10,0},0}];
, gfx = Graphics[{labels , pts} , Axes->axes , ImageSize->imagesize , AspectRatio->aspectratio];
];
];

Return[gfx];
]
320



---


(*Interact , by Joakim*)
Interact::novar = "The Protocol contains no variables";
325 Interact[Protocol[pd_List]]:=DynamicModule[
{newpd , newmodpd , VarTable , FacTable , ctrls , imagesize=400, individcloud , opacity , colorfac },
newpd=pd;
330 (*Check to see if the protocol contains any variables*)
If[Length[newpd[[VARIABLES]]]===0, Message[Interact::novar]; Abort[]];
];

```

```

VarTable = Table[k,{k,Length[newpd[[VARIABLES]]]]}];
335 FacTable = Table[k,{k,Length[newpd[[FACTORS]]]]};;

Dynamic[Grid[{{Column[{ctrls},Dividers->Center,Spacings->3,BaselinePosition->Bottom], Show[
    individcloud]}},{
340 Frame->All, Alignment->{Left,Top},ItemStyle->Directive[FontFamily->"Helvetica"], Spacings
    ->{{1->1,2->2},{1->1}}], SynchronousUpdating->False]
,
Initialization:>(
345 opacity=1;
colorfac=None;
toolfac=None;
labelfac=None;
labelorientation = South;
350 labeloffset = 0;
fontsize = 10;
symbfac = None;
sizevar=None;
symbssize=0.05;
355 sizescale={1,1};

individcloud := Cloud[Protocol[newpd],Variables->{1,2},Opacity->Dynamic@opacity,ImageSize->
    Dynamic@imagesize,ColorFactor->colorfac,ShapeFactor->symbfac,TooltipFactor->toolfac,
    LabelFactor->labelfac,LabelOrientation->labelorientation,LabelPlacementOffset->labeloffset,
    FontSize->fontsize,SymbolSize->symbssize,SizeVariable->sizevar,SizeScale->sizescale];
360

ctrls := Column[{
(*Main Controls*)
365 Grid[{{Style["Main Controls",Bold]}, {"Image Size",InputField[Dynamic[imagesize]]}}, Spacings ->{1,{1,2,{0.5}}}, Alignment->{Left,Center}], (*Cloud Controls*)
Grid[{{Style["Cloud Controls",Bold]}, {"Factor to use for coloring",PopupMenu[Dynamic[colorfac],Prepend[FacTable,None]]}, {"Factor to use for Symbols",PopupMenu[Dynamic[symbfac],Prepend[FacTable,None]]}, {"Factor to use for Tooltip",PopupMenu[Dynamic[toolfac],Prepend[FacTable,None]]}, {"Factor to use for Labels",PopupMenu[Dynamic[labelfac],Prepend[FacTable,None]]}, {"Variable to use for Symbol Sizes",PopupMenu[Dynamic[sizevar],Prepend[VarTable,None]]}], {"Label Orientation",SetterBar[Dynamic[labelorientation],{North,East,West,South}]}, {"Label Placement Offset",InputField[Dynamic[labeloffset]]}, {"Font Size",InputField[Dynamic[fontsize]]}, {"Symbol Size",InputField[Dynamic[symbssize]]}, {"Size Scale X",InputField[Dynamic[sizescale[[1]]]]}, {"Size Scale Y",InputField[Dynamic[sizescale[[2]]]]}, {"Opacity",Slider[Dynamic[opacity],ContinuousAction->False]}}, Spacings ->{1,{1,2,{0.5}}}, Alignment->{Left,Center}]
},
385 Dividers->Center, Spacings ->3, BaselinePosition ->Bottom];
385

)
]
390 Interact[Protocol[pd1>List],Protocol[pd2>List]]:=DynamicModule[
{newpd,newmodpd,VarTable,FacTable,ctrls,imagesize=400,individcloud,modcloud,opacity,colorfac,
    toolfac,labelfac,labelorientation,labeloffset,fontsize,symbfac},
    newpd=pd1;
    newmodpd=pd2;
    individcloud := Cloud[Protocol[newpd],Opacity->Dynamic@opacity[[1]]];
    modcloud := Cloud[Protocol[newmodpd],Opacity->Dynamic@opacity[[2]],LabelFactor->Dynamic@labelfac
    [[2]]];
400 (*Check to see if the protocol contains any variables*)
If[Length[newpd[[VARIABLES]]]===0,Message[Interact::novar]; Abort[]];
    Dynamic[Grid[{{Column[{ctrls},Dividers->Center,Spacings->3,BaselinePosition->Bottom], Show[
        individcloud,modcloud]}},{
395 Frame->All, Alignment->{Left,Top},ItemStyle->Directive[FontFamily->"Helvetica"], Spacings
        ->{{1->1,2->2},{1->1}}], SynchronousUpdating->False]

```

```

    Initialization:>(
410  VarTable = {Table[k,{k,Length[newpd[[VARIABLES]]]}], Table[k,{k,Length[newmodpd[[VARIABLES]]]}]};
    FacTable = {Table[k,{k,Length[newpd[[FACTORS]]]}], Table[k,{k,Length[newmodpd[[FACTORS]]]}]};

    opacity={1,1};
    colorfac={None,None};
    toolfac={None,None};
    labelfac={None,None};
    labelorientation = {South,South};
    labeloffset = {0,0};
    fontsize = {10,10};
    symbfac = {None,None};

415  ctrls := Column[{
    (*Main Controls*)
    Grid[{Style["Main Controls",Bold],
        {"Image Size",InputField[Dynamic[imagesize]]}
    },Spacings ->{1,{1,2,{0.5}}}, Alignment->{Left,Center}],
420
    (*Cloud Controls*)
    Grid[{Style["Cloud Controls",Bold],
        {"Factor to use for coloring",PopupMenu[Dynamic[colorfac[[1]]],Prepend[FacTable[[1]],None]]},
        {"Factor to use for Symbols",PopupMenu[Dynamic[symbfac[[1]]],Prepend[FacTable[[1]],None]]},
        {"Factor to use for Tooltip",PopupMenu[Dynamic[toolfac[[1]]],Prepend[FacTable[[1]],None]]},
        {"Factor to use for Labels",PopupMenu[Dynamic[labelfac[[1]]],Prepend[FacTable[[1]],None]]},
        {"Label Orientation",SetterBar[Dynamic[labelorientation[[1]]],{North,East,West,South}]},
        {"Label Placement Offset",InputField[Dynamic[labeloffset[[1]]]]},
        {"Font Size",InputField[Dynamic[fontsize[[1]]]]},
        {"Opacity",Slider[Dynamic[opacity[[1]]],ContinuousAction->False]}
    },Spacings ->{1,{1,2,{0.5}}}, Alignment->{Left,Center}],
425
    (*Cloud Controls*)
    Grid[{Style["Cloud Controls",Bold],
        {"Factor to use for coloring",PopupMenu[Dynamic[colorfac[[2]]],Prepend[FacTable[[2]],None]]},
        {"Factor to use for Symbols",PopupMenu[Dynamic[symbfac[[2]]],Prepend[FacTable[[2]],None]]},
        {"Factor to use for Tooltip",PopupMenu[Dynamic[toolfac[[2]]],Prepend[FacTable[[2]],None]]},
        {"Factor to use for Labels",PopupMenu[Dynamic[labelfac[[2]]],Prepend[FacTable[[2]],None]]},
        {"Label Orientation",SetterBar[Dynamic[labelorientation[[2]]],{North,East,West,South}]},
        {"Label Placement Offset",InputField[Dynamic[labeloffset[[2]]]]},
        {"Font Size",InputField[Dynamic[fontsize[[2]]]]},
        {"Opacity",Slider[Dynamic[opacity[[2]]],ContinuousAction->False]}
    },Spacings ->{1,{1,2,{0.5}}}, Alignment->{Left,Center}]
430
435
440
445
450
455
460
    },Dividers->Center, Spacings ->3, BaselinePosition ->Bottom];
)

```

---

```

465 (**GetCoordinates, by Joakim**)
GetCoordinates::Variable="Error in the list of variables";
GetCoordinates[Protocol[pd_List],var_List]:=Module[{F},
470  If[Max[var]>Length[pd[[VARIABLES]]], Message[GetCoordinates::Variable]];
    F= pd[[VARIABLES,var,2]];
    Return[Transpose[F]];
]
475 End[]

```

**EndPackage** []