Contents

A Morphological opening implementations and proofs **1**

B Normalisation implementation **6**

C Peak alignment algorithms and implementation **8**

D Surrogate variable analysis implementation **14**

E Supplementary detail of classification methods **16**

F Pareto Fronts for variable ranking **20**

Appendix A

Morphological opening implementations and proofs

A.1 Simple top-hat implementation

Consider $x \in \{1, 2, \ldots, n\} = X$ and $f(x) \in \mathbb{R} \quad \forall x \in X$. Because of the assumed even spacing of the elements of X, the R-function below simply requires the vector of intensities, f, and the size of the SE.

A.1.1 Top-hat implementation

```
1  erode<-function(f,sesize) #f are the intensities
2  {
3     if(!(sesize %% 2)) return(NULL) #SE must be of odd length
4     nx<-length(f); erode<-rep(0,nx); halfse<-(sesize-1)/2;
5     for(i in 1:nx) #for each m/z point across the spectrum
6        erode[i]<-min(f[max(1,i-halfse):min(nx,i+halfse)])
7     return(erode)
8  }
9  ### dilate is the same as erode except use max instead of min ...OR...
10 dilate<-function(f,sesize) return(-erode(-f,sesize))
11 ### as defined τ_B(f) = f − (f ⊖ B) ⊕ B
12 tophat<-function(f,sesize) return(f-dilate(erode(f,sesize),sesize))
```

Note, the code checks the SE provided is an odd integer as a centred, symmetric SE is required. The maximum and minimum statements on line 6 of Code Segment ??, namely `max(1,i-halfse)` and `min(nx,i+halfse)`, check for when the SE sits over the 'edge' on the left or right of the series, respectively. This ensures only defined f values will be used.

A.2 Line segment erosion implementation

```
erode.quick<-function(f,se.size){
  nx<-length(f)
  k<-se.size
  t1<-proc.time()[3] ### get start time
  if(k>=nx){
    cat("Warning: structuring element is >= in length as the input\n")
    cat("The input vector has been output \n")
    return(f)
  }else{
    if((k%%2) != 1){
      k<-k-1
      cat("Structring Element not symmetric, using SE length -1 =",k,"\n")
    }
    k.left<-(k-1)/2
    add.pix<-k-(nx%%k)
    isMultiple<-(add.pix==k)
    if(!isMultiple){
      f<-c(f,rep(+Inf,add.pix))
      rem.indxs<-(nx+1):(nx+add.pix)
      nx<-nx+add.pix
    }
    g<-rep(0,nx)
    h<-rep(0,nx)
    r<-rep(0,nx)
    j<-nx
    for(i in 1:nx){
      if(i%%k==1){
        g[i]<-f[i]
      }else{
        g[i]<-min(g[i-1],f[i])
      }
      if(j%%k==0){
        h[j]<-f[j]
      }else{
        h[j]<-min(h[j+1],f[j])
      }
      j<-j-1
    }
    r[1:(k.left+1)]<-g[(k.left+1):k]
    r[(k.left+2):(nx-k.left-1)]<-pmin(g[(k+1):(nx-1)],h[2:(nx-k)])
    r[(nx-k.left):nx]<-h[(nx-k+1):(nx-k.left)]
    if(!isMultiple) r<-r[-rem.indxs]

    delta.t<-sprintf("%.2f",proc.time()[3]-t1) ### time elapsed
    cat("Completed morphological erosion in",delta.t,"seconds\n")
    return(r)
  }
}
```

A.3 Naive implementation of a morphological erosion for unequally spaced values

Algorithm A.1 (Calculate $\epsilon_B(f)(x_i)$ for all $x_i \in X \subset \mathbb{R}$).

Require: $X = \{x_1, x_2, \ldots, x_n\}, f(X), k = SE.length$
  for all $x_i \in X$ ***do***
    find x_l *s.t.* $x_{l-1} < x_i - \frac{k}{2} \leq x_l$
    find x_u *s.t.* $x_u \leq x_i + \frac{k}{2} < x_{u+1}$
    $\epsilon_B(f)(x_i) \leftarrow \min\{f(x_l), f(x_{l+1}), \ldots, f(x_{i-1}), f(x_i), f(x_{i+1}), \ldots, f(x_{u-1}), f(x_u)\}$
  end for

A vector of all the lower bounds (LB) x_l in Algorithm A.1 for each corresponding x_i can be created using a $O(n)$ algorithm using two pointers along the input vector X. One pointer is the current position, the other is a lagging pointer that moves along the vector when required. A simple algorithm to do this is described in Algorithm A.2. To find the upper bound the same algorithm would be employed, but the pointers will start from the right and move down the vector with the second point lagging to the right.

Algorithm A.2 (Find a vector of length n of the x_ls (LBs) from Algorithm A.1).

Require: $X = \{x_1, x_2, \ldots, x_n\}, k = SE.length$
  create the vector X_{LB} *of length* n
  $pointer_{curr} \leftarrow 2$
  $pointer_{LB} \leftarrow 1$
  $X_{LB}[1] \leftarrow 1$ *# {x_1 is its own LB}*
  while $pointer_{curr} \leq n$ ***do***
    if $x_{pointer_{curr}} - x_{pointer_{LB}} \leq \frac{k}{2}$ ***then***
      $X_{LB}[pointer_{curr}] \leftarrow pointer_{LB}$
      $pointer_{curr} \leftarrow pointer_{curr} + 1$
    else
      $pointer_{LB} \leftarrow pointer_{LB} + 1$ *# {check next element for LB}*
    end if
  end while
  return X_{LB}

A.4 Continuous line segment erosion implementation

```
erode.cts.quick<-function(x,f,se.span){

  out.vector<-NULL
  nx<-length(x)
  x.span<-x[nx]-x[1]
  k<-se.span
  t1<-proc.time()[3]
  isAppend<-FALSE

  if(k>=x.span){
    cat("Warning: structuring element spans the entire input set \n")
    cat("The input f vector has been output \n")
    return(f)
  }else{
    m<-ceiling(x.span/k)
    mk<-m*k
    if(!((x[1]+mk) == x[nx])){
      x<-c(x,x[1]+mk)
      f<-c(f,+Inf)
      isAppend<-TRUE
      nx<-nx+1
      x.span<-x[nx]-x[1]
    }
    k.blocks<-c(0,findInterval(x,seq(x[1],x[1]+(m-1)*k,by=k)),m+1)
    p<-k.blocks[1]
    q<-k.blocks[nx+2]
    g<-rep(0,nx)
    h<-rep(0,nx)
    r<-rep(0,nx)
    i<-1
    j<-nx
    while(i<=nx){
      this.p<-k.blocks[i+1]
      this.q<-k.blocks[j+1]
      if(p==this.p){
        g[i]<-min(g[i-1],f[i])
      }else{
        g[i]<-f[i]
      }
      if(q==this.q){
        h[j]<-min(h[j+1],f[j])
      }else{
        h[j]<-f[j]
      }
      p<-this.p
      q<-this.q
      i<-i+1
```

```
48        j<-j-1
49      }
50      k.left<-k/2
51      low.bound.index<-nx-rev(findInterval(rev(-x),rev(-(x+k.left))))+1
52      upp.bound.index<-findInterval(x+k.left,x)
53      out.vector<-pmin(h[low.bound.index],g[upp.bound.index])
54      which.low<-which(k.blocks[low.bound.index]==k.blocks[upp.bound.index+1])
55      out.vector[which.low]<-h[low.bound.index[which.low]]
56      which.hi<-which(k.blocks[low.bound.index+1]==k.blocks[upp.bound.index+2])
57      out.vector[which.hi]<-g[upp.bound.index[which.hi]]
58      if(isAppend) out.vector<-out.vector[-nx]
59      cat("Completed morphological erosion (cts scale) in"
60        ,sprintf("%.2f",proc.time()[3]-t1),"seconds \n")
61      return(out.vector)
62    }
63  }
```

Appendix B

Normalisation implementation

B.1 Empirical quantile normalisation implementation

```r
### Input: msData - the spectra intensities in matrix
###        where columns are spectra 1,2,...,n
msEmpiricalQuantNorm<-function(msData)
{
  msD<-msData
  nSpec<-ncol(msD)
  nDim<-nrow(msD)

  orders<-apply(msD, 2, order)
  reorders<-apply(orders, 2, order)
  # order each column into ascending order
  for(i in 1:nSpec) msD[,i]<-msD[orders[,i],i]
  #replace ordered columns with row means
  rmeans<-rowMeans(msD)
  for(i in 1:nSpec) msD[,i]<-rmeans
  #put back into the original order (with changed values)
  for(i in 1:nSpec) msD[,i]<-msD[reorders[,i],i]

  return(msD)
}
```

B.2 MA normalisation implementation

```
### adjM(): Used by intensAdj(), performs LOESS regression on ordered MA-vals
### Input: ordered dependent variable A with corresponding M
### Returns: adjusted M values, M_t^*
adjM<-function(ordered.M,ordered.A)
{
  MA.finites<-is.finite(ordered.M) #only include values > -∞
  finites.M<-ordered.M[MA.finites]
  finites.A<-ordered.A[MA.finites]

  MAloess<-loess(finites.M~finites.A
    ,span=0.40,degree=2,family="symmetric",normalize=FALSE)

  finites.M<-finites.M-MAloess$fitted # make adjustments
  ordered.M[MA.finites]<-finites.M # and return adjusted values
  return(ordered.M)
}

### intensAdj(): Perform MA adjustment on two vectors
### Input: Two spectra vectors F_1 and F_2
### Returns: MA adjusted F_1 and F_2 values, F_1^* and F_2^* respectively
intensAdj<-function(F1,F2)
{
  t1<-proc.time()[3] ### get start time
  V1<-log2(F1) # Will produce -∞ for log_2(0)
  V2<-log2(F2)
  M<-V1-V2
  A<-(V1+V2)/2
  ### A is the dependent regression variable,
  ### ordering required for adjM function
  ordered.indxs<-order(A)
  ordered.A<-A[ordered.indxs]
  ordered.M<-M[ordered.indxs]
  ordered.M<-adjM(ordered.M,ordered.A)
  ### get indexes of original ordering
  orig.order<-order(ordered.indxs)
  M.dash<-ordered.M[orig.order]

  orig.finites<-is.finite(M) #update values requiring updating
  F1[orig.finites]<-2^(A[orig.finites]+M.dash[orig.finites]/2)
  F2[orig.finites]<-2^(A[orig.finites]-M.dash[orig.finites]/2)

  delta.t<-sprintf("%.2f",proc.time()[3]-t1) ### time elapsed
  cat("Completed MA Normalisation in",delta.t,"seconds \n")
  return(list(F1adj=F1,F2adj=F2))
}
```

Appendix C

Peak alignment algorithms and implementation

C.1 NM-alignment code

```
##############################################################################
#################################### Function ################################
##############################################################################
#
# Wmatrix(): Create a peak similarity matrix between an N- and M-alignment
#
##############################################################################
#################################### Input ###################################
##############################################################################
#
# Nmatchedpeaks: K x N matrix of matched pairs of the N alignment
# Npeaklists: a list object containing N matricies of
#         (time,intensityVector) pairs: (n_a x (n_Comp+1) matrix, a=1,...,N)
# Mmatchedpeaks: L x M matrix of matched pairs of the M alignment
# Mpeaklists: a list object containing M matricies of
#         (time,intensityVector) pairs: (n_b x (n_Comp+1) matrix, b=1,...,M)
#
##############################################################################
#
# e.g. Nmatchedpeaks =
# [ 1 0 1 0 ]
# [ 0 1 2 1 ]
# [ 0 0 0 2 ]
# [ 2 2 0 0 ]
# [ 3 3 3 3 ]
# [ 0 4 0 4 ]
# [ . . . . ]
# [ . . . . ]
# [ . . . . ]
```

```
#
# here K x N (N=4) matrix
#
# Npeaklist=
# [[1]]
# [t_{1,1} t_{1,2} ... t_{1,n_1} ]
# [x_{1,1} x_{1,2} ... x_{1,n_1} ]
#
# [[2]]
# [t_{2,1} t_{2,2} ... t_{2,n_2} ]
# [x_{2,1} x_{2,2} ... x_{2,n_2} ]
#
# ...
#
# [[N]]
# [t_{N,1} t_{N,2} ... t_{N,n_N} ]
# [x_{N,1} x_{N,2} ... x_{N,n_N} ]
#
# where t_{i,j} is the time point j-th peak for the i-th spectrum
# where x_{i,j} is the vector of intensities (n_{Comp} long)
#      i.e. n_{Comp} x 1 matrix for the j-th peak for the i-th spectrum
# NB: each list item is a (n_{Comp}+1) x n_N matrix
#
# the _i_ co-ord is the row in the Nmatchedpeaks
# the _a_ co-ord is the column number of Nmatchedpeaks (the spectrum number)
# the _p_ co-ord is the peak number for the _a_th spectrum
#
# the _j_,_b_ and _q_ co-ords are defined similarly for the M-alignment

Wmatrix<-function(Nmatchedpeaks,Npeaklists,Mmatchedpeaks,Mpeaklists
          ,D,expon,lambda){

   K<-nrow(Nmatchedpeaks)
   N<-ncol(Nmatchedpeaks)
   L<-nrow(Mmatchedpeaks)
   M<-ncol(Mmatchedpeaks)
   W<-matrix(0,nrow=K,ncol=L)

   for(i in 1:K){
      for(j in 1:L){
         numerator<-0
         denominator<-0
         for(a in 1:N){
            p<-Nmatchedpeaks[i,a]
            if(p>0){
               for(b in 1:M){
                  q<-Mmatchedpeaks[j,b]
                  if(q>0){
                     t_a<-Npeaklists[[a]][1,p]
                     p_a<-Npeaklists[[a]][-1,p]
                     t_b<-Mpeaklists[[b]][1,q]
                     p_b<-Mpeaklists[[b]][-1,q]
```

```
                    numerator<-numerator+
                      PeakSim(p_a,t_a,p_b,t_b,D,expon,lambda)
                    denominator<-denominator+1
                  }
                }
              }
          }
          if(denominator>0) W[i,j]<-numerator/denominator
          else W[i,j]<-0
      }
    }
    return(W)
}
```

```
##############################################################################
#################################### Function ################################
##############################################################################
#
# guideTreePeakAlign(): for peak list data, create successive N- and M-alignments
#              until all spectra are aligned.
#
##############################################################################
#################################### Input ###################################
##############################################################################
#
# msD: MS Data, a T x n matrix of MS intensities. One column per spectra.
# peaklistlist: see below
# in.param: [ D expon lambda G maxM ]-tuple as a vector
#
##############################################################################
#################################### Output ##################################
##############################################################################
#
### A list containing the following elements:
# dendro:
# stepwise.peaks: a list where each element is the successive amalgamation data
# amalpeaks: the final matrix of aligned peaks. Columns are named spectra, rows
#    are
#
##############################################################################

### peaklistlist=
# [[1]]
# [t_{1,1} t_{1,2} ... t_{1,n_1} ]
# [x_{1,1} x_{1,2} ... x_{1,n_1} ]
#
# [[2]]
# [t_{2,1} t_{2,2} ... t_{2,n_2} ]
# [x_{2,1} x_{2,2} ... x_{2,n_2} ]
#
# .
# .
```

```
# .
#
# [[N]]
# [t_{N,1} t_{N,2} ... t_{N,n_N} ]
# [x_{N,1} x_{N,2} ... x_{N,n_N} ]
#
# where t_{i,j} is the time point j-th peak for the i-th spectrum
# where x_{i,j} is the vector of intensities (nComp long i.e. nComp x 1 matrix)
#         for the j-th peak for the i-th spectrum
# NB: each list item is a (n_Comp+1) x n_N matrix

guideTreePeakAlign<-function(msD,peaklistlist,in.param)
{

  D<-in.param[1]
  nC<-nrow(peaklistlist[[1]])-1
  expon<-in.param[2]
  lambda<-in.param[3]
  G<-in.param[4]
  maxM<-in.param[5]

  nPat<-length(peaklistlist)
  Pats<-1:nPat

  cat("Calculating merge sequence for spectra \n")

  fordist<-t(msD$intensity)
  hc<-hclust(as.dist(fordist,diag=FALSE,upper=FALSE),"average")

  ### find amalgamation sequence
  ### see ?hclust for information on the merge matrix:
  #    "an n-1 by 2 matrix. Row i of merge describes the merging of clusters
  #    at step i of the clustering. If an element j in the row is negative,
  #    then observation -j was merged at this stage. If j is positive then
  #    the merge was with the cluster formed at the (earlier) stage j of the
  #    algorithm. Thus negative entries in merge indicate agglomerations of
  #    singletons, and positive entries indicate agglomerations of
  #    non-singletons."
  amalg<-hc$merge

  nAmal<-nrow(amalg)
  # alignment of peaklistlist (a.pll)
  a.pll<-vector(length=nAmal,mode="list")
  Npeaks<-NULL
  Npeaklist<-NULL
  Mpeaks<-NULL
  Mpeaklist<-NULL

  # start
  for(aindx in 1:nAmal)
  {

```

```
91      # if patsToGetN or patsToGetM are positive,
92      #    ... it is a single spectrum (1-alignment)
93      # if negative, it is a previous N/M-alignment (N,M>1)
94      patsToGetN<--amalg[aindx,1]
95      patsToGetM<--amalg[aindx,2]
96
97      printPatsN<-sprintf("%03d",patsToGetN)
98      printPatsM<-sprintf("%03d",patsToGetM)
99      amalg.str<-"Amalgamting patient"
100     if(patsToGetN>0 && patsToGetM>0){
101       cat(amalg.str,"s ",printPatsN," and ",printPatsM,"\n",sep="")
102     }else if(patsToGetN>0 && patsToGetM<0){
103       cat(amalg.str,printPatsN,"to previously amalgamated patients\n")
104     }else if(patsToGetN<0 && patsToGetM>0){
105       cat(amalg.str,printPatsM,"to previously amalgamated patients\n")
106     }else{
107       cat("Amalgamting two clusters of previously amalgamated patients\n")
108     }
109
110     ### prepare N-Alignment data
111     if(patsToGetN>0){ # if a single spectrum (1-alignment)
112       Npeaks<-matrix(1:ncol(peaklistlist[[patsToGetN]]),ncol=1)
113       Npeaklist<-peaklistlist[patsToGetN]
114     }else{ # if a previously aligned N-alignment (N>1)
115       Npeaks<-a.pll[[-patsToGetN]]
116       patsToGetN<-as.numeric(colnames(Npeaks))
117       Npeaklist<-peaklistlist[patsToGetN]
118     }
119
120     ### prepare M-Alignment data
121     if(patsToGetM>0){ # if a single spectrum (1-alignment)
122       Mpeaks<-matrix(1:ncol(peaklistlist[[patsToGetM]]),ncol=1)
123       Mpeaklist<-peaklistlist[patsToGetM]
124     }else{ # if a previously aligned M-alignment (M>1)
125       Mpeaks<-a.pll[[-patsToGetM]]
126       patsToGetM<-as.numeric(colnames(Mpeaks))
127       Mpeaklist<-peaklistlist[patsToGetM]
128     }
129
130     ### use Wmatrix() function
131     Wm<-Wmatrix(Npeaks,Npeaklist,Mpeaks,Mpeaklist,D,expon,lambda)
132     ### use S-W alignment function to estimate maximum path
133     ### see: https://code.google.com/p/swalign/
134     estPM<-SWalign(Wm,G,maxM)
135     ### estPM is a data.frame of (i,j) locations of the maximum path
136     ### the data.frame is 2 columns for i,j points
137
138     nN<-ncol(Npeaks) ### no. of peaks in N-align
139     nM<-ncol(Mpeaks) ### no. of peaks in M-align
140     nK<-nrow(estPM)  ### no. of peaks in new N:M-align
141     ### apllTemp:
142     ###      (a)lignment of (p)eak (l)ist (l)ist, (temp)orary
```

```
143     ### Matrix of peak indicators. The n_K rows represent the n_K peaks
144     ###    from the N:M-alignment.
145     ### Entries apllTemp[i,j] are ==
146     ### { 0 if that N:M-aligned peak does not exist in spec j (column j)
147     ### { _else_ a non-zero indicator, the peak number from within the
148     ###                           1-alignment from spectrum j (column j)
149     apllTemp<-matrix(0,nrow=nK,ncol=nN+nM)
150     mzValsTemp<-NULL
151     AveMzValsTemp<-NULL
152     for(n.k in 1:nK)
153     {
154       if(estPM[n.k,1]>0) # if the peak exists in the N-alignment
155       {
156         # transfer peak info from N-align to new N:M-align matrix
157         apllTemp[n.k,1:nN]<-Npeaks[estPM[n.k,1],]
158         for(i in 1:nN) if(apllTemp[n.k,i]>0) mzValsTemp<-
159             c(mzValsTemp,Npeaklist[[i]][1,apllTemp[n.k,i]])
160       }
161       if(estPM[n.k,2]>0) # if the peak exists in the M-alignment
162       {
163         # transfer peak info from N-align to new N:M-align matrix
164         apllTemp[n.k,(nN+1):(nN+nM)]<-Mpeaks[estPM[n.k,2],]
165         for(i in (nN+1):(nN+nM)) if(apllTemp[n.k,i]>0) mzValsTemp<-
166             c(mzValsTemp,Mpeaklist[[i-nN]][1,apllTemp[n.k,i]])
167       }
168       # get ave m/z of all aligned peaks
169       AveMzValsTemp<-c(AveMzValsTemp,mean(mzValsTemp))
170       mzValsTemp<-NULL
171     }
172     ### change row order if averaging m/z has changed peak location order
173     mzReOrder<-order(AveMzValsTemp)
174     apllTemp<-apllTemp[mzReOrder,]
175
176     allPat<-c(patsToGetN,patsToGetM)
177     colnames(apllTemp)<-allPat
178     ### clean up N:M-alignment to preserve spectrum order
179     patOrder<-order(allPat)
180     apllTemp<-apllTemp[,patOrder]
181
182     a.pll[[aindx]]<-apllTemp
183   }
184   ### return list() object of peak amalgamation/alignment,
185   ###     including intermediate steps
186   outlist<-list(dendro=hc,stepwise.peaks=a.pll[-nAmal],amalpeaks=a.pll[[nAmal]])
187   return(outlist)
188
189 }
```

Appendix D

Surrogate variable analysis implementation

Please note the function `getH()` (line 47 below) is the code available in the `DanteR` package to determine the number of significant surrogate variables. The function `mulReg(Y,X)` performs sequential linear regressions on the columns of the input `Y` using a fixed effects design matrix `X`. `mulReg()` returns a list containing the following vectors and matrices: $\texttt{RES}_{n\times P}$, residual matrix after `Y` has been regressed; $\texttt{BETA}_{d\times P}$, matrix of the regression coefficients, P columns for each regression; $\texttt{TVALS}_{d\times P}$, the corresponding t-statistics; $\texttt{PVALS}_{d\times P}$ the corresponding p-values of `TVALS`; $\texttt{FPVALS}_{P\times 1}$, p-value for each linear regression corresponding to the null model F-statistic.

```
1  ################################### FUNCTION ###################################
2  #### doSVA: Perform SVA using the model:
3  ####        Y_j = μ_j + Xα_j + Zβ_j + Wδ_j + e_j
4  ################################### INPUTS #####################################
5  #### Y: is a n × p matrix, where each p columns are regressed
6  #### Intecept: boolean; do we want to fit a mean value? (yes, in most cases)
7  #### X: is a n × d_α design matrix of the factors of interest
8  #### Z: is a n × d_β design matrix of the incidental experimental factors
9  #### nosigsv: the number (referred to as H in some papers) of significant
10 ####         eigen vecs if NULL, the function will determine. If less than
11 ####         1, no W computed
12 #### verbose: boolean, whether the surragate variable matrix, W is returned
13 #### seed: an integer to feed into 'set.seed()' for reproducable results
14 ################################### OUTPUTS ####################################
15 #### Ytilde: the Y matrix with Zβ_j + Wδ_j removed
16 #### pvals: the p-values of Ytilde regressed on μ_j + Xα_j + Zβ_j + Wδ_j
17 #### tvals: the corresponding t-statistics
18 #### betas: the corresponding α_j,β_j,δ_j estimates
19 #### paramlabels: a combination of I (intercept), X, Z, W to signify the
20 ####         relevent rows of p-vals/tvals/betas
21 #### W: the eigen vectors matrix
```

```
#### H: the number of columns of W (used eigen-vectors)
##############################################################################
doSVA<-function(
  Y,Intercept=TRUE,X=NULL,Z=NULL,nosigsv=NULL,verbose=FALSE,seed=NULL
){
  n<-nrow(Y)
  thisInt<-IXZ<-NULL
  if(Intercept) thisInt<-matrix(1,nrow=n,ncol=1,dimnames=list(NULL,"Intcpt"))
  if(is.null(thisInt) && is.null(X) && is.null(Z))
  {
    cat("At least one of: Intercept, X and Z must be specified \n")
    return(NULL)
  } else IXZ<-cbind(thisInt,X,Z)
  kparam<-ncol(IXZ)
  colmarkers<-rep("",kparam)
  indx<-0
  if(!is.null(thisInt)) colmarkers[indx<-indx+1]<-"I"
  if(!is.null(X)) colmarkers[(indx<-indx+1):(indx<-indx+ncol(X)-1)]<-"X"
  if(!is.null(Z)) colmarkers[(indx<-indx+1):(indx<-indx+ncol(Z)-1)]<-"Z"

  RIXZ<-multReg(Y,IXZ,createNAvals=TRUE,seed=seed)
  thissvd<-svd(RIXZ$RES)

  W<-H<-NULL
  if(is.null(nosigsv)){
    H<-getH(RIXZ$RES,IXZ,nullsig=0.1,verbose=FALSE)
    if(H<1) cat("No significant surrogate variables found \n")
  }else{
    H<-nosigsv
  }
  if(H<1){
    cat("No surrogate variables will be used \n")
  }else{
    cat("Using H=",H," significant surrogate variables \n",sep="")
    W<-as.matrix(thissvd$u[,1:H])
    colnames(W)<-paste("W",1:H,sep="")
    colmarkers<-c(colmarkers,rep("W",H))
  }
  IXZW<-cbind(IXZ,W)
  Rtilde<-multReg(Y,IXZW)
  removecols<-colmarkers %in% c("Z","W")
  ZBetaWDelta<-0
  if(sum(removecols)) ZBetaWDelta<-as.matrix(IXZW[,removecols]) %*%
                as.matrix(Rtilde$BETA[removecols,])
  Ytilde<-Y-ZBetaWDelta
  if(verbose) return(list(Ytilde=Ytilde,paramlabels=colmarkers,W=W,H=H))
  else return(Ytilde)
}
```

Appendix E

Supplementary detail of classification methods

E.1 Pairwise fusion linear discriminant analysis code

```
####### FUNCTION: createPFldaobj()
### estimate parameters of PF-DA model, so that a discrim function created

####### input:
### X: a n x p matrix, of n obs and p variables
### Xclass: a vector of length n of the classes (must be a factor variable)
### priors: a vector of length K (#classes) with elements in (0,1)

createPFldaobj<-function(X,Xclass,lambdar=1,priors=NULL,alph=NULL,wts=NULL)
{
  N<-length(Xclass)
  P<-ncol(X)
  nks<-table(Xclass)
  classnames<-levels(Xclass)
  K<-length(classnames)

  ### if not supplied, make π̂_k data proportions
  if(is.null(priors)) priors<-nks/N

  if(length(priors)!=K){
    cat("The length of priors and the total number
        of groups must be equal \n")
    return(NULL)
  }else if(is.null(alph) & (N<P)){
    cat("Alpha is suggested for n<p data \n")
  }else if(N!=nrow(X)){
    cat("The length of Xclass and the number
        of rows in X must agree \n")
    return(NULL)
```

```
30    }else if(!all(nks>1)){
31      cat("There needs to be at least two obs in each
32          group for variances to be computed \n")
33      return(NULL)
34    }
35
36    Xclassint<-as.integer(Xclass)
37    transMeans<-colMeans(X)
38    X<-X-matrix(rep(transMeans,N),nrow=N,byrow=TRUE)
39
40    ##### create μ_k = [μ_k1,...,μ_kp] vectors
41    ##### place on top of each other to get KxP matrix
42    MuMat<-matrix(0,nrow=K,ncol=P)
43    for(k in 1:K) MuMat[k,]<-colMeans(X[Xclassint==k,])
44    MuIter<-MuMat
45
46    ##### create Σ
47    Sigma<-matrix(0,nrow=P,ncol=P)
48    for(k in 1:K)
49    {
50      rowuse<-which(Xclassint==k)
51      Sigma<-Sigma+length(rowuse)*cov.wt(X[rowuse,],cor=FALSE
52                                         ,center=TRUE,method="ML")$cov
53    }
54    Sigma<-Sigma/N
55    ##### extract Diag elements
56    sigmasqs<-diag(Sigma)
57
58    if(!(is.null(alph) | is.null(wts))) sigmasqs<-sigmasqs+alph*wts
59    else if(!is.null(alph)) sigmasqs<-sigmasqs+rep(alph,P)
60
61    #### Now start iterative estimation of the ℓ_1 penalised means
62    #### Note "squig" is used for the ML estimates
63    G<-matrix(0,nrow=K,ncol=K)
64    deltatol<-1e-10
65    deltaMu<-1
66    maxIter<-500
67    itcount<-0
68    while(deltaMu>(1e-5) && itcount<maxIter)
69    {
70      deltaMuNumer<-0
71      deltaMuDenom<-0
72      itcount<-itcount+1
73
74      ### For each of the features
75      for(j in 1:P)
76      {
77        beta.t.j<-MuIter[,j]
78        musqig.j<-MuMat[,j]
79        sqigY<-X[,j]
80        sqigX<-matrix(0,nrow=N,ncol=K)
81        G<-matrix(0,nrow=K,ncol=K)
```

```
82
83        ### sum_{k=1}^{K-1} sum_{k_dash=k+1}^{K}
84        for(k in 1:(K-1))
85        {
86          for(kdash in (k+1):K)
87          {
88            PFweight<-1/abs(musqig.j[k]-musqig.j[kdash])
89            ### assign updated iterations, or tol value if "zero"
90            muDiffIter<-max(abs(beta.t.j[k]-beta.t.j[kdash]),deltatol)
91            G[k,kdash]<-G[kdash,k]<- -PFweight/muDiffIter
92          }
93        }
94        for(k in 1:K)
95        {
96          sqigX[which(Xclassint==k),k]<-1
97          ### note the diag elements of G can be calculated as the sum of the column
98          G[k,k]<- -sum(G[,k])
99        }
100       #### M^ = (B^T B + lambda sigma_j^2 G)^{-1} B^T J
101       MuIter[,j]<-solve(t(sqigX)%*%sqigX+lambdar*sigmasqs[j]*G)%*%(t(sqigX)%*%sqigY)
102       deltaMuNumer<-deltaMuNumer+sum(abs(MuIter[,j]-beta.t.j))
103       deltaMuDenom<-deltaMuDenom+sum(abs(beta.t.j))
104     }
105     ### our break loop value
106     deltaMu<-deltaMuNumer/deltaMuDenom
107
108   }
109   cat("Iterations performed to aquire a solution:",itcount
110        ,"| Final tol val:",deltaMu," \n")
111
112   MuIter[which(MuIter<deltatol)]<-0
113
114   ### 1/2 sum_{j=1}^{p} mu^2_{kj} constant term
115   consts<-rep(0,K)
116   for(k in 1:K) consts[k]<-0.5*sum(MuIter[k,]^2/sigmasqs)
117   ### mu^2_{kj}/sigma^2_p term
118   lins<-vector(mode="list",length=K)
119   for(k in 1:K) lins[[k]]<-MuIter[k,]/sigmasqs
120
121   ### return calculated information as list object
122   return(list(classes=classnames,consts=consts,lins=lins,prior=priors
123               ,meanadj=transMeans,MuIter=MuIter,InitMu=MuMat))
124 }
```

```r
####### FUNCTION: PFldapredict()
### estimate probabilities and class of new inputs

####### input:
### ldaobj: an object created by createPFldaobj()
### Xnew: a n x p matrix, of n obs and p variables.
###         May also be a single numeric vector (n=1) of length p
### priors: a vector of length K (#classes) with elements in (0,1)

PFldapredict<-function(ldaobj,Xnew,priors=NULL)
{
  if(is.vector(Xnew,mode="numeric")){
    Xnew<-matrix(Xnew,nrow=1)
  }else if(!is.matrix(Xnew)){
    cat("Xnew must be numeric and either a vector or matrix \n ")
  }

  Nnew<-nrow(Xnew)
  ### centre each feature at 0, as done on the training data
  Xnew<-Xnew-matrix(rep(ldaobj$meanadj,Nnew),nrow=Nnew,byrow=TRUE)
  Xclasses<-ldaobj$classes
  K<-length(Xclasses)
  if(is.null(priors)) priors<-ldaobj$prior

  outposteriors<-matrix(0,nrow=Nnew,ncol=K)
  colnames(outposteriors)<-Xclasses

  ### discrim function: $\delta_k(x_{new})$
  for(i in 1:Nnew){
    for(k in 1:K){
      outposteriors[i,k]<-log(priors[k]) - ldaobj$consts[k]
                                  + sum(Xnew[i,]*ldaobj$lins[[k]])
    }
  }

  ### predicted class is arg max
  ### $p(G_K|x_{new}) = \frac{P(x_{new}|G_k)P(G_k)}{\sum_{i=1}^{K} P(x_{new}|G_i)P(G_i)}$
  predclasses<-rep(0,Nnew)
  for(i in 1:Nnew){
    outposteriors[i,]<-exp(outposteriors[i,])
    predclasses[i]<-which.max(outposteriors[i,])
    outposteriors[i,]<-outposteriors[i,]/sum(outposteriors[i,])
  }
  outpred<-factor(Xclasses[predclasses])

  return(list(pred=outpred,posteriors=outposteriors))
}
```

Appendix F

Pareto Fronts for variable ranking

F.1 Dominating features

Below is the core of the Pareto Front code, finding features that are the dominated as per the definition. Written in `C` to be compiled to a `.so` file (or `.dll` on Windows operating systems) that in turn can be loaded into `R`.

```c
#include <R.h>

void domfeat(int *n, double *obja, double *objb, int *domvec)
{
  int i, j, nonDomI;
  for (i=0; i<*n; i++)
  {
    j=0;
    nonDomI=1;
    while(nonDomI && j<*n)
    {
      if((obja[i]<obja[j]) && (objb[i]<=objb[j])) nonDomI=0;
      else if((objb[i]<objb[j]) && (obja[i]<=obja[j])) nonDomI=0;
      j+=1;
    }
    domvec[i]=-nonDomI+1;
  }
}
```

F.2 Pareto Front wrapper functions

```r
library(animation)
dyn.load("domfeat.so")

##########################################
```

```
5  ##########################################
6  #
7  # Pairwise case of Pareto Fronts
8  #     obj is the 2 × n matrix of the two vectors of length n for
9  #      the features/observations of the 2 criteria/objective functions
10 #     istomin is a boolean vector of whether the criteria obj_1, obj_2
11 #      are to be minimised (=TRUE), respectively
12 #
13 ##########################################
14 ##########################################
15
16 #
17 # This function returns a vector of 'dominated' observations (Boolean,
18 # length n vector) FALSE=Pareto front, TRUE=dominated observation
19 #
20
21 domFeaturesPW<-function(obj,istomin)
22 {
23   #if to be minimised then just make negative and maximise
24   obj[,istomin]<- -obj[,istomin]
25   n<-as.integer(nrow(obj))
26   obj1<-as.double(obj[,1])
27   obj2<-as.double(obj[,2])
28   domvec<-as.integer(rep(0,n)) #output vector
29
30   return(as.logical(.C("domfeat",n,obj1,obj2,domvec)[[4]]))
31 }
32
33 ##########################################
34 ##########################################
35 #
36 # General case
37 #     objmatrix is n × m matrix. n features/observations and
38 #     m criteria/objective functions istominvec is a boolean vec
39 #     of length m to say whether the criteria are to be minimised
40 #
41 ##########################################
42 ##########################################
43
44 #
45 # This function returns a vector of 'dominated' observations (Boolean,
46 # length n vector) FALSE=Pareto front, TRUE=dominated observation
47 # same as pairwise but the input can take more than two objective functions
48 #
49
50 domFeatures<-function(objmatrix,istominvec)
51 {
52   n<-as.integer(nrow(objmatrix))
53   m<-ncol(objmatrix)
54   objmatrix[,istominvec]<- -objmatrix[,istominvec]
55   vecdomvec<-rep(1,n)
56   indxs<-combn(m,2)
```

```
57    nm<-ncol(indxs)
58    i<-0
59    while(i<nm)
60    {
61      i<-i+1
62      # call pairwise function, take the intersection of previous
63      # dominated observations remembering the intersection(s)
64      # of dominated in the same as unions(s) of Pareto fronts
65      vecdomvec<-vecdomvec * .C("domfeat"
66                ,n
67                ,as.double(objmatrix[,indxs[1,i]])
68                ,as.double(objmatrix[,indxs[2,i]])
69                ,as.integer(rep(0,n)))[[4]]
70    }
71    return(as.logical(vecdomvec))
72  }
73
74  ##########################################
75  ##########################################
76  #
77  # Sucessive Pareto Fronts
78  #     noFronts is the # of pareto fronts required
79  #
80  #     fn returns a vector of length n
81  #           each element is labelled the pareto front #,
82  #           0 is dominated even after noFronts found
83  #
84  ##########################################
85  ##########################################
86
87  paretoFronts<-function(noFronts,objmatrix,istominvec)
88  {
89    objmatrix[,istominvec]<- -objmatrix[,istominvec]
90    n<-as.integer(nrow(objmatrix))
91    m<-ncol(objmatrix)
92    pfvec<-rep(0,n) #output vector
93    #once a front is found we need to set the correponding values to ∞ or
94    # −∞ so they won't be chosen again
95    #try: as.numeric(c(TRUE,FALSE,TRUE))*2-1 to see what the next line is doing
96    #if Min then set 1, ifMax then set -1 (the sign of the Inf if we find front
97    #   and have to put to a value)
98    ourInfs<-min(objmatrix)-1
99    allFrontsFound<-FALSE
100
101   i<-0
102   while(i<noFronts && !allFrontsFound) #go thru all fronts required
103   {
104     i<-i+1
105     df<-domFeatures(objmatrix,rep(FALSE,m)) #general m obj vectors function
106     # pf.i are the indexs of the output vector that need to be updated
107     # with the pareto front number
108     pf.i<-(!df) & (pfvec<1)
```

```
109       pfvec[pf.i]<-i
110       # re-assign values were pareto front found
111       objmatrix[pf.i,]<-ourInfs
112       if(all(pfvec>0)) allFrontsFound<-TRUE
113     }
114     return(pfvec)
115   }
116
117   ##########################################
118   ##########################################
119   #
120   # Leave-one-out/k-fold feature ranking
121   #
122   # returns a vector of length n with values ∈ (0,1] for feature importance
123   #
124   ##########################################
125   ##########################################
126
127   #
128   # Same inputs of previous functions, with folds (aka k-fold cross
129   # validation) and reps is the number of times we re-do the cross
130   # validation fold=1 or the length of the input (i.e. n) creates
131   # leave-one-out cross validation
132
133   paretoRanking<-function(objmatrix,istominvec,noFronts=20,folds=1,reps=5)
134   {
135     objmatrix[,istominvec]<- -objmatrix[,istominvec]
136     m<-ncol(objmatrix)
137     n<-nrow(objmatrix)
138     pfmetric<-rep(0,n) #output vector
139     nfolds<-n
140     if(folds>1) nfolds<-folds
141     if(nfolds==n) reps<-1
142
143     blocks<-kfcv(nfolds,n)
144     block.nos<-rep(1:nfolds,blocks)
145
146     for(r in 1:reps)
147     {
148       indxs<-sample(1:n) #fresh randomisation each repetition
149       k.f.mat<-cbind(indxs,block.nos) # create the fold 'blocks' of data
150       for(i in 1:nfolds)
151       {
152         rows<-k.f.mat[k.f.mat[,2]==i,1] # find the ith fold to leave out
153         # call general function with ith fold removed
154         calcfronts<-paretoFronts(noFronts,objmatrix[-rows,],rep(FALSE,m))
155         # which are non-dominated
156         whichnondom<-calcfronts>0
157         # if you are ont the first front you get 1, second=1/2, third=1/3,
158         # ..., jth=1/j, else 0
159         pfmetric[-rows][whichnondom]<-
160             pfmetric[-rows][whichnondom]+1/calcfronts[whichnondom]
```

```
    }
  }
  #now divide by maximum posible value i.e. (nfolds-1)*reps so output in [0,1]
  pfmetric<-pfmetric/((nfolds-1)*reps)
  return(pfmetric)
}

##############################################################################
# Below are three metrics that can possibly describe the value of
#   variables/fetures to discriminate betwwen classes
##############################################################################

# minIntraClassVar(): find the minimum WITHIN class variance of the K groups
# interClassVar(): find the variance of means/centroids of the K groups
# maxInterClassDist(): possibly correlated with interClassVar, find the dist
#              MAXIMUM between the K group's centroids/means

# The rationale of the last metric is that a variable/feature that only
# seperates two of the K classes  is undervalued by the Fisher score because
# it may not separate the K-2 classes remaining well.
# ... And a separation of two classes (in conjunction with other variables)
#      is important information for the discriminant model

##############################################################################
#### ds: a data.frame or matrix (numeric values only/factors will be dealt
####  with as integers)  class vec: a vector correspong to the class of
####    the rows of ds
##############################################################################

minIntraClassVar<-function(ds,class.vec){

  dsfs<-ds
  if(!is.matrix(dsfs)) dsfs<-data.matrix(dsfs)

  p<-ncol(dsfs)
  K<-length(levels(class.vec))
  n.all<-length(class.vec)
  n.i<-0

  intraClassVar<-Inf
  mean.j<-colMeans(dsfs)

  for(i in 1:K){
    true.vec<-(as.integer(class.vec)==i)
    n.i<-length(which(true.vec))
    mean.class<-colMeans(as.matrix(dsfs[true.vec,],ncol=p))
    var.class<-colSums(as.matrix((dsfs[true.vec,]-rep(mean.class,each=n.i))^2,ncol=p))
    intraClassVar<-pmin(intraClassVar,var.class/(n.i-1))
  }
  return(intraClassVar)
}

```

```
213 interClassVar<-function(ds,class.vec){
214
215   dsfs<-ds
216   if(!is.matrix(dsfs)) dsfs<-data.matrix(dsfs)
217
218   p<-ncol(dsfs)
219   K<-length(levels(class.vec))
220
221   interClassVar<-0
222   mean.j<-colMeans(dsfs)
223
224   for(i in 1:K){
225     true.vec<-(as.integer(class.vec)==i)
226     n.i<-length(which(true.vec))
227     mean.class<-colMeans(as.matrix(dsfs[true.vec,],ncol=p))
228     var.class<-((mean.j-mean.class)^2)
229     interClassVar<-interClassVar+var.class
230   }
231   return(interClassVar/(K-1))
232 }
233
234 maxInterClassDist<-function(ds,class.vec){
235
236   dsfs<-ds
237   if(!is.matrix(dsfs)) dsfs<-data.matrix(dsfs)
238
239   p<-ncol(dsfs)
240   K<-length(levels(class.vec))
241
242   interClassDist<--Inf
243   mean.j<-colMeans(dsfs)
244
245   for(i in 1:K){
246     true.vec<-(as.integer(class.vec)==i)
247     mean.class<-colMeans(as.matrix(dsfs[true.vec,],ncol=p))
248     interClassDist<-pmax(interClassDist,abs(mean.j-mean.class))
249   }
250   return(interClassDist)
251 }
```