Approaches to Scoring Clinical Patient Notes

Lejana Dietrich
faculty IWI
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
ailaboratory2324@gmail.com

Yi-Hsuan Hung
faculty IWI
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
husanhung036@gmail.com

Leonard Paal
faculty IWI
Karlsruhe University of Applied Sciences
Karlsruhe, Germany
leompaal@gmail.com

***Abstract*—We present single-model (as opposed to ensemble model) NLP solutions to scoring clinical patient notes and compare the results. We use an LSTM as well as BERT and DeBERTa models to determine which physical and diagnostic features were found by medical students evaluating patients and where in their notes they can be found. Judging by F1 scores over the predicted ranges and ground truth data for each feature, we find that DeBERTa based implementations fine tuned on all layers give the most satisfactory results even when utilizing limited training resources.**

Keywords—machine learning, NLP, LSTM, BERT, DeBERTa

I. Introduction

This final report describes an effort to evaluate different approaches to the NBME — Score Clinical Patient Notes Kaggle competition [1]. While the winning solutions all consist of ensemble models combining the results of around 5 singular models, we aim to evaluate the performance of singular models. Findings could in turn be used to optimize ensembles.

Furthermore, the problem, competition goal data structure is described in a way which makes introductions into this topic easier. We implement a LSTM architecture using BERT embeddings, a Google BERT-based model and a Microsoft DeBERTa-based model, the latter of which we fine-tune on just some layers for efficiency.

Our results can be found on Github: https://github.com/Tsunetsuki/AILab2023/tree/main/Project.

II. Problem Statement

A. Goal

This challenge was developed with the goal of reducing personnel cost for training medical students in mind [1]. Evaluations of patient notes made by students are still done manually by trained doctors who need to search them for all the physical features and symptoms the students should be able to detect. To facilitate machine learning-based scoring necessitates automatically determining which features were detected by students as well as their location in the clinical patient notes so results can be controlled manually. Models should achieve maximum possible F1 scores, meaning character-wise matches of feature locations predicted in a patient note and the ground truth provided for that patient note.

Note that the goal is not for the models to learn to extract patient features like their age or symptoms from a clinical evaluation but to evaluate how well and where medical students documented such features in their notes.

B. Data

Ground truth consists of features which can be found in each patient note and their locations. Multiple features exist in each patient note and identical features can exist in multiple patient notes. There are clinical notes for 10 different patients by multiple students, more specifically portions of patient histories written by the medical students.

Ground truth data points which are included as training data are unique. They each denote one specific feature in one patient note and the character ranges where these can be found. There can be none, one or multiple ranges of characters for each unique training data point. Training annotations exist only for 1000 of the patient notes, 100 patient notes per case. There are 14301 training data points, an average of 14 per annotated patient note, as well as a list of all 901 different case-specific features and about 40000 patient notes in total, each in a separate file.

There is no annotated test data so we used 20% of the training dataset as test data. Additionally, we identified some inaccuracies in the dataset, such as the spelling error in "Last-Pap-smear-I-year-ago" within the feature_text, which should be corrected to "Last-Pap-smear-1-year-ago." There were also instances of incorrect annotations, for example, where a feature in a patient's history should read "father heart attack," but in the dataset, it was annotated as "ather heart attack." Consequently, we corrected a total of 42 instances of annotation errors in the dataset, while also updating the indices of these datasets with the correct features.

III. Related Work

Both in the official competition and in our own experiments, the best results were achieved by utilizing Google's BERT model or derivations of this model like DeBERTa and fine-tuning them. Therefore, we include short descriptions of the way these models work and why this makes them appropriate for our task:

A. BERT

BERT (Bidirectional Encoder Representations from Transformers) is a transformer type NLP model [2]. Unlike popular generator models, it is trained (deeply) bi-directionally, meaning it can determine context both before and after a specific token in an equally accurate way. The specific training methods are trained masked language modeling (MLM) and next sentence prediction (NSP). Training data was extracted from the Toronto Book Corpus and Wikipedia. There are pre-trained versions in multiple sizes as well as case-sensitive and uncased versions. Its base size consists of 12 transformer block layers. It can operate on multiple levels, from token level to sentence pair level.

BERT can't be used to generate text, but is highly performative for other NLP tasks like classification, language inference and question answering from given texts. Its developers note that just one additional output layer is needed to create state-of-the-art implementations for these tasks, making task-specific architecture modifications unnecessary.

Furthermore, its contextual embeddings can be extracted and used in other models [3].

B. DeBERTa

DeBERTa is a newer model developed by Microsoft in 2020 and derived from BERT, which is widely considered state-of-the-art for non-generative NLP tasks today [4, 5]. Its most significant difference lies in its disentangled attention mechanism. While BERT adds absolute positions of tokens to their context embedding vectors, DeBERTa keeps these properties separate, creating 2 vector embeddings. From these disentangled attention can be calculated in a more complex but also more accurate way than is the case for BERT.

While version 1 was a BPE (Byte Pair Encoding)–based, versions 2 and 3 use a SentencePiece tokenizer which in addition to BPT tokenizer functionality can more easily handle different types of token separators (for example "_"). Like BERT, its base size consists of 12 layers in which multiple functionalities are nested. There are multiple versions for each iteration which were pre-trained on different vocabularies and sometimes also vary in their number of parameters.

This model can be used for the same tasks as BERT and can be expected to perform better, though the more complicated structure makes training more expensive.

Other models like DistillBERT or RoBERTa were not chosen for several reasons. Because available hardware was performant enough there were no advantages to using the smaller and slightly less accurate DistillBERT model. As for another popular model named RoBERTa, its overall performance is worse than DeBERTas for existing public solutions to this challenge while not giving significant advantages like BERTs faster training and better documentation.

IV. Implementation

We used Binary Cross Entropy With Logits Loss (which corresponds to BCE loss with a sigmoid function applied beforehand) for training and evaluation, as well as an F1 score only for evaluation. We exclusively used supervised learning.

A. LSTM

The simplest model we tested to achieve the task is a vanilla LSTM [9] model (more precisely, a network with two LSTM layers) using BERT embeddings. This model's process is as follows.

1) Data Preprocessing

a) The feature and the patient note are tokenized and embedded with BERT.

b) Each embedded token is assigned a "relevance" score for training, expressing whether the token is in the target range.

2) Model. (see also fig. 2 in appendix)

a) The model takes the embedded features and patient notes.

b) The feature vector is run through the first LSTM layer, which outputs a vector with the length of one single patient history token.

c) The compressed feature vector is appended to each token in the patient note sequence.

d) The second LSTM takes as input the concatenation of patient note tokens + feature vector. It outputs a single vector.

e) The vector is run through a hidden layer and finally through a sigmoid to create a floating point output between 0 and 1 for each patient history token. Its output should be the token's relevancy score.

3) Criterion and Optimizer

BCELossWithLogits loss is used. This function creates a binary cross entropy, but data with a positive label is weighted more strongly than negative label data. This is important, as most of the tokens in the patient notes will not be relevant to the feature description, thus making the dataset unbalanced.

A positive weight of 44 was chosen since there is approximately a 44:1 ratio between negatively and positively labeled tokens in the training data. This is to prevent the model from taking the "safe" strategy of always outputting zero.

The PyTorch SGD optimizer was used.

B. BERT and DeBERTa

We use very similar approaches for training and evaluation BERT [2] and DeBERTa[4]-based models.

1) Data

The objective of this chapter is to delineate the process of transforming patient history and annotation using the DeBERTa model through a tokenizer. We will delve into the handling of Input and Label within the DeBERTa model, encompassing aspects such as dataset structure, how the tokenization works, transformation steps, and pertinent considerations. Initially, the requisite contents for Input and Label are outlined as follows:

a) Input Dataset: Patient history and the rubric of features (or key concepts) for each clinical case.

b) Label Dataset: Patient history and the index position of the annotations to be predicted within the patient history.

Prior to commencing the tokenizer for Input and Label, it is imperative to determine the maximum length employed by the model. BERT, and other Transformer-based models, necessitate a fixed maximum sequence length during input processing due to the self-attention mechanism of the Transformer model. This ensures consistent attention matrix calculations for both training and inference phases. The calculation of the maximum length during data transformation involves the following steps:

c) Once the patient history undergoes tokenization, the token IDs are obtained through input_ids frome tokenize results, resulting in a sequence of Token IDs representing the patient history data. input_ids is the index of the tokens in the tokenizer's vocabulary. It maps each token to a unique ID in the vocabulary.

d) Subsequently, for each tokenized patient history, the length of the sequence is determined, and the maximum length among these patient histories is obtained. This includes both the tokenized patient history and the tokenized feature as they will be provided as a combined input for training the model.

e) Finally, the maximum length is computed by summing the maximum length max_len among the tokenized patient history and feature sequences.

Regarding the processing steps for the input dataset, incorporating both patient history and annotations into the model involves simultaneous tokenization of these two sources of information. This process allows the adding of special tokens, such as [CLS] for denoting the beginning, [SEP] for indicating sentence breaks, and [PAD] for padding sections with no values when setting the max_length parameter. Ultimately, the tokenized output, including input_ids, token_type_ids, and attention_mask, serves as the input for the DeBERTa model.

In the label processing steps for our machine learning model, it is essential to incorporate both patient history and feature location. Feature location denotes the index position of a particular feature within the patient's historical data. To ensure a standardized input sequence for the model, we adopt a binary {0,1} encoding scheme with a fixed length. In this scheme, setting a feature token to 1 indicates the desired prediction for the corresponding part of the model, while regular tokens are set to 0. Padding or special tokens are designated as -1. Consequently, we utilize the results of tokenization, specifically the input_ids and offset_mapping. The primary purpose of offset_mapping is to identify the original index of tokens in the sentence after tokenization. This knowledge is crucial for labeling which tokens correspond to annotations. If a token contains the content of the feature, it is labeled as 1. To illustrate the labeling process, consider the example with the text "I have a headache," a fixed length of 10, and the feature being "headache."

a) Tokenizing the sentence "I have a headache." yields the following results. It is important to note that each value in offset_mapping contains the start and end index positions of the corresponding token in the sentence, with the end position not inclusive. The tokenized result will be like the following: input_ids: [1, 100, 33, 10, 19344, 4, 2, 0, 0, 0]; offset_mapping: [(0, 0), (0, 1), (1, 6), (6, 8), (8, 17), (17, 18), (0, 0), (0, 0), (0, 0), (0, 0)]; decode: ["[CLS]", "I", " have", " a", " headache", ".", "[SEP]", "[PAD]", "[PAD]", "[PAD]"].

b) The feature "headache" is identified in the sentence at positions (9, 18), calculated by adding the index 9, where "headache" starts, to the length of "headache," which is 18. Note that index 18 does not include the word "headache." Create an array named 'result' with zeros, based on the length of input_ids: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0].

c) Mark special tokens in 'result' as -1 to inform the model that these are not part of the learning content, resulting in [-1, 0, 0, 0, 0, 0, -1, -1, -1, -1]. Iterate through each value in offset_mapping to identify which token positions contain the feature (9, 18). While searching for token positions, account for the possibility that the feature spans multiple tokens and determine the starting and ending tokens. Mark special tokens in 'result' as -1 to inform the model that these are not part of the learning content, resulting in [-1, 0, 0, 0, 0, 0, -1, -1, -1, -1]. Iterate through each value in offset_mapping to identify which token positions contain the feature (9, 18). While searching for token positions, account for the possibility that the feature spans multiple tokens and determine the starting and ending tokens. Find the index of the last token that includes the feature. When the feature end index is less than or equal to the end index in offset_mapping's value, this index represents the last position of the feature in tokens and should be marked as 1.

d) Collecting the feature information, we identify that the feature's position in 'result' is (8, 17), corresponding to index 4. The final 'result' is [-1, 0, 0, 0, 1, 0, -1, -1, -1, -1], which can be converted into a tensor.

2) Model

Firstly, the DeBERTa model architecture is based on Microsoft's DeBERTa (Decoding-enhanced BERT with Disentangled Attention) framework, which enhances BERT by introducing a disentangled attention mechanism to improve the model's ability to capture relationships between different words. This results in significant performance improvements in natural language processing tasks. We used pre-trained DeBERTa to leverage extensive pre-training on vast datasets for better natural language understanding. It features a fully connected layer that processes the hidden states from DeBERTa, reducing the dimensionality from 768 to 1.

DeBERTa and BERT both models can be used to predict the probability of whether each character in a student-note text is part of a given feature. We add a single linear layer with output size 1 after the last BERT/DeBERTa layer. We used the newest base-sized uncased versions since larger vocabularies did not greatly improve results among Kaggle competitors for this challenge and since the patient notes are not cased consistently.

The model incorporates a dropout mechanism (set at a probability of 0.2) for regularization to reduce overfitting and employs meticulous weight initialization for different layers. For the fully connected and embedding layers, weights are initialized following a normal distribution with a mean of 0.0 and a standard deviation defined by the model's configuration initializer range. For layer normalization, a specific value initialization is employed: biases are set to zero, and weights are filled with a value of 1.0. This particular initialization in layer normalization ensures that the initial output of the layer is normalized, avoiding the extreme variations in the early stages of training, which can be detrimental to the learning process.

Additionally, it defines a feature extraction method, extracting the last layer of hidden states from the DeBERTa model and passing these features through the dropout layer, then to the fully connected layer for the final output.

3) Criterion and Optimizer

As mentioned, BCELossWithLogits loss is used here as well, with the same reasoning as for the LSTM evaluation. We use the AdamW optimizer with a weight decay of 0.1 to avoid overfitting and also rely on this optimizer for an adaptable learning rate.

4) Prediction

Characters with probabilities over 0.5 are then marked in a binary array as part of a feature and ranges are constructed by including consecutive characters with an index that denotes value 1 in this binary array. The ranges constructed like this are continuous (meaning there aren't any instances of singular characters within a word being left out) without additional operations because internally, BERT and DeBERTa work token-wise unless otherwise specified.

5) Training

The BERT model, which can be trained faster but doesn't yield significant improvements in F1 scores for several epochs, we train for 10 epochs. We use a starting learning rate of 2e-6 for the encoder to make this model less prone to overfitting after around epoch 5. We experimented with fine-tuning only some layers, but the accuracy was not satisfactory and training speed did not improve by more than a factor of 2, which matches the experience other developers had with this model [6].

As for the DeBERTa model, we train for 5 epochs and use a starting learning rate of 2e-5 for the encoder stack. We experimented with training only the linear layer after the models themselves, though since this layer only reduces output dimensions to 1, this was not very promising. We also experimented with training 3 layers in total, our linear layer and the last two layers of the models (which in practice consist of multiple layers in an attention-intermediate-output scheme). We also fine-tuned the whole model once in order to compare the accuracy. These experiments otherwise used the same parameters.

A structure inspired by k-fold validation is used for training the BERT-based implementation in order to use all the available training data and still get test results without the original kaggle test data. For this the data is split into fifths, each of five folds training on four fifths of the data and testing results on the remaining fifth. This also let us test for variance in performance based on selection of the test set. For the DeBERTa-based approaches, one fifth of the data is simply designated as test data, since this model would otherwise take excessive time to train and proved to perform in a more stable manner, no matter which data was chosen for testing, during early test runs.

These models were trained with a single NVIDIA GTX2070 GPU. The training function automatically saves the weights which yield the best test results which are measured after each completed epoch. A seed was used for repeatability.

V. Evaluation

The average loss was calculated based on the pytorch BCEWithLogitsLoss, which is the same loss function used for training. The F1 score was calculated character-wise. The results presented here are the best results achieved for each implementation.

Because of the numerous true negatives in the data and because models tended to underestimate rather than overestimate probabilities despite the use of a loss function which can counteract this disbalance, F1 scores generally started improving only once the loss fell below approximately 0.05.

A. LSTM

1) Loss

The LSTM model's loss improved over time, but by a small margin (from an initial weighted BCE of 1.36 to 1.28). Most of the improvements happened in the first seven epochs.

2) F1 Score

The LSTM network was able to obtain a token F1 score of 10.7 at best. It is notable that it was a flat zero from epochs 1-5, then jumped to 0.9 during epoch 6, and then nearly flatlined again.

3) Summary

The LSTM network seems to have failed to learn any significant features of the training data. While it did reduce the training loss and increased the F1 score beyond the initial 0% value, it never surpassed 11%. Furthermore, it was doing this using all of the training data, and being evaluated only on the training data.

We did not attempt to progress beyond this point, as it seems that the model is too poor in comparison to the transformer counterparts to merit further effort. As a result, the metrics which were taken for the BERT and DeBERTa models were not taken on the LSTM model (letter F1 score and same loss function).

The best results were achieved with a learning rate of 1e-2 and with the weighted *BCEWithLogitsLoss* to account for the unbalanced dataset. In fact, without this weighing, no training was observable.

B. BERT

The best score was achieved after 9 epochs at 0.5454 with an average test loss of 0.0342.

In epoch 4 the average test loss was even lower at 0.0399, though the more significant F1 score results were worse at 0.5080. This shows that decreasing loss does not always equal better F1 score evaluation results. Nevertheless, scores can improve further during the following epochs while using this loss function.

Since scores improve slowly and inconsistently during the last three epochs, training for more epochs with similar parameters would be unlikely to yield further improvements.

Training took an average 358 seconds per epoch and 10 epochs in total, meaning around 3580 seconds or 1 hour. It should be noted that this model was implemented last and not trained in a Jupyter notebook which accelerated training on this specific model by about a factor of 4.

C. DeBERTa fully fine-tuned

The fully fine-tuned DeBERTa-based model yielded the best results out of all models, at the cost of time-consuming training.The best F1 score was achieved after 5 epochs at 0.8599 with an average test loss of 0.0123.

After just 1 epoch of training the model achieved scores above 0.7800 with test losses under 0.05, outperforming the BERT-based implementations results after 10 epochs. Training took over 6000 seconds or per epoch and the model was trained for 5 epochs, making the total training time over 30000 seconds or 10 hours.

It is significant that a Juyter notebook was used to make this initial implementation easy to work with, which can as much as quadruple training time as observed in our BERT experiments.

D. DeBERTa partially fine-tuned

1) 1 Layer

Fine-tuning just the linear layer after the model makes this even faster than BERT at 4-5 minutes per epoch, at the cost of adequate performance. Even after 10 epochs of training the score reaches only 0.0282 with a loss of about 0.06, while after the usual 5 epochs the score stays at 0.0000. This approach was deemed unserviceable because training success was minimal.

2) 3 Layers

Fine-tuning the last 3 layers – the manually added linear layer as well as the last 2 of 12 DeBERTa layers – yields comparable results to fine-tuning the whole model. The validation loss reaches values of about 0.016 for the last two epochs and an F1 score of up to 0.8069 after 4 epochs.

Training takes around 3980 seconds or slightly over an hour — 50 minutes for training and 15 minutes for validation per epoch when using a Jupyter notebook file. Training for over 5 epochs is unlikely to yield further improvements, as the best results are usually achieved in epoch 4.

Results from epoch 4 are comparable to results of the fully fine-tined model after between 1 and 3 epochs depending on which 20% of data is used for testing. Therefore, this method is less efficient than training the whole model for a smaller number of epochs.

E. Overview

TABLE I. Overview of each model's best performance

Results	**Model**			
	LSTM	***BERT fine-tuned***	***DeBERTa fully fine-tuned***	***DeBERTa 3 layers fine-tuned***
avg. loss	N/A	0.0342	0.0123	0.016
F1 score	0.107 (tokens scored, training data only)	0.5454	0.8599	0.8069
best epoch / total epochs	15/20	9/10	5/5	4/5
total training + test time	6 hours (Jupyter, training only)	2 hours (Jupyter, training only)	2 hours (Jupyter, training only)	2 hours (Jupyter, training only)

[a] Using a Jupyter notebook instead, like for the DeBERTa-based model, slows training by about a factor of 4.

VI. Conclusion

In this work, we compare several singular models' performance in finding given features in human-written clinical patient notes. This provides us insights into which models could be successfully used in an ensemble type model which has proven ideal for this task.

We conclude that models based on Microsoft's DeBERTa architecture will yield ideal results which — as expected — can't be approached by simple LSTMs and neither by the faster but older and less performant BERT architecture.

With efficiency being part of our aim, we attempted to fine-tune just the last three layers of the DeBERTa-based implementation. However, it proved to be both more accurate and more time efficient to fine-tune a DeBERTa-based model on all layers with a low variable learning rate starting at 2e-5. To improve accuracy, training for more than 5 epochs may be promising, since improvements slow down but do not plateau entirely at epoch 5.

While BERT and DeBERTa are trained on English data , it can be assumed that this approach would work for other languages for which DeBERTa performs similarly well or for which language-specific models have been derived, e.g. CamemBERTa which is based on the same DeBERTa version used in our experiments and trained exclusively on French text data [7, 8].

We conclude that training DeBERTa-based models while only fine-tuning all layers with a learning rate of 2e-5 for at least 5 epochs, is likely to yield ideal results for this type of task.

References

[1] Le An Ha, Maggie, Ryan Holbrook, Victoria_Yaneva. (2022). NBME - Score Clinical Patient Notes. Kaggle. https://kaggle.com/competitions/nbme-score-clinical-patient-notes

[2] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), J. Burstein, C. Doran und T. Solorio, Hrsg.,Minneapolis, Minnesota: Association for Computational Linguistics, Juni 2019, S. 4171–4186. doi: 10.18653/v1/N19-1423.

[3] A. Prakash, 3 Types of Contextualized Word Embeddings From BERT Using Transfer Learning, en, Jan. 2021. Adresse: https://towardsdatascience.com/3-types-of-contextualized-word-embeddings-from-bert-using-transfer-learning-81fcefe3fe6d (visited 15. 12. 2023).

[4] P. He, X. Liu, J. Gao und W. Chen, DeBERTa: Decoding-enhanced BERT with Disentangled Attention, arXiv:2006.03654 [cs], Okt. 2021. doi: 10.48550/arXiv.2006.03654.

[5] A. Hagen, Microsoft DeBERTa surpasses human performance on SuperGLUE benchmark, en-US, Jan. 2021. Adresse: https://www.microsoft.com/en-us/research/blog/microsoft-deberta-surpasses-human-performance-on-the-superglue-benchmark/ (visited 15. 12. 2023).

[6] T. Hustache, Bert Fine tuning takes a lot of time, Forum post, Okt. 2020. Adresse: https://stackoverflow.com/q/64573292 (visited 04. 01. 2024)

[7] R. Scheible, F. Thomczyk, P. Tippmann, V. Jaravine und M. Boeker, GottBERT: a pure German Language Model, en, Dez. 2020.

[8] [7] W. Antoun, B. Sagot und D. Seddah, Data-Efficient French Language Modeling with CamemBERTa, arXiv:2306.01497 [cs], Juni 2023. doi: 10 . 48550 / arXiv . 2306 . 01497.

[9] S. Hochreiter; J. Schmidhuber (1997). "Long short-term memory". Neural Computation. PMID 9377276. S2CID 1915014.

APPENDIX

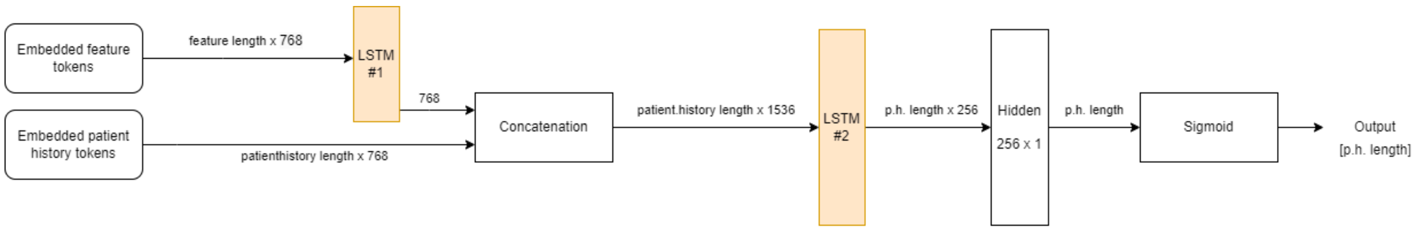

Fig. 2: Structure of used LSTM network