Appendix B: Python code for "Beyond validation loss"

Charles B. Delahunt, Courosh Mehanian, Daniel Shea, and Matthew P. Horning

delahunt@uw.edu

Appendix B: Python function defs

(i) 2-sided standard deviations

Asymmetrical distributions are imperfectly described by the standard deviation with gaussian assumption. A quick way to more precision is to calculate two standard deviations, right- and left-handed, using just the points to the right (or left) of the median.

```
import numpy as np
def calculateTwoSidedStdDev_fn(x, middleVal=None)
   """
   Calculate two standard deviations, one for the left and one
   for the right, by flipping samples across the median (not
   across the mean, on the grounds that in asymmetrical
   distributions the median is likely closer to the peak value).

   Parameters
   -----
   x : list-like or np.array vector
   middleVal: float or int. If you don't want to use the
      median. Default = None, ie use median.

   Returns
   ----
   stdDevRight : scalar float
   stdDevLeft : scalar float
   """
   if len(x) <= 1:
      stdDevRight = -1
      stdDevLeft = -1
   else:
      if middleVal != None:
         m = middleVal
      else:
         m = np.median(x)
      x = x - m
      xH = x[x >= 0] # top half, shifted to a nominal 0
```

```
      center
      xL = x[x <= 0] # bottom half
      stdDevRight = np.std(np.concatenate((-xH, xH)))
      stdDevLeft = np.std(np.concatenate((xL, -xL)))

   return stdDevRight, stdDevLeft
```

(ii) z-scale alignment

The first def below extracts the parameters of the distributions in each split. The second def applies these parameters to z-map the scores of each split.

```
import numpy as np

def standardizeSvmScoresForKArrayGivenParams_fn(x, p):
   """
   Given an array of model output scores from k splits adjust
   them using parameters pre-calculated by
   'calculateStandardizationParamsForKFoldScores_fn' (above).
   We apply vector operations to, for each split's score,
      1. Subtract that split's median to center the scores at
      (hypothetical) zeros
      2. Scale each values by its split's stdDevs (RH and LH)
      to roughly match the canonical std dev
      3. Shift all the values to the canonical median.
   NOTE: Function exits if there is a zero in splitRhStdDevs
   or splitLhStdDevs. This should not occur, because the
   training samples that generated the std devs would have to
   have identical scores.

   Parameters
   -----
   x : list of floats, len = number of splits, ie number of
      models
   p : dict with keys 'canonicalMedian', 'canonicalStdDev',
      'splitMedians',
      'splitRhStdDevs', 'splitLhStdDevs' All entries except
      'canonicalMedian' and 'canonicalStdDev' will be vectors
      if we are adjusting all splits at once.

   Returns
   ----
   adjX : list of floats, same length as argin 'x'.
   """

   rhStd = np.array(p['splitRhStdDevs'])
   lhStd = np.array(p['splitLhStdDevs'])
   canonStd = np.array(p['canonicalStdDev'])
   canonMedian = np.array(p['canonicalMedian'])
```

```
   splitMedians = np.array(p['splitMedians'])
   if np.sum(rhStd == 0) > 0 or np.sum(lhStd == 0) > 0: # should
      not happen
      print('splitRhStdDevs or splitLhStdDevs contains a 0 value.')
      adjX = x
   else:
      t0 = x - splitMedians # subtract each split's median
      # Apply column vector operations:
      for k in range(len(splitMedians)): # Process each
      column in turn:
         tk = t0[:, k]
         if np.sum(tk > 0) > 0:
            tk[tk > 0] = tk[tk > 0] * canonStd / rhStd[k] # to
      right of median
         if np.sum(tk < 0) > 0:
            tk[tk < 0] = tk[tk < 0] * canonStd / lhStd[k] # to
      left of median
         t0[:, k] = tk
      adjX = t0 + canonMedian * np.ones(x.shape)

   return adjX

#---------------------

def standardizeKFoldScoresForVectorGivenParams_fn(x,split,p):
   """
   Given a vector of scores from a model, along with a
   vector of their splits,
   standardize the scores of each fold using the dictionary
   of parameters.
   The argins will likely be the scores on k test sets,
   concatenated into a vector, where the scores came from k
   models built on a k-fold split.

   Parameters
   -----
   x : list-like of floats (scores)
   split : list-like of ints (fold indices). same len as 'scores'
   p : dict of params (see unpacking in first few lines)

   Returns
   ----
   adjScores : list-like of floats. same len as 'scores'
   """

   foldVals = np.unique(split) # hopefully 0,1,2, etc. But does
   not need to be.
   canonicalMedian = p['canonicalMedian']
   canonicalStdDev = p['canonicalStdDev']
   splitMedians = p['splitMedians']
```

```
   splitRhStdDevs = p['splitRhStdDevs']
   splitLhStdDevs = p['splitLhStdDevs']
   adjX = x.copy()
   for k in range(len(foldVals)):
      # Adjust all scores in this fold:
      inds = split == foldVals[k]
      t = x[inds]
      m = splitMedians[k]
      r = splitRhStdDevs[k]
      l = splitLhStdDevs[k]
      t0 = t - m
      if r > 0: # in case r == 0, don't divide (a catch)
         t0[t0 > 0] = t0[t0 >> 0] * p['canonicalStdDev'] / r
         # to the right of median
      else:
         pass
         print(f'rh stdDev=0 on {np.sum(t0>0)} cases > median.')
      if l > 0: # ditto
         t0[t0 < 0] = t0[t0 < 0] * p['canonicalStdDev'] / l
         # to the left of median
      else:
         pass
         print(f'lh stdDev=0 on np.sum(t0>0) cases < median.')
      adjX[inds] = t0 + p['canonicalMedian']
   # optional: could clip adjX at 0 and 1.

   return adjX
```

(iii) n% sliver AUC

```
import numpy as np
from sklearn.metrics import roc_curve
def calculateSliverAuc_fn(y, yHat, targetSpec):
   """
   Given scores, binary labels, and a minimum specificity:
   calculate the normalized AUC within the leftmost sliver
   of an ROC curve.

   Parameters
   -----
   y : list-like of ints (0s and 1s)
   yHat : list-like of floats (scores). same len as 'y'
   targetSpec : int (1 to 99)

   Returns
   ----
   sliverAuc : float
   """

   maxFprForAucLoss = (100 - targetSpec) / 100
```

```
[fpr, tpr, op] = roc_curve(y, yHat, pos_label=1)
inds = np.where(fpr <= maxFprForAucLoss)[0]
f = list(fpr[inds]) + [maxFprForAucLoss] # postpend an
   endpoint fpr
t = list(tpr[inds]) + [tpr[inds[-1]]] # postpend the
   corresponding tpr value
sessionSliverAuc = 0
for i in range(len(f) - 1):
   sliverAuc += (f[i+1] - f[i]) * (0.5 * (t[i+1] + t[i]))
   # Trapezoid rule
sliverAuc = sliverAuc / maxFprForAucLoss # normalize
return sliverAuc
```