HW4_RETAIN

October 3, 2025

1 HW4 RETAIN

1.1 Overview

Previously, you tried heart failure predictioin with classical machine learning models, neural network (NN), and recurrent neural network (RNN).

In this question, you will try a different approach. You will implement RETAIN, a RNN model with attention mechanism, proposed by Choi et al. in the paper RETAIN: An Interpretable Predictive Model for Healthcare using Reverse Time Attention Mechanism.

---

[1]:
```
import os
import pickle
import random
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
```

[2]:
```
# set seed
seed = 24
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
os.environ["PYTHONHASHSEED"] = str(seed)

# define data path
DATA_PATH = "../HW4_RETAIN-lib/data/"
```

1.2 About Raw Data

We will perform heart failure prediction using the diagnosis codes. We will use the same dataset from HW3 RNN, which is synthesized from MIMIC-III.

The data has been preprocessed for you. Let us load them and take a look.

[3]:
```
pids = pickle.load(open(os.path.join(DATA_PATH,'train/pids.pkl'), 'rb'))
vids = pickle.load(open(os.path.join(DATA_PATH,'train/vids.pkl'), 'rb'))
hfs = pickle.load(open(os.path.join(DATA_PATH,'train/hfs.pkl'), 'rb'))
seqs = pickle.load(open(os.path.join(DATA_PATH,'train/seqs.pkl'), 'rb'))
types = pickle.load(open(os.path.join(DATA_PATH,'train/types.pkl'), 'rb'))
rtypes = pickle.load(open(os.path.join(DATA_PATH,'train/rtypes.pkl'), 'rb'))

assert len(pids) == len(vids) == len(hfs) == len(seqs) == 1000
assert len(types) == 619
```

where

- `pids`: contains the patient ids
- `vids`: contains a list of visit ids for each patient
- `hfs`: contains the heart failure label (0: normal, 1: heart failure) for each patient
- `seqs`: contains a list of visit (in ICD9 codes) for each patient
- `types`: contains the map from ICD9 codes to ICD-9 labels
- `rtypes`: contains the map from ICD9 labels to ICD9 codes

Let us take a patient as an example.

[4]:
```
# take the 3rd patient as an example

print("Patient ID:", pids[3])
print("Heart Failure:", hfs[3])
print("# of visits:", len(vids[3]))
for visit in range(len(vids[3])):
    print(f"\t{visit}-th visit id:", vids[3][visit])
    print(f"\t{visit}-th visit diagnosis labels:", seqs[3][visit])
    print(f"\t{visit}-th visit diagnosis codes:", [rtypes[label] for label in␣
 ↪seqs[3][visit]])
```

```
Patient ID: 47537
Heart Failure: 0
# of visits: 2
        0-th visit id: 0
        0-th visit diagnosis labels: [12, 103, 262, 285, 290, 292, 359, 416, 39,
225, 275, 294, 326, 267, 93]
        0-th visit diagnosis codes: ['DIAG_041', 'DIAG_276', 'DIAG_518',
'DIAG_560', 'DIAG_567', 'DIAG_569', 'DIAG_707', 'DIAG_785', 'DIAG_155',
'DIAG_456', 'DIAG_537', 'DIAG_571', 'DIAG_608', 'DIAG_529', 'DIAG_263']
        1-th visit id: 1
        1-th visit diagnosis labels: [12, 103, 240, 262, 290, 292, 319, 359,
510, 513, 577, 307, 8, 280, 18, 131]
        1-th visit diagnosis codes: ['DIAG_041', 'DIAG_276', 'DIAG_482',
'DIAG_518', 'DIAG_567', 'DIAG_569', 'DIAG_599', 'DIAG_707', 'DIAG_995',
'DIAG_998', 'DIAG_V09', 'DIAG_584', 'DIAG_031', 'DIAG_553', 'DIAG_070',
'DIAG_305']
```

Note that seqs is a list of list of list. That is, seqs[i][j][k] gives you the k-th diagnosis codes for the j-th visit for the i-th patient.

And you can look up the meaning of the ICD9 code online. For example, DIAG_276 represetns *disorders of fluid electrolyte and acid-base balance*.

Further, let see number of heart failure patients.

[5]:
```
print("number of heart failure patients:", sum(hfs))
print("ratio of heart failure patients: %.2f" % (sum(hfs) / len(hfs)))
```

```
number of heart failure patients: 548
ratio of heart failure patients: 0.55
```

1.3 1 Build the dataset [15 points]

1.3.1 1.1 CustomDataset [5 points]

This is the same as HW3 RNN.

First, let us implement a custom dataset using PyTorch class Dataset, which will characterize the key features of the dataset we want to generate.

We will use the sequences of diagnosis codes seqs as input and heart failure hfs as output.

[6]:
```
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, seqs, hfs):
        self.x = seqs
        self.y = hfs

    def __len__(self):
        # number of patients
        return len(self.x)

    def __getitem__(self, index):
        # return raw Python objects; collate_fn will tensorize/pad
        return self.x[index], self.y[index]


dataset = CustomDataset(seqs, hfs)
```

[7]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

dataset = CustomDataset(seqs, hfs)

assert len(dataset) == 1000
```

1.3.2 1.2 Collate Function [5 points]

This is the same as HW3 RNN.

As you note that, we do not convert the data to tensor in the built `CustomDataset`. Instead, we will do this using a collate function `collate_fn()`.

This collate function `collate_fn()` will be called by `DataLoader` after fetching a list of samples using the indices from `CustomDataset` to collate the list of samples into batches.

For example, assume the `DataLoader` gets a list of two samples.

```
[ [ [0, 1, 2], [8, 0] ],
  [ [12, 13, 6, 7], [12], [23, 11] ] ]
```

where the first sample has two visits `[0, 1, 2]` and `[8, 0]` and the second sample has three visits `[12, 13, 6, 7]`, `[12]`, and `[23, 11]`.

The collate function `collate_fn()` is supposed to pad them into the same shape (3, 4), where 3 is the maximum number of visits and 4 is the maximum number of diagnosis codes.

```
[ [ [0, 1, 2, *0*], [8, 0, *0*, *0*], [*0*, *0*, *0*, *0*]  ],
  [ [12, 13, 6, 7], [12, *0*, *0*, *0*], [23, 11, *0*, *0*] ] ]
```

Further, the padding information will be stored in a mask with the same shape, where 1 indicates that the diagnosis code at this position is from the original input, and 0 indicates that the diagnosis code at this position is the padded value.

```
[ [ [1, 1, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0] ],
  [ [1, 1, 1, 1], [1, 0, 0, 0], [1, 1, 0, 0] ] ]
```

Lastly, we will have another diagnosis sequence in reversed time. This will be used in our RNN model for masking. Note that we only flip the true visits.

```
[ [ [8, 0, *0*, *0*], [0, 1, 2, *0*], [*0*, *0*, *0*, *0*]  ],
  [ [23, 11, *0*, *0*], [12, *0*, *0*, *0*], [12, 13, 6, 7] ] ]
```

And a reversed mask as well.

```
[ [ [1, 1, 0, 0], [1, 1, 1, 0], [0, 0, 0, 0] ],
  [ [1, 1, 0, 0], [1, 0, 0, 0], [1, 1, 1, 1], ] ]
```

We need to pad the sequences into the same length so that we can do batch training on GPU. And we also need this mask so that when training, we can ignored the padded value as they actually do not contain any information.

[8]:
```python
# 1.2 - Collate function
import torch

def collate_fn(data):
    """
    Returns:
      x, masks, rev_x, rev_masks: (B, Vmax, Cmax)
      y: (B,)
    """
```

```python
    sequences, labels = zip(*data)
    y = torch.tensor(labels, dtype=torch.float)

    B = len(sequences)
    Vmax = max(len(p) for p in sequences) if B else 0
    Cmax = max((len(v) for p in sequences for v in p), default=1)

    x         = torch.zeros((B, Vmax, Cmax), dtype=torch.long)
    masks     = torch.zeros((B, Vmax, Cmax), dtype=torch.bool)
    rev_x     = torch.zeros((B, Vmax, Cmax), dtype=torch.long)
    rev_masks = torch.zeros((B, Vmax, Cmax), dtype=torch.bool)

    # fill x/masks
    for b, patient in enumerate(sequences):
        for v, visit in enumerate(patient):
            c = min(len(visit), Cmax)
            if c > 0:
                x[b, v, :c] = torch.tensor(visit[:c], dtype=torch.long)
                masks[b, v, :c] = True

    # reverse only TRUE visits per patient
    for b, patient in enumerate(sequences):
        vlen = len(patient)
        if vlen > 0:
            idx = torch.arange(vlen-1, -1, -1)
            rev_x[b, :vlen]     = x[b, :vlen][idx]
            rev_masks[b, :vlen] = masks[b, :vlen][idx]

    return x, masks, rev_x, rev_masks, y
```

[9]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

from torch.utils.data import DataLoader

loader = DataLoader(dataset, batch_size=10, collate_fn=collate_fn)
loader_iter = iter(loader)
x, masks, rev_x, rev_masks, y = next(loader_iter)

assert x.dtype == rev_x.dtype == torch.long
assert y.dtype == torch.float
assert masks.dtype == rev_masks.dtype == torch.bool

assert x.shape == rev_x.shape == masks.shape == rev_masks.shape == (10, 3, 24)
assert y.shape == (10,)
```

Now we have `CustomDataset` and `collate_fn()`. Let us split the dataset into training and validation sets.

[10]:
```
from torch.utils.data.dataset import random_split

split = int(len(dataset)*0.8)

lengths = [split, len(dataset) - split]
train_dataset, val_dataset = random_split(dataset, lengths)

print("Length of train dataset:", len(train_dataset))
print("Length of val dataset:", len(val_dataset))
```

```
Length of train dataset: 800
Length of val dataset: 200
```

1.3.3 1.3 DataLoader [5 points]

This is the same as HW3 RNN.

Now, we can load the dataset into the data loader.

[11]:
```
# 1.3 - DataLoader
from torch.utils.data import DataLoader

def load_data(train_dataset, val_dataset, collate_fn):
    batch_size = 32
    train_loader = DataLoader(train_dataset, batch_size=batch_size,
                              shuffle=True,  collate_fn=collate_fn)
    val_loader   = DataLoader(val_dataset,   batch_size=batch_size,
                              shuffle=False, collate_fn=collate_fn)
    return train_loader, val_loader
```

[12]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

train_loader, val_loader = load_data(train_dataset, val_dataset, collate_fn)

assert len(train_loader) == 25, "Length of train_loader should be 25, instead␣
 ↪we got %d"%(len(train_loader))
```

1.4 2 RETAIN [70 points]

RETAIN is essentially a RNN model with attention mechanism.

The idea of attention is quite simple: it boils down to weighted averaging. Let us consider machine translation in class as an example. When generating a translation of a source text, we first pass the source text through an encoder (an LSTM or an equivalent model) to obtain a sequence of encoder hidden states $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T$. Then, at each step of generating a translation (decoding), we selectively attend to these encoder hidden states, that is, we construct a context vector $\boldsymbol{c}_i$ that is a weighted average of encoder hidden states.

$$\boldsymbol{c}_i = \sum_j a_{ij} \boldsymbol{h}_j$$

We choose the weights a_{ij} based both on encoder hidden states $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T$ and decoder hidden states $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_T$ and normalize them so that they encode a categorical probability distribution $p(\boldsymbol{h}_j|\boldsymbol{s}_i)$.

$$\boldsymbol{a}_i = \text{Softmax}\left(a(\boldsymbol{s}_i, \boldsymbol{h}_j)\right)$$

RETAIN has two different attention mechanisms. - One is to help figure out what are the important visits. This attention α_i, which is scalar for the i-th visit, tells you the importance of the i-th visit. - Then we have another similar attention mechanism. But in this case, this attention ways β_i is a vector. That gives us a more detailed view of underlying cause of the input. That is, which are the important features within a visit.

Unfolded view of RETAIN's architecture: Given input sequence $\mathbf{x}_1, ..., \mathbf{x}_i$, we predict the label $\mathbf{y}_i$. - Step 1: Embedding, - Step 2: generating α values using RNN-α, - Step 3: generating β values using RNN-β, - Step 4: Generating the context vector using attention and representation vectors, - Step 5: Making prediction.

Note that in Steps 2 and 3 we use RNN in the reversed time.

Let us first implement RETAIN step-by-step.

1.4.1 2.1 Step 2: AlphaAttention [20 points]

Implement the alpha attention in the second equation of step 2.

[13]:
```python
# 2.1 - AlphaAttention
class AlphaAttention(torch.nn.Module):
    def __init__(self, hidden_dim):
        super().__init__()
        self.a_att = nn.Linear(hidden_dim, 1)

    def forward(self, g):
        # g: (B, V, H) -> scores: (B, V, 1) -> softmax along V
        scores = self.a_att(g)
        alpha = torch.softmax(scores, dim=1)
        return alpha
```

[14]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[14]: '\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'

1.4.2 2.2 Step 3: BetaAttention [20 points]

Implement the beta attention in the second equation of step 3.

[15]:
```python
# 2.2 - BetaAttention
class BetaAttention(torch.nn.Module):
    def __init__(self, hidden_dim):
        super().__init__()
        self.b_att = nn.Linear(hidden_dim, hidden_dim)

    def forward(self, h):
        # h: (B, V, H) -> beta: (B, V, H)
        beta = torch.tanh(self.b_att(h))
        return beta
```

[16]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[16]: '\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'

1.4.3 2.3 Attention Sum [30 points]

Implement the sum of attention in step 4.

[17]:
```python
# 2.3 - Attention Sum (vectorized)
def attention_sum(alpha, beta, rev_v, rev_masks):
    """
    alpha: (B, V, 1)   beta: (B, V, H)   rev_v: (B, V, H)
    rev_masks: (B, V, C)  -> visit_mask: (B, V, 1)
    Returns c: (B, H)
    """
    # visit is valid if any code is valid
    visit_mask = rev_masks.any(dim=-1, keepdim=True)       # (B, V, 1)
    # mask out padded visits
    weighted = alpha * beta * rev_v * visit_mask           # (B, V, H)
    c = weighted.sum(dim=1)                                # (B, H)
    return c
```

[18]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[18]: '\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'

1.4.4 2.4 Build RETAIN

Now, we can build the RETAIN model.

[19]:
```python
def sum_embeddings_with_mask(x, masks):
    """
    Mask select the embeddings for true visits (not padding visits) and then␣
↪sum the embeddings for each visit up.

    Arguments:
        x: the embeddings of diagnosis sequence of shape (batch_size, # visits,␣
↪# diagnosis codes, embedding_dim)
        masks: the padding masks of shape (batch_size, # visits, # diagnosis␣
↪codes)

    Outputs:
        sum_embeddings: the sum of embeddings of shape (batch_size, # visits,␣
↪embedding_dim)
    """

    x = x * masks.unsqueeze(-1)
    x = torch.sum(x, dim = -2)
    return x
```

[20]:
```python
class RETAIN(nn.Module):

    def __init__(self, num_codes, embedding_dim=128):
        super().__init__()
        # Define the embedding layer using `nn.Embedding`. Set `embDimSize` to␣
↪128.
        self.embedding = nn.Embedding(num_codes, embedding_dim)
        # Define the RNN-alpha using `nn.GRU()`; Set `hidden_size` to 128. Set␣
↪`batch_first` to True.
        self.rnn_a = nn.GRU(embedding_dim, embedding_dim, batch_first=True)
        # Define the RNN-beta using `nn.GRU()`; Set `hidden_size` to 128. Set␣
↪`batch_first` to True.
        self.rnn_b = nn.GRU(embedding_dim, embedding_dim, batch_first=True)
        # Define the alpha-attention using `AlphaAttention()`;
        self.att_a = AlphaAttention(embedding_dim)
        # Define the beta-attention using `BetaAttention()`;
        self.att_b = BetaAttention(embedding_dim)
        # Define the linear layers using `nn.Linear()`;
        self.fc = nn.Linear(embedding_dim, 1)
        # Define the final activation layer using `nn.Sigmoid().
        self.sigmoid = nn.Sigmoid()

    def forward(self, x, masks, rev_x, rev_masks):
        """
        Arguments:
```

```
            rev_x: the diagnosis sequence in reversed time of shape (# visits,␣
↪batch_size, # diagnosis codes)
            rev_masks: the padding masks in reversed time of shape (# visits,␣
↪batch_size, # diagnosis codes)

        Outputs:
            probs: probabilities of shape (batch_size)
        """
        # 1. Pass the reversed sequence through the embedding layer;
        rev_x = self.embedding(rev_x)
        # 2. Sum the reversed embeddings for each diagnosis code up for a visit␣
↪of a patient.
        rev_x = sum_embeddings_with_mask(rev_x, rev_masks)
        # 3. Pass the reversed embeggings through the RNN-alpha and RNN-beta␣
↪layer separately;
        g, _ = self.rnn_a(rev_x)
        h, _ = self.rnn_b(rev_x)
        # 4. Obtain the alpha and beta attentions using `AlphaAttention()` and␣
↪`BetaAttention()`;
        alpha = self.att_a(g)
        beta = self.att_b(h)
        # 5. Sum the attention up using `attention_sum()`;
        c = attention_sum(alpha, beta, rev_x, rev_masks)
        # 6. Pass the context vector through the linear and activation layers.
        logits = self.fc(c)
        probs = self.sigmoid(logits)
        return probs.squeeze()


# load the model here
retain = RETAIN(num_codes = len(types))
retain
```

[20]:
```
RETAIN(
  (embedding): Embedding(619, 128)
  (rnn_a): GRU(128, 128, batch_first=True)
  (rnn_b): GRU(128, 128, batch_first=True)
  (att_a): AlphaAttention(
    (a_att): Linear(in_features=128, out_features=1, bias=True)
  )
  (att_b): BetaAttention(
    (b_att): Linear(in_features=128, out_features=128, bias=True)
  )
  (fc): Linear(in_features=128, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

[21]:
```
assert retain.att_a.a_att.in_features == 128, "alpha attention input features␣
↪is wrong"
assert retain.att_a.a_att.out_features == 1, "alpha attention output features␣
↪is wrong"
assert retain.att_b.b_att.in_features == 128, "beta attention input features is␣
↪wrong"
assert retain.att_b.b_att.out_features == 128, "beta attention output features␣
↪is wrong"
```

1.5 3 Training and Inferencing [10 points]

Then, let us implement the `eval()` function first.

[22]:
```
# 3 - eval
from sklearn.metrics import precision_recall_fscore_support, roc_auc_score

def eval(model, val_loader):
    model.eval()
    y_pred  = torch.LongTensor()
    y_score = torch.Tensor()
    y_true  = torch.LongTensor()

    with torch.no_grad():
        for x, masks, rev_x, rev_masks, y in val_loader:
            y_logit = model(x, masks, rev_x, rev_masks)      # (B,)
            y_hat   = (y_logit > 0.5).long()                 # predicted class

            y_score = torch.cat((y_score, y_logit.cpu()), dim=0)
            y_pred  = torch.cat((y_pred,  y_hat.cpu()),   dim=0)
            y_true  = torch.cat((y_true,  y.cpu()),       dim=0)

    p, r, f, _ = precision_recall_fscore_support(y_true, y_pred,␣
↪average='binary')
    roc_auc = roc_auc_score(y_true, y_score)
    return p, r, f, roc_auc
```

Now let us implement the `train()` function. Note that `train()` should call `eval()` at the end of each training epoch to see the results on the validation dataset.

[23]:
```
# 3 - train
def train(model, train_loader, val_loader, n_epochs):
    model.train()
    for epoch in range(n_epochs):
        train_loss = 0.0
        for x, masks, rev_x, rev_masks, y in train_loader:
            optimizer.zero_grad()
            y_hat = model(x, masks, rev_x, rev_masks)   # (B,)
```

```
            loss = criterion(y_hat, y)                 # BCE on probabilities
            loss.backward()
            optimizer.step()
            train_loss += loss.item()

        train_loss /= len(train_loader)
        print(f'Epoch: {epoch+1} \t Training Loss: {train_loss:.6f}')
        p, r, f, roc_auc = eval(model, val_loader)
        print(f'Epoch: {epoch+1} \t Validation p: {p:.2f}, r:{r:.2f}, f:{f:.
↪2f}, roc_auc:{roc_auc:.2f}')
    return round(roc_auc, 2)
```

[24]:
```
# load the model
retain = RETAIN(num_codes = len(types))

# load the loss function
criterion = nn.BCELoss()
# load the optimizer
optimizer = torch.optim.Adam(retain.parameters(), lr=1e-3)

n_epochs = 5
train(retain, train_loader, val_loader, n_epochs)
```

```
Epoch: 1         Training Loss: 0.646113
Epoch: 1         Validation p: 0.75, r:0.83, f:0.78, roc_auc:0.83
Epoch: 2         Training Loss: 0.469833
Epoch: 2         Validation p: 0.77, r:0.74, f:0.75, roc_auc:0.84
Epoch: 3         Training Loss: 0.293150
Epoch: 3         Validation p: 0.78, r:0.81, f:0.79, roc_auc:0.83
Epoch: 4         Training Loss: 0.148171
Epoch: 4         Validation p: 0.77, r:0.82, f:0.79, roc_auc:0.84
Epoch: 5         Training Loss: 0.068694
Epoch: 5         Validation p: 0.82, r:0.79, f:0.80, roc_auc:0.85
```

[24]: 0.85

[25]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[25]: '\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'

1.6 4 Sensitivity analysis [5 points]

We will train the same model but with different hyperparameters. We will be using 0.1 and 0.001 for learning rate, and 16, 128 for embedding dimensions. It shows how model performance varies with different values of learning rate and embedding dimensions.

[26]:
```
# 4 - Sensitivity analysis
results = {}
for lr in [1e-1, 1e-3]:
    for embedding_dim in [8, 128]:
        print('='*50)
        print({'learning rate': lr, 'embedding_dim': embedding_dim})
        print('-'*50)

        retain = RETAIN(num_codes=len(types), embedding_dim=embedding_dim)
        criterion = nn.BCELoss()
        optimizer = torch.optim.Adam(retain.parameters(), lr=lr)

        roc_auc = train(retain, train_loader, val_loader, n_epochs=5)
        results[f'lr:{lr},emb:{embedding_dim}'] = roc_auc
```

```
==================================================
{'learning rate': 0.1, 'embedding_dim': 8}
--------------------------------------------------
Epoch: 1         Training Loss: 0.671265
Epoch: 1         Validation p: 0.72, r:0.66, f:0.69, roc_auc:0.78
Epoch: 2         Training Loss: 0.563677
Epoch: 2         Validation p: 0.69, r:0.85, f:0.77, roc_auc:0.81
Epoch: 3         Training Loss: 0.530108
Epoch: 3         Validation p: 0.69, r:0.75, f:0.72, roc_auc:0.78
Epoch: 4         Training Loss: 0.527535
Epoch: 4         Validation p: 0.72, r:0.83, f:0.77, roc_auc:0.79
Epoch: 5         Training Loss: 0.465458
Epoch: 5         Validation p: 0.68, r:0.91, f:0.78, roc_auc:0.81
==================================================
{'learning rate': 0.1, 'embedding_dim': 128}
--------------------------------------------------
Epoch: 1         Training Loss: 1.892332
Epoch: 1         Validation p: 0.53, r:0.99, f:0.69, roc_auc:0.56
Epoch: 2         Training Loss: 2.636945
Epoch: 2         Validation p: 0.56, r:0.89, f:0.69, roc_auc:0.58
Epoch: 3         Training Loss: 1.763409
Epoch: 3         Validation p: 0.52, r:0.96, f:0.68, roc_auc:0.56
Epoch: 4         Training Loss: 1.917141
Epoch: 4         Validation p: 0.54, r:0.99, f:0.70, roc_auc:0.63
Epoch: 5         Training Loss: 1.788670
Epoch: 5         Validation p: 0.55, r:1.00, f:0.71, roc_auc:0.64
==================================================
{'learning rate': 0.001, 'embedding_dim': 8}
--------------------------------------------------
Epoch: 1         Training Loss: 0.696755
Epoch: 1         Validation p: 0.54, r:0.92, f:0.68, roc_auc:0.60
Epoch: 2         Training Loss: 0.683072
```

```
Epoch: 2         Validation p: 0.55, r:0.93, f:0.70, roc_auc:0.63
Epoch: 3         Training Loss: 0.673072
Epoch: 3         Validation p: 0.58, r:0.92, f:0.71, roc_auc:0.66
Epoch: 4         Training Loss: 0.667294
Epoch: 4         Validation p: 0.59, r:0.92, f:0.72, roc_auc:0.68
Epoch: 5         Training Loss: 0.662536
Epoch: 5         Validation p: 0.60, r:0.92, f:0.73, roc_auc:0.69
==================================================
{'learning rate': 0.001, 'embedding_dim': 128}
--------------------------------------------------
Epoch: 1         Training Loss: 0.644957
Epoch: 1         Validation p: 0.73, r:0.81, f:0.77, roc_auc:0.83
Epoch: 2         Training Loss: 0.461738
Epoch: 2         Validation p: 0.73, r:0.73, f:0.73, roc_auc:0.82
Epoch: 3         Training Loss: 0.280822
Epoch: 3         Validation p: 0.73, r:0.72, f:0.73, roc_auc:0.81
Epoch: 4         Training Loss: 0.147574
Epoch: 4         Validation p: 0.74, r:0.71, f:0.72, roc_auc:0.81
Epoch: 5         Training Loss: 0.069153
Epoch: 5         Validation p: 0.72, r:0.81, f:0.76, roc_auc:0.82
```

[27]:

```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert results['lr:0.1,emb:128'] < 0.7, "auc roc should be below 0.7! Since␣
↪higher learning rate of 0.1 will not allow the model to converge."
```

[28]:

```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[28]: '\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'

[ ]:

```
```