```
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 21 06:23:50 2018

@author: Hi
"""

def apply_rules_set_based(X,
        rule_lower_corners,
        rule_upper_corners):
    # store sparse rules
#    rule_upper_corners_sparse = csr_matrix(
#        rule_upper_corners - RULE_UPPER_CONST, dtype=np.float64)
#    rule_lower_corners_sparse = csr_matrix(
#        rule_lower_corners - RULE_LOWER_CONST, dtype=np.float64)
    # create output matrix
    rule_mask = np.zeros(
        [X.shape[0], rule_lower_corners.shape[0]], dtype=np.int32)
    sorted_feats=np.zeros(X.shape,dtype=np.float64)
    sorted_indxs=np.zeros(X.shape,dtype=np.int32)
    sorted_datapoint_posns=np.zeros(X.shape,dtype=np.int32)
    for j in np.arange(X.shape[1]):
        sorted_indxs[:,j]= np.argsort(X[:,j], axis=-1, kind='quicksort')
        sorted_feats[:,j]=X[sorted_indxs[:,j],j]
        i=0
        for k in sorted_indxs[:,j]:
            sorted_datapoint_posns[k,j]=i
            i=i+1
    apply_rules_set_based_c(
        X.astype(
            np.float64),
        sorted_feats,
        sorted_indxs,
        sorted_datapoint_posns,
        rule_lower_corners,
        rule_upper_corners,
        rule_mask)
    return np.asarray(rule_mask, dtype=bool)

    ############################################################
    def apply_rules_set_based_c(np.ndarray[float64, ndim=2] X,
                np.ndarray[float64, ndim=2] sorted_feats,
                np.ndarray[int32, ndim=2] sorted_indxs,
                np.ndarray[int32, ndim=2] sorted_datapoint_posns,
                object rule_lower_corners,
                object rule_upper_corners,
                np.ndarray[int32, ndim=2] out):
    # sort X feats
#    cdef np.ndarray[float64, ndim=2] sorted_feats
#    sorted_feats=np.zeros(X.shape)#,dtype=float64)
#    cdef np.ndarray[int32, ndim=2] sorted_indxs=np.zeros(X.shape)#,dtype=int32)
#    cdef np.ndarray[int32, ndim=2] sorted_datapoint_posns=np.zeros(X.shape)#,dtype=int32)
#    for j in np.arange(X.shape[1]):
#        sorted_indxs[:,j]= np.argsort(X[:,j], axis=-1, kind='quicksort')
#        sorted_feats[:,j]=X[sorted_indxs[:,j],j]
#        i=0
#        for k in sorted_indxs[:,j]:
#            sorted_datapoint_posns[k,j]=i
#            i=i+1
    if issparse(rule_lower_corners):
        pass # DENSE NOT IMPLEMENTED
    else:
        _apply_rules_set_based(<float64*> (<np.ndarray> X).data,
              <float64*> (<np.ndarray> rule_lower_corners).data ,
              <float64*> (<np.ndarray> rule_upper_corners).data ,
              <float64*> (<np.ndarray> sorted_feats).data,
              <int32*> (<np.ndarray> sorted_indxs).data ,
              <int32*> (<np.ndarray> sorted_datapoint_posns).data ,
             X.shape[0],
             X.shape[1],
             rule_lower_corners.shape[0],
             <int32*> (<np.ndarray> out).data)

    ############################################################


    cdef void _apply_rules_set_based(float64 *X,
                       float64 *rule_lower_corners,
                           float64 *rule_upper_corners,
                           float64 *sorted_feats,
                          int32 *sorted_indxs,
                          int32 *sorted_datapoint_posns,
                           Py_ssize_t n_samples,
                          Py_ssize_t n_features,
                          Py_ssize_t n_rules,
                          int32 *out):
    """   """
    #DTYPE_t
#    cdef float64* lower_data = <float64*>(<np.ndarray> rule_lower_corners.data).data
#    cdef INT32_t* lower_indices = <INT32_t*>(<np.ndarray> rule_lower_corners.indices).data
#    cdef INT32_t* lower_indptr = <INT32_t*>(<np.ndarray> rule_lower_corners.indptr).data
#    cdef float64* upper_data = <float64*>(<np.ndarray> rule_upper_corners.data).data
#    cdef INT32_t* upper_indices = <INT32_t*>(<np.ndarray> rule_upper_corners.indices).data
#    cdef INT32_t* upper_indptr = <INT32_t*>(<np.ndarray> rule_upper_corners.indptr).data
    cdef int32 res
    cdef int32 rule_start
    cdef int32 rule_end
    cdef Py_ssize_t i
    cdef Py_ssize_t j
    cdef Py_ssize_t r
    cdef int32 j_test
    cdef int32 i_f
    cdef int32 i_ff
    cdef int32 insert_pos
    cdef int32 dirn
    cdef np.ndarray[np.int32_t, ndim=1] viable_set = np.empty(n_samples, dtype=np.int32)
    cdef np.ndarray[np.int32_t, ndim=1] feat_sets=np.zeros([n_features*2*4],dtype=np.int32)

    #cdef int32 viable_set[n_samples]
    cdef int32 viable_set_size
    cdef int32 viable_set_size_this
    cdef int32 i_viable
    cdef int32 min_viable_size
    cdef int32  min_viable_index
    # apply each rule
    for r in range(n_rules):
        i_f=0
        for j in range(n_features): #np.arange(X.shape[1],dtype=np.int32):
            if rule_lower_corners[j + n_features * r]!=RULE_LOWER_CONST: #j * n_rules + r
                insert_pos=_search_sorted(sorted_feats,j*n_samples, n_samples,rule_lower_corners
                feat_sets[0*2*n_features+ i_f]=j#[j,-1,insert_pos,n_samples-insert_pos]
                feat_sets[1*2*n_features+ i_f]=-1
                feat_sets[2*2*n_features+ i_f]=insert_pos
                feat_sets[3*2*n_features+ i_f]=n_samples-insert_pos
                i_f=i_f+1
            if rule_upper_corners[j + n_features * r]!=RULE_UPPER_CONST: #j * n_rules + r
                insert_pos=_search_sorted(sorted_feats,j*n_samples, n_samples,rule_upper_corners
                #feat_sets[i_f,:]=[j,1,insert_pos,insert_pos]
                feat_sets[0*2*n_features+ i_f]=j#[j,-1,insert_pos,n_samples-insert_pos]
                feat_sets[1*2*n_features+ i_f]=1
                feat_sets[2*2*n_features+ i_f]=insert_pos
                feat_sets[3*2*n_features+ i_f]=insert_pos
                i_f=i_f+1
        if i_f==0:
            for i in range(n_samples):
                out[r + n_rules * i]=1 #i * n_rules + r
            #viable_pts=np.arange(n_samples,dtype=np.int32)
        else:
            #feat_sets=feat_sets[0:i_f,:]
            #feat_sets=feat_sets[feat_sets[:,3].argsort(),:]
            min_viable_size=100000
            min_viable_index=-1
            for i_ff in range(i_f):
                if feat_sets[3*2*n_features+ i_ff]<min_viable_size:
                    min_viable_size=feat_sets[3*2*n_features+ i_ff]
                    min_viable_index=i_ff
            i_ff=min_viable_index # start with minimum because the size of the first set is an
            j=feat_sets[0*2*n_features+ i_ff]
            insert_pos=feat_sets[2*2*n_features+ i_ff]
            dirn=feat_sets[1*2*n_features+ i_ff]
            viable_set_size=feat_sets[3*2*n_features+ i_ff]

            if dirn==-1:
                for i in range(viable_set_size):
                    viable_set[i]=sorted_indxs[(i+insert_pos)*n_features + j  ] #j*n_samples +
                #viable_pts=sorted_indxs[insert_pos:,j]
            else:
                for i in range(viable_set_size):
                    viable_set[i]=sorted_indxs[i*n_features + j ] #j*n_samples + (i)
                #viable_pts=sorted_indxs[0:insert_pos,j]

            for i_ff in range(0,i_f):
                if i_ff !=min_viable_index:
                    j=feat_sets[0*2*n_features+ i_ff]
                    insert_pos=feat_sets[2*2*n_features+ i_ff]
                    dirn=feat_sets[1*2*n_features+ i_ff]
                    viable_set_size_this=feat_sets[3*2*n_features+ i_ff]
                    if dirn==-1:
                        i_viable=0
                        for i in range(viable_set_size):
                            if  sorted_datapoint_posns[viable_set[i]*n_features + j ]>=insert_po
                                viable_set[i_viable]=viable_set[i]
                                i_viable=i_viable+1
                        viable_set_size=i_viable
                        #viable_pts=viable_pts[sorted_datapoint_posns[viable_pts,j]>=insert_pos
                    else:
                        i_viable=0
                        for i in range(viable_set_size):
                            if  sorted_datapoint_posns[viable_set[i]*n_features + j  ]<insert_po
                                viable_set[i_viable]=viable_set[i]
                                i_viable=i_viable+1
                        viable_set_size=i_viable
            #viable_pts=viable_pts[sorted_datapoint_posns[viable_pts,j]<insert_pos]

if viable_set_size>0:
    for i in range(viable_set_size) :
        out[viable_set[i]*n_rules + r]=1
```