from functools import partial

from twitter.cortex.ml.embeddings.deepbird.grouped\_metrics.configuration import (

GroupedMetricsConfiguration,

)

from twitter.cortex.ml.embeddings.deepbird.grouped\_metrics.helpers import (

extract\_prediction\_from\_prediction\_record,

)

# checkstyle: noqa

def score\_loss\_at\_n(labels, predictions, lightN):

"""

Compute the absolute ScoreLoss ranking metric

Args:

labels (list) : A list of label values (HeavyRanking Reference)

predictions (list): A list of prediction values (LightRanking Predictions)

lightN (int): size of the list at which of Initial candidates to compute ScoreLoss. (LightRanking)

"""

assert len(labels) == len(predictions)

if lightN <= 0:

return None

labels\_with\_predictions = zip(labels, predictions)

labels\_with\_sorted\_predictions = sorted(

labels\_with\_predictions, key=lambda x: x[1], reverse=True

)[:lightN]

labels\_top1\_light = max([label for label, \_ in labels\_with\_sorted\_predictions])

labels\_top1\_heavy = max(labels)

return labels\_top1\_heavy - labels\_top1\_light

def cgr\_at\_nk(labels, predictions, lightN, heavyK):

"""

Compute Cumulative Gain Ratio (CGR) ranking metric

Args:

labels (list) : A list of label values (HeavyRanking Reference)

predictions (list): A list of prediction values (LightRanking Predictions)

lightN (int): size of the list at which of Initial candidates to compute CGR. (LightRanking)

heavyK (int): size of the list at which of Refined candidates to compute CGR. (HeavyRanking)

"""

assert len(labels) == len(predictions)

if (not lightN) or (not heavyK):

out = None

elif lightN <= 0 or heavyK <= 0:

out = None

else:

labels\_with\_predictions = zip(labels, predictions)

labels\_with\_sorted\_predictions = sorted(

labels\_with\_predictions, key=lambda x: x[1], reverse=True

)[:lightN]

labels\_topN\_light = [label for label, \_ in labels\_with\_sorted\_predictions]

if lightN <= heavyK:

cg\_light = sum(labels\_topN\_light)

else:

labels\_topK\_heavy\_from\_light = sorted(labels\_topN\_light, reverse=True)[:heavyK]

cg\_light = sum(labels\_topK\_heavy\_from\_light)

ideal\_ordering = sorted(labels, reverse=True)

cg\_heavy = sum(ideal\_ordering[: min(lightN, heavyK)])

out = 0.0

if cg\_heavy != 0:

out = max(cg\_light / cg\_heavy, 0)

return out

def \_get\_weight(w, atK):

if not w:

return 1.0

elif len(w) <= atK:

return 0.0

else:

return w[atK]

def recall\_at\_nk(labels, predictions, n=None, k=None, w=None):

"""

Recall at N-K ranking metric

Args:

labels (list): A list of label values

predictions (list): A list of prediction values

n (int): size of the list at which of predictions to compute recall. (Light Ranking Predictions)

The default is None in which case the length of the provided predictions is used as L

k (int): size of the list at which of labels to compute recall. (Heavy Ranking Predictions)

The default is None in which case the length of the provided labels is used as L

w (list): weight vector sorted by labels

"""

assert len(labels) == len(predictions)

if not any(labels):

out = None

else:

safe\_n = len(predictions) if not n else min(len(predictions), n)

safe\_k = len(labels) if not k else min(len(labels), k)

labels\_with\_predictions = zip(labels, predictions)

sorted\_labels\_with\_predictions = sorted(

labels\_with\_predictions, key=lambda x: x[0], reverse=True

)

order\_sorted\_labels\_predictions = zip(range(len(labels)), \*zip(\*sorted\_labels\_with\_predictions))

order\_with\_predictions = [

(order, pred) for order, label, pred in order\_sorted\_labels\_predictions

]

order\_with\_sorted\_predictions = sorted(order\_with\_predictions, key=lambda x: x[1], reverse=True)

pred\_sorted\_order\_at\_n = [order for order, \_ in order\_with\_sorted\_predictions][:safe\_n]

intersection\_weight = [

\_get\_weight(w, order) if order < safe\_k else 0 for order in pred\_sorted\_order\_at\_n

]

intersection\_score = sum(intersection\_weight)

full\_score = sum(w) if w else float(safe\_k)

out = 0.0

if full\_score != 0:

out = intersection\_score / full\_score

return out

class ExpectedLossGroupedMetricsConfiguration(GroupedMetricsConfiguration):

"""

This is the Expected Loss Grouped metric computation configuration.

"""

def \_\_init\_\_(self, lightNs=[]):

"""

Args:

lightNs (list): size of the list at which of Initial candidates to compute Expected Loss. (LightRanking)

"""

self.lightNs = lightNs

@property

def name(self):

return "ExpectedLoss"

@property

def metrics\_dict(self):

metrics\_to\_compute = {}

for lightN in self.lightNs:

metric\_name = "ExpectedLoss\_atLight\_" + str(lightN)

metrics\_to\_compute[metric\_name] = partial(score\_loss\_at\_n, lightN=lightN)

return metrics\_to\_compute

def extract\_label(self, prec, drec, drec\_label):

return drec\_label

def extract\_prediction(self, prec, drec, drec\_label):

return extract\_prediction\_from\_prediction\_record(prec)

class CGRGroupedMetricsConfiguration(GroupedMetricsConfiguration):

"""

This is the Cumulative Gain Ratio (CGR) Grouped metric computation configuration.

CGR at the max length of each session is the default.

CGR at additional positions can be computed by specifying a list of 'n's and 'k's

"""

def \_\_init\_\_(self, lightNs=[], heavyKs=[]):

"""

Args:

lightNs (list): size of the list at which of Initial candidates to compute CGR. (LightRanking)

heavyK (int): size of the list at which of Refined candidates to compute CGR. (HeavyRanking)

"""

self.lightNs = lightNs

self.heavyKs = heavyKs

@property

def name(self):

return "cgr"

@property

def metrics\_dict(self):

metrics\_to\_compute = {}

for lightN in self.lightNs:

for heavyK in self.heavyKs:

metric\_name = "cgr\_atLight\_" + str(lightN) + "\_atHeavy\_" + str(heavyK)

metrics\_to\_compute[metric\_name] = partial(cgr\_at\_nk, lightN=lightN, heavyK=heavyK)

return metrics\_to\_compute

def extract\_label(self, prec, drec, drec\_label):

return drec\_label

def extract\_prediction(self, prec, drec, drec\_label):

return extract\_prediction\_from\_prediction\_record(prec)

class RecallGroupedMetricsConfiguration(GroupedMetricsConfiguration):

"""

This is the Recall Grouped metric computation configuration.

Recall at the max length of each session is the default.

Recall at additional positions can be computed by specifying a list of 'n's and 'k's

"""

def \_\_init\_\_(self, n=[], k=[], w=[]):

"""

Args:

n (list): A list of ints. List of prediction rank thresholds (for light)

k (list): A list of ints. List of label rank thresholds (for heavy)

"""

self.predN = n

self.labelK = k

self.weight = w

@property

def name(self):

return "group\_recall"

@property

def metrics\_dict(self):

metrics\_to\_compute = {"group\_recall\_unweighted": recall\_at\_nk}

if not self.weight:

metrics\_to\_compute["group\_recall\_weighted"] = partial(recall\_at\_nk, w=self.weight)

if self.predN and self.labelK:

for n in self.predN:

for k in self.labelK:

if n >= k:

metrics\_to\_compute[

"group\_recall\_unweighted\_at\_L" + str(n) + "\_at\_H" + str(k)

] = partial(recall\_at\_nk, n=n, k=k)

if self.weight:

metrics\_to\_compute[

"group\_recall\_weighted\_at\_L" + str(n) + "\_at\_H" + str(k)

] = partial(recall\_at\_nk, n=n, k=k, w=self.weight)

if self.labelK and not self.predN:

for k in self.labelK:

metrics\_to\_compute["group\_recall\_unweighted\_at\_full\_at\_H" + str(k)] = partial(

recall\_at\_nk, k=k

)

if self.weight:

metrics\_to\_compute["group\_recall\_weighted\_at\_full\_at\_H" + str(k)] = partial(

recall\_at\_nk, k=k, w=self.weight

)

return metrics\_to\_compute

def extract\_label(self, prec, drec, drec\_label):

return drec\_label

def extract\_prediction(self, prec, drec, drec\_label):

return extract\_prediction\_from\_prediction\_record(prec)