Revisiting Neural Models for Hospital Readmission: A Replication and Extension Study

Christopher Hynes, Felipe Oliveira

I. Abstract

In this project, we replicate Xiao et al. (2018), who proposed the CONTENT model for hospital readmission prediction and claimed it outperforms state-of-the-art baselines. However, we find that the reported performance gains are not statistically significant. We reimplemented the CONTENT model and the GRU benchmark in PyTorch and evaluated them on the same synthetic dataset. Our results show no significant difference in performance under the original setup. To extend the work, we introduced a grid search for hyperparameter tuning. With this optimization, CONTENT outperforms the baseline, supporting the model's potential under improved training conditions ($p < 0.0001$).

Link to video - Google Drive
Link to Public Github Repo - Github

II. Introduction

Hospital readmissions have been a substantial challenge for hospitals due to their frequent occurrence, high costs, and negative impact on patient health. In the United States, for example, about 17.6% of hospital-admitted patients are readmitted within 30 days, making up about $17.9 billion in annual Medicare spending—with 76% of these readmissions being avoidable. Traditional approaches to predict readmission typically use manually engineered features from electronic health records (EHRs) and simple machine learning models. However, these methods struggle to capture the complex, heterogeneous, and dynamic nature of medical data.

The paper, "Readmission prediction via deep contextual embedding of clinical concepts," introduces CONTENT, a deep learning model designed to predict 30-day hospital readmission for Congestive Heart Failure (CHF) patients (Xiao et al., 2018). CONTENT combines Gated Recurring Unit (GRU) and topic modeling on EHRs, using both short-term medical events and long-term history of patient data to create patient similarity representations that help improve prediction accuracy. This has allowed healthcare providers to better identify high-risk patients and find interventions to reduce unnecessary readmissions.

CONTENT was evaluated using real-world EHR data from a cohort of 5,393 CHF patients, including information on diseases, lab tests, and medication codes. The model had higher predictive accuracy (PR-AUC of 0.3894±0.0153) compared to other state-of-the-art models (best baseline PR-AUC was 0.3862±0.0136). This can help hospitals understand underlying phenotypes that lead to CHF readmission and prepare for these events accordingly. As a result patients can receive better care and hospitals will have lower costs with better allocated resources, potentially saving lives.

III. Scope of Reproducibility

Our goal was not only to reproduce the readmission prediction performance of the CONTENT model, but also to explore potential improvements in the model's implementation. Specifically, we tested the following hypotheses:

- **Model Reimplementation and Performance Validation:** The original CONTENT implementation used Theano, an outdated and discontinued framework. We tested if translating the model to PyTorch would still have the claimed performance improvement (PR-AUC of 0.3894 ± 0.0153) over the next-best baseline GRU model (PR-AUC was 0.3862 ± 0.0136).
  - Reimplement the CONTENT architecture in PyTorch
  - Reimplement the GRU architecture in PyTorch
  - Compare their performance to the results reported in Xiao *et al.* (2018)
- **Improving Hyperparameter Optimization:** The authors mention optimizing hyperparameters on validation data but do not clearly specify their method. We tested if we could further improve model performance with a custom grid search.
  - Create a grid search to explore hyperparameter combinations
  - Compare the CONTENT model's ROC-AUC obtained using our custom grid search against the parameters and performance reported in Xiao *et al.* (2018)
- **Assessment of Statistical Significance in Original Results:** In reviewing the original results table presented in Xiao *et al.* (2018), we noticed that the reported improvements were potentially not statistically significant. The paper does not include any statistical tests or measures to support the claim of superior performance.
  - Run 30 independent trials (up from 10) of the CONTENT and GRU models to increase statistical power
  - Perform statistical testing on the differences in performance

IV. Methodology

Dataset Descriptions

The authors use a real and a synthetic dataset. The real dataset includes EHR of 5,393 congestive heart failure (CHF) patients. Input features are binary encoded disease, lab test, and medication codes indicating presence/absence per visit. Each visit is treated as a set of simultaneous events represented by a binary vector. The authors did not publish this dataset (likely due to confidentiality concerns), so we experimented with the published synthetic dataset (found here) created to reproduce the claimed results.

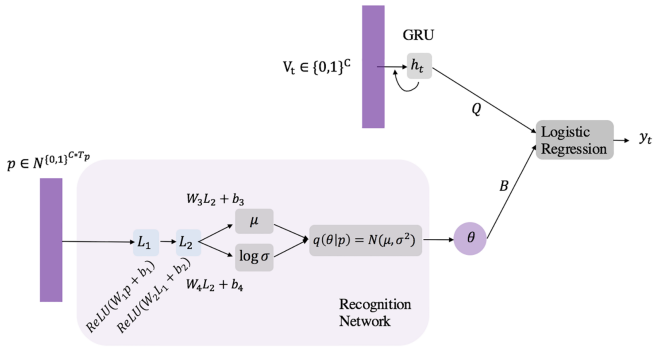

Fig. 1: CONTENT Model Architecture (Xiao et al., 2018). This illustrates the two-branch, hybrid CONTENT model. This image comes from Figure 2 in their original paper.

The synthetic dataset is simulated from a real world patient dataset. They randomly select 30% to 50% of visits from each original patient record, drop the rest, and shuffle patient indices. Each new patient record is then randomly paired with another, aligning the second record's event times to the first. The combined result is a single synthetic patient record that contains 3,000 patients and 239,936 visits, with an average of 80 visits and 228 clinical events per patient. There are 618 unique event codes. The label distribution is imbalanced, with 54,427 occurrences of readmitted patients (29.34%) and 185,509 non-readmitted (70.64%).

For data processing, the authors apply three steps: 1) they keep only inpatient-hospital visits, flagging each visit 1 if the next inpatient admission occurs within 30 days, else 0; 2) they remove overly frequent "stop-word" medical events, and map the remaining words to unique indices; 3) for every patient they create a nested list of indices (representing medical events) that occurred in each visit (i.e., docs[i] = [[medevent_indices_day1], [medevent_indices_day2], ...]) , allowing for the model to pick up on temporal patterns when predicting readmission at each time step. To match the original setup, we then split the dataset into 2,000 patients for training, 500 for validation, and 500 for testing.

Model Description

The CONTENT model can be understood as a language model. The entire patient cohort is a corpus: every patient is a document, each hospital visit is a paragraph, and the clinical codes recorded on a given day are the "words" (medical events) that appear together. In this analogy a patient's record becomes a sequence of paragraphs, while all codes within a single visit are treated as occurring simultaneously.

The model is composed of a "GRU branch" and a "Recognition Network branch" (Figure 1). A GRU processes the visit sequence and produces hidden states h=(h_1,...,h_t). The Recognition Network branch calculates a separate latent vector θ, which summarizes the patient's overall clinical context. The predicted 30-day readmission probabilities y=(y_1,...,y_N) are generated by combining the local information in ht with the global information in θ.

The baseline GRU model strips away this global-context branch and relies solely on the sequential GRU branch. Training therefore reduces to minimizing binary cross-entropy on the GRU's logits. Using this model as a baseline allows us to measure the difference in performance due to including long-term context vector θ in 30-day readmission prediction.

Below we describe each building block of both models in the order it is applied:

- **CONTENT architecture:**
  - **GRU Branch:**
    - Code-embedding layer (`Linear(490 → 100)`)—compresses a sparse 490-dimensional bag-of-words (medical events) into a dense 100-D vector for dimensionality reduction. Pre-trained with a custom Word2Vec matrix via Gensim.
    - Local-context GRU (`GRU(100 → 200)`)—inputs the sequence of code embeddings and outputs a 200-D hidden state h_t for each daily visit, capturing both short- and long-term medical events.
  - **Recognition Network Branch:**
    - Nonlinear Dense layers (`Linear(490 → 200) → ReLU → Linear(200 → 200) → ReLU`)—transforms the raw 490-D visit vector into a nonlinear 200-D representation.
    - μ and $\log \sigma^2$ layers (`Linear(200 → 50)` each)—map dense features to the mean and log-variance of a 50-D Gaussian for smooth latent-space embedding.
    - ThetaLayer (`Linear(100 → 50)`)—uses μ and $\log \sigma^2$ to sample a logit vector, then applies softmax to produce the patient topic vector θ.
    - Topic layer B (`Linear(490 → 50)`)—projects the current day's medical indices into the 50-D topic space; the elementwise product $B(x) \odot \theta$ is averaged to yield a scalar topic score, adding global context to the visit-level prediction.
    - Output layer (`Linear(200 → 1)`)—combines the GRU hidden state h_t with the topic score, then applies a sigmoid to compute the readmission probability.

- **GRU Architecture:**
  - Local-context GRU (`GRU(100 → 200)`)—processes daily code embeddings into a 200-D hidden state h_t, capturing short-term temporal patterns.
  - Output layer (`Linear(200 → 1)` + sigmoid)—maps h_t to a readmission probability. Unlike CONTENT, this model relies solely on the GRU's internal memory without a global context vector.

Training Objectives

CONTENT is optimized by maximizing the evidence lower bound (ELBO) loss function, which is essentially adding the binary cross-entropy (BCE) loss to the KL divergence regularizer (See "Training" Section for more details). The GRU baseline however optimizes only the pure BCE classification loss.

Fig. 2: Training and validation loss curves for CONTENT across 10 epochs. The model becomes overfit to the training data after the second epoch.

Method	Source	PR-AUC	ROC-AUC	ACC
CONTENT	Xiao et al., 2018	0.6011±0.0191	0.6886±0.0074	0.6934±0.0090
GRU	Xiao et al., 2018	0.5929±0.0100	0.6881±0.0048	0.6856±0.0082
CONTENT	Replication	0.6391±0.0025	0.7970±0.0014	0.8342±0.0021
CONTENT w/o Word2Vec	Replication	**0.6404±0.0022**	0.7985±0.0019	0.8355±0.0017
GRU	Replication	0.6401±0.0017	0.7979±0.0013	0.8356±0.0008

TABLE I: Averages and standard deviations of performance metrics over 30 trials (2 epochs each). The original results use the authors' hyperparameters, 10 trials, and 6 epochs. "CONTENT w/o Word2Vec" denotes ablation of pre-trained embeddings, with random initialization instead.

Both models share identical optimization settings (Adam with default β-values, gradient clipping, single-patient batches) and are trained for six epochs on validation ROC-AUC, with each experiment repeated ten times to yield robust mean ± SD performance estimates.

V. Training

Methods

CONTENT and GRU models are instantiated and optimized with Adam (learning rate 0.001, $\beta_1 = 0.9, \beta_2 = 0.999$) plus gradient clipping at 100. For each training batch we 1) zero the gradients, 2) forward-pass to compute predictions and for CONTENT the KL term, 3) compute masked binary cross-entropy loss (plus KL), 4) back-propagate, 5) clip gradients, and 6) step the optimizer. We record per-epoch training loss and save topic vectors.

CONTENT is trained with the objective function of maximizing the variational evidence lower bound (ELBO):

$$L = \mathrm{BCE}(p, y) + \frac{\mathrm{KL}\big(q(\theta) \,\|\, \mathcal{N}(0,1)\big)}{\mathrm{MaxVisitLength}}$$

The first term of the loss function is the binary cross-entropy between true and predicted labels, and the second is the KL divergence that regularizes the latent topics. No additional weighting was reported, so both terms contribute equally. The baseline GRU minimizes binary cross-entropy alone.

The original experiment trained each model for 6 epochs and repeated the process 10 times. We initially trained the original model for 10 epochs and observed that validation loss began increasing after epoch 2 (Figure 2), indicating overfitting. To address this, we trained each model for only 2 epochs and used the saved computation to run 30 trials per model. This provided more statistical power to detect differences between models and test the authors' claims. Increasing the number of trials reduced the standard error of our PR-AUC estimates and allowed us to leverage the central limit theorem for more reliable statistical comparison.

Training the CONTENT model also required training a separate Word2Vec model using the Gensim library to obtain 490 vectors of 100-D, each representing a medical event word found in the synthetic EHR data. The output Word2Vec matrix initializes a pre-trained weight matrix of the code-embedding layer in the GRU branch. Computing this was trivial, taking only 4.5 seconds. However, the computation may become more expensive on a larger real-world dataset or with more complex settings (e.g., larger vectors).

The network is not particularly computationally demanding to train, but without CUDA support, training can be time-consuming, especially with so many trials. To expedite the process, we utilized Google Colab's free T4 GPU. With the original paper's parameters, the average runtime per epoch was approximately 22 seconds for CONTENT and 18 seconds for the GRU. With the tuned parameters, the runtime was reduced to 10 seconds per epoch for CONTENT and 6 seconds for GRU, respectively. The total GPU runtime for both models was 46 minutes.

VI. Evaluation

The original paper compares model performance using PR-AUC, ROC-AUC, and accuracy. We report the same metrics, using PR-AUC as our primary evaluation metric due to the imbalanced nature of the dataset and the importance of correctly identifying readmission cases. While accuracy can sometimes overstate performance by favoring the majority class, we include it here to align with the evaluation approach used by the original authors. PR-AUC provides a better measure of performance in identifying at-risk patients.

VII. Results

Table 1 presents the means and standard deviations across 30 trials for both models on the test data, using the original experiment's hyperparameters.

CONTENT without Word2Vec initialization achieved the highest performance metrics by a small margin, but the numbers are too close to draw strong conclusions. Interestingly, CONTENT performed significantly better without Word2Vec initialization than with it based on a difference of means test ($p = 0.0367$), potentially indicating issues with our Word2Vec embeddings. This was the only statistically significant difference observed between the three configurations. Overall, we find no evidence that CONTENT meaningfully outperforms the GRU baseline.

Our performance metrics were universally much higher than those reported in the original paper on the synthetic data,

despite using the same model architecture. There are several possible reasons for this discrepancy:

1) **PyTorch vs. Theano:** We used PyTorch to implement both models, whereas the original paper used Theano. Differences in default behaviors or even numerical precision and stability could have subtly impacted training dynamics and final performance.

2) **Synthetic data differences:** We may be using a different version of the synthetic data than the authors. While the synthetic dataset was not included in their GitHub repository, they hosted a data file alongside their paper on plos.org, an academic journal website. It is possible that the data has been updated or modified since publication.

3) **Implementation variation:** Despite our efforts to faithfully replicate the model architecture, our reimplementation may differ from the original in ways that either improved performance (e.g., more stable training) or introduced issues (e.g., data leakage). Similarly, it is possible the original implementation contained inefficiencies or overlooked optimizations that limited its performance.

Next, we used LLMs to brainstorm potential extensions and ablations. To make the prompts more effective, we prompted it to generate multiple ideas based on its analysis of the original paper. This approach allowed us to explore several relevant options, which we could then test and select from. The LLM proposed ideas such as incorporating side information and handling missing data—approaches that were not feasible due to the lack of access to the real dataset.

It also suggested architectural changes, such as using alternative RNNs. We tested several modifications, including replacing the GRU with an LSTM, adding batch normalization and dropout, and experimenting with different activation functions. We tested these changes both individually and in combination, but all led to worse performance.

Another suggestion was hyperparameter tuning—the authors mention using the validation data to optimize model parameters but do not provide any details about how this was done. We found no evidence of a systematic approach to tuning in their paper or code, so to improve the replicability and performance of the model, we added functionality for a grid search. This extension also facilitates further model extensions or ablations by providing a clear and systematic method for hyperparameter optimization.

We choose to optimize hyperparameters to maximize PR-AUC, as the primary goal of the model is to identify the positive case of readmission, which is much rarer than the negative class. We found that performance did not change significantly with additional training epochs, so we opted to use only 1 epoch for the initial search. This choice allowed us to more efficiently explore hyperparameters beyond the depth of training, ensuring a broader search for optimal settings.

Since the original paper provides limited detail on how the Word2Vec embeddings were trained, we used the same Word2Vec embeddings we had already trained, with 100-dimensional vectors, and avoided retraining them or varying their parameters. We fixed the embedding size at 100, consistent with the authors' implementation, and focused our tuning efforts on each model's own hyperparameters, including batch size, hidden size, learning rate, and number of topics (relevant to CONTENT only).

Method	Tuning	PR-AUC	ROC-AUC	ACC
CONTENT	Xiao et al., 2018	0.6391±0.0025	0.7970±0.0014	0.8342±0.0021
CONTENT	Tuned (Grid)	**0.6464±0.0022**	0.8026±0.0014	0.8368±0.0013
CONTENT w/o Word2Vec	Xiao et al., 2018	0.6404±0.0022	0.7985±0.0019	0.8355±0.0017
CONTENT w/o Word2Vec	Tuned (Grid)	0.6406±0.0018	0.7990±0.0013	0.8358±0.0011
GRU	Xiao et al., 2018	0.6401±0.0017	0.7979±0.0013	0.8356±0.0008
GRU	Tuned (Grid)	0.6391±0.0025	0.797±0.0014	0.8342±0.0021

TABLE II: Averages and standard deviations of performance metrics over 30 trials (using tuned hyperparameters: 2 epochs per trial). The paper results use the authors' hyperparameters, 10 trials, and 6 epochs. "CONTENT w/o Word2Vec" denotes ablation of pre-trained embeddings, with random initialization instead.

Given the increased computational cost of a grid search, we limited the initial run to a single trial per configuration to keep training times reasonable. Based on the results of that run, we selected several top-performing configurations and re-tested them using five trials and two epochs, again evaluating on the validation set. The test set metrics are reported in Table 2. Specifically, we found that decreasing CONTENT's hidden size to 200 and increasing the number of topics to 150, along with decreasing the GRU's hidden size to 200 and increasing the learning rate to 0.005, yielded the best results.

With these optimized settings, full CONTENT has the highest performance metrics. On PR-AUC, it significantly outperforms both the GRU ($p < 0.0001$) and CONTENT without Word2Vec initialization ($p < 0.0001$). This suggests that CONTENT's original underperformance may have been due to suboptimal hyperparameter choices rather than the architecture itself.

VIII. Discussion

Certain omissions prevent exact replication. The authors do not include the pre-trained Word2Vec embeddings used to train their model (or the procedure to train them). Additionally, the real-world CHF dataset referenced in the paper is not available, so we could not replicate the original experiments that evaluated model performance on real EHR data.

Despite the omission of the real-world dataset, we were able to partially reproduce the experiment on the synthetic data, which is available. The paper and code provide enough detail to replicate the CONTENT model architecture. We trained new Word2Vec embeddings on the synthetic data, which, while structurally similar, may lack the nuance and variability of real-world text and fail to capture key patterns in the real-world data.

Regarding the claim that "the proposed model outperforms state-of-the-art methods," the reproducibility of the benchmark models is limited:

- **Word2Vec + LR:** The paper provides the Word2Vec embedding size (100) but does not specify the logistic regression implementation.
- **Med2Vec + LR:** No details are given on the Med2Vec model or pre-trained embeddings, nor on the logistic regression implementation.
- **GRU:** Parameters such as hidden size (200) and training settings (Adam optimizer, learning rate 0.001, batch size 1) may be inferred from the CONTENT model's GRU branch, but are not explicitly reported.
- **GRU + Word2Vec:** Information on the Word2Vec embeddings and their integration with the GRU model is limited.
- **RETAIN:** Only a citation of the RETAIN paper is provided, with no implementation details from the authors.

The GRU was the only model we felt reasonably confident replicating to match the authors' setup. While even that implementation lacked full transparency, it offered more reproducible detail than the others. Notably, this GRU baseline was the second-best performing model in the original paper (after the proposed model), making it a reasonable choice for comparison.

Even if the benchmark implementations were fully replicable, the performance gains reported by the authors are not statistically significant. On the real-world data, a difference of means test between CONTENT's PR-AUC (0.3984 ± 0.0153) and GRU's (0.3862 ± 0.0136) yields a p-value of 0.6271. Similarly, on the synthetic dataset, CONTENT achieves a PR-AUC of 0.6011 ± 0.0191 compared to GRU's 0.5929 ± 0.0100, with a corresponding p-value of 0.2447. The results from our initial experiment support the observation that, under the original hyperparameter setup, there is no statistically significant difference between CONTENT and the benchmark GRU model in terms of PR-AUC. However, after applying a grid search to find optimal hyperparameters, the full CONTENT model significantly outperformed both the GRU and the CONTENT variant using Word2Vec initialization. Notably, it achieved this while training in roughly half the time. In an extension of the original paper's results, we demonstrate the statistical significance of this improvement.

This suggests that the limited performance we initially observed may have been due to suboptimal hyperparameter settings in our reimplementation. The original paper's configuration may not have transferred cleanly to our version of the model or dataset. At the same time, it's also possible that the original implementation itself was not fully optimized.

Notably, performance gains in our extension occurred only in the full CONTENT model using Word2Vec. In our optimized configuration, the number of topics increased from 50 to 150, possibly compensating for issues in how our Word2Vec embeddings were trained or integrated. This may suggest that the synergy between embeddings and topic modeling is sensitive to training conditions and must be carefully tuned.

In summary, our findings partially support the authors' claims. While we were able to replicate the CONTENT architecture and observed superior performance after optimizing hyperparameters, the original setup did not yield statistically significant improvements over the GRU baseline. Additionally, the authors' own reported results lacked statistical significance, further weakening the strength of the claim. These findings suggest that CONTENT has potential under the right conditions, but its performance is highly sensitive to implementation details and parameter tuning.

Improving reproducibility would require clearer documentation of benchmark implementations, hyperparameter selection, access to or guidance on training Word2Vec embeddings, and greater transparency in the original experimental setup. Without these, the claim that CONTENT consistently outperforms state-of-the-art models remains only partially substantiated.

IX. Authors' Contributions

Christopher Hynes

- Implemented data pre-processing and evaluation functionalities.
- Implemented the benchmark GRU model.
- Added support for GRU in all existing functionalities.
- Trained Word2Vec embeddings from synthetic EHR data.
- Conducted extensive hyper-parameter tuning (topic count, hidden size, learning rate, etc.) using Felipe's grid search.
- Ran statistical checks on results to choose optimal hyperparameters for CONTENT and GRU.
- Tested architectural variations—e.g., LSTM layers and LeakyReLU activations.
- Completed 30-trial replication runs for all models with the tuned hyperparameters.
- Conducted statistical testing on the original and replicated results to verify performance differences.
- Wrote Abstract, Training, Evaluation, Results, Discussion, and maintained the running project task list.

Felipe Oliveira

- Implemented the CONTENT model.
- Implemented the training loop.
- Established the GitHub repository, conda environment YAML, and parallel notebooks for local and Colab execution, ensuring smooth reproducibility.
- Implemented improved Gensim Word2Vec training
- Added a PR-AUC–driven grid-search module to tune hyperparameters using the validation set.
- Automated training/validation loss logging, generated over-fitting curves.
- Computed label-distribution statistics.
- Completed 30-trial replication runs for all models with the paper parameters.
- Wrote the Introduction, Scope of Reproducibility, and Methodology.
- Moved the report into the LaTeX template.

REFERENCES

[1] Xiao, C., Ma, T., Dieng, A. B., Blei, D. M., & Wang, F. (2018, April 9). *Readmission prediction via deep contextual embedding of clinical concepts.* PloS one. https://pmc.ncbi.nlm.nih.gov/articles/PMC5890980/

Appendix

DATA PROCESSING

First prompt:

I have a healthcare dataset in a .txt format that includes patient visit data. I also have vocab and stop word files. Can you help me write code to:

1. Unzip and load the data
2. Build a vocabulary excluding rare words and stop words
3. Convert the raw visit sequences into numerical tensors for a GRU-based model
4. Save the processed data into train/validation/test .pkl files

First output:

```
import os

import pickle

import zipfile

from collections import Counter, defaultdict

import torch


# 1. Unzip and load data

def unzip_data(zip_path, extract_to):

    with zipfile.ZipFile(zip_path, 'r') as zip_ref:

        zip_ref.extractall(extract_to)


def load_txt_file(filepath):

    with open(filepath, 'r') as f:

        return [line.strip() for line in f]


# 2. Build vocabulary (excluding rare words + stop words)

def build_vocab(sequences, stop_words, min_freq=5):
```

```
    word_counts = Counter()
    for seq in sequences:
        tokens = seq.split()
        word_counts.update(tokens)

    vocab = {
        word for word, count in word_counts.items()
        if count >= min_freq and word not in stop_words
    }

    word2idx = {word: idx + 1 for idx, word in enumerate(sorted(vocab))}  # idx 0 = PAD
    word2idx['<PAD>'] = 0
    return word2idx

# 3. Convert sequences to tensors
def convert_to_tensor(sequences, word2idx, max_len=None):
    tensor_data = []
    for seq in sequences:
        tokens = seq.split()
        indices = [word2idx.get(token, 0) for token in tokens]  # Unknowns → PAD (or add <UNK> later)
        if max_len:
            indices = indices[:max_len]
            indices += [0] * (max_len - len(indices))  # Pad
        tensor_data.append(torch.tensor(indices, dtype=torch.long))
    return tensor_data
```

```
# 4. Save data
def save_splits(data, split_ratio=(0.7, 0.15, 0.15), output_dir='processed'):
    os.makedirs(output_dir, exist_ok=True)
    total = len(data)
    train_end = int(split_ratio[0] * total)
    val_end = train_end + int(split_ratio[1] * total)

    splits = {
        'train.pkl': data[:train_end],
        'val.pkl': data[train_end:val_end],
        'test.pkl': data[val_end:]
    }

    for fname, d in splits.items():
        with open(os.path.join(output_dir, fname), 'wb') as f:
            pickle.dump(d, f)
```

Discussion:

The LLM's (ChatGPT 4o) initial response provided a good starting point, but ultimately wasn't very helpful in processing the data. While the general steps aligned with what we expected from the prompt, the response didn't account for the complexity of our data structure or the specifics of the CONTENT model architecture.

This was due to a gap in our initial prompt—we didn't mention the structure of the data or clarify how we expected the raw visit data to be transformed. We were aiming to replicate the authors' processing steps, which required a very specific approach. It took us 14 additional prompts, along with referencing the authors' original code, to understand and follow their logic closely and arrive at the final working version.

Model Implementation

First Prompt:

I am trying to replicate the CONTENT model for EHR data. It uses a GRU followed by a variational topic layer. Can you help implement this model in PyTorch, including KL divergence regularization?

First Output:

Assumptions:

- Input shape: [batch_size, sequence_length, vocab_size] (multi-hot vectors per timestep).
- Output: Binary classification per patient.
- Topic modeling is done using a Gaussian latent space with a softmax to get topic proportions (like a VAE).

Model Architecture Overview:

- GRU for visit-level temporal representation.
- Mean and log-variance layers from final GRU hidden state.
- Reparameterization trick to sample z ~ N(mu, sigma).
- Softmax on z to get topic distribution theta.
- Linear layer to map theta to output prediction.
- KL divergence term computed during forward pass.

PyTorch Code:

```
import torch

import torch.nn as nn

import torch.nn.functional as F


class ThetaLayer(nn.Module):

    def __init__(self, hidden_dim, topic_dim):

        super().__init__()

        self.fc_mu = nn.Linear(hidden_dim, topic_dim)

        self.fc_logsigma = nn.Linear(hidden_dim, topic_dim)

        self.kl_term = 0.0  # KL divergence (used during training)


    def forward(self, h):

        mu = self.fc_mu(h)
```

```
        log_sigma = self.fc_logsigma(h)
        eps = torch.randn_like(mu)
        z = mu + torch.exp(0.5 * log_sigma) * eps
        theta = F.softmax(z, dim=1)

        # KL divergence between N(mu, sigma) and N(0,1)
        kl = -0.5 * torch.sum(1 + log_sigma - mu.pow(2) - log_sigma.exp(), dim=1)
        self.kl_term = kl.mean()
        return theta

class ContentModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, topic_dim):
        super().__init__()
        self.rnn = nn.GRU(input_dim, hidden_dim, batch_first=True)
        self.theta_layer = ThetaLayer(hidden_dim, topic_dim)
        self.output_layer = nn.Linear(topic_dim, 1)

    def forward(self, x, mask=None):
        _, h_n = self.rnn(x)                # h_n: [1, B, H]
        h_n = h_n.squeeze(0)                # h_n: [B, H]
        theta = self.theta_layer(h_n)       # [B, T]
        logits = self.output_layer(theta)   # [B, 1]
        preds = torch.sigmoid(logits).squeeze(1)  # [B]
        return preds, h_n, theta
```

```
    @property
    def kl_term(self):
        return self.theta_layer.kl_term
```

Loss Function:

```
def compute_loss(preds, y_true, mask, kl_term):
    bce_elem = F.binary_cross_entropy(preds, y_true, reduction='none')
    bce = (bce_elem * mask).sum() / mask.sum()
    return bce + kl_term
```

Let me know your: input_dim (i.e. size of your vocabulary / concept space), hidden_dim for GRU, and topic_dim (number of topics), and I can plug those in or modify things like dropout, embedding layers, or patient-level pooling if needed.

Discussion:

The first LLM answer was somewhat helpful because it included the main architectural elements of the CONTENT model, such as KL divergence regularization, a variational topic layer with the reparameterization method, and a GRU encoder. But because my question was unstructured and primarily consisted of code fragments, the LLM mistook it as a continuation job and made small formatting and logical mistakes such as redundant return statements and unclear inputs. A more detailed prompt outlining the model structure and intended implementation from the original paper would have guided the LLM to provide a better output. Nevertheless, the LLM helped me create a functional baseline, which I then improved over 12 more prompts.

TRAINING LOOP

First Prompt:

Can you write a training loop in PyTorch that logs average loss per epoch and evaluates the model using PR-AUC on a validation set?

First Output:

Sure! Here's a PyTorch training loop that logs average loss per epoch and evaluates the model using Precision-Recall AUC (PR-AUC) on a validation set:

```
import torch

import torch.nn.functional as F

from sklearn.metrics import average_precision_score

from tqdm import tqdm


def train(model, train_loader, val_loader, optimizer, config):

    best_val_pr_auc = 0

    for epoch in range(config.num_epochs):

        model.train()

        total_loss = 0.0

        total_examples = 0


        for x_batch, y_batch, mask_batch in tqdm(train_loader, desc=f"Epoch
{epoch+1}/{config.num_epochs}"):

            x_batch = x_batch.to(config.device)

            y_batch = y_batch.to(config.device)

            mask_batch = mask_batch.to(config.device)


            optimizer.zero_grad()

            preds, *_ = model(x_batch, mask_batch)  # adjust based on your model's output

            bce_elem = F.binary_cross_entropy(preds, y_batch, reduction="none")

            bce = (bce_elem * mask_batch).sum() / mask_batch.sum()

            loss = bce + getattr(model, "kl_term", 0.0)  # if model has KL term

            loss.backward()

            torch.nn.utils.clip_grad_norm_(model.parameters(), 100)
```

```
            optimizer.step()

            total_loss += loss.item() * mask_batch.sum().item()
            total_examples += mask_batch.sum().item()

        avg_loss = total_loss / total_examples
        val_pr_auc = evaluate_pr_auc(model, val_loader, config)

        print(f"Epoch {epoch+1}: Avg Loss = {avg_loss:.4f}, Val PR-AUC = {val_pr_auc:.4f}")

        if val_pr_auc > best_val_pr_auc:
            best_val_pr_auc = val_pr_auc
            # Optional: save model checkpoint

    print(f"Best Validation PR-AUC: {best_val_pr_auc:.4f}")
    return best_val_pr_auc


def evaluate_pr_auc(model, loader, config):
    model.eval()
    all_true, all_pred = [], []

    with torch.no_grad():
        for x_batch, y_batch, mask_batch in loader:
            x_batch = x_batch.to(config.device)
            y_batch = y_batch.to(config.device)
```

```
        mask_batch = mask_batch.to(config.device)


        preds, *_ = model(x_batch, mask_batch)
        all_true.append(y_batch.cpu().numpy())
        all_pred.append(preds.cpu().numpy())


    true = torch.cat([torch.tensor(t) for t in all_true]).numpy().flatten()
    pred = torch.cat([torch.tensor(p) for p in all_pred]).numpy().flatten()


    return average_precision_score(true, pred)
```

Assumptions:

- train_loader and val_loader yield (x_batch, y_batch, mask_batch)
- Model's output is (preds, ...), and preds are probabilities in [0,1]
- config includes num_epochs and device

Let me know if you'd like it customized for a specific model structure (e.g. CONTENT model, RNN, etc.) or if you're using weighted BCE, multi-label output, etc.

Discussion:

A general-purpose PyTorch training loop that calculated PR-AUC on the validation set and monitored average loss per epoch was given by the first LLM response. Although it made sense and served as a good foundation, it ignored the unique architecture and output format of the CONTENT model, which incorporates KL divergence and produces latent vectors like theta and h_n. Additionally, it didn't cover how to gather intermediate vectors (such as theta) for each epoch or how to use m_batch to mask out padding. The output wasn't enough for specialized models, but it was useful for scaffolding the high-level training structure. Later I had to make changes to the loop in order to handle model-specific training (CONTENT vs GRU), add KL loss, and save latent representations for later use. This took 13 more prompts to achieve.

EVALUATION

First prompt:

I'm building a PyTorch model for hospital readmission prediction based on sequences of patient visits. I now need help evaluating my model on test data. Can you help me write functions to:

1. *Run my model on a test dataset and collect predictions*
2. *Calculate relevant metrics for a binary classification task like this*

My model outputs probabilities between 0 and 1 for each element in a sequence, and I'm only interested in the non-padded elements. I'm using DataLoader that gives me x, y, and a mask tensor.

First output:

Absolutely, I can help you with that! Based on your setup:

- You're working with sequences (likely padded).
- Your model outputs probabilities per time step.
- You're using a mask tensor to identify which elements are not padding.
- It's a binary classification task.

Here's a clean and modular way to:

1. Run inference on your test data
2. Collect predictions and targets (filtered by the mask)
3. Calculate common metrics: accuracy, precision, recall, F1 score, ROC AUC

Step 1: Run Model and Collect Predictions

```
import torch


def evaluate_model(model, dataloader, device):
    model.eval()
    all_preds = []
    all_probs = []
    all_targets = []

    with torch.no_grad():
```

```
    for x, y, mask in dataloader:
        x, y, mask = x.to(device), y.to(device), mask.to(device)
        probs = model(x)  # output: [batch, seq_len], probabilities

        # Filter out padded elements
        probs_masked = probs[mask.bool()]
        y_masked = y[mask.bool()]
        preds_masked = (probs_masked >= 0.5).long()

        all_probs.append(probs_masked.cpu())
        all_preds.append(preds_masked.cpu())
        all_targets.append(y_masked.cpu())

    # Concatenate all batches
    all_probs = torch.cat(all_probs)
    all_preds = torch.cat(all_preds)
    all_targets = torch.cat(all_targets)
    return all_probs.numpy(), all_preds.numpy(), all_targets.numpy()
```

Step 2: Compute Metrics

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score

def compute_metrics(probs, preds, targets):
    metrics = {
        'Accuracy': accuracy_score(targets, preds),
        'Precision': precision_score(targets, preds),
        'Recall': recall_score(targets, preds),
```

```
    'F1 Score': f1_score(targets, preds),
    'ROC AUC': roc_auc_score(targets, probs)
  }
  return metrics
```

Example Usage

```
# Assuming your model, test_loader, and device are already set
probs, preds, targets = evaluate_model(model, test_loader, device)
metrics = compute_metrics(probs, preds, targets)
for name, value in metrics.items():
  print(f"{name}: {value:.4f}")
```

Let me know if you want to support:

- sequence-level metrics (e.g., did this patient ever get readmitted?)
- saving predictions for later analysis
- thresholds other than 0.5
- handling multi-label/multi-class cases

Happy to help tweak it!

Discussion:

It took a total of five prompts to get the evaluation working as expected. The first prompt already got quite a bit right–the output was relevant and helpful. It correctly handled masking, predicted probabilities, applied a threshold to convert them into binary classifications, and computed most of the performance metrics we were interested in. This makes sense: once you have the model's predictions, using them for evaluation is relatively straightforward, so it's not surprising that an LLM might handle that part well.

However, we left out some important details in the initial prompt. As a result, it didn't include PR-AUC as a metric, and it also didn't compute the loss, which we wanted to track throughout evaluation. Additionally, we overlooked the need for model-specific logic–the prompt didn't differentiate between CONTENT and GRU models. This became an issue later: we initially got the evaluation working for CONTENT, but had to revise the code afterward to add support for GRU.