A2_part3_clinical_BERT_embeddings_and_readmission_prediction

November 2, 2022

1 Assignment 2 - part 3 - BERT Embeddings For Prediction

In this part of the assignment, we will attempt the same prediction task as part 2, but with two differences.

Different subsequencing strategy Models for sequence data need fixed sequence lengths. In part 2 we just used just the first ~500 words of each note. In part 3 we will break each note into 500-word chunks and train the model to classify each chunk separately. Then we will combine the chunked predictions into one prediction for the whole note. This is sometimes referred to as a 'sliding window' or 'binning'. (Here is a discussion of strategies for long-text modeling with BERT.)

Different embedding strategy We need to convert sequences of word tokens to a vector representation that we can then use in a prediction model. In part 2 we converted each of the first 500 words into 500 Word2Vec embedding vectors, and then passed that sequence of 500 vectors to an LSTM prediction model. In part 3 we will instead convert each note sequence to a single vector. This vector is something we can get from BERT, a popular transformer model. Specifically we will be using a BERT model trained on biomedical and clinical data, similar to the ClinicalBert paper.

In the next cell replace `ROOT` with your path.

[1]:
```
import readmission_utils
import tensorflow as tf
import pandas as pd
import random
import pickle
import numpy as np
import matplotlib.pyplot as plt
import bert_utils

ROOT = "/home/jupyter/cs271_assign2/ROOT"  # Put your root path here"
tf.keras.backend.set_floatx("float32")
```

1.1 Preprocessing text data and visualization

Execute the code in the next cell, which will take about 60mins the first time you run it. It will save its results to a file in `ROOT/saved_data/texts_to_labels_5000.pkl`. This is the same code as part 2, except now we have 5000 notes instead of 1000.

If the file already exists then calling the function will just load the results. We also break the notes and labels into train/val/test sets.

[2]:
```
notes, labels = readmission_utils.get_notes_and_labels(ROOT, 5000)
```

```
Found file /home/jupyter/cs271_assign2/ROOT/saved_data/texts_to_labels_5000.pkl,
loading
```

Run the following code which loads a a pretrained Bert model from the HuggingFace transformers library. This library provides a standard interface for tokenizers and transformers in Tensorflow and PyTorch. HuggingFace also provides a platform for reserachers to share pretrained models. For example we are using this BERT model + tokenizer that has been trained on a dataset of biomedical texts.

This code should take less than 1 minute to run.

[3]:
```
# install transformers library
!pip install transformers==4.11.3
```

```
Requirement already satisfied: transformers==4.11.3 in
/opt/conda/lib/python3.7/site-packages (4.11.3)
Requirement already satisfied: huggingface-hub>=0.0.17 in
/opt/conda/lib/python3.7/site-packages (from transformers==4.11.3) (0.10.1)
Requirement already satisfied: packaging>=20.0 in /opt/conda/lib/python3.7/site-
packages (from transformers==4.11.3) (21.3)
Requirement already satisfied: sacremoses in /opt/conda/lib/python3.7/site-
packages (from transformers==4.11.3) (0.0.53)
Requirement already satisfied: requests in /opt/conda/lib/python3.7/site-
packages (from transformers==4.11.3) (2.28.1)
Requirement already satisfied: filelock in /opt/conda/lib/python3.7/site-
packages (from transformers==4.11.3) (3.8.0)
Requirement already satisfied: numpy>=1.17 in /opt/conda/lib/python3.7/site-
packages (from transformers==4.11.3) (1.19.5)
Requirement already satisfied: tokenizers<0.11,>=0.10.1 in
/opt/conda/lib/python3.7/site-packages (from transformers==4.11.3) (0.10.3)
Requirement already satisfied: pyyaml>=5.1 in /opt/conda/lib/python3.7/site-
packages (from transformers==4.11.3) (6.0)
Requirement already satisfied: regex!=2019.12.17 in
/opt/conda/lib/python3.7/site-packages (from transformers==4.11.3) (2022.9.13)
Requirement already satisfied: importlib-metadata in
/opt/conda/lib/python3.7/site-packages (from transformers==4.11.3) (4.11.4)
Requirement already satisfied: tqdm>=4.27 in /opt/conda/lib/python3.7/site-
packages (from transformers==4.11.3) (4.64.1)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/opt/conda/lib/python3.7/site-packages (from huggingface-
hub>=0.0.17->transformers==4.11.3) (3.10.0.2)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/opt/conda/lib/python3.7/site-packages (from
packaging>=20.0->transformers==4.11.3) (3.0.9)
```

```
Requirement already satisfied: zipp>=0.5 in /opt/conda/lib/python3.7/site-
packages (from importlib-metadata->transformers==4.11.3) (3.10.0)
Requirement already satisfied: charset-normalizer<3,>=2 in
/opt/conda/lib/python3.7/site-packages (from requests->transformers==4.11.3)
(2.1.1)
Requirement already satisfied: certifi>=2017.4.17 in
/opt/conda/lib/python3.7/site-packages (from requests->transformers==4.11.3)
(2022.9.24)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/opt/conda/lib/python3.7/site-packages (from requests->transformers==4.11.3)
(1.26.11)
Requirement already satisfied: idna<4,>=2.5 in /opt/conda/lib/python3.7/site-
packages (from requests->transformers==4.11.3) (3.4)
Requirement already satisfied: click in /opt/conda/lib/python3.7/site-packages
(from sacremoses->transformers==4.11.3) (8.1.3)
Requirement already satisfied: joblib in /opt/conda/lib/python3.7/site-packages
(from sacremoses->transformers==4.11.3) (1.2.0)
Requirement already satisfied: six in /opt/conda/lib/python3.7/site-packages
(from sacremoses->transformers==4.11.3) (1.15.0)
```

[4]:

```
from transformers import AutoTokenizer, TFAutoModel
import readmission_utils

hf_model = "cambridgeltl/SapBERT-from-PubMedBERT-fulltext"
tokenizer = AutoTokenizer.from_pretrained(hf_model)
bert_model = TFAutoModel.from_pretrained(hf_model)
```

```
2022-11-01 05:17:55.418344: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-01 05:17:55.428626: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-01 05:17:55.430370: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-01 05:17:55.432994: I tensorflow/core/platform/cpu_feature_guard.cc:142]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2022-11-01 05:17:55.433789: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
```

```
node, so returning NUMA node zero
2022-11-01 05:17:55.435705: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-01 05:17:55.437407: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-01 05:17:55.991730: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-01 05:17:55.993623: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-01 05:17:55.995282: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-01 05:17:55.996870: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 13642 MB memory:  -> device:
0, name: Tesla T4, pci bus id: 0000:00:04.0, compute capability: 7.5
All model checkpoint layers were used when initializing TFBertModel.

All the layers of TFBertModel were initialized from the model checkpoint at
cambridgeltl/SapBERT-from-PubMedBERT-fulltext.
If your task is similar to the task the model of the checkpoint was trained on,
you can already use TFBertModel for predictions without further training.
```

Now run the following data preparation code. We'll explain what it does later. It will take about ~20mins the first time it's run and it will save data to `{ROOT}/saved_data/bert_datasets.pkl`. For later runs, it will just load this file.

[5]:
```
data = bert_utils.prepare_bert_datasets(ROOT, notes, labels, bert_model,
 ↪tokenizer)
```

```
File /home/jupyter/cs271_assign2/ROOT/saved_data/bert_datasets.pkl exists.
Loading it
```

Q3.1 BERT architecture

LSTMs model sequence dependencies using recurrence; they are recurrent neural networks or RNNs. In RNNs we pass elements of a sequence through the model one a time (sequentially). Each pass through the RNN updates an internal state vector. Future passes through the RNN are a function of the state vector. This is how RNNs can model dependencies between elements in a sequence.

On the other hand, Bert has a transformer architecture. Transformers are state-of-the-art in most standard tasks in language modelling. Instead of processing sequence data one-at-a-time, transformers process entire sequences at once. But they still model dependencies between sequence elements. Briefly describe the mechanism that transformers use to model sequence dependencies. (You can refer to the lecture slides, or the major transformers paper, Attention is all you need).

1.1.1 Written answer: Transformers use attention vectors in order to model dependencies. Attention vectors are vectors which portray how relevant one word is to every other word in the input sequence. For example, say our problem was to translate an english sentence into a french one. If our sentences was "The big red dog", the attention vector would model how much focus we should put on each word for translation. For example, the vector may look like [0.71, 0.04, 0.07, 0.18], which tells our model how much we should focus on each word in the sequence. We often use multi-headed attention which is simply a weighted average of different attention vectors for one word. This is how the transformer architecture models sequence dependencies.

Q3.2 BERT pretraining

Briefly describe the 2 pretraining tasks discussed in the introduction to the BERT paper.

1.1.2 Written answer: 1. Masked LM: In order to train a deep bidirectional representation, the authors simply mask some percentage of the input tokens at random, and then predict those masked tokens. After masking, the final hidden vectors corresponding to the masked tokens are given to an output softmax over the vocabulary in order to predict the masked word. However, the fine-tuning task of the model is not going to see the [MASK] token as its input. To mitigate this, we do not always replace masked words with the actual [MASK] token. Instead, 80% of the time, the 15% tokens are masked; 10% of the time, 15% tokens are replaced with random tokens; and 10% of the time, they are kept as is.

1.1.3 2. Next Sentence Prediction (NSP): In order to train a model that understands sentence relationships, the authors pre-train for binarized a next sentence prediction task. When chosing sentences A and B for a pre-training example, 50% of the time the correct subsequent next sentence is chosen as B, and 50% of the time B is chosen as a random sentence from the corpus of sentences. This method ensures that the model trains on multiple sequences.

Q3.3 datasets for BERT pretraining

What is the benefit of using a BERT model that has been pretrained on biomedical text compared with, for example, a BERT model trained on Wikipedia?

1.1.4 Written answer: This is because we have more domain-specific words/sentences in our corpus. In typical settings such as Wikipedia, medical terms (such as benign, biopsy, etc.) and medical phrases/sentences will be quite rare to come by. Thus, relevant words/phrases are often not available to the model at prediction time since they are not (or are very rarely) in the vocabulary. As a result, models designed for general purpose language understanding often obtain poor performance in biomedical text mining tasks.

Q3.4 data chunking strategy

Let's look at some of the data we created earlier when we ran `bert_utils.prepare_bert_datasets`. First we did a train/val/test split for the notes and labels. All these variables have the suffix, `_FULL`, indicating that this is the full note, before chunking.

[6]:
```
[
    train_notes_FULL,
    train_labels_FULL,
    val_notes_FULL,
    val_labels_FULL,
    test_notes_FULL,
    test_labels_FULL,
] = data["FULL"]

all_note_lengths = [len(train_notes_FULL), len(val_notes_FULL),␣
 ↪len(test_notes_FULL)]
print(f"train/val/test lengths {all_note_lengths} \n")
print(f"Which sum to {sum(all_note_lengths)}")
print(f"Original notes len {len(notes)}")
```

```
train/val/test lengths [2853, 951, 951]

Which sum to 4755
Original notes len 4755
```

Now we do chunking for the train, val and test sets separately (this is what we described in the "Different subsequencing strategy" section at the start of the assignment).

Take the train set for example. We break the notes into ~500 word chunks, `train_notes_CHUNKS`. We copy the labels into `train_labels_CHUNKS`. Finally, `train_idxs_CHUNKS` tells you which FULL note this CHUNK is from. Suppose `train_idxs_CHUNKS[30]=6`; this means `train_notes_CHUNKS[30]` is a subsequence of the note `train_notes_FULL[6]`.

Here is an example: - If `train_notes_FULL[0]` is about 1200 words long, then we create 3 note-chunks that will be in `train_notes_CHUNKS[0:3]` - If the label is `train_notes_FULL[0]=1` then we copy that label for each note-chunk, so `train_labels_CHUNKS[0:3]=1`. - Since these chunks are all subsequences of `train_notes_FULL[0]`, we set `train_idxs_CHUNKS[0:3]=0`.

We print the labels and indxs for the first 25 entries. You should verify that the results match your understanding of this dataset.

```
[7]: [
    train_notes_CHUNKS,
    train_labels_CHUNKS,
    train_idxs_CHUNKS,
    val_notes_CHUNKS,
    val_labels_CHUNKS,
    val_idxs_CHUNKS,
    test_notes_CHUNKS,
    test_labels_CHUNKS,
    test_idxs_CHUNKS,
] = data["CHUNKS_DATA"]


print("First 20 chunks:")
print(f"Labels      : {train_labels_CHUNKS[:25]}")
print(f"Indexes.    : {train_idxs_CHUNKS[:25]}")
```

```
First 20 chunks:
Labels      : [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1]
Indexes.    : [0 0 0 0 0 0 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4]
```

Briefly discuss the pros and cons of this data chunking strategy compared to the truncation strategy used in part 2.

1.1.5 Written answer: The pro of this truncation strategy is that our prediction takes the entire sequence into account. For example, if the first 512 words are introdcutory sentences that do not give much context about the state of the patient, then we will not be able to make a very meaningful prediction. In this strategy, we break the sequence up into chunks of 500 words, make predictions for each chunk, and combine chunk predictions into one prediction for the whole note. Say our middle group of words or last 500 words gave context regarding the patient's condition. In this strategy, we will be able to use those in our overall prediction, leading to better performance. A con of this truncation strategy is that it takes up much more computational power. We will need to process many more chunks and make multiple predictions for every sentence, which will take up a lot more time and resources.

Q3.5 BERT embeddings

Finally we took these chunked notes and put them into BERT pooled embeddings.

```
[8]: [
    train_bert_pool_embeddings_CHUNKS,
    val_bert_pool_embeddings_CHUNKS,
    test_bert_pool_embeddings_CHUNKS,
] = data["CHUNKS_EMEDDINGS"]
print(f"Length of train_notes_CHUNKS               {len(train_notes_CHUNKS)}")
print(
```

```
    f"Shape of train_bert_pool_embeddings_CHUNKS␣
 ↪{train_bert_pool_embeddings_CHUNKS.shape}"
)
```

```
Length of train_notes_CHUNKS               12336
Shape of train_bert_pool_embeddings_CHUNKS (12336, 768)
```

For our prediction task: - The x-data is `train_bert_pool_embeddings_CHUNKS`. - They y-labels are `train_labels_CHUNKS`.

Look at the shape of `train_bert_pool_embeddings_CHUNKS` printed in the above cell. There is one single BERT embedding vector for each note chunk. This is called the "pooled BERT embedding", and is also the h_{CLS} token output discussed in lecture.

How is this embedding different to the embeddings used in part 2? Specifically talk about the shape of the data that we will pass into a prediction model.

1.1.6 Written answer: In part 2, the shape of our word embeddings was (500,32), where there was a 32 dimensional embedding vector for each word in the 500 word vocabulary. Now, our embedding is of shape (12336, 768), where there is a 768 dimensional embedding vector for each training note chunk. Each embedding in this schema represents a chunk (sequence of up to 500 words) rather than a single word in a vocabulary. Additionally, this input embedding is the sum of multiple separate embeddings: a token embedding, a segment embedding (indicating which segment the token belongs to), and a positional embedding (where the token is located in the sequence).

1.1.7 Prediction model

Q3.6 build and run prediction model The inputs to our model are single-vector BERT embeddings. These embeddings should do a very good job of summarising the text, such that our prediction model can be extremely simple: - The input is the BERT embedding vector. - We have one Dense layer with 1 node output and sigmoid activation (no hidden layers).

Compile this model with. - Adam. - Binary cross entropy loss. - Metrics for accuracy and AUC.

Train it for 100 epoch with batch size 128, and pass in the validation dataset.

(Optional: you can experiment with adding extra dense layers and dropout. See if you can avoid overfitting.)

[9]:
```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
```

[10]:
```
train_x = train_bert_pool_embeddings_CHUNKS
train_y = np.array(train_labels_CHUNKS)
val_x = val_bert_pool_embeddings_CHUNKS
val_y = np.array(val_labels_CHUNKS)
test_x = test_bert_pool_embeddings_CHUNKS
test_y = np.array(test_labels_CHUNKS)
```

```python
# YOUR CODE HERE #
model = Sequential()
model.add(Dense(1, activation = 'sigmoid'))

optimizer = tf.keras.optimizers.Adam()
loss = tf.keras.losses.BinaryCrossentropy()
metrics = ['accuracy', tf.keras.metrics.AUC()]

model.compile(optimizer, loss, metrics)

epochs = 100
hist = model.fit(
    train_x,
    train_y,
    batch_size = 128,
    epochs = epochs,
    validation_data = (val_x, val_y)
)


# END CODE #
```

```
Epoch 1/100
```

```
2022-11-01 05:17:57.789043: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR
Optimization Passes are enabled (registered 2)
```

```
97/97 [==============================] - 1s 6ms/step - loss: 0.6968 - accuracy:
0.5340 - auc: 0.5376 - val_loss: 0.6798 - val_accuracy: 0.5711 - val_auc: 0.5985
Epoch 2/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6780 - accuracy:
0.5674 - auc: 0.5935 - val_loss: 0.6653 - val_accuracy: 0.5945 - val_auc: 0.6261
Epoch 3/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6738 - accuracy:
0.5805 - auc: 0.6070 - val_loss: 0.6614 - val_accuracy: 0.5972 - val_auc: 0.6343
Epoch 4/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6711 - accuracy:
0.5852 - auc: 0.6151 - val_loss: 0.6594 - val_accuracy: 0.6025 - val_auc: 0.6393
Epoch 5/100
97/97 [==============================] - 0s 4ms/step - loss: 0.6681 - accuracy:
0.5902 - auc: 0.6227 - val_loss: 0.6580 - val_accuracy: 0.6091 - val_auc: 0.6407
Epoch 6/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6666 - accuracy:
0.5909 - auc: 0.6258 - val_loss: 0.6572 - val_accuracy: 0.6033 - val_auc: 0.6436
Epoch 7/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6659 - accuracy:
0.5906 - auc: 0.6282 - val_loss: 0.6585 - val_accuracy: 0.6062 - val_auc: 0.6464
```

```
Epoch 8/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6643 - accuracy:
0.5947 - auc: 0.6319 - val_loss: 0.6549 - val_accuracy: 0.6108 - val_auc: 0.6474
Epoch 9/100
97/97 [==============================] - 1s 6ms/step - loss: 0.6637 - accuracy:
0.5950 - auc: 0.6332 - val_loss: 0.6543 - val_accuracy: 0.6106 - val_auc: 0.6484
Epoch 10/100
97/97 [==============================] - 1s 5ms/step - loss: 0.6616 - accuracy:
0.6023 - auc: 0.6394 - val_loss: 0.6602 - val_accuracy: 0.6028 - val_auc: 0.6494
Epoch 11/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6610 - accuracy:
0.6008 - auc: 0.6400 - val_loss: 0.6543 - val_accuracy: 0.6113 - val_auc: 0.6505
Epoch 12/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6619 - accuracy:
0.6020 - auc: 0.6383 - val_loss: 0.6546 - val_accuracy: 0.6047 - val_auc: 0.6509
Epoch 13/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6590 - accuracy:
0.5988 - auc: 0.6439 - val_loss: 0.6525 - val_accuracy: 0.6123 - val_auc: 0.6512
Epoch 14/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6591 - accuracy:
0.6043 - auc: 0.6438 - val_loss: 0.6521 - val_accuracy: 0.6152 - val_auc: 0.6519
Epoch 15/100
97/97 [==============================] - 0s 4ms/step - loss: 0.6601 - accuracy:
0.6004 - auc: 0.6406 - val_loss: 0.6516 - val_accuracy: 0.6128 - val_auc: 0.6529
Epoch 16/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6565 - accuracy:
0.6069 - auc: 0.6506 - val_loss: 0.6544 - val_accuracy: 0.6120 - val_auc: 0.6530
Epoch 17/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6566 - accuracy:
0.6057 - auc: 0.6490 - val_loss: 0.6631 - val_accuracy: 0.5945 - val_auc: 0.6534
Epoch 18/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6571 - accuracy:
0.6058 - auc: 0.6481 - val_loss: 0.6589 - val_accuracy: 0.5991 - val_auc: 0.6536
Epoch 19/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6566 - accuracy:
0.6038 - auc: 0.6493 - val_loss: 0.6506 - val_accuracy: 0.6113 - val_auc: 0.6541
Epoch 20/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6553 - accuracy:
0.6099 - auc: 0.6529 - val_loss: 0.6528 - val_accuracy: 0.6142 - val_auc: 0.6546
Epoch 21/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6563 - accuracy:
0.6095 - auc: 0.6500 - val_loss: 0.6523 - val_accuracy: 0.6074 - val_auc: 0.6544
Epoch 22/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6556 - accuracy:
0.6100 - auc: 0.6514 - val_loss: 0.6513 - val_accuracy: 0.6125 - val_auc: 0.6537
Epoch 23/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6539 - accuracy:
0.6085 - auc: 0.6551 - val_loss: 0.6508 - val_accuracy: 0.6069 - val_auc: 0.6550
```

```
Epoch 24/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6553 - accuracy:
0.6078 - auc: 0.6516 - val_loss: 0.6617 - val_accuracy: 0.5925 - val_auc: 0.6541
Epoch 25/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6527 - accuracy:
0.6129 - auc: 0.6580 - val_loss: 0.6502 - val_accuracy: 0.6064 - val_auc: 0.6561
Epoch 26/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6543 - accuracy:
0.6089 - auc: 0.6535 - val_loss: 0.6781 - val_accuracy: 0.5770 - val_auc: 0.6551
Epoch 27/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6541 - accuracy:
0.6111 - auc: 0.6549 - val_loss: 0.6541 - val_accuracy: 0.6076 - val_auc: 0.6558
Epoch 28/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6530 - accuracy:
0.6141 - auc: 0.6580 - val_loss: 0.6493 - val_accuracy: 0.6128 - val_auc: 0.6560
Epoch 29/100
97/97 [==============================] - 0s 4ms/step - loss: 0.6519 - accuracy:
0.6142 - auc: 0.6600 - val_loss: 0.6489 - val_accuracy: 0.6094 - val_auc: 0.6560
Epoch 30/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6516 - accuracy:
0.6154 - auc: 0.6606 - val_loss: 0.6496 - val_accuracy: 0.6128 - val_auc: 0.6561
Epoch 31/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6525 - accuracy:
0.6105 - auc: 0.6573 - val_loss: 0.6524 - val_accuracy: 0.6084 - val_auc: 0.6561
Epoch 32/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6504 - accuracy:
0.6134 - auc: 0.6623 - val_loss: 0.6489 - val_accuracy: 0.6101 - val_auc: 0.6559
Epoch 33/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6506 - accuracy:
0.6159 - auc: 0.6620 - val_loss: 0.6636 - val_accuracy: 0.5960 - val_auc: 0.6555
Epoch 34/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6509 - accuracy:
0.6165 - auc: 0.6616 - val_loss: 0.6548 - val_accuracy: 0.6084 - val_auc: 0.6563
Epoch 35/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6499 - accuracy:
0.6155 - auc: 0.6636 - val_loss: 0.6484 - val_accuracy: 0.6072 - val_auc: 0.6574
Epoch 36/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6496 - accuracy:
0.6162 - auc: 0.6640 - val_loss: 0.6484 - val_accuracy: 0.6111 - val_auc: 0.6570
Epoch 37/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6492 - accuracy:
0.6151 - auc: 0.6647 - val_loss: 0.6509 - val_accuracy: 0.6113 - val_auc: 0.6566
Epoch 38/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6497 - accuracy:
0.6178 - auc: 0.6642 - val_loss: 0.6514 - val_accuracy: 0.6120 - val_auc: 0.6563
Epoch 39/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6503 - accuracy:
0.6157 - auc: 0.6623 - val_loss: 0.6490 - val_accuracy: 0.6103 - val_auc: 0.6569
```

```
Epoch 40/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6494 - accuracy:
0.6167 - auc: 0.6641 - val_loss: 0.6486 - val_accuracy: 0.6076 - val_auc: 0.6577
Epoch 41/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6482 - accuracy:
0.6191 - auc: 0.6671 - val_loss: 0.6477 - val_accuracy: 0.6074 - val_auc: 0.6575
Epoch 42/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6501 - accuracy:
0.6161 - auc: 0.6622 - val_loss: 0.6575 - val_accuracy: 0.6042 - val_auc: 0.6572
Epoch 43/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6499 - accuracy:
0.6184 - auc: 0.6629 - val_loss: 0.6484 - val_accuracy: 0.6064 - val_auc: 0.6573
Epoch 44/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6474 - accuracy:
0.6172 - auc: 0.6676 - val_loss: 0.6475 - val_accuracy: 0.6045 - val_auc: 0.6586
Epoch 45/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6491 - accuracy:
0.6198 - auc: 0.6647 - val_loss: 0.6481 - val_accuracy: 0.6062 - val_auc: 0.6578
Epoch 46/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6490 - accuracy:
0.6204 - auc: 0.6650 - val_loss: 0.6539 - val_accuracy: 0.6047 - val_auc: 0.6581
Epoch 47/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6490 - accuracy:
0.6152 - auc: 0.6650 - val_loss: 0.6554 - val_accuracy: 0.6076 - val_auc: 0.6578
Epoch 48/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6474 - accuracy:
0.6208 - auc: 0.6675 - val_loss: 0.6526 - val_accuracy: 0.6045 - val_auc: 0.6561
Epoch 49/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6500 - accuracy:
0.6158 - auc: 0.6615 - val_loss: 0.6472 - val_accuracy: 0.6084 - val_auc: 0.6587
Epoch 50/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6475 - accuracy:
0.6213 - auc: 0.6678 - val_loss: 0.6484 - val_accuracy: 0.6081 - val_auc: 0.6582
Epoch 51/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6463 - accuracy:
0.6187 - auc: 0.6696 - val_loss: 0.6474 - val_accuracy: 0.6052 - val_auc: 0.6581
Epoch 52/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6484 - accuracy:
0.6205 - auc: 0.6663 - val_loss: 0.6483 - val_accuracy: 0.6084 - val_auc: 0.6579
Epoch 53/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6465 - accuracy:
0.6218 - auc: 0.6694 - val_loss: 0.6497 - val_accuracy: 0.6106 - val_auc: 0.6569
Epoch 54/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6461 - accuracy:
0.6234 - auc: 0.6704 - val_loss: 0.6498 - val_accuracy: 0.6072 - val_auc: 0.6577
Epoch 55/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6468 - accuracy:
0.6188 - auc: 0.6691 - val_loss: 0.6479 - val_accuracy: 0.6055 - val_auc: 0.6569
```

```
Epoch 56/100
97/97 [==============================] - 0s 4ms/step - loss: 0.6461 - accuracy:
0.6210 - auc: 0.6699 - val_loss: 0.6502 - val_accuracy: 0.6113 - val_auc: 0.6575
Epoch 57/100
97/97 [==============================] - 0s 4ms/step - loss: 0.6460 - accuracy:
0.6188 - auc: 0.6703 - val_loss: 0.6521 - val_accuracy: 0.6084 - val_auc: 0.6578
Epoch 58/100
97/97 [==============================] - 0s 4ms/step - loss: 0.6457 - accuracy:
0.6240 - auc: 0.6716 - val_loss: 0.6533 - val_accuracy: 0.6069 - val_auc: 0.6574
Epoch 59/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6467 - accuracy:
0.6204 - auc: 0.6686 - val_loss: 0.6492 - val_accuracy: 0.6076 - val_auc: 0.6578
Epoch 60/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6467 - accuracy:
0.6239 - auc: 0.6693 - val_loss: 0.6480 - val_accuracy: 0.6081 - val_auc: 0.6578
Epoch 61/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6452 - accuracy:
0.6240 - auc: 0.6722 - val_loss: 0.6497 - val_accuracy: 0.6098 - val_auc: 0.6577
Epoch 62/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6445 - accuracy:
0.6242 - auc: 0.6733 - val_loss: 0.6476 - val_accuracy: 0.6123 - val_auc: 0.6578
Epoch 63/100
97/97 [==============================] - 0s 4ms/step - loss: 0.6455 - accuracy:
0.6226 - auc: 0.6722 - val_loss: 0.6475 - val_accuracy: 0.6081 - val_auc: 0.6582
Epoch 64/100
97/97 [==============================] - 0s 4ms/step - loss: 0.6456 - accuracy:
0.6235 - auc: 0.6716 - val_loss: 0.6474 - val_accuracy: 0.6047 - val_auc: 0.6578
Epoch 65/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6451 - accuracy:
0.6238 - auc: 0.6718 - val_loss: 0.6483 - val_accuracy: 0.6130 - val_auc: 0.6577
Epoch 66/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6470 - accuracy:
0.6241 - auc: 0.6683 - val_loss: 0.6515 - val_accuracy: 0.6072 - val_auc: 0.6579
Epoch 67/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6447 - accuracy:
0.6220 - auc: 0.6726 - val_loss: 0.6610 - val_accuracy: 0.5999 - val_auc: 0.6576
Epoch 68/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6454 - accuracy:
0.6221 - auc: 0.6707 - val_loss: 0.6483 - val_accuracy: 0.6094 - val_auc: 0.6579
Epoch 69/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6448 - accuracy:
0.6248 - auc: 0.6736 - val_loss: 0.6479 - val_accuracy: 0.6055 - val_auc: 0.6577
Epoch 70/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6472 - accuracy:
0.6201 - auc: 0.6674 - val_loss: 0.6520 - val_accuracy: 0.6076 - val_auc: 0.6580
Epoch 71/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6447 - accuracy:
0.6273 - auc: 0.6733 - val_loss: 0.6532 - val_accuracy: 0.6074 - val_auc: 0.6572
```

```
Epoch 72/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6440 - accuracy:
0.6238 - auc: 0.6749 - val_loss: 0.6498 - val_accuracy: 0.6115 - val_auc: 0.6570
Epoch 73/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6441 - accuracy:
0.6256 - auc: 0.6744 - val_loss: 0.6511 - val_accuracy: 0.6072 - val_auc: 0.6581
Epoch 74/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6447 - accuracy:
0.6253 - auc: 0.6725 - val_loss: 0.6478 - val_accuracy: 0.6098 - val_auc: 0.6584
Epoch 75/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6447 - accuracy:
0.6231 - auc: 0.6725 - val_loss: 0.6472 - val_accuracy: 0.6106 - val_auc: 0.6578
Epoch 76/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6442 - accuracy:
0.6255 - auc: 0.6739 - val_loss: 0.6474 - val_accuracy: 0.6067 - val_auc: 0.6576
Epoch 77/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6439 - accuracy:
0.6235 - auc: 0.6737 - val_loss: 0.6478 - val_accuracy: 0.6081 - val_auc: 0.6580
Epoch 78/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6427 - accuracy:
0.6280 - auc: 0.6771 - val_loss: 0.6516 - val_accuracy: 0.6038 - val_auc: 0.6583
Epoch 79/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6429 - accuracy:
0.6269 - auc: 0.6753 - val_loss: 0.6471 - val_accuracy: 0.6057 - val_auc: 0.6579
Epoch 80/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6438 - accuracy:
0.6263 - auc: 0.6742 - val_loss: 0.6471 - val_accuracy: 0.6047 - val_auc: 0.6581
Epoch 81/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6431 - accuracy:
0.6255 - auc: 0.6758 - val_loss: 0.6534 - val_accuracy: 0.6074 - val_auc: 0.6567
Epoch 82/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6434 - accuracy:
0.6254 - auc: 0.6752 - val_loss: 0.6474 - val_accuracy: 0.6076 - val_auc: 0.6586
Epoch 83/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6428 - accuracy:
0.6273 - auc: 0.6762 - val_loss: 0.6469 - val_accuracy: 0.6033 - val_auc: 0.6583
Epoch 84/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6422 - accuracy:
0.6308 - auc: 0.6780 - val_loss: 0.6492 - val_accuracy: 0.6103 - val_auc: 0.6587
Epoch 85/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6425 - accuracy:
0.6277 - auc: 0.6768 - val_loss: 0.6542 - val_accuracy: 0.6042 - val_auc: 0.6586
Epoch 86/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6429 - accuracy:
0.6275 - auc: 0.6759 - val_loss: 0.6469 - val_accuracy: 0.6059 - val_auc: 0.6582
Epoch 87/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6425 - accuracy:
0.6250 - auc: 0.6765 - val_loss: 0.6467 - val_accuracy: 0.6059 - val_auc: 0.6586
```

```
Epoch 88/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6440 - accuracy:
0.6274 - auc: 0.6746 - val_loss: 0.6500 - val_accuracy: 0.6089 - val_auc: 0.6584
Epoch 89/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6423 - accuracy:
0.6265 - auc: 0.6767 - val_loss: 0.6487 - val_accuracy: 0.6096 - val_auc: 0.6584
Epoch 90/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6418 - accuracy:
0.6278 - auc: 0.6780 - val_loss: 0.6531 - val_accuracy: 0.6052 - val_auc: 0.6579
Epoch 91/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6439 - accuracy:
0.6223 - auc: 0.6736 - val_loss: 0.6586 - val_accuracy: 0.6035 - val_auc: 0.6576
Epoch 92/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6419 - accuracy:
0.6295 - auc: 0.6783 - val_loss: 0.6479 - val_accuracy: 0.6084 - val_auc: 0.6589
Epoch 93/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6421 - accuracy:
0.6252 - auc: 0.6771 - val_loss: 0.6511 - val_accuracy: 0.6091 - val_auc: 0.6584
Epoch 94/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6421 - accuracy:
0.6268 - auc: 0.6772 - val_loss: 0.6469 - val_accuracy: 0.6035 - val_auc: 0.6581
Epoch 95/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6412 - accuracy:
0.6300 - auc: 0.6785 - val_loss: 0.6527 - val_accuracy: 0.6081 - val_auc: 0.6585
Epoch 96/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6438 - accuracy:
0.6270 - auc: 0.6742 - val_loss: 0.6533 - val_accuracy: 0.6057 - val_auc: 0.6582
Epoch 97/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6447 - accuracy:
0.6222 - auc: 0.6721 - val_loss: 0.6518 - val_accuracy: 0.6067 - val_auc: 0.6585
Epoch 98/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6438 - accuracy:
0.6260 - auc: 0.6740 - val_loss: 0.6486 - val_accuracy: 0.6045 - val_auc: 0.6585
Epoch 99/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6423 - accuracy:
0.6289 - auc: 0.6767 - val_loss: 0.6515 - val_accuracy: 0.6059 - val_auc: 0.6580
Epoch 100/100
97/97 [==============================] - 0s 3ms/step - loss: 0.6424 - accuracy:
0.6288 - auc: 0.6773 - val_loss: 0.6525 - val_accuracy: 0.6062 - val_auc: 0.6586
```

Q3.7 assessing model performance

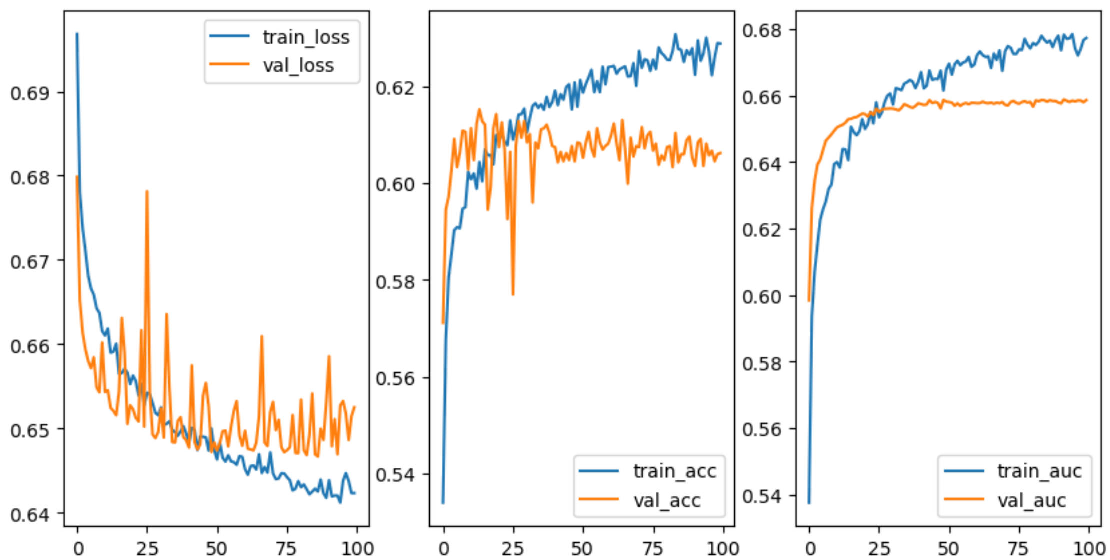
Make 3 plots: one each for loss, accuracy and AUC. Each plot should have train and validation scores labeled.

[11]:
```
f, axs = plt.subplots(1, 3, figsize=(10, 5))
axs[0].plot(hist.history["loss"], label="train_loss")
axs[0].plot(hist.history["val_loss"], label="val_loss")
axs[0].legend()
```

```
axs[1].plot(hist.history["accuracy"], label="train_acc")
axs[1].plot(hist.history["val_accuracy"], label="val_acc")
axs[1].legend()

axs[2].plot(hist.history["auc"], label="train_auc")
axs[2].plot(hist.history["val_auc"], label="val_auc")
axs[2].legend()
```

[11]: <matplotlib.legend.Legend at 0x7f280c7915d0>



Q3.8 combining chunked predictions to full predictions for readmission

In the ClinicalBert paper, the authors did pretraining with MIMIC-III, and in section 3.3.2, they make predictions of hospital readmission on MIMIC-III, which is the same task we are doing.

We've broken our notes into chunks and made predictions for each chunk. The ClinicalBert authors propose a method to combine the chunked notes predictions into one prediction per note (equation 4 in the paper).

Implement this function in the next cell for the *validation* set with `c=1`. Use the combined predictions to compute the AUC. You should get AUC >= 0.65 (we got 0.69).

[12]:
```
from sklearn import metrics
from itertools import compress
```

[15]:
```
def predict_FULL_note_readmission_clinicalBert(
    model, bert_pool_embeddings_CHUNKS, idxs_CHUNKS, c=1
):
    """
    Combine CHUNK predictions to FULL predictions using equation 4 of
```

```
        https://arxiv.org/pdf/1904.05342.pdf

    Args:
    model (tf.keras.Model): a trained prediction model.
    bert_embedding_CHUNK (np.array[float,float]): chunked dataset of bert␣
↪embeddings
        for running prediction.
    idxs_CHUNKS np.array([int]): idxs_CHUNKS[i]=j means chunk i is a␣
↪subsequence of
        a note i.

    Returns:
        y_pred_FULL (np.array([int]))
    """
    n_unique = len(np.unique(idxs_CHUNKS))
    idxs_CHUNKS = np.array(idxs_CHUNKS)
    y_pred_score_CHUNKS = model.predict(bert_pool_embeddings_CHUNKS)

    y_pred_FULL = np.zeros(n_unique)
    for i in range(n_unique):
        # YOUR CODE HERE #
        idx_array = (idxs_CHUNKS == i)
        idxs = np.array(list(compress(range(len(idx_array)), idx_array)))
        pred = np.array([y_pred_score_CHUNKS[j] for j in idxs])
        y_pred_FULL[i] = (np.max(pred) + np.mean(pred)*(len(idxs)/c))/(1 +␣
↪len(idxs)/c)
        # END CODE #
    return y_pred_FULL


c = 1
y_pred_FULL = predict_FULL_note_readmission_clinicalBert(
    model, val_bert_pool_embeddings_CHUNKS, val_idxs_CHUNKS, c=c
)

y_score_FULL = val_labels_FULL
auc = None
# YOUR CODE HERE #
fpr, tpr, thresholds = metrics.roc_curve(y_score_FULL, y_pred_FULL)
auc = metrics.auc(fpr, tpr)
# END CODE #
print(f"AUC {auc:.5f}")
```

```
AUC 0.71946
```

Q3.9 hyperparameter tuning

Run `predict_FULL_note_readmission_clinicalBert` and compute the AUC for a range of `c` values. We will use the best AUC to choose a value of `c`. This is hyperparameter tuning, and so we should do this on the validation set.

[16]:
```
for c in [0.01, 0.1, 0.5, 1, 2, 5, 10, 20, 50]:
    auc = None
    # YOUR CODE HERE #
    y_pred_FULL = predict_FULL_note_readmission_clinicalBert(
        model, val_bert_pool_embeddings_CHUNKS, val_idxs_CHUNKS, c=c)
    fpr, tpr, thresholds = metrics.roc_curve(y_score_FULL, y_pred_FULL)
    auc = metrics.auc(fpr, tpr)
    # END CODE #
    print(f"c {c}\t AUC: {auc:.5f}")
```

```
c 0.01   AUC: 0.71897
c 0.1    AUC: 0.71887
c 0.5    AUC: 0.71943
c 1      AUC: 0.71946
c 2      AUC: 0.71894
c 5      AUC: 0.71686
c 10     AUC: 0.71431
c 20     AUC: 0.71098
c 50     AUC: 0.70796
```

Q3.10 evaluation

Now that you have chosen a `c` value, let's evaluate on the test set. Run `predict_FULL_note_readmission_clinicalBert` and compute the AUC. In the next cell you just have to fill in your value of `c` and compute the `auc`.

[17]:
```
c = 1  # choose your best value

y_pred_FULL = predict_FULL_note_readmission_clinicalBert(
    model, test_bert_pool_embeddings_CHUNKS, test_idxs_CHUNKS, c=c
)
y_score_FULL = test_labels_FULL
auc = None
# YOUR CODE HERE #
fpr, tpr, thresholds = metrics.roc_curve(y_score_FULL, y_pred_FULL)
auc = metrics.auc(fpr, tpr)
# END CODE #
print(f"Test set auc {auc:.5f}")
```

```
Test set auc 0.68030
```

Your test set AUC may be different to tha validation set. Explain why, and give one strategy for getting more consistent results between validation and test.

1.1.8 Written answer: The test set AUC may be different from the validation set AUC since the validation set is not entirely unbiased. The validation set is used for hyperparameter tuning, meaning that it chooses the values for each hyperparameter that give it the best outcome. However, this means that the validation data is inherently used to select the optimal outcome and cannot be used to judge the quality of the model. Thus, the test AUC may be different since the validation can be optimistic. In order to get more consistent results between validation and test sets, I suggest using k-fold cross validation. The results are more consistent between validation and test it gives your model the opportunity to train on multiple train-val-test splits.

[ ]:

```

```