Beyond validation loss: Improving a model's clinical performance using clinically-relevant optimization metrics

Charles B. Delahunt DELAHUNT@UW.EDU
Courosh Mehanian COUROSH@UOREGON.EDU
Daniel Shea SHEA.DAN@GMAIL.COM
Matthew P. Horning MATTHEW.P.HORNING@GMAIL.COM
Global Health Labs, Bellevue, WA.

Abstract

A key task in ML is to optimize models at various stages, e.g. by choosing hyperparameters or picking a stopping point. A traditional ML approach is to apply the training loss function on a validation set to guide these optimizations.

However, ML for healthcare has a distinct goal from traditional ML: Models must perform well relative to specific clinical requirements, vs. relative to the loss function used for training. These clinical requirements can be captured more precisely by tailored metrics. Since many optimization tasks do not require the driving metric to be differentiable, they allow a wider range of options, including the use of metrics tailored to be clinically-relevant.

In this paper we describe two controlled experiments which show how the use of clinically-relevant metrics provide superior model optimization compared to validation loss, in the sense of better performance on the clinical task.

The use of clinically-relevant metrics for optimization entails some extra effort, to define the metrics and to code them into the pipeline. But it can yield models that better meet the central goal of ML for healthcare: strong performance in the clinic.

Keywords: Metrics; healthcare; optimization

Data and Code Availability The *Loa loa* dataset is in preparation for open-access publication (Kamgno and et al., 2025). The fetal ultrasound dataset (Pokaprakarn et al., 2022) is not yet available publicly. Relevant Python code is included in Appendix B.

Institutional Review Board (IRB) (1) CE N° 0094/CRERSHC/2023 Yaoundé, Cameroon (*Loa loa* study). (2) UNC School of Medicine IRB# 24-1415 (ultrasound study).

1. Introduction

Metrics are fundamental to optimizing machine learning (ML) algorithms, including tasks such as hyperparameter optimization and selection of stopping point for training. Certain metrics inherited from ML, e.g. cross-entropy loss, are engrained in ML convention, are straight-forward to implement thanks to well-curated libraries, and are often highly effective. In addition, using the same loss function on train and validation sets is automatic in ML frameworks such as PyTorch. As a result these metrics are embedded in the practice of ML as a standard way to optimize models, on the implicit assumption that they will lead to an optimization that performs well in the clinic.

However, in ML for health care, performing well relative to the training loss function is not a model's actual goal. Rather, the true goal of algorithms developed for medical use cases is to meet specifications determined by clinical needs. These needs are characterized by metrics that often diverge significantly from standard ML metrics.

A clear example is object-level vs. patient-level metrics: An algorithm is often trained at the object-level - that is, the unit passed through the model is a patch of a histology slide, a suspected parasite, a thumbnail of an object of interest, etc, and algorithm performance is optimized using metrics at this level; in contrast the clinician cares about a patient, whose sample under test contains many objects. There is necessarily a transform, usually non-linear, from object-level results to patient-level results. For example, the algorithm might evaluate several image patches from a histology slide, then patient disposition is based on some function of the patch results. A model optimized for maximum performance at the object-level has no guarantee of optimal performance

at the patient-level, because the metrics that define good performance are different.

More generally, if a model is optimized according to a Figure of Merit (FoM) that does not accurately encode the clinical requirements, the model will not attain a clinically-optimal state, because it is by definition being directed towards some other optimum. This is analogous to heading northwest when the actual destination is due north - "close" might work if the problem is tractable enough, but for a difficult problem it carries risk that the algorithm will fail to meet the clinical requirements.

An optimized model certainly can (and arguably should) be evaluated using metrics tailored to the clinical use case requirements, both to guide overall development and for final reporting. For a concrete example see Delahunt et al. (2024). However, in this case the impacts of the chosen optimization metrics are already baked in prior to model evaluation.

In this paper, we examine the effect of applying clinically-focused metrics earlier, during model optimizations. We describe two controlled experiments that demonstrate a clear benefit of applying metrics that are closely tailored to use case-specific demands to optimize a model, rather than relying on standard flavors of ML metrics.

We distinguish this approach from the familiar construction of loss functions, e.g. with form $L = L_{a_1} + L_{a_2} + \alpha L_r$, where L_{a_i} are variants of losses like cross-entropy (CE) and L_r is a regularizer. Though these can work well from an ML perspective, they usually must be differentiable, and this constraint can prevent tailoring them to specific, concrete clinical performance requirements.

The approaches described are readily applicable to any ML project, and are potentially valuable if the FoMs defining clinical performance are different from the loss functions that drive ML training (as is usually the case). The approach requires two steps:

1. Accurately capture the clinical performance requirements as a FoM that is a function of algorithm outputs, by combining the {algorithm outputs $\rightarrow$ patient disposition} function with the clinical requirements (e.g. the minimum acceptable patient-level specificity for deployment). A valuable guide to metrics for health care tasks is found in (Maier-Hein et al., 2022).

2. Inject this metric at the relevant stage to drive the algorithm optimization or selection.

The next two sections of the paper describe the two experiments, each in a Background-Methods-Result structure. They are: (i) Optimizing hyperparameters using a patient-level FoM rather than an object-level FoM; and (ii) Choosing a stopping point for a DNN model using clinically-relevant metrics rather than the usual validation loss curve.

To clarify expectations, we note that the details of the particular datasets and models are not important for this paper. They are vehicles to illustrate the topic of metric choices for algorithm optimization.

2. Experiment 1: Hyperparameter optimization

This section describes an experiment comparing the effects of a patient-level vs object-level loss function as a driver of hyperparameter optimization. The details of the model are not important (they are described in (Delahunt and et al., 2025)). The important element is that an identical model architecture (trained at the object-level) has hyperparameter optimization done in two ways: driven by patient-level FoM or by object-level FoM.

2.1. Background

We consider an imaging device that looks at videos of fresh blood in capillaries to diagnose *Loa loa* infections by detecting motion of filariae in fresh blood for the "Test and Not Treat" use case in Mass Drug Administration (Kamgno et al., 2017). A clinically-important issue arose during field trials: If the blood in the capillary coagulates due to some delay in imaging, the filariae cannot move so the motion detection algorithm returns a potentially dangerous False Negative. For example images, see Figure 1. Therefore a module to detect coagulation was required to return a label (coagulated or not) for the patient.

2.2. Methods

For the *Loa loa* diagnosis task, the inputs are 7 videos taken along the length of the capillary. For the coagulation task, 1 still frame is extracted from each video, so the inputs are 7 frames (7 "objects") per patient.

An SVM (Pedregosa et al., 2011) acts on FFT spectrum features (chosen because coagulated blood acquires a "checkerboard" appearance) to give a score to each video ("video-level", "frame-level", and

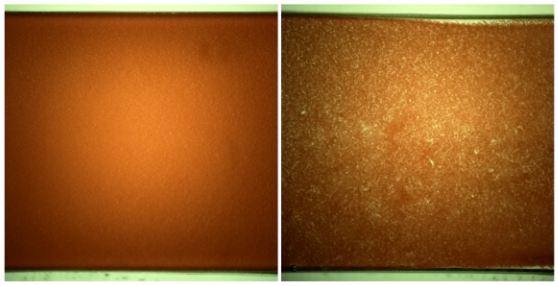

Figure 1: Frames of blood capillary videos, normal (top) and very coagulated (bottom). The dataset contains a range of coagulations.

"object-level" are equivalent for this task). To assign a patient-level label, the N^{th} video's score is used (N is a hyperparameter). So the SVM is trained at the object-level to classify objects (videos) as coagulated or not. But the required clinical output is a patient-level classification, which is a non-linear function of the object scores.

Model hyperparameters include SVM parameters, feature selection parameters, and a few others. For optimization we used the hyperopt library (Bergstra et al., 2015), which is guided by the Tree of Parzen Estimators method (Bergstra et al., 2011). Differentiability is not required.

For this experiment, everything in the set-up is kept fixed except for the choice of loss function to drive the hyperopt optimization: in one case AUC over videos is used, matching the object-level of the model's training[1]; in the other, AUC over patients is used, matching the clinically-relevant output. (The actual loss used by hyperopt is 1 - AUC.)

Results are reported for the validation sets in a 5-fold cross-validation. The different fold scores are combined and made comparable using a novel (to our knowledge) z-mapping technique. The method is described in Appendix A.

2.3. Results

The crucial finding is that patient-level and object-level optima are not correlated, and optimizing the object-level metric, though perfectly sensible from a ML perspective (since the model is trained at object-level) leads to inferior patient-level performance. The following figures illustrate this disconnect between object-level and patient-level losses:

1. Figure 2 show the ROC curves for the best iteration in the video-level run (on left) and for the best iteration in the patient-level run (on right). The video-level ROCs are very similar, but the patient-level ROC is much better when patient-level metrics drive the optimization.

2. Figure 3 **A** show the patient-level and object-level AUC values for each iteration of the hyperopt run driven by patient-level AUC, with iteration order sorted by increasing patient-level AUC. Note that the object-level AUCs are simply a cloud: optimizing patient-level AUC does not optimize object-level AUC.

3. Figure 3 **B** shows the equivalent plot for the hyperopt run driven by object-level AUC, sorted by increasing object-level AUC. As above (but reversed), optimizing object-level AUC does not optimize patient-level AUC.

4. Figure 3 **C** shows a scatterplot of patient-level vs. object-level AUCs per iteration (from the object-level run). Note the lack of correlation between object- and patient-level results.

Thus, if the goal is to optimize clinical performance it is best to drive hyperparameter optimization not by the model's loss function, but by a loss function that more closely aligns with clinical requirements.

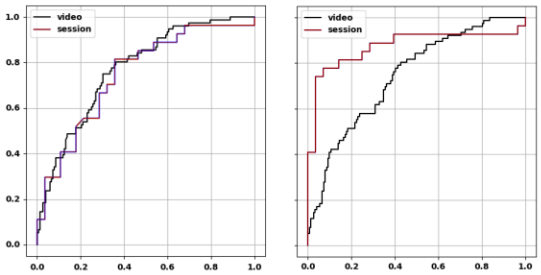

Figure 2: Best ROC curves for patient-level (red) and video-level (black). Per subplot, the two curves are from the best hyperopt iteration, as driven by: **(left)** object-level AUC, **(right)** patient-level AUC. Patient-level optimization gives much better patient-level performance.

1. The SVM's hinge loss is not visible. Our manual hinge loss gave results similar to object-level AUC.

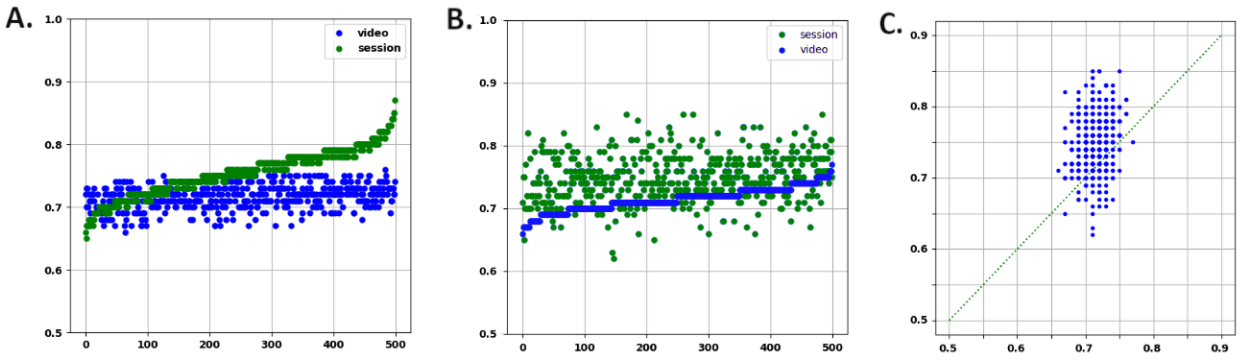

Figure 3: Patient-level and video-level AUC values for different iterations of hyperopt. x-axis = sorted iteration index, y-axis = AUC value. Patient-level AUCs in green, video-level AUCs in blue. **A**: Optimizer driven by patient-level AUC, iterations sorted by patient-level AUC values. **B**: optimizer driven by video-level AUC, iterations sorted by video-level AUC values. **C**: Scatterplot of patient-level vs video-level AUCs for data in subplot B. Note the lack of correlation between the two performance metrics.

3. Experiment 2: Stopping point optimization

3.1. Background

DNNs are trained over several epochs, and an important optimization task is to choose the best epoch's model for testing or deployment. The usual method is to consult the loss curves for train and validation sets, where the same loss function is applied to both train and val sets. Then the final epoch (i.e. model) is chosen based on the validation loss curve, e.g. where it starts to increase, implying that overfitting has begun. This is an effective method to choose the best model in the narrow sense of performance as measured by the training loss function.

But clinical performance requirements are not, in general, captured by the training objective. Although the loss is often structured to generally reflect the medical context, it is also often shaped by reasons of convention, differentiability, or ease of implementation (e.g. the existence of built-in functions).

Therefore this experiment assesses whether the validation loss curve is in fact an optimal guide to choosing the best epoch in terms of performance on the clinical task. We train a DNN for 40 epochs and then plot, for each epoch, multiple FoMs: not just the validation loss but also other metrics that more tightly reflect the clinical use case. We also plot per-epoch histograms of the validation exam scores to assess the stopping points selected by each FoM.

As before, the details of the model are not important (these are described in (Mehanian and et al., 2025)). The important point is that an identical model has its stopping point optimized in one of two ways: with the usual validation loss curve, or with clinically-relevant metrics along with per-epoch assessment plots such as score histograms.

We consider the case of a DNN trained to distinguish twins vs singleton fetuses using blind (unguided) ultrasound sweep videos. This is clinically important because twin pregnancies tend to be higher risk and should be identified. The blind sweeps with algorithms enable this diagnostic in low resource settings where trained sonographers are not available.

3.2. Methods

The data consist of blind sweep ultrasound exams collected in Zambia and the USA by a team led by U. of North Carolina's Global Women's Health group (Pokaprakarn et al., 2022). Twins are the "positive" class, singletons are the control.

A DNN architecture delivers an exam-level score. It is trained in Pytorch (Paszke et al., 2019) and Pytorch Lightning (Falcon and the PyTorch Lightning team, 2019) with balanced cross-entropy loss. The Pytorch library, as typical for ML frameworks, cal-

culates the same CE loss on the validation set, and produces a validation set loss curve. We plot 1 - validation loss.

We also plot alternate metrics tailored to the clinical goal. To generate these metrics requires some extra effort. First, the exam scores must be saved at each epoch, either via a custom callback function during training, or by running each epoch's model on the validation set as inference. Then custom code is needed to calculate metrics and generate plots. This is a cost of extending optimization beyond built-in ML functions. The following FoMs are plotted:

1. Validation loss returned by PyTorch (here, balanced CE).
2. Standard area under the ROC curve (AUC).
3. 90% "sliver" AUC. The sliver AUC considers only the subset of the AUC where specificity is $> 90\%$, i.e. the area under the ROC in the leftmost $1/10^{th}$ of the normal ROC. This FoM is valuable because the minimum acceptable clinical specificity (for this project) was 90%, so performance in the entire righthand region of the ROC is not clinically relevant.

   The $n\%$ sliver AUC can be calculated by summing trapezoids over the operating points with False Positive Rate between 0.0 and $(1-n/100)$, normalized by $(1-n/100)$. For an example of sliver AUC, see Figure 4. Code is provided in Appendix B.
4. Sensitivity at 90% specificity. If the clinically-acceptable specificity is known (here 90%), the corresponding sensitivity is a scalar that directly measures the model's potential performance at a clinically-relevant operating point.
5. Fisher distance, defined for two distributions by $\frac{|\mu_1 - \mu_2|}{\sigma_1 + \sigma_2}$ where μ_i, $\sigma_i =$ means and standard deviations. We use right and lefthand std devs which are useful for asymmetric distributions (code is given in Appendix B).

We also plot, per epoch, some histograms and scatterplots, because these are useful for qualitative assessment of what the models are doing relative to the FoMs given above.

1. Scatterplots and histograms of scores, separated by class.
2. Plots of sensitivity and specificity curves vs threshold on scores.
3. Scatterplots of scores vs gestational age (GA).

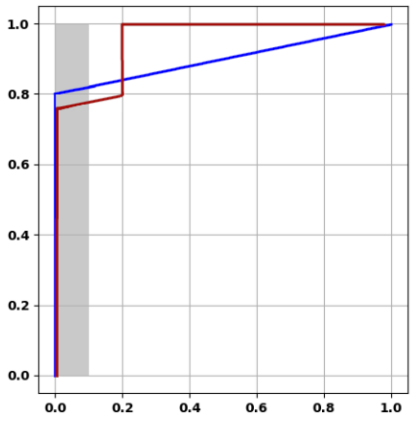

Figure 4: Sliver AUC example: The blue ROC has lower overall AUC than the red (0.9 vs 0.96), but it is stronger at high specificities and thus has a higher 90% sliver AUC (in the grey column, 0.82 vs 0.77).

3.3. Results

The crucial finding is that, when these various FoMs are calculated at each epoch on the validation set, the validation loss curve suggests a substantially different "best" stopping point than do the clinically-tailored FoMs. In addition, the per-epoch scatterplots and sensitivity-specificity indicate that the stopping point suggested by the training loss is too early and will likely give inferior results in the clinical task.

A 5-fold split was used for cross-validation purposes, so 5 models were trained. Results are given below for one of the 5 folds - all folds showed similar behavior.

The disagreement between "best" stopping points is seen in the per-epoch time-series of PyTorch's val loss and the alternate FoMs, shown in Figure 5 (the negative of val loss is shown, so that for all FoMs higher is better). The val loss maxes out at epoch 22, while the the clinically-tailored FoMs keep steadily increasing up to epoch 39. Specifically:

1. The validation loss (using the training loss function) maxes out at epoch 22, then decreases in a distinct though noisy way.

2. The standard AUC maxes out at around epoch 20, and shows no further change after this. That is, standard AUC cannot distinguish between the models of epochs 20 - 39.

3. The 90% sliver AUC shows steady increase up to epoch 39, i.e. it argues for later epochs.

4. The "sensitivity at 90% specificity" FoM increases until epoch 29, then mostly stays at this maximum. So it argues for ≥ 29 epochs.

5. The Fisher Distance matches the behavior of the sliver AUC, steadily increasing to a maximum at epoch 35.

The question now arises: Which stopping point is in fact the best? The separation behavior of the per-epoch models, for epochs $\{0, 3, 12, 22, 29, 35, 39\}$, are shown in the scatterplots and histograms in Figure 6. Epoch 22, despite having the best validation loss, has poor separation compared to epochs 29 - 39.

The later epochs also show greater stability around the clinically-relevant operating point, seen in the per-epoch sensitivity and specificity plots of Figure 7. Stability is indicated by shallower slopes in the two curves at the required operating point ($\geq 90\%$ specificity), since this reflects greater robustness to changes in operating point. Epoch 35 is strongest in this regard.

An interesting effect specific to the particular dataset/use-case is seen in Figure 8, which shows per-epoch scatterplots of scores vs GA. Twin fetuses with low GA, e.g. < 80 days, are much harder to classify for biological reasons, and all epochs give lower scores to low GA cases. However, the model at epoch 35 "sacrifices" these cases, giving them very low scores, in order to further reduce singletons' scores, ensuring better overall separation. By contrast, epoch 22 has higher scores for low GA cases, but at the cost of spread-out, higher scores for singletons. Thus, epochs ≥ 35 separate the two classes better, except for the "lost cause" low-GA cases. This effect is germane to the clinical use case, so the ability to notice it enables better optimization for the clinic.

In summary: The standard validation loss curve suggests stopping relatively early, but the clinically-tailored FoMs, backed up by the behaviors in the per-epoch assessment plots, indicate that training much longer is better for the clinical task. This crucial information is not revealed by the validation loss curve.

4. Discussion

A key task in ML is to optimize of models at various stages, e.g. by choosing hyperparameters or picking a stopping point. A standard ML approach is to use the training loss function to to guide these optimizations, because these meet the differentiability requirements of backprop and gradient descent, and also perhaps because of convention and ease of use (e.g. with built-in functions and libraries).

However, ML for healthcare is a distinct field from traditional ML, and it has a distinct goal: The model performance must meet rigorous clinical specifications. These demands can be captured more precisely by metrics customized to the clinical needs than by the more usual ML objective functions. Since many optimization tasks do not require differentiability, they admit of a wider range of metric options.

In this paper we gave two examples of how metrics tailored to be clinically-relevant provide superior model optimization, in the sense of better performance on the clinical task (they give worse performance in the sense of an ML-centric task definition). This result is not surprising: To excel at a task, it makes sense to tailor the optimization to that task. For ML applied to health care, this requires moving beyond the metrics and conventions inherited from traditional ML.

This clinically-focused approach requires encoding the specifics of the use case into usable Figures of Merit: What metrics matter to clinical deployment, and how the outputs of a model feed into a clinically-relevant disposition (e.g. the object-level to patient level transform).

The use of clinically-relevant metrics for optimization entails some extra effort, to define appropriate metrics and code them into the pipeline, but in return it can yield models that better meet the central goal of ML for healthcare: Optimal performance in the clinic.

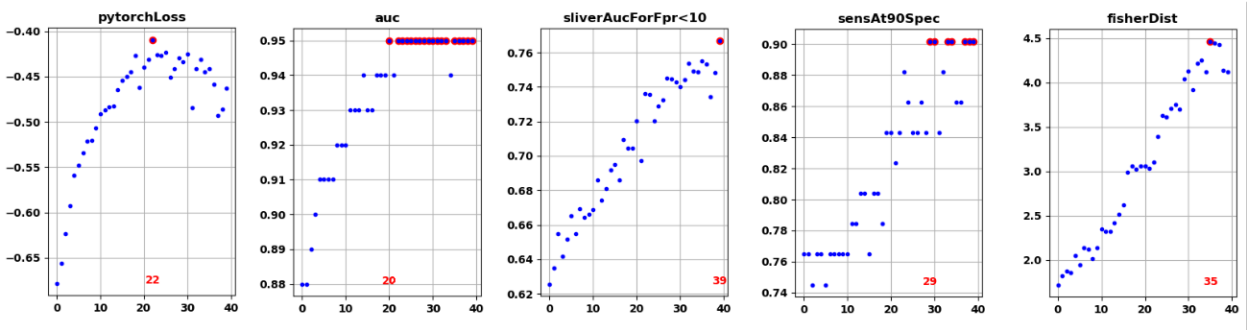

Figure 5: Figure of Merit time-series over training epochs. x-axis: epoch indices, y-axis: FoM value. **L - R:** (1) -1 * Standard validation loss; (2) AUC; (3) 90% Sliver AUC; (4) Sensitivity at 90% specificity; (5) Fisher distance. x-axis: epoch. y-axis: value (higher is better). Highest scores marked in red. Clinically-relevant FoMs suggest much later stopping points than that of validation loss.

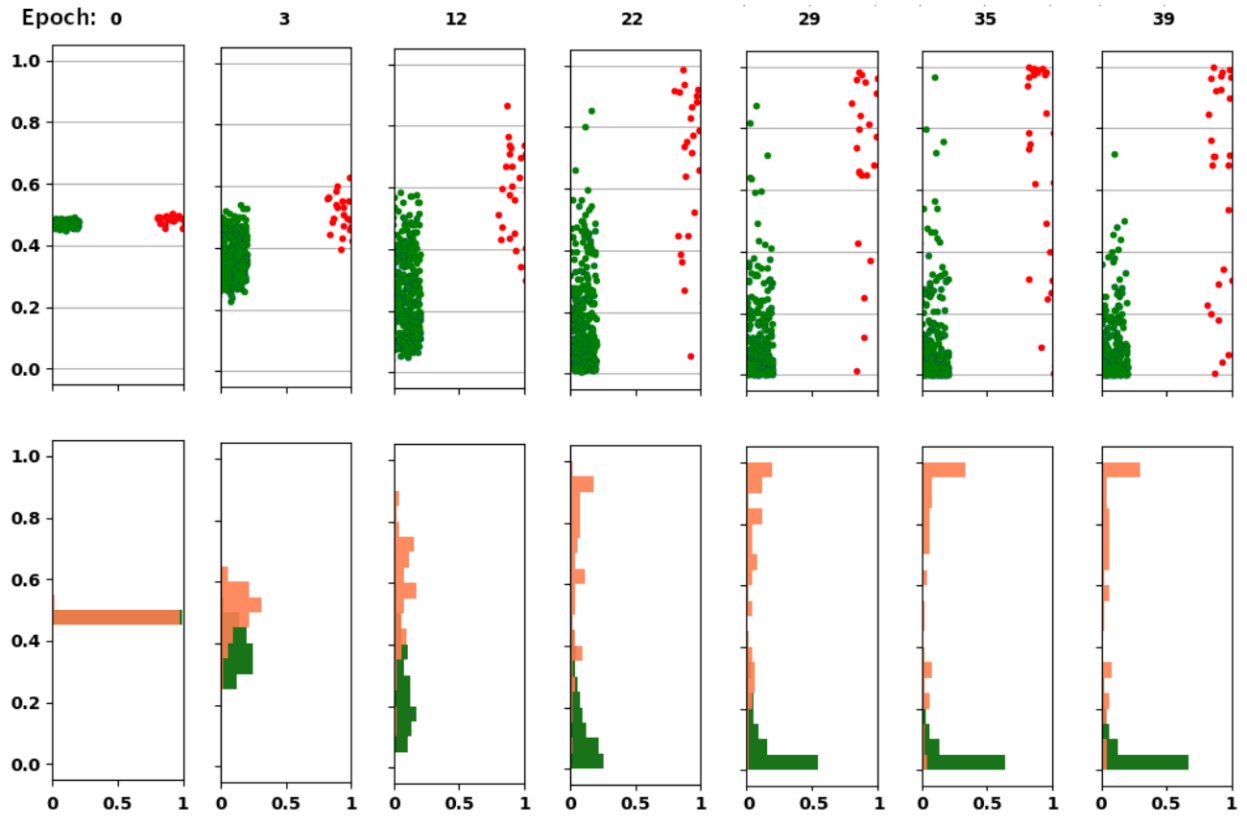

Figure 6: Exam score distributions per epoch: **Top:**Score scatterplots: Singletons in green, twins in red. y-axis = model score. **Bottom:** Histograms of scores: Singletons in green, twins in red. Class separation continues to improve after epoch 22 (which has best validation loss), indicating that the model is still usefully learning.

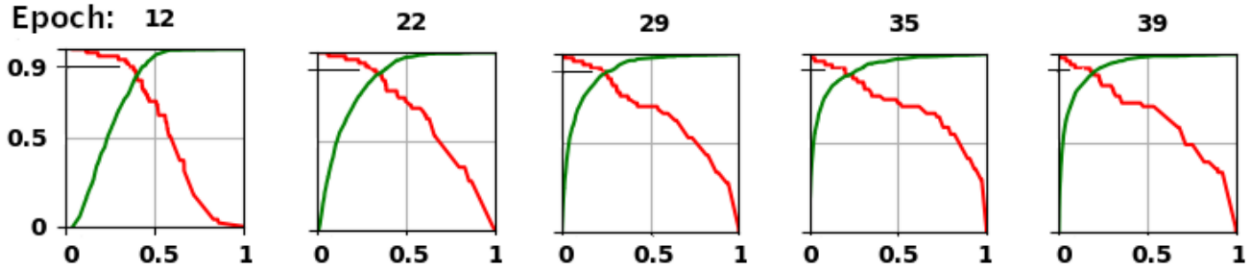

Figure 7: Sensitivity (green) and specificity (red) vs threshold, per epoch. x-axis: threshold. y-axis: value (of sens or spec) Shallower curves around the clinically-relevant operating point ($\geq 90\%$ specificity, marked by short gray lines) indicate more stability.

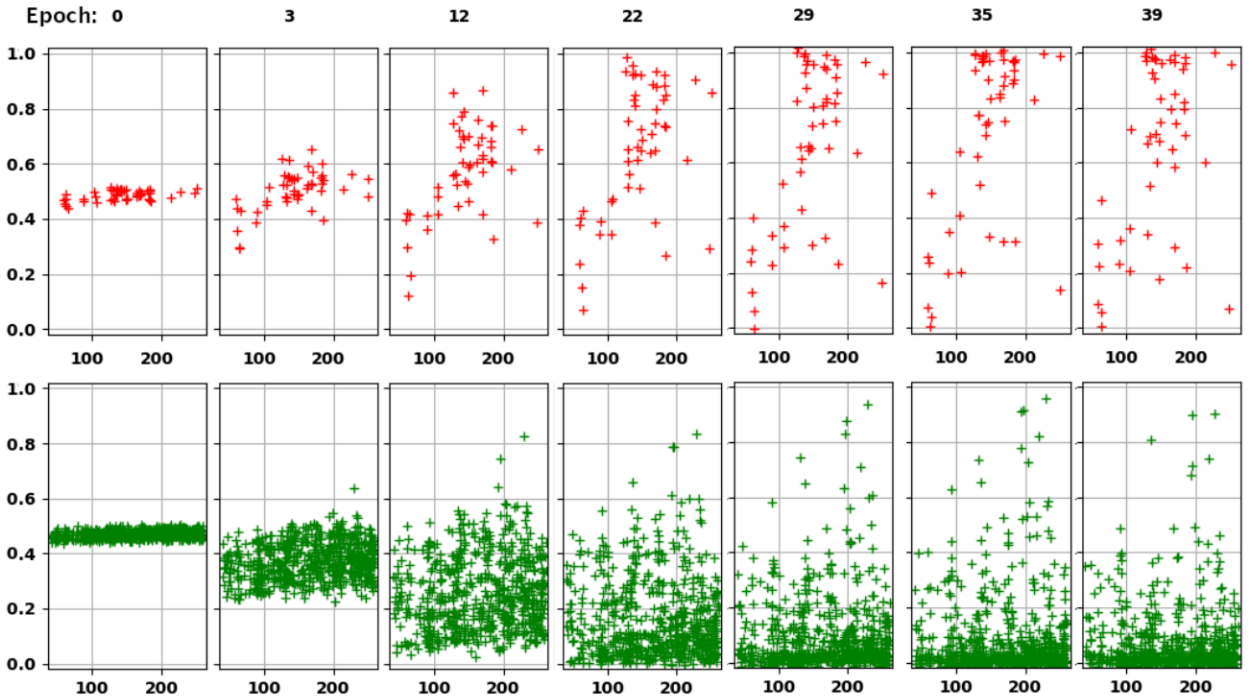

Figure 8: Exam scores vs GA, per epoch: x-axis = GA. y-axis = score. Red = Twin, green = singletons. Low GA cases are always difficult. Later epochs (e.g. 35 vs. 22) "sacrifice" them to force singleton scores closer to 0 and twin scores closer to 1, giving overall better separation.

Acknowledgments

Like all ML researchers, we owe everything to those who create the datasets:

Our thanks to Dr. Joseph Kamgno and his team at the Center for Research on Filariasis and other Tropical Diseases (CRFilMT) in Cameroon for collecting the *Loa loa* dataset, to all the generous participants, and to Anne-Laure Nye for annotations.

Our thanks to Dr. Jeffrey Stringer, the Global Women's Health group at U. of North Carolina School of Medicine, and their Zambian collaborators for collecting and curating the FAMLI dataset, and to all the mothers-to-be who generously participated.

This work was funded by Global Health Labs, Inc. (www.ghlabs.org).

References

J Bergstra, R Dardenet, Y Bengio, and B Kegl. Algorithms for hyper-parameter optimization. *NeurIPS*, 2011.

J Bergstra, B Komer, C Eliasmith, D Yamins, and DD Cox. Hyperopt: a Python library for model selection and hyperparameter optimization. *Computational Science Discovery*, 2015.

CB Delahunt and et al. Algorithms to detect *Loa loa* in fresh blood samples. *In preparation*, 2025.

CB Delahunt, N Gachuhi, and MP Horning. Metrics to guide development of machine learning algorithms for malaria diagnosis. *Frontiers Malaria*, 2024.

W Falcon and the PyTorch Lightning team. PyTorch Lightning. 2019. URL https://github.com/Lightning-AI/pytorch-lightning.

J Kamgno and et al. A dataset of videos for *Loa loa*. *In preparation*, 2025.

J Kamgno, SD Pion, CB Chesnais, MH Bakalar, MV Michael V. D'Ambrosio, CD Mackenzie, HC Nana-Djeunga, R Gounoue-Kamkumo, G-R Njitchouang, P Nwane, JB Tchatchueng-Mbouga, S Wanji, WA Stolk, DA Fletcher, AD Klion, TB Nutman, and M Boussinesq. A Test-and-Not-Treat Strategy for Onchocerciasis in *Loa loa*-Endemic Areas. *NEJM*, 2017. doi: 10.1056/NEJMoa1705026.

L. Maier-Hein, A. Reinke, and et al. Metrics reloaded: Pitfalls and recommendations for image analysis validation. *arXiv*, 2022. https://arxiv.org/abs/2206.01653.

C Mehanian and et al. Algorithms to detect fetal age features from ultrasound blind sweeps. *github repo, exact name TBD*, 2025.

A Paszke, S Gross, F Massa, A Lerer, J Bradbury, G Chanan, T Killeen, Z Lin, N Gimelshein, L Antiga, A Desmaison, A Kopf, E Yang, Z DeVito, M Raison, A Tejani, S Chilamkurthy, B Steiner, L Fang, J Bai, and S Chintala. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 2019.

F Pedregosa, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, J Vanderplas, A Passos, D Cournapeau, M Brucher, M Perrot, and E Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

T Pokaprakarn, JC Prieto, JT Price, MP Kasaro, N Sindano, HR Shah, M Peterson, MM Akapelwa, FM Kapilya, YV Sebastião, W 3rd Goodnight, EM Stringer, BL Freeman, LM Montoya, BH Chi, DJ Rouse, Vwalika B Cole, SR, MR Kosorok, and JSA Stringer. AI estimation of gestational age from blind ultrasound sweeps in low-resource settings. *NEJM Evid.*, 2022.

Appendix A. Scaling method to combine k-folds

A K-fold split setup yields K distinct models and associated scores on their respective validation sets. Each sample has exactly one output score when treated as a validation sample, as calculated by one of the K models. Performance of the K models on their validation sets offers valuable insight into model variability. But these per-model results can be highly volatile for medical datasets that have small patient counts.

In such cases, it can be desirable to combine all the samples into one set, to give a larger validation set on which to evaluate model performance. The catch is that the different models are calibrated differently, i.e. they have different score ranges so that a given operating point will correspond to different thresholds in each split, as seen in the left subplot of Figure 9. Therefore we offer here a technique to combine the model scores on their validation sets into a single consistent set of scores for which a single threshold can be used.

We assume two classes, one of which is the "control" class. For example, blood samples that are uncoagulated (the control) or coagulated, with class labels 0 and 1. Let samples be ${s_{i,k}}^c$, where $c \in \{0, 1\}$ is the class, i is the sample index, and k is the split. The technique basically consists of mapping, for each split, the distribution of its control sample validation scores to a common z-scale and common median. The coagulated sample scores also get mapped according to this transform, i.e. they "go along for the ride".

1. For each fold k, calculate the median and the two-sided standard deviations of the control samples, $m_k = \text{median}({s_i}^0)$ and similarly $\sigma_{r,k}, \sigma_{l,k}$. See code in Appendix B.
2. Choose a target median and right-hand standard deviation m_t and $\sigma_{r,t}$, e.g. $m_t = 0.3, \sigma_{r,t} = 0.2$.
3. For each k, scale all the sample scores in the k^{th} validation set to a new normalized score n_i: $n_i = \sigma_{r,t}({s_{k,i}}^c - m_k)/\sigma_{r,k}$ (if $s_{k,i} > m_k$; similarly use $\sigma_{l,t}$ if $s_{k,i} < m_k$).

The $\{n_i\}$ are all directly comparable, in the following sense: if s_{i,k_1} is x std devs above m_{k_1} and s_{j,k_2} is x std devs above m_{k_2}, then $n_i = n_j$, so an x std dev threshold treats them both the same, whether it is in fold k_1, k_2, or in the common scale. See Figure 9 for an example. Code is provided in Appendix B.

This method works best when scores are not already pushed to the rails (if scores are clustered at 0 and 1, the method has little impact vs. simply combining the various folds' scores as-is).

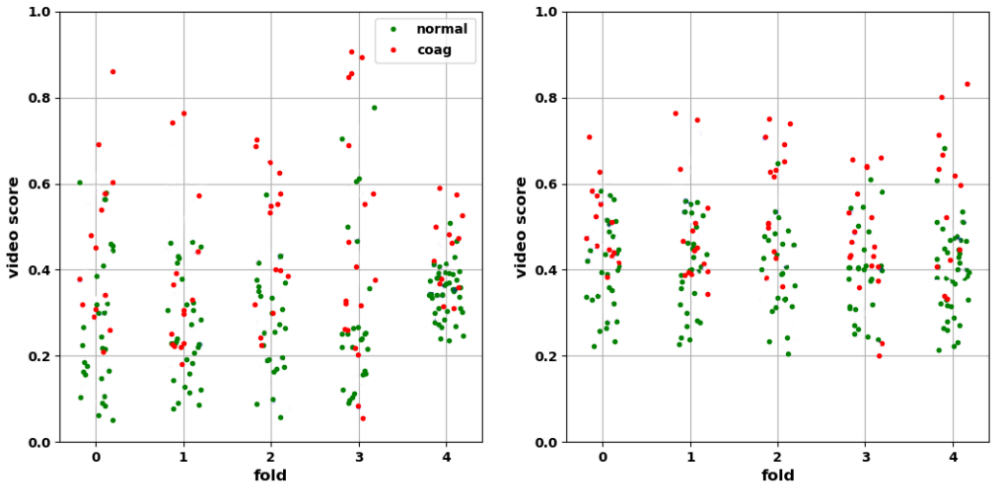

Figure 9: Effect of z-scale alignment. y-axis = scores. Green = controls, red = coagulated. **Left:** Raw output scores per fold: Each fold's model requires a different threshold for a given specificity. **Right:** mapped scores per fold: One threshold gives the same specificity for all folds.

Appendix B. Python function defs

B.1. 2-sided standard deviations

Asymmetrical distributions are imperfectly described by the standard deviation with gaussian assumption. A quick way to more precision is to calculate two standard deviations, right- and lefthanded, using just the points to the right (or left) of the median.

```
import numpy as
def calculateTwoSidedStdDev_fn(x, middleVal=None)
  """
  Calculate two standard deviations, one for the left and one for the right, by flipping samples across
  the median (not across the mean, on the grounds that in asymmetrical distributions the median
  is likely closer to the peak value).

  Parameters
  ----------
  x : list-like or np.array vector
  middleVal: float or int. If you don't want to use the median. Default = None, ie use median.

  Returns
  -------
  stdDevRight : scalar float
  stdDevLeft : scalar float
  """

  if len(x) <= 1:
    stdDevRight = -1
    stdDevLeft = -1
  else:
    if middleVal != None:
      m = middleVal
    else:
      m = np.median(x)
    x = x - m
    xH = x[x >= 0] # top half, shifted to a nominal 0 center
    xL = x[x <= 0] # bottom half

    stdDevRight = np.std(np.concatenate((-xH, xH)))
    stdDevLeft = np.std(np.concatenate((xL, -xL)))

  return stdDevRight, stdDevLeft

# End of calculateTwoSidedStdDev_fn
```

B.2. z-scale alignment

The first def below extracts the parameters of the distributions in each split. The second def applies these parameters to z-map the scores of each split.

```
import numpy as np
```

```
def standardizeSvmScoresForKArrayGivenParams_fn(x, p):
    """
    Given an array of model output scores from k splits adjust them using parameters
    pre-calculated by 'calculateStandardizationParamsForKFoldScores_fn' (above).
    We apply vector operations to, for each split's score,
      1. Subtract that split's median to center the scores at (hypothetical) zeros
      2. Scale each values by its split's stdDevs (RH and LH) to roughly match the canonical std dev
      3. Shift all the values to the canonical median.
    NOTE: Function exits if there is a zero in splitRhStdDevs or splitLhStdDevs. This should not occur,
    because the training samples that generated the std devs would have to have identical scores.
    Parameters
    ----------
    x : list of floats, len = number of splits, ie number of models
    p : dict with keys 'canonicalMedian', 'canonicalStdDev', 'splitMedians',
      'splitRhStdDevs', 'splitLhStdDevs' All entries except 'canonicalMedian' and
      'canonicalStdDev' will be vectors if we are adjusting all splits at once.

    Returns
    -------
    adjX : list of floats, same length as argin 'x'.

    """
    rhStd = np.array(p['splitRhStdDevs'])
    lhStd = np.array(p['splitLhStdDevs'])
    canonStd = np.array(p['canonicalStdDev'])
    canonMedian = np.array(p['canonicalMedian'])
    splitMedians = np.array(p['splitMedians'])

    if np.sum(rhStd == 0) > 0 or np.sum(lhStd == 0) > 0: # should not happen
      print('splitRhStdDevs or splitLhStdDevs contains a 0 value. Returning input scores.')
      adjX = x
    else:
      t0 = x - splitMedians # subtract each split's median
      # Apply column vector operations:
      for k in range(len(splitMedians)): # Process each column in turn:
        tk = t0[:, k]
        if np.sum(tk > 0) > 0:
          tk[tk > 0] = tk[tk > 0] * canonStd / rhStd[k] # to right of median
        if np.sum(tk < 0) > 0:
          tk[tk < 0] = tk[tk < 0] * canonStd / lhStd[k] # to left of median
        t0[:, k] = tk
      adjX = t0 + canonMedian * np.ones(x.shape)

    return adjX

# End of standardizeSvmScoresGivenParams_fn
#-----------------------------

def standardizeKFoldScoresForVectorGivenParams_fn(x, split, p):
    """
    Given a vector of scores from a model, along with a vector of their splits, standardize the
```

```
    scores of each fold using the dictionary of parameters.
    The argins will likely be the scores on k test sets, concatenated into a vector, where the
    scores came from k models built on a k-fold split.

    Parameters
    ----------
    x : list-like of floats (scores)
    split : list-like of ints (fold indices). same len as 'scores'
    p : dict of params (see unpacking in first few lines)

    Returns
    -------
    adjScores : list-like of floats. same len as 'scores'
    """

    foldVals = np.unique(split) # hopefully 0,1,2, etc. But does not need to be.
    canonicalMedian = p['canonicalMedian']
    canonicalStdDev = p['canonicalStdDev']
    splitMedians = p['splitMedians']
    splitRhStdDevs = p['splitRhStdDevs']
    splitLhStdDevs = p['splitLhStdDevs']

    adjX = x.copy()
    for k in range(len(foldVals)):
        # Adjust all scores in this fold:
        inds = split == foldVals[k]
        t = x[inds]
        m = splitMedians[k]
        r = splitRhStdDevs[k]
        l = splitLhStdDevs[k]
        t0 = t - m
        if r > 0: # in case r == 0, don't divide (a catch)
            t0[t0 > 0] = t0[t0 >¿ 0] * p['canonicalStdDev'] / r # to the right of median
        else:
            pass
            print('rh stdDev = 0 on ' + str(np.sum(t0 > 0)) + ' cases ¿ median.')
        if l > 0: # ditto
            t0[t0 < 0] = t0[t0 < 0] * p['canonicalStdDev'] / l # to the left of median
        else:
            pass
            print('lh stdDev = 0 on ' + str(np.sum(t0 > 0)) + ' cases < median.')
        adjX[inds] = t0 + p['canonicalMedian']

    # optional: could clip adjX at 0 and 1.
    return adjX

# End of standardizeKFoldScoresForVectorGivenParams_fn
```

B.3. n% sliver AUC

```
import numpy as np
from sklearn.metrics import roc_curve
def calculateSliverAuc_fn(y, yHat, targetSpec):
    """
    Given scores, binary labels, and a minimum specificity: calculate the normalized AUC
    within the leftmost sliver of an ROC curve.

    Parameters
    ----------
    y : list-like of ints (0s and 1s)
    yHat : list-like of floats (scores). same len as 'y'
    targetSpec : int (1 to 99)
    Returns
    -------
    sliverAuc : float
    """

    maxFprForAucLoss = (100 - targetSpec) / 100
    [fpr, tpr, op] = roc_curve(y, yHat, pos_label=1)
    inds = np.where(fpr <= maxFprForAucLoss)[0]
    f = list(fpr[inds]) + [maxFprForAucLoss] # postpend an endpoint fpr
    t = list(tpr[inds]) + [tpr[inds[-1]]] # postpend the corresponding tpr value
    sessionSliverAuc = 0
    for i in range(len(f) - 1):
        sliverAuc += (f[i+1] - f[i]) * (0.5 * (t[i+1] + t[i])) # Trapezoid rule
    sliverAuc = sliverAuc / maxFprForAucLoss # normalize

    return sliverAuc

# End of calculateSliverAuc_fn
```