Final Report - Individual Report: Andrew Ash

Introduction

This group project was inspired by our experiences in undergraduate studies as members of Toastmasters [1], an organization that helps members improve their public speaking abilities through a combination of extemporaneous and prepared speaking activities. One of the roles of a Toastmasters meeting is the Ah Counter who keeps a report of all filler words used by all members who speak throughout the meeting and delivers this report at the end of the meeting. Filler words can be easy to miss during a meeting, leading us to explore ways to help automate the process. We determined that a deep neural network designed to detect filler words with a preference for high precision to avoid distracting speakers with false positive classifications could help streamline the responsibilities of the Ah Counter in a modern approach to the role. This report summarizes my contributions to BERFA (Binary Estimator for Robust Filler Analysis) designed to complete this filler word detection task.

Contributions

Dataset Selection: In the early stages of the project I was exploring datasets that might be able to improve our generalization. This led me to the PodcastFillers dataset [2] as a great additional dataset for testing or validation alongside training on the originally planned dataset. After identifying inconsistencies and inaccuracies in the original dataset, we transitioned training over to use only the PodcastFillers dataset.

Transfer Learning Model Selection: Early in the project I was tasked with selecting a pretrained model to use for feature extraction alongside a custom classifier head. I referenced the PyTorch documentation [3] for built in models to ease the transfer learning process and learned it would be quite easy to use Wav2Vec2-based models as well as different iterations of HuBERT. After referencing documentation and tutorials on the models [4], [5], I learned that the Basic Wav2Vec2 would be well suited to the task without any unnecessary layers to complicate the process or transform the data in ways that would be unhelpful to standard speech processing tasks. At this time I wrote a basic script (wav2vec2_model.py) for testing the transfer model with a custom classifier. The flow of the script was inspired by the transfer learning lab from this course, but uses my own original code without outside sources.

Model Training, Data Augmentation, and Refining: I was the primary decision maker in all training and model improvements. I would occasionally report my findings to Marcus and get his inputs on the process, but aside from those conversations this process was entirely my contribution. Rather than write the same information a second time, I am going to simply give a brief summary of my methodology and copy in the relevant section from the group report below.

I used ChatGPT to write boilerplate code for saving the model checkpoints to the file system and plotting. I then modified the code to clean up the presentation of data, file structures, and other details to my preferences. The end result of this is that train.py, test.py, plotting.py, and threshold_test.py are approximately 80% code written or edited by me with some boilerplate

written by ChatGPT and a single method (CustomBCE) being based on a recommendation from a StackOverflow post answer [6] that I then modified to my needs. The results section reflects the in depth description of my model development, data augmentation, and hyperparameter exploration from the group report.

Results

All our trained models use the Wav2Vec2 Basic model for voice audio feature extraction without fine-tuning, as this would have greatly increased the time required to train new models between tests and could have led to overfitting due to the much smaller training dataset compared with the Wav2Vec2 training data.

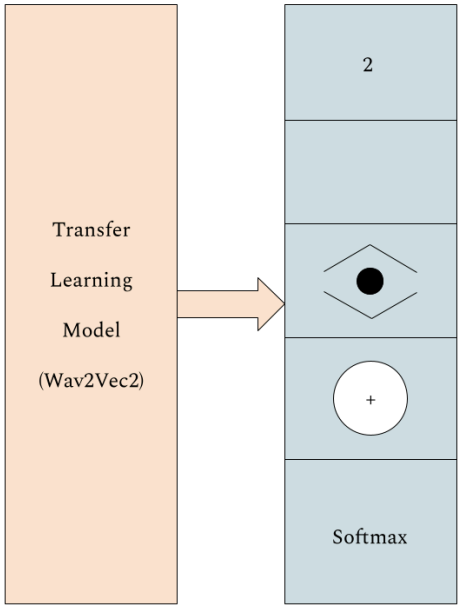

Fig. 1: The Wav2Vec2 basic model is followed by a single fully connected layer with two neurons and a softmax output to be fed into a cross-entropy loss function.

Fig. 1 shows the basic structure of our first model with a single linear layer classifier with two neurons fed into a softmax activation function to create confidence scores for the filler and non-filler classes. The confidence values are then used by a cross-entropy loss function.

This basic classifier trained in under 2 minutes per epoch, allowing for many epochs of training while maintaining quick evaluation. PyTorch's built-in CrossEntropyLoss method includes an optional weight value to help with class imbalance problems [7]. For our application, this weight allows us to punish loss related to the positive class more heavily than the negative class by increasing the weight for Filler class loss and leaving Non-Filler unchanged. The loss equation is shown below.

$$\text{Cross Entropy Loss}(x, y) = -w_y \cdot \log\left(\frac{e^{x_y}}{\sum_j e^{x_j}}\right)$$

Giving double the weight to Filler classifications in the loss function, this model achieved an F0.5 score of 0.865 after training for 49 epochs. At this point we determined exploring other

loss functions may improve our model's performance and streamline our training process. The slowly climbing F0.5 score made training take longer than necessary. In addition, cross-entropy loss depends on the logits of all classes in the denominator. Our custom weight is only intended to punish false positives, but dependence on all classes' logits leads our weight to interact indirectly with the non-filler class decision.

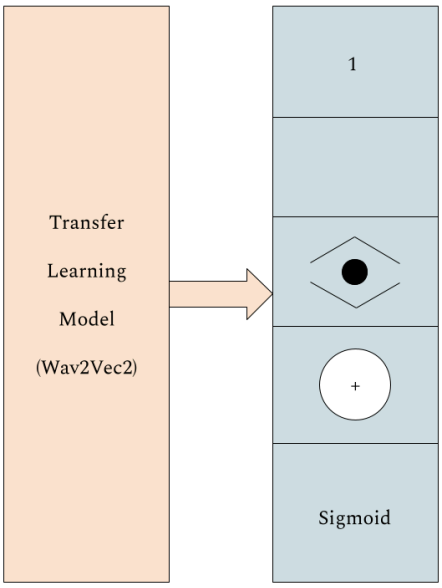

Fig. 2: The Wav2Vec2 basic model is followed by a single fully connected layer with one neuron and a sigmoid output to be fed into a binary cross-entropy loss function.

Based on recommendations from a Stack Overflow question [8] and answer [6], the next model we attempted was a custom binary cross-entropy (BCE) loss. To accommodate the change in loss function, the model needed slight modifications, shown in Fig. 2. A single neuron outputs a logit that is passed through a sigmoid activation function and then given to the custom BCE function, calculated as shown below. The ε here is just a small value to ensure that log has a non-zero value for stability. w_{fn} and w_{fp} are the custom weights for the negative and positive classes, y is the target class, and p is the sigmoid output (a value between 0 and 1).

$$\text{Custom BCE} = -\big(w_{\text{fn}} \cdot y \cdot \log(p + \varepsilon) + w_{\text{fp}} \cdot (1 - y) \cdot \log(1 - p + \varepsilon)\big)$$

Training for this network was highly unstable. Fig. 3 shows the F0.5, accuracy, precision, and recall all wildly oscillating between minimum and maximum values from one epoch to the next. This is because the custom loss function does not implement any maximum magnitude for the output, while PyTorch's built-in BCEloss function caps the loss at a magnitude of 100 for stability while training [9]. A maximum magnitude could have been easily implemented, but further exploration of the PyTorch documentation [10] and forums [11] revealed that BCEWithLogitsLoss could meet our needs without writing additional custom code.

The next model is identical to that of Fig. 2, but the loss function is replaced with PyTorch's BCEWithLogitsLoss method. The pos_weight variable allows us to punish false

positives more heavily than false negatives by setting a weight between 0 and 1. Weights greater than 1 would punish false negatives instead of false positives. This is equivalent to the custom BCE equation where w_{fn} is set to 1 and w_{fp} is pos_weight, which is determined based on the intended ratio of the two weights. Fig. 4 shows the baseline performance for the model with this loss function; a pos_weight of 1.0 ensures standard performance without weighting to punish false positives. Fig. 5 shows the minor changes that start to appear with a small emphasis on the positive class using a pos_weight of 0.75. Finally, Fig. 6 shows the differences in the model's behavior with a pos_weight of 0.5. The recall occasionally drops below 0.8, but the F0.5 is getting closer to 0.9 due to precision gains. The model's accuracy is slighter higher with this pos_weight because the dataset is slightly weighted towards fillers.

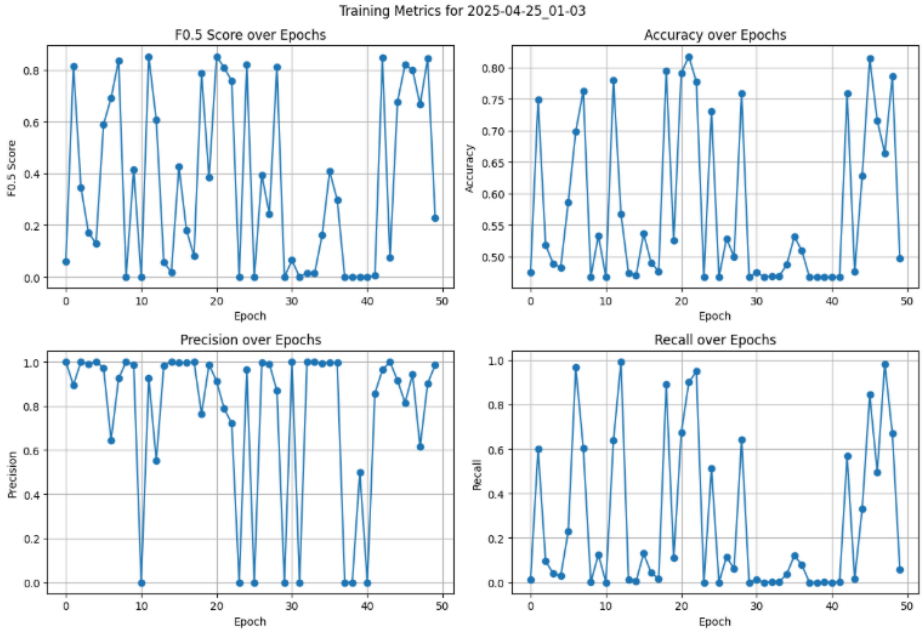

Fig. 3: The F0.5, accuracy, precision, and recall of the model using custom BCE Loss. Training is highly unstable with the model oscillating between classifying everything as a filler and then as a non-filler.

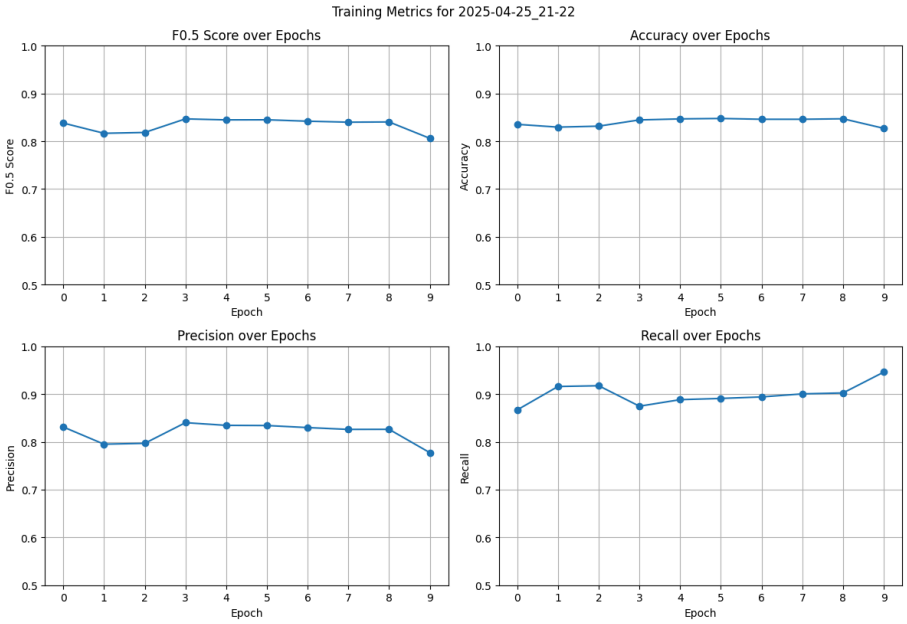

Fig. 4: A baseline example of BCEWithLogitsLoss using a pos_weight of 1. The loss function has the default behavior in this configuration.

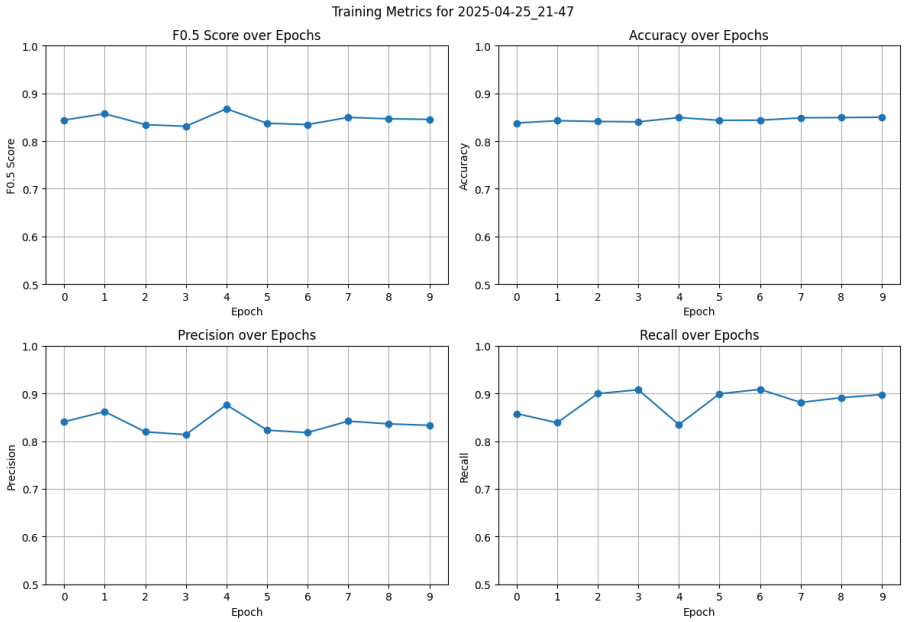

Fig. 5: BCEWithLogitsLoss using a pos_weight of 0.75. The fifth epoch shows the training tradeoffs between precision and recall are impacting performance with minimal change in accuracy compared to the baseline.

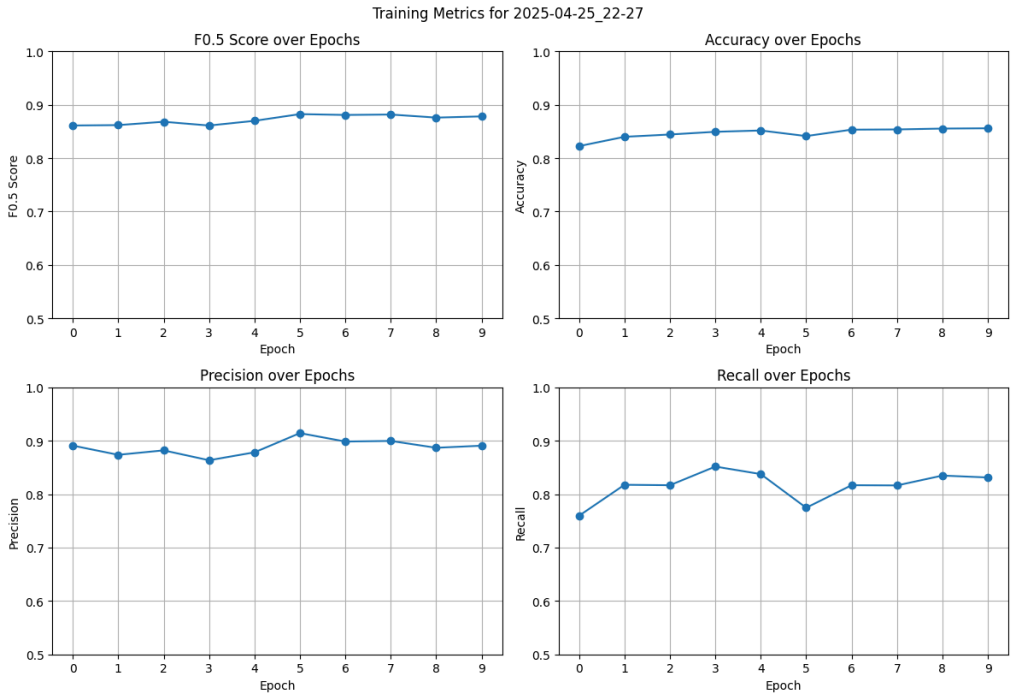

Fig. 6: BCEWithLogitsLoss using a pos_weight of 0.5. The later epochs show that the tradeoffs between precision and recall slowly increase F0.5 at the cost of lower recall compared to the baseline.

At this stage in the model design, we implemented learning rate scheduling to help with stability while training. Fig. 7 shows 50 training epochs with a learning rate scheduler that reduces the rate by a factor of 0.33 after 2 epochs without improving the F0.5 score. The training has largely stopped around 20 epochs as the learning rate hits its minimum. Fig. 8 shows the performance differences with a pos_weight of 0.75 with the same learning rate scheduler. The model now overemphasizes recall for our purposes, resulting in a slightly lower F0.5 maximum. In this stage of design, we achieved a maximum F0.5 of 0.881 in the 10th epoch of training using a pos_weight of 0.5 (0.023 better than the cross-entropy loss and 39 epochs faster).

After this, we added early stopping to avoid wasting unnecessary training time if no improvements were noted as the learning rate decreased to identify a near-optimal network. From this stage onward, we tested with three different pos_weight values to explore the small performance differences between a weighting of 0.75 (small extra punishment for false positives), 0.5 (heavy false positive punishment), and 0.625 (a balance between the first two weights).

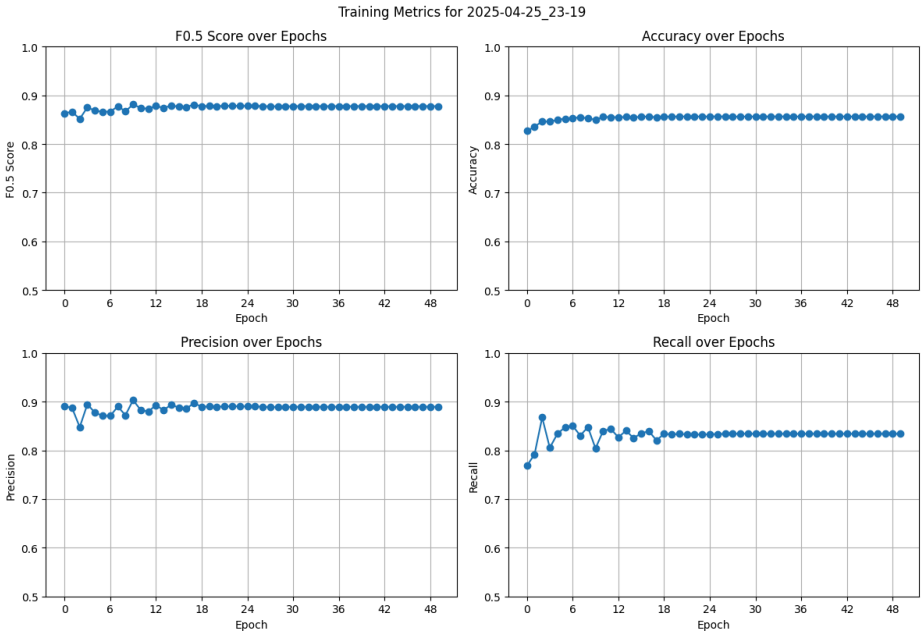

Fig. 7: A pos_weight of 0.5 with learning rate scheduled to reduce by 1/3 after 2 epochs without F0.5 score improvement. Results have plateaued after ~20 epochs.

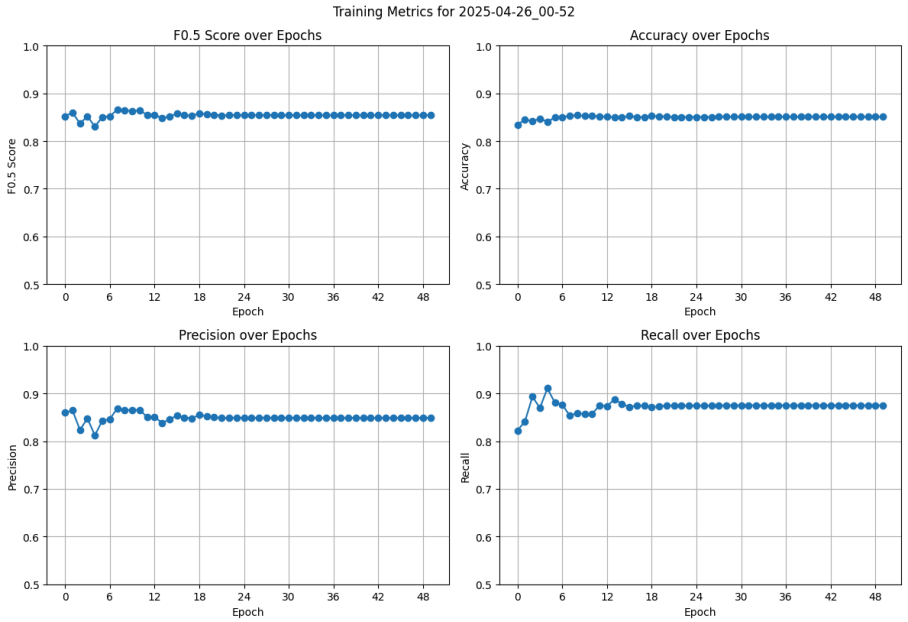

Fig. 8: A pos_weight of 0.75 with learning rate scheduling. Results emphasize recall more than precision and plateau after ~20 epochs.

Reading through the full details of the PodcastFillers dataset [2] revealed a potential overfitting issue in the data. The audio clips that include a filler are always centered on the filler; this quirk of the dataset could easily lead the network to assume a filler should be in the middle of sampled audio. To resolve this issue, we added data augmentation code that takes a random amount of audio off the front or back of the clip and then appends the same amount of silence onto the clip's other side. This effectively moves the remaining audio forward or backward in

time to attain more diverse filler locations, thereby achieving better translation invariance in the time dimension. Fig. 9 shows the best model with data augmentation used a pos_weight of 0.625 and random shifting between $plus.minus$ 150 milliseconds to achieve an F0.5 of 0.882 in 7 epochs (0.001 higher and 3 epochs faster).

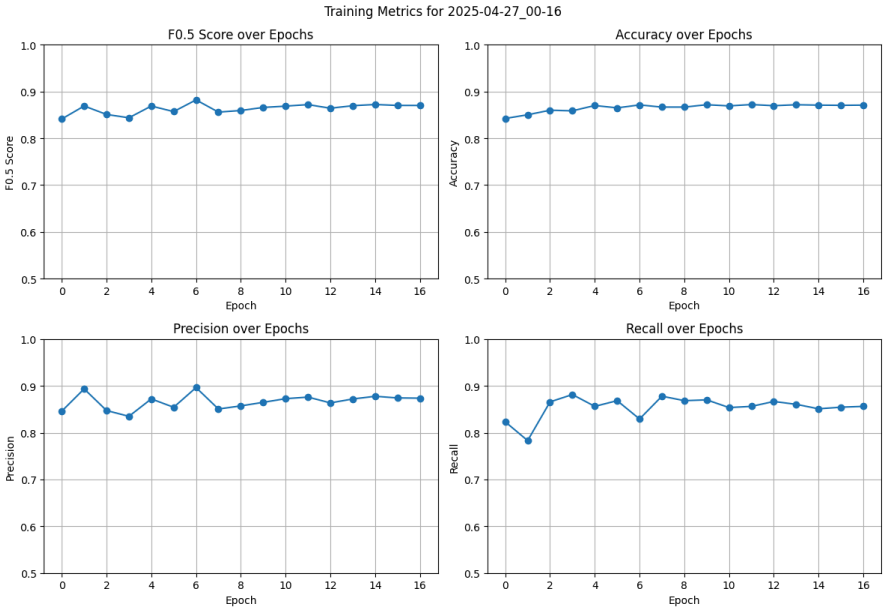

Fig. 9: Using a pos_weight of 0.625 and adding time-shifting data augmentation yields slightly better performance than prior models. Early stopping prevented unnecessary training after epoch 17.

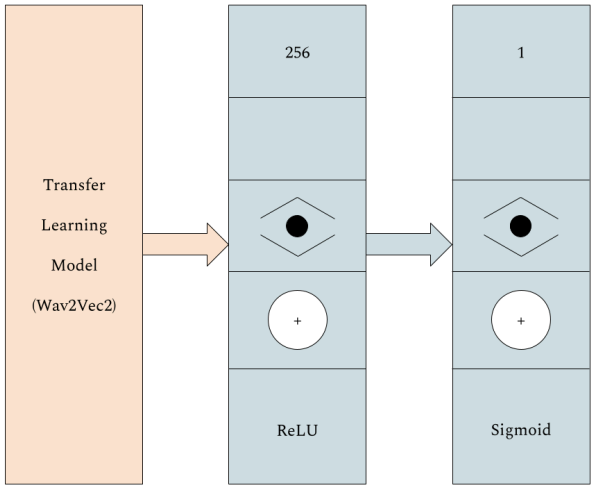

Fig. 10: The Wav2Vec2 basic model is followed by a simple multi-layer perceptron. The first fully connected layer extracts more feature information, and the second is the classifier with a sigmoid output to be fed into a cross-entropy loss function.

The final modification to the model was to add another linear layer to the classifier. The new model is shown in Fig. 10. The classifier head is now a small multi-layer perceptron with additional feature extraction capabilities from the added hidden layer. The extra layer adds 10-15 seconds onto each epoch, maintaining efficient training and testing.

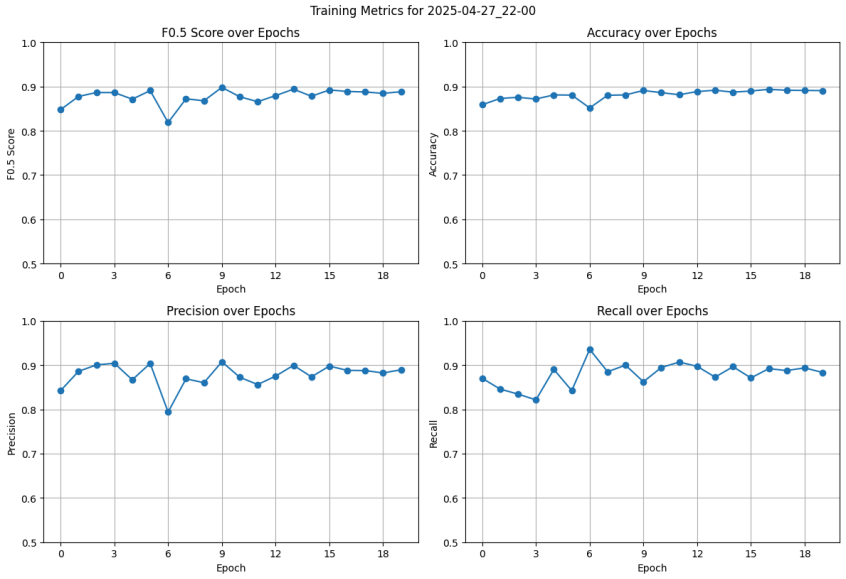

Fig. 11: A pos_weight of 0.75 provides more balance of precision and recall than past models using this weight. Both metrics stay near 0.9. The F0.5 also follows this trend.

Fig. 11 shows the new model trained with a pos_weight of 0.75; this model achieves a good balance of precision and recall. However, the recall values are consistently greater than precision, indicating this may not be the best value for our use case. Fig. 12 shows a better balance of precision and recall, with both values staying much closer together and a slightly better precision than recall. Fig. 13 shows the best-performing model achieved an F0.5 score of 0.903 in 7 epochs using a pos_weight of 0.5. This comes at the cost of a lower recall that is far less stable throughout the training. Regardless of pos_weight, the accuracy is largely unaffected, slowly climbing towards 0.9.

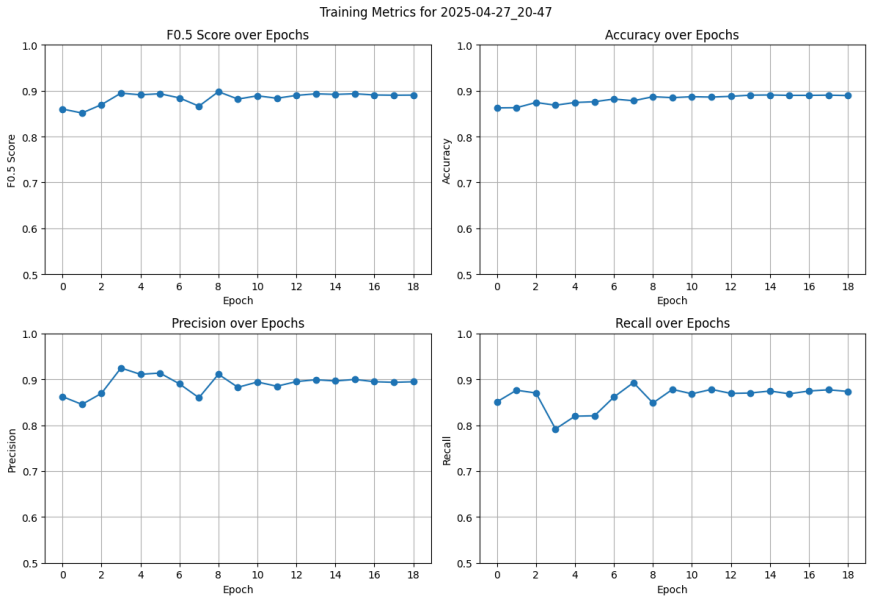

Fig. 12: A pos_weight of 0.625 benefits precision well, with values consistently near or above 0.9. Despite oscillations in recall, the F0.5 remains steady between 0.89 and 0.9.

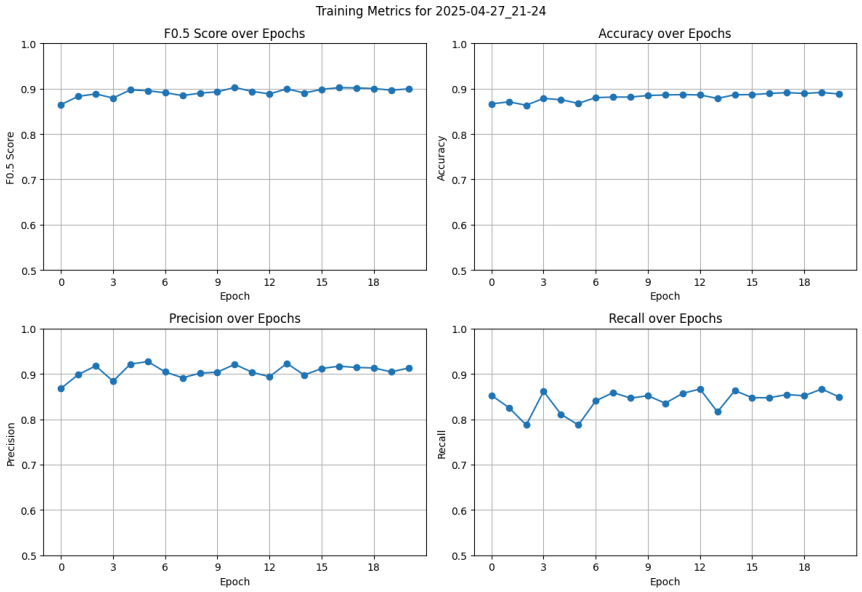

Fig. 13: A pos_weight of 0.5 benefits precision well, with values consistently above 0.9. F0.5 climbs above 0.9 despite recall scores staying between 0.8 and 0.85.

The final improvement came from manually adjusting the threshold delineating filler and non-filler classifications. By default, the model assumes a threshold of 0.5 and then rounds up or down to the nearest classification label. The F0.5 score can be further tuned by allowing a custom threshold value to filter out additional false positives. SkLearn's precision_recall_curve method tests all different threshold values that would make a change in classification accuracy [12]. For example, if a test dataset included three samples that yielded logits of 0.3, 0.56, and 0.8 then the method would test precision and recall at these threshold values as well as one threshold value less than the minimum and one threshold greater than the maximum to force all classifications to each of the two classes. The precision, recall, and list of thresholds checked are all returned for easy plotting.

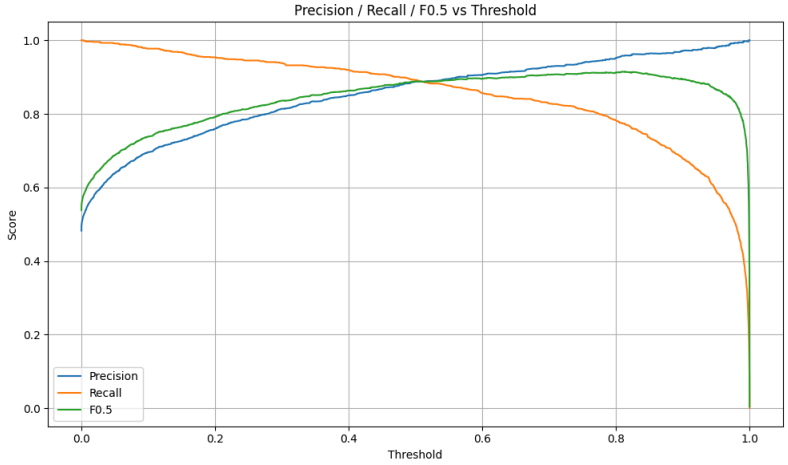

Fig. 14: Threshold shifting for epoch 16 with a pos_weight of 0.75 (see Fig. 11) achieves a maximum of 0.915 at a threshold of 0.809.

Fig. 14 shows the result of threshold shifting using epoch 16 of the model trained with a pos_weight of 0.75. It is interesting to note that the precision and recall curves intersect at a threshold of approximately 0.5, indicating the default threshold is well balanced for cases where both metrics are equally important. Shifting the threshold to 0.809 yields an F0.5 of 0.915, an improvement of 0.026 from the default threshold at the cost of much lower recall.

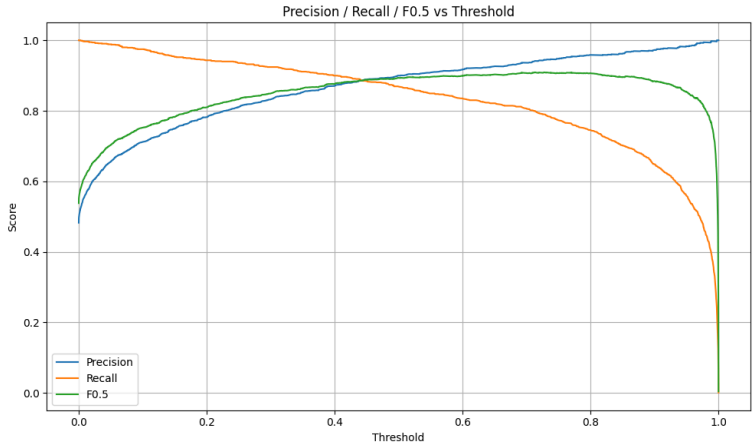

Fig. 15: Threshold shifting for epoch 15 with a pos_weight of 0.625 (see Fig. 12) achieves a maximum of 0.909 at a threshold of 0.713.

Fig. 15 shows the improvements possible from epoch 15 of the 0.625 pos_weight model. A balanced precision and recall would require a threshold near 0.425, clearly showing the model's preference to improve precision even before threshold shifting. The maximum F0.5 reaches 0.909 at a threshold of 0.713, yielding a 0.016 improvement.

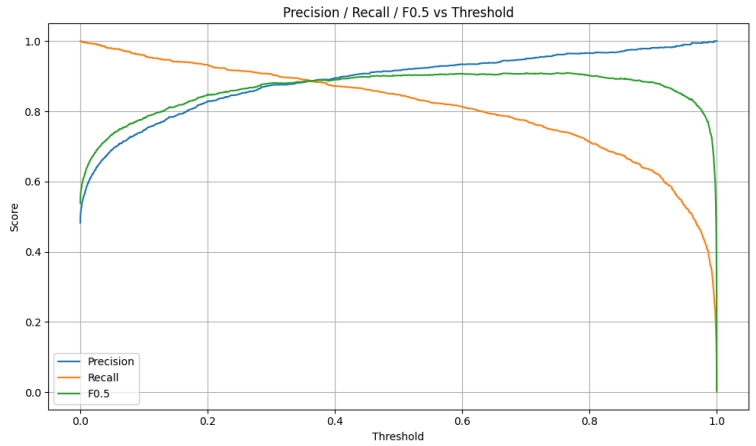

Fig. 16: Threshold shifting for epoch 16 with a pos_weight of 0.5 (see Fig. 13) achieves a maximum of 0.910 at a threshold of 0.764.

Fig. 16 shows the improvements possible at epoch 16 of the 0.5 pos_weight model. The threshold for balancing precision and recall now drops below 0.4, indicating the expected extreme bias towards precision in this model. Shifting the threshold to 0.764 leads to a maximum F0.5 of 0.910, an improvement of 0.008 from the default threshold.

This final hyperparameter adjustment allowed for a maximum F0.5 of 0.915 by using a pos_weight of 0.75 and a threshold of 0.809. This is not recommended as the false negative rate quickly increases with a threshold greater than 0.6. For a practical system, the small precision increases are not worth the large recall losses. Instead, a threshold between 0.5 and 0.75 should improve any of these models moderately at a lower cost to recall.

Conclusion

There are a few key takeaways from my contributions to the project. First, even a simple classifier head can get excellent results with the right selection of a pre-trained feature extraction network. Even at its most complex, the custom portion was still just a two-layer MLP. Second, dataset development and curation is a challenging task that cannot be taken for granted in a neural network design problem. Our first dataset selection looked good, until we noticed how inaccurate the filler word labeling was and the PodcastFillers dataset had some overfitting issues at first because of quirks in the way the audio was formatted. Finally, the threshold shift plots (Fig. 14-16) show some interesting points about the tradeoffs of precision and recall in situations where the two metrics cannot be treated equally. It is very interesting to see that the best F0.5 metric came from letting the recall plummet to get just a few extra percentage points of precision.

There are a variety of next steps that I believe could improve the network, as I mentioned in the group report. Rather than rehash them in detail, I will list them briefly and refer to the group report for full details.

- Additional data augmentation: pitch shift and faster/slower speaking rate
- A more diverse set of speaker accents
- Fine-tuning the feature extraction of Wav2Vec2 basic for our particular task
- Adding a second neural network to identify different speakers to fully automate the Ah Counter concept this project is built upon

Overall, the project went quite well. Thanks to the combined efforts of Marcus and I, we have a real-time classifier that can run on constrained laptop hardware and is capable of an ah count at least as accurate as a human Toastmasters member.

References

[1] “Toastmasters International -Club Meeting Roles.” Accessed: May 05, 2025. [Online]. Available: https://www.toastmasters.org/membership/club-meeting-roles

[2] G. Zhu, J.-P. Caceres, and J. Salamon, “Filler Word Detection and Classification: A Dataset and Benchmark,” in 23rd Annual Cong.~of the Int.~Speech Communication Association (INTERSPEECH), Incheon, Korea, Sep. 2022. [Online]. Available: https://arxiv.org/abs/2203.15135

[3] “torchaudio.pipelines — Torchaudio 2.5.0.dev20241105 documentation.” Accessed: April 05, 2025. [Online]. Available: https://docs.pytorch.org/audio/main/pipelines.html#wav2vec-2-0-hubert-wavlm-ssl

[4] “Wav2Vec2Model — Torchaudio 2.5.0.dev20241105 documentation.” Accessed: April 05, 2025. [Online]. Available: https://docs.pytorch.org/audio/main/generated/torchaudio.models.Wav2Vec2Model.html#torchaudio.models.Wav2Vec2Model
[5] “Speech Recognition with Wav2Vec2 — Torchaudio 2.5.0.dev20241105 documentation.” Accessed: April 05, 2025. [Online]. Available: https://docs.pytorch.org/audio/main/tutorials/speech_recognition_pipeline_tutorial.html#sphx-glr-tutorials-speech-recognition-pipeline-tutorial-py
[6] J. Vuurens, “Answer to "Using weights in CrossEntropyLoss and BCELoss (PyTorch)".” Accessed: Apr. 24, 2025. [Online]. Available: https://stackoverflow.com/a/67778392
[7] “CrossEntropyLoss — PyTorch 2.7 documentation.” Accessed: May 05, 2025. [Online]. Available: https://pytorch.org/docs/stable/generated/ torch.nn.CrossEntropyLoss.html
[8] clueless, “Using weights in CrossEntropyLoss and BCELoss (PyTorch).” Accessed: Apr. 24, 2025. [Online]. Available: https:// stackoverflow.com/q/67730325
[9] “BCELoss — PyTorch 2.7 documentation.” Accessed: May 05, 2025. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn. BCELoss.html
[10] “BCEWithLogitsLoss — PyTorch 2.7 documentation.” Accessed: May 05, 2025. [Online]. Available: https://pytorch.org/docs/stable/generated/ torch.nn.BCEWithLogitsLoss.html
[11] “BCELoss with class weights - vision.” Accessed: Apr. 23, 2025. [Online]. Available: https://discuss.pytorch.org/t/bceloss-withclass-weights/196991
[12] “precision_recall_curve.” Accessed: May 06, 2025. [Online]. Available: https://scikit-learn/stable/modules/generated/sklearn.metrics.precision_ recall_curve.html