DQF For Clustering

2023-03-23

The Algorithm

1. Run data through `dqf.subset`
2.

Dependent Functions

Calculate winsorized standard deviation

```
sd.w <- function(x, k) {
  # computes the windsorized standard deviation, called by dqf.outlier
  # Inputs:
  ## k: (integer) number of observations at each extreme to alter
  ## x: (vector) numeric data values
  k <- floor(k)
  if (k == 0) {  #corresponds to non-robust adaptive DQF
    return(sd(x))
  } else {
    x <- sort(x)
    n <- length(x)
    x[1:k] <- x[k]
    x[(n-k+1):n] <- x[n-k+1]
    return(sd(x))
  }
}
```

Subsample pairs of points (random is observations are scrambled)

```
subsamp.dqf <- function(n.obs, subsample) {
  # called by dqf.outlier, computes random subset of pairs
  pairs <- c()
  subsample <- floor(subsample/2)*2
  for (i in 1:n.obs) {
    for (j in (i+1):(i+subsample/2)) {
      pairs <- rbind(pairs, c(i,j*(j <= n.obs) + (j-n.obs)*(j > n.obs)))
    }
  }
  return(pairs)
}
```

Ploting functions

```
plot.dqf <- function(dqf,labels=NULL,xlab='',ylab='',main=''){
  x <- seq(.01,1,.01)

  n.functions <- length(dqf[,1])
```

```
  if(is.null(labels)) labels <- rep(1,n.functions)

  plot(x,dqf[1,],t='l',ylim=c(0,max(dqf)),col=labels[1],xlab=xlab,ylab=ylab,main=main)
  for(i in 2:n.functions){
    lines(x,dqf[i,],col=labels[i])
  }
}
```

```
show <- function(length,s){
  labels <- rep(1,length)
  labels[s] <- 2
  return(labels)
}
```

Original Anomaly Detection Function

```
dqf.outlier <- function(data = NULL, gram.mat = NULL, g.scale=2, angle=c(30,45,60), kernel="linear", p1=
  # kernelized version of depthity
  #
  # inputs:
  #   data (matrix or data frame) - a data matrix of explanatory variables
  #   kernel - of form "linear", "rbf" or "poly", or a user defined function
  #   g.scale (scalar) - scales the base distribution G
  #   angle (numeric vector of length 3)- angles of cone from midline, must live between 0 and 90
  #   p1 - first parameter for kernel
  #   p2 - second parameter for kernel
  #   n.splits (integer) - the number of split points at which the DQF is computed
  #   subsample (integer)- the number of random pairs for each observation
  #   z.scale (logical) - should the data be z-scaled first
  #   k.w (integer) - the number of points altered in the windsorized standard deviation
  #   adaptive - if TRUE, uses windsorized standard deviation to scale base distribution
  #   G - base distribution:   "norm" or "unif"
  ##
  # Output:
  #   angle  - vector of angles used, same as inputted
  #   dqf1, dqf2, dqf3 - matrices of depth quantile functions, rows are observations
  if (G=="norm") {
    param1 <- 0; param2 <- 1 }
  if (G=="unif") {
    param1 <- -1; param2 <- 1 }
  if (is.null(data) & is.null(gram.mat))
    stop("Either a data set or Gram matrix must be provided")
  if (min(angle) <= 0 | max(angle) >= 90)
    stop("Angles must be between 0 and 90")
  if (is.null(data))
    n.obs <- nrow(gram.mat)
  if (is.null(gram.mat))
    n.obs <- nrow(data)
  scram <- sample(n.obs)
  pairs <- subsamp.dqf(n.obs, subsample)
  if (is.null(gram.mat)) {
    if (z.scale==TRUE)
      data <- apply(data, 2, scale) #z-scale data
```

```
  if (is.function(kernel)==TRUE)
    kern <- kernel
  if (kernel == "linear") {
    kern <- function(x,y)
      return(sum(x*y))
  }
  if (kernel == "rbf") {
    kern <- function(x,y)
      return(exp(-sum((x-y)^2)/p1))
  }
  if (kernel == "poly") {
    kern <- function(x,y)
      return((sum(x*y)+p2)^p1)
  }
  data <- data[scram,]
  gram <- matrix(0,n.obs, n.obs)
  for (i in 1:n.obs) {
    for (j in i:n.obs) {
      gram[i,j] <- kern(data[i,], data[j,])
      gram[j,i] <- gram[i,j]
    }
  }
} else {
  if (diff(dim(gram.mat))!=0)
    stop("Gram matrix must be square")
  if (isSymmetric(gram.mat) == FALSE)
    stop("Gram matrix must be symmetric")
  gram <- gram.mat
  gram <- gram[scram,]
  gram <- gram[,scram]
}
splits <- get(paste("q", G, sep=""))((1:n.splits)/(n.splits+1),param1, param2) * g.scale
depthity1 <- depthity2 <- depthity3 <- rep(0,length(splits))
norm.k2 <- error.k <- k.to.mid <- rep(0, n.obs)
dep1 <- dep2 <- dep3 <- matrix(0, nrow=nrow(pairs), ncol=n.splits)
qfs1 <- qfs2 <- qfs3 <- matrix(0, nrow=nrow(pairs), ncol=100)
for (i.subs in 1:nrow(pairs)) {
  i <- pairs[i.subs,1];  j <- pairs[i.subs,2]
  for (k in 1:n.obs) {
    norm.k2[k] <- gram[k,k] + 1/4*(gram[i,i]+gram[j,j]) + 1/2*gram[i,j]-gram[k,i]-gram[k,j]
    k.to.mid[k] <- (gram[k,i]-gram[k,j]+1/2*(gram[j,j]-gram[i,i]))/sqrt(gram[i,i]+gram[j,j]-2*gram[i,
    error.k[k] <- sqrt(abs(norm.k2[k] - k.to.mid[k]^2))
  }
  for (c in 1:length(splits)) {
    good <- rep(1, n.obs)
    s <- splits[c] * (sd.w(k.to.mid, k.w)*adaptive + (adaptive==FALSE))
    good[k.to.mid/s > 1] <- 0  #points on other side of cone tip removed
    d.to.tip <- abs(k.to.mid - s)
    good1 <- good * (abs(atan(error.k / d.to.tip)) < (angle[1]/360*2*pi))  #points outside of cone re
    good1 <- good1 * (1 - 2*(sign(k.to.mid)==sign(s)))  #which side of midpoint are they on
    depthity1[c] <- min(c(sum(good1==-1), sum(good1==1)))
    good2 <- good * (abs(atan(error.k / d.to.tip)) < (angle[2]/360*2*pi))  #points outside of cone re
    good2 <- good2 * (1 - 2*(sign(k.to.mid)==sign(s)))  #which side of midpoint are they on
```

```
      depthity2[c] <- min(c(sum(good2==-1), sum(good2==1)))
      good3 <- good * (abs(atan(error.k / d.to.tip)) < (angle[3]/360*2*pi))  #points outside of cone re
      good3 <- good3 * (1 - 2*(sign(k.to.mid)==sign(s)))  #which side of midpoint are they on
      depthity3[c] <- min(c(sum(good3==-1), sum(good3==1)))
    }
    qfs1[i.subs,] <- quantile(depthity1, seq(0,1,length=100), na.rm=TRUE)
    qfs2[i.subs,] <- quantile(depthity2, seq(0,1,length=100), na.rm=TRUE)
    qfs3[i.subs,] <- quantile(depthity3, seq(0,1,length=100), na.rm=TRUE)
  }
  dqf1 <- dqf2 <- dqf3 <- matrix(0,n.obs, 100)
  for (i in 1:n.obs) {
    dqf1[i,] <- apply(qfs1[which(pairs[,1]==i | pairs[,2]==i),],2,mean, na.rm=TRUE)
    dqf2[i,] <- apply(qfs2[which(pairs[,1]==i | pairs[,2]==i),],2,mean, na.rm=TRUE)
    dqf3[i,] <- apply(qfs3[which(pairs[,1]==i | pairs[,2]==i),],2,mean, na.rm=TRUE)
  }
  dqf1 <- dqf1[order(scram),]  #map back to original indicies
  dqf2 <- dqf2[order(scram),]
  dqf3 <- dqf3[order(scram),]

  return(list(angle=angle, dqf1=dqf1, dqf2=dqf2, dqf3=dqf3))
}
```

Subsetable DQFs

Inputs: Same of original DQF function

Returns: Original DQF data, Pairs of points, List of good vectors corresponding to pairs of points, `k.to.mids` vector for adapativity.

```
dqf.subset <- function(data = NULL, gram.mat = NULL, g.scale=2, angle=c(45), kernel="linear", p1=1, p2=
  # kernelized version of depthity
  #
  # inputs:
  #   data (matrix or data frame) - a data matrix of explanatory variables
  #   kernel - of form "linear", "rbf" or "poly", or a user defined function
  #   g.scale (scalar) - scales the base distribution G
  #   angle (numeric vector of length 3)- angles of cone from midline, must live between 0 and 90
  #   p1 - first parameter for kernel
  #   p2 - second parameter for kernel
  #   n.splits (integer) - the number of split points at which the DQF is computed
  #   subsample (integer)- the number of random pairs for each observation
  #   z.scale (logical) - should the data be z-scaled first
  #   k.w (integer) - the number of points altered in the windsorized standard deviation
  #   adaptive - if TRUE, uses windsorized standard deviation to scale base distribution
  #   G - base distribution:   "norm" or "unif"
  ##
  # Output:
  #   angle  - vector of angles used, same as inputted
  #   dqf1, dqf2, dqf3 - matrices of depth quantile functions, rows are observations
  if (G=="norm") {
    param1 <- 0; param2 <- 1 }
  if (G=="unif") {
    param1 <- -1; param2 <- 1 }
  if (is.null(data) & is.null(gram.mat))
```

```
  stop("Either a data set or Gram matrix must be provided")
if (min(angle) <= 0 | max(angle) >= 90)
  stop("Angles must be between 0 and 90")
if (is.null(data))
  n.obs <- nrow(gram.mat)
if (is.null(gram.mat))
  n.obs <- nrow(data)


#scram <- sample(n.obs)
#unscram <- c()
#for(i in 1:length(scram)){
#  unscram <- c(unscram,which(scram==i))
#}

pairs <- subsamp.dqf(n.obs, subsample)
if (is.null(gram.mat)) {
  if (z.scale==TRUE)
    data <- apply(data, 2, scale) #z-scale data
  if (is.function(kernel)==TRUE)
    kern <- kernel
  if (kernel == "linear") {
    kern <- function(x,y)
      return(sum(x*y))
  }
  if (kernel == "rbf") {
    kern <- function(x,y)
      return(exp(-sum((x-y)^2)/p1))
  }
  if (kernel == "poly") {
    kern <- function(x,y)
      return((sum(x*y)+p2)^p1)
  }
  #data <- data[scram,]
  gram <- matrix(0,n.obs, n.obs)
  for (i in 1:n.obs) {
    for (j in i:n.obs) {
      gram[i,j] <- kern(data[i,], data[j,])
      gram[j,i] <- gram[i,j]
    }
  }
} else {
  if (diff(dim(gram.mat))!=0)
    stop("Gram matrix must be square")
  if (isSymmetric(gram.mat) == FALSE)
    stop("Gram matrix must be symmetric")
  gram <- gram.mat
  #gram <- gram[scram,]
  #gram <- gram[,scram]
}
splits <- get(paste("q", G, sep=""))((1:n.splits)/(n.splits+1),param1, param2) * g.scale
depthity1 <- rep(0,length(splits))
norm.k2 <- error.k <- k.to.mid <- rep(0, n.obs)
```

```
  dep1 <- matrix(0, nrow=nrow(pairs), ncol=n.splits)
  qfs1 <- matrix(0, nrow=nrow(pairs), ncol=100)

  ret.pairs <- matrix(0,nrow=nrow(pairs),ncol=2)
  k.to.mids <- matrix(ncol = n.obs, nrow = nrow(pairs))

  ret.gs <- matrix(ncol = n.obs, nrow = n.splits)  # skeleton for goods data.frame
  ret.goods <- list()

  # pairs.goods <- list(pairs=list(c(0,0)),goods=list(data.frame(matrix(ncol = n.obs, nrow = 100)))) #

  for (i.subs in 1:nrow(pairs)) {
    i <- pairs[i.subs,1];  j <- pairs[i.subs,2]

    # pairs.goods$pairs <- append(pairs.goods$pairs,list(c(scram[i],scram[j]))) # record pairs of point

    #ret.pairs[i.subs,] <- c(scram[i],scram[j])
    ret.pairs[i.subs,] <- c(i,j)

    for (k in 1:n.obs) {
      norm.k2[k] <- gram[k,k] + 1/4*(gram[i,i]+gram[j,j]) + 1/2*gram[i,j]-gram[k,i]-gram[k,j]
      k.to.mid[k] <- (gram[k,i]-gram[k,j]+1/2*(gram[j,j]-gram[i,i]))/sqrt(gram[i,i]+gram[j,j]-2*gram[i,
      error.k[k] <- sqrt(abs(norm.k2[k] - k.to.mid[k]^2))
    }

    k.to.mids[i.subs,] <- k.to.mid

    for (c in 1:length(splits)) {
      good <- rep(1, n.obs)
      s <- splits[c] * (sd.w(k.to.mid, k.w)*adaptive + (adaptive==FALSE))
      good[k.to.mid/s > 1] <- 0  #points on other side of cone tip removed
      d.to.tip <- abs(k.to.mid - s)
      good1 <- good * (abs(atan(error.k / d.to.tip)) < (angle[1]/360*2*pi))  #points outside of cone re
      good1 <- good1 * (1 - 2*(sign(k.to.mid)==sign(s)))  #which side of midpoint are they on
      ret.gs[c,] <- good1
      depthity1[c] <- min(c(sum(good1==-1), sum(good1==1)))
    }
    ret.goods[[i.subs]] <- ret.gs
    qfs1[i.subs,] <- quantile(depthity1, seq(0,1,length=100), na.rm=TRUE)
  }
  dqf1 <- matrix(0,n.obs, 100)
  for (i in 1:n.obs) {
    dqf1[i,] <- apply(qfs1[which(pairs[,1]==i | pairs[,2]==i),],2,mean, na.rm=TRUE)
  }

  # reorder
  # dqf1 <- dqf1[unscram,]  #map back to original indicies

  return(list(dqf1=dqf1,ret.pairs=ret.pairs,ret.goods=ret.goods,k.to.mids=k.to.mids))
}
```

Extract DQFs

Inputs: Output of `dqf.subset`

Returns: dqfs for subset of points

```
extract.dqf <- function(dqf,subset){
  n.subset <- length(subset)
  depthity1 <- rep(0,100)

  indices <- which(dqf$ret.pairs[,1] %in% subset & dqf$ret.pairs[,2] %in% subset)
  pairs <- dqf$ret.pairs[indices,]

  qfs1 <- matrix(0, nrow=length(indices), ncol=100)

  qfs.count <- 1
  for(j in indices){
    gs <- dqf$ret.goods[[j]][,subset]
    for(i in 1:nrow(gs)){
      g <- gs[i,]
      depthity1[i] <- min(c(sum(g==-1), sum(g==1)))
    }
    qfs1[qfs.count,] <- quantile(depthity1, seq(0,1,length=100), na.rm=TRUE)
    qfs.count <- qfs.count+1
  }

  dqf1 <- matrix(0,n.subset, 100)
  for (i in 1:n.subset) {
    dqf1[i,] <- apply(qfs1[which(pairs[,1]==i | pairs[,2]==i),],2,mean,na.rm=TRUE)
  }

  return(dqf1)
}
```

Extract Depths

Inputs: Output of `dqf.subset`

Returns: Depths dataframe

```
extract.depths <- function(dqf,subset){
  n.subset <- length(subset)
  depthity1 <- rep(0,100)

  indices <- which(dqf$ret.pairs[,1] %in% subset & dqf$ret.pairs[,2] %in% subset)
  pairs <- dqf$ret.pairs[indices,]

  depths <- matrix(0, nrow=length(indices), ncol=100)

  depths.count <- 1
  for(j in indices){
    gs <- dqf$ret.goods[[j]][,subset]
    for(i in 1:nrow(gs)){
      g <- gs[i,]
      depthity1[i] <- min(c(sum(g==-1), sum(g==1)))
    }
    depths[depths.count,] <- depthity1
    depths.count <- depths.count+1
  }
```

```
  return(depths)
}
```

calculcate.qfs

Input: Depths dataframe, k2mid vector, splits vector

Returns: qfs dataframe

```
calculate.qfs <- function(depths,k2mid,splits){
  qfs <- matrix(0,nrow=nrow(depths),ncol=length(splits))

  for(i in 1:nrow(depths)){
    qf <- rep(0,100)
    d <- depths[i,]
    k.sd <- sd.w(k2mid[i,],3)

    q.prev <- 1
    min.s <- 1; max.s <- -1
    for(j in 0:max(d)){
      s <- splits[which(d==j)]
      min.s <- min(c(s,min.s)); max.s <- max(c(s,max.s))
      q <- ceiling((pnorm(max.s,sd=k.sd)-pnorm(min.s,sd=k.sd))*100)

      if(q>q.prev & q <= 100){
        qf[q.prev:q] <- j
        q.prev <- q+1
      }

    }
    qfs[i,] <- qf

  }

  return(qfs)

}
```

Clustering

```
initial.cluster <- function(data,n.clusters){
  distance_mat <- dist(data, method = 'euclidean')
  Hierar_cl <- hclust(distance_mat, method = "single")
  clusters <- cutree(Hierar_cl, k = n.clusters)

  return(clusters)
}
```

```
combine.clusters <- function(clusters,combine){
  new.clusters <- clusters

  indices <- which(clusters %in% combine)
  new.clustersclusters[indices] <- clusters[indices[1]]
```

```
  return(new.clusters)
}
```

```
interact <- function(){
  str=""
  while(str != "exit"){
    str = readline()
    print(str)
    plot(seq(0,1,.01),seq(0,1,.01))
  }
}
```