Beyond validation loss: Clinically-tailored metrics for hyperparameter optimization improve a model's clinical performance

Charles B. Delahunt[1,2], Courosh Mehanian[2], Daniel Shea[2], and Matthew P. Horning[2]

[1] University of Washington, Seattle, WA delahunt@uw.edu
[2] formerly Global Health Labs, Bellevue, WA (lab has ceased operations)
{courosh@uoregon.edu, shea.dan@gmail.com,matthew.p.horning@gmail.com

Abstract. Hyperparameter optimization is a key task in ML, and the standard ML approach of validation loss, i.e. applying the training loss function on a validation set, is arguably optimal from a strictly ML-centric perspective. However, ML applied to healthcare has a distinct goal from traditional ML: to deploy, a model's performance must meet stringent clinical requirements that are imperfectly captured by the training loss function. Furthermore, the loss function chosen for hyperparameter optimization has enormous impact on the final state of the model, including which tasks it excels at or falls short on, raising the stakes for this choice. Therefore we describe two controlled experiments, taken from deployment-focused projects, which show how using clinically-tailored loss metrics for hyperparameter optimization yields superior models than using validation loss, in the sense of delivering markedly better performance on the clinical task. Our goal is to highlight this currently underutilized opportunity to improve the clinical utility of ML models.

Keywords: metrics · optimization · hyperparameters · healthcare · machine learning.

1 Introduction

Metrics are fundamental to optimizing machine learning (ML) algorithms, including tasks such as hyperparameter optimization. Certain metrics inherited from ML, e.g. cross-entropy loss, are engrained in ML convention, are straightforward to implement thanks to well-curated libraries, and are often highly effective from an ML perspective. 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 hyperparameters. But in ML for healthcare, performance relative to the training loss function is not a model's true goal: to deploy, a model must meet stringent performance requirements, determined by the clinical use case, that may be best captured by metrics that 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, i.e. the unit passed through the model is a patch of a histology slide, a thumbnail of an object of interest, etc, and algorithm performance is optimized using differentiable metrics at this object 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 clinically-driven function of the patch results. A model optimized for performance at the object-level has no guarantee of optimal performance at the patient-level, because the metrics that define good performance are different.

Previous work: A valuable literature, often written by clinicians, has called out the mismatch between traditional ML metrics and clinical needs [22, 19, 23, 6, 8, 11, 9, 15, 20, 3], noting for example, "The disconnect between the metrics for algorithm performance and the realities of a clinician's workflow and decision-making process is a fundamental but often overlooked issue" [19]; and "None of these measures [traditional ML metrics] ultimately reflect what is most important to patients, namely whether the use of the model results in a beneficial change in patient care" [11]. [13] offers evaluation metrics, organized by medical imaging areas, that are more clinically salient than traditional ML metrics. Some authors have developed metrics that are closely tailored to clinical needs: [6] derives a "net benefit" metric to measure clinical actionability; [3] derives metrics for malaria use cases; [15] develops a simulation model to assess ML model effects in provider-constrained settings; and [20] itemizes properties that diagnoses should have and then derives a metric to match these properties.

However, this literature consistently speaks in terms of how to evaluate and report performance of a trained, locked model, passing over (for reasons unclear) the use of clinically-tailored metrics for hyperparameter optimization. The only example to our knowledge is in [2], where a hyperparameter tuning loss is tailored to the application's regulatory path. But algorithm development is "greedy" (i.e. each decision point constricts later options). By the time evaluation metrics are applied, the impacts of earlier optimization metric choices are baked in.

Contribution: This paper illustrates, via two controlled experiments from real-world deployment-focused projects, the clear benefit to clinical utility of applying loss functions that are closely tailored to use case-specific demands, instead of the usual validation loss, for hyperparameter optimization. They are: (i) optimizing hyperparameters using a patient-level loss rather than an object-level loss; and (ii) choosing a stopping point for a DNN model using clinically-relevant metrics rather than the usual validation loss curve. We note that the hyperparameter optimization task allows more freedom as to metric than the training task: Loss functions for training, 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, must usually be differentiable, a constraint not required during hyperparameter optimization.

The principle we illustrate here is readily applicable to any ML-for-healthcare project, and potentially valuable whenever the Figures of Merit (FoMs) defining the performance required for clinical deployment are at variance with the loss functions that drive ML training (as is usually the case). In our experience, the use of clinically-driven metrics for hyperparameter optimization can provide the crucial boost needed to pass the performance gate towards deployment.

Sections 2 and 3 describe the experiments, each in a Background-Methods-Result structure. To clarify expectations, we note that the details of the particular datasets and models are not the main topic of this paper. References below contain full technical details, and relevant code is in the Appendix. Rather, the examples are vehicles to illustrate the impact of metric choices for model optimization in the following general situation: we have a model which we need to optimize such that it meets stringent, clinically-defined performance specifications as a necessary condition for deployment.

2 Experiment 1: Hyperparameter optimization

Background:
This section describes an experiment in which an identical model architecture (trained at the object-level) had hyperparameter optimization done in two ways: driven by either an object-level FoM or a patient-level FoM.

The context is a diagnostic device and algorithm for the neglected tropical filarial disease *Loa loa*. An imaging device [12] takes videos of fresh blood in capillaries. An algorithm then detects the motion of swimming filariae to diagnose and quantitate *Loa loa* infections [4]. This system serves the "Test and Not Treat" use case in Mass Drug Administration for onchocerciasis, in which patients with high loiasis burdens do not receive ivermectin to avoid adverse events [10].

A clinically-important issue arose during field trials of this system: If the blood in the capillary coagulates due to some delay, the filariae cannot move so the motion detection algorithm returns a potentially dangerous error: a high loiasis burden is labeled as safe to receive ivermectin due to an apparent low burden. This raised the need for a module to detect coagulation. The experiment describes the hyperparameter optimization of this coagulation detector.

Methods:
Videos were drawn from the main loiasis dataset [5]. Each patient visit generated a "session" of 7 sequential videos taken along a capillary of whole blood. For the coagulation task, 385 videos (55 sessions) were manually annotated. In a given session, videos could vary from normal to coagulated due to passage of time. *Session* labels were "normal" or "coagulated", according to whether the session contained at least one coagulated video. The videos were divided into 5-fold train-val splits stratified by session.

Full algorithm details are found in [4]. Briefly: For the coagulation detection task, the inputs were one still frame from each video, so each patient had 7

"objects", hereafter videos. An SVM [17] acts on FFT spectrum features to give a score to each video. 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 video-level to classify videos as coagulated or not, while the required clinical output is a patient-level classification, found via a non-linear linkage function of the video scores. Hyperparameters to be optimized included SVM parameters, feature selection parameters, and a few others. The hyperopt library [1] was used.

For this experiment, everything in the set-up was kept fixed except for the choice of loss function to drive the hyperopt optimization: in one arm, AUC over videos was used, matching the video-level of the model's training; in the other arm, AUC over patients was used, matching the clinically-relevant output. Results are reported for the validation sets in a 5-fold cross-validation. The validation scores from each fold were combined into a single validation set using a novel (to our knowledge) z-mapping technique (described in the online repo).

Results:
The crucial finding is that patient-level and video-level optima are uncorrelated, and that optimizing the video-level metric, though sensible from a ML perspective since the model is trained at this level, leads to inferior patient-level performance. The following figures illustrate this disconnect between video-level and patient-level losses:

1. Fig 1 shows the ROC curves for the best iteration in the video-level run (A) and for the best iteration in the patient-level run (B). The video-level ROCs are very similar, but the patient-level ROC is much better when patient-level metrics drive the optimization.
2. Fig 1 **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.

No test set was assembled because the validation set performance of the module was insufficient to pass the gate towards deployment and development of the module was halted. However, in terms of clinically-relevant performance, the validation set results shown above strongly favored the model optimized with the patient-level metric.

3 Experiment 2: Stopping point optimization

Background:
We consider the case of a DNN trained to distinguish twins vs singleton fetuses using blind (unguided) ultrasound (US) sweep videos. This is clinically important because twin pregnancies tend to be higher risk and should be identified. The blind sweeps plus algorithm enable this diagnostic in low resource settings where trained sonographers are not available. The data consisted of blind sweep ultrasound exams, collected in Zambia and the USA [18].

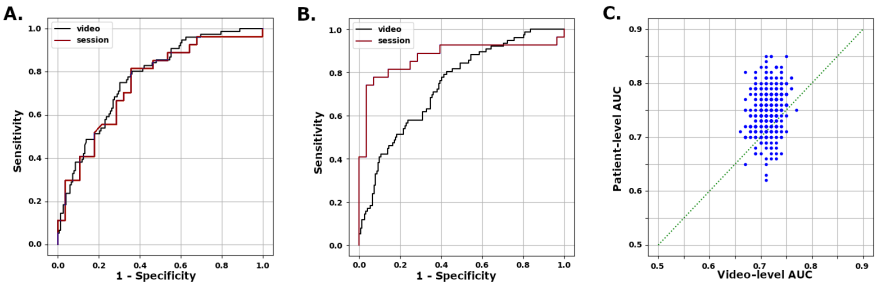

Fig. 1. (A, B) ROC curves for patient-level (red) and video-level (black). Per subplot, the two curves are from the best hyperopt iteration, as driven by video-level AUC as loss (**A**) or patient-level AUC as loss (**B**). **C**: Scatterplot of patient-level vs video-level AUCs. Note the lack of correlation between the two performance metrics, and the superiority of patient-level ROC in B.

DNNs are trained over several epochs, and an important optimization task is choosing the best epoch's model for testing or deployment. The standard method is to consult the validation loss curve (the training loss function applied to the validation set). Then the stopping point is chosen based on the optimal epoch in the validation loss curve, giving a model that is "best" in the narrow sense of the training loss function. But although the training loss is structured to generally reflect the medical context, it is also often shaped by reasons of convention, differentiability, or ease of implementation (e.g. built-in functions), and thus does not well capture clinical performance requirements.

Therefore this experiment assessed whether the validation loss is an optimal guide to choosing the best epoch in terms of performance on the clinical task. We trained a DNN for 40 epochs and then plotted, for each epoch, multiple FoMs: not just the validation loss but also other metrics that more tightly reflect the clinical use case. We also plotted per-epoch histograms of the validation exam scores to assess the stopping points selected by each FoM. Thus we can see whether validation loss chose a clinically-useful stopping point.

Methods:
Full model details are in [14]. Briefly: The twin detection pipeline consists of a CNN feature generator followed by an attention-based Multiple Instance Learning DNN module and a fully-connected classification layer. The CNN is pretrained on a separate US task, then the full pipeline is trained on a train-val fold in Pytorch [16] and Pytorch Lightning [7] with balanced cross-entropy (CE) loss. A US exam is the "bag", and frames from the various sweeps are the "instances". Given an exam as input, the model randomly selects 300 frames (Matérn sampling to avoid clumping) and delivers an exam-level score.

The Pytorch library automatically calculates a validation set loss curve using the training loss function. We also calculated and plotted alternate metrics tailored to the particular clinical goal:

1. Validation loss as returned by PyTorch (here, balanced CE). We plot 1 - CE, so higher is better.
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 the entire righthand region of the ROC is irrelevant. 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 $(1 - n/100)$. Code is in the online repo.
4. Sensitivity at 90% specificity. If the minimum 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. See [21] for a (non-ML) example from tuberculosis, where sensitivity is set to 90% and the corresponding specificity is evaluated.
5. Fisher distance, defined for two distributions by $\frac{|\mu_1 - \mu_2|}{\sigma_1 + \sigma_2}$ where $\mu_i,\ \sigma_i =$ mean and standard deviation for distribution i. We used right- and lefthand std devs, which are useful for asymmetric distributions (code in online repo).

Results:
The crucial finding is that these clinically-tailored FoMs, calculated at each epoch on the validation set, suggest a much later stopping point than the validation loss curve does. In addition, the per-epoch scatterplots and sensitivity-specificity indicate that the stopping point chosen by the training loss gives inferior results in the clinical task.

A 5-fold split was used for cross-validation, so 5 models were trained. Typical results are given below for Fold 3 (results for all folds are in Table 1 and Fig 4).

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 Fig 2 (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.

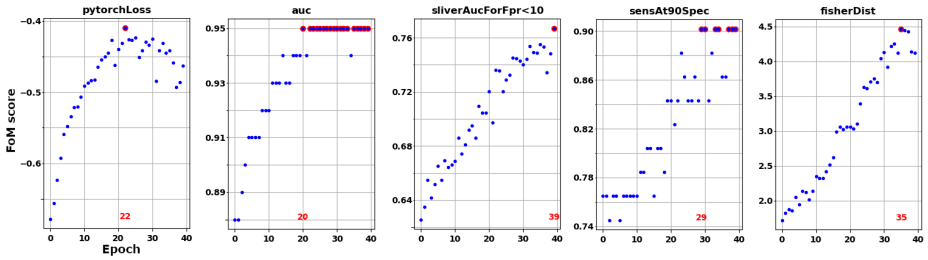

Fig. 2. FoMs (y-axis) at each epoch (x-axis). **L - R:** (1) -1 * Standard validation loss; (2) AUC; (3) 90% Sliver AUC; (4) Sensitivity at 90% specificity; (5) Fisher distance. x-axis: epoch. y-axis: value (higher is better). Highest scores marked in red. Clinically-relevant FoMs suggest much later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

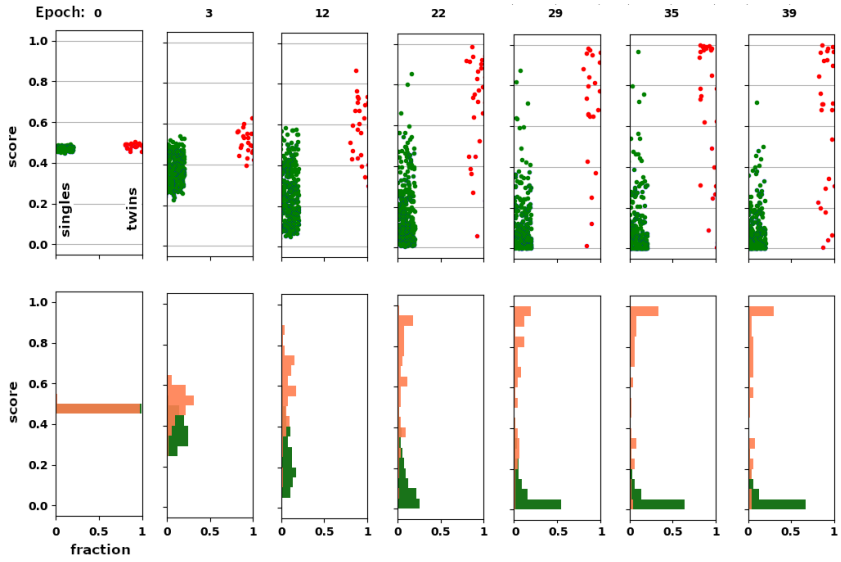

Fig. 3. Exam score distributions per epoch: **Top:** Score scatterplots: Singletons in green, twins in red. y-axis = model score. **Bottom:** Histograms of scores: Singletons in green, twins in red. Class separation continues to improve after epoch 22 (which has best validation loss), indicating that the model is still usefully learning.

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 Fig 3. Epoch 22, despite having the best validation loss, has poor separation compared to epochs 29 - 39.

All folds showed similar behavior, as seen in Table 1 (for further detail, see [14]). In particular, the validation loss was always much worse at the epoch chosen by clinically-based metrics, while the clinically-based metrics were better,

Table 1. Per fold, metrics for the best epochs as determined by Val Loss (1st row) and clinically-tailored metrics (2nd row). Values are ×100 except for Fisher distance. Deltas are percentages relative to the value at the val loss epoch: "+" means the clinically-tailored metrics had a better metric score, "-" means they had a worse score.

Fold,	**Epoch**	**val loss,**	Δ%	**AUC,**	Δ%	**sliver10,**	Δ%	**sens90,**	Δ%	**fisher,**	Δ%
0	29	-41		93		74		84		4.2	
	38	-42	-2	93	0	75	+1	84	0	4.7	+12
1	10	-62		78		40		48		1.1	
	38	-83	-34	83	+6	54	+35	66	+38	2.6	+150
2	50	-33		96		82		89		4.5	
	50	-38	-15	96	0	82	0	91	+2	5.1	+13
3	22	-41		95		74		84		3.1	
	39	-47	-15	95	0	77	+4	90	+7	4.1	+32
4	27	-43		92		79		85		3.7	
	39	-65	-28	90	-2	80	+1	85	+0	5.1	+38

often markedly so. The class separation was always better at the epoch chosen by clinically-based metrics, often markedly so, as seen in Fig 4.

Thus the later stopping point, urged by clinically-tailored FoMs was better for the clinical task. This crucial information is not revealed by the validation loss curve. Based on this we used a late stopping point for the final model, which hit the performance bar on the test set (sensitivity 89.7%, specificity 95.6%).

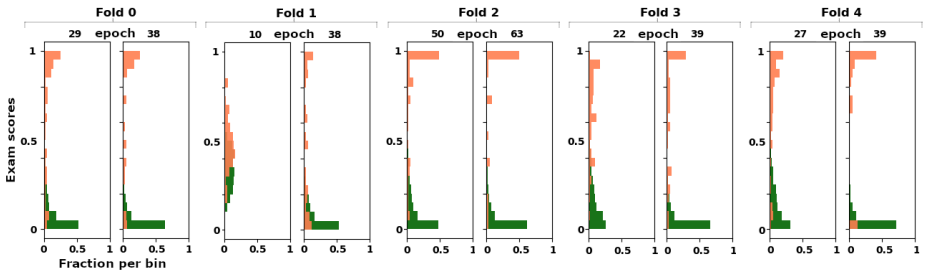

Fig. 4. For each fold, the histograms of model scores for twin and singleton exams at the epochs chosen by val loss (lefthand) and clinically-tailored metrics (righthand). In each fold, the epoch chosen by clinically-tailored metrics has better class separation (especially, singleton scores driven towards zero), sometimes markedly so (e.g. folds 1, 3, 4).

4 Discussion

Using validation loss to guide hyperparameter optimization is standard ML practice, because this loss meets the differentiability requirements of backprop and

gradient descent, and also perhaps because of habit, convention, and ease of use (e.g. built-in functions). However, ML for healthcare is a distinct field from traditional ML, and it has a distinct goal: model performance must meet rigorous clinical requirements that can be better captured by metrics customized to the clinical needs than by standard ML losses. This paper gave two examples of how metrics tailored to be clinically-relevant provide superior model optimization, in the sense of giving 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 optimization to that task.

Implementation of this approach is a bespoke process, because the metric selection details are highly task-specific, depending on both the particular clinical use case and the chosen ML framework. We view defining these clinically-relevant metrics as an essential technical task in ML for healthcare. Consultation with non-ML literature and domain experts, before any ML coding, can identify relevant quantitative clinical requirements. Our treatment of coagulation was grounded in the details of the "test and not treat" use case, as defined by the clinical field teams; and the ultrasound metrics were constructed to capture sens/spec targets developed through consultation with clinicians and use case landscaping (e.g. strategy documents from groups like the Gates Foundation).

We do not advocate patient-level training losses, nor all-in-one/end-to-end losses (though ML convention favors these), because an object-based loss plus an object-to-patient linkage is often more effective than (and not equivalent to) a direct patient-level loss, for three reasons: (i) The patient count is often much lower than object count, so an object-based loss has smoother behavior. (ii) Objects are often a fundamentally easier target; e.g. CNNs are beautifully suited to identifying object thumbnails, but directly identifying an "infected patient" is harder to frame as an ML task. (iii) Showing salient objects to the clinician serves as a trust-building tool and also opens options for aligning with clinical workflow and easing regulatory burden (e.g. decision support vs diagnosis).

This paper describes just two examples out of many possible {use case, model} pairings in ML for healthcare and carries no universal guarantee. However, the method has proven consistently valuable when tuning a model to meet challenging clinical performance requirements.

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).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Bergstra, J., Komer, B., Eliasmith, C., Yamins, D., Cox, D.: Hyperopt: a Python library for model selection and hyperparameter optimization. Computational Science and Discovery (2015)
2. Buturovic, L., Sweeney, T., et alia.: Development of machine learning classifiers for blood-based diagnosis and prognosis of suspected acute infections and sepsis. Proc 4th Machine Learning for Health Symposium, PMLR (2024)
3. Delahunt, C., Gachuhi, N., Horning, M.: Metrics to guide development of machine learning algorithms for malaria diagnosis. Frontiers Malaria (2024)
4. Djune-Yemeli, L., Delahunt, C., et alia.: Automated detection of *Loa loa*: a field trial in Cameroon. In review (2026), preprint at https://charlesdelahunt.github.io/assets/djuneYemeli_loa2026.pdf
5. Djune-Yemeli, L., Zoelko-Manego, R., Kamgno, J., et alia: A multi-site loiasis dataset of whole blood videos. MICCAI Open Data (2026)
6. Ehrmann, D., Eytan, D., et alia.: Making machine learning matter to clinicians: model actionability in medical decision-making. npj Digital Medicine (2023)
7. Falcon, W., the PyTorch Lightning team: PyTorch Lightning (2019), https://github.com/Lightning-AI/pytorch-lightning
8. Hicks, S., Sravanthi, P., et alia.: On evaluation metrics for medical applications of artificial intelligence. Nature Scientific Reports (2022)
9. Huang, C., Krumholz, H., et alia.: Performance metrics for the comparitive analysis of clinical risk prediction models employing machine learning. Circ Cardiovasc Qual Outcomes (2021)
10. Kamgno, J., Boussinesq, M., et alia.: A Test-and-Not-Treat Strategy for Onchocerciasis in *Loa loa*-Endemic Areas. NEJM (2017)
11. Kelly, C., King, D., et alia.: Key challenges for delivering clinical impact with artificial intelligence. BMC Medicine (2019)
12. de Leon Derby, M., Fletcher, D., et alia: Ntdscope: A multi-contrast portable microscep for disease diagnosis. PLOS NTDs (2025)
13. Maier-Hein, L., Reinke, A., et alia.: Metrics reloaded: Pitfalls and recommendations for image analysis validation. arXiv (2022), https://arxiv.org/abs/2206.01653
14. Mehanian, C., Shea, D., et alia.: Algorithms to detect fetal age features from ultrasound blind sweeps: documentation and codebase. Gates Open Research (2025), https://gatesopenresearch.org/gateways/ghlabs and https://github.com/Global-Health-Labs/OBUS-GHL
15. Misic, V., Rajaram, K., Gabel, E.: A simulation-based evaluation of machine learning models for clinical decision support: application and anlysis using hospital readmission. npj Digital Medicine (2021)
16. Paszke, A., Chintala, S., et alia.: Pytorch: An imperative style, high-performance deep learning library. Advances in Neural Information Processing Systems (2019)
17. Pedregosa, F., Duchesnay, E., et alia.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research (2011)
18. Pokaprakarn, T., Stringer, J., et alia.: AI estimation of gestational age from blind ultrasound sweeps in low-resource settings. NEJM Evid. (2022)
19. Reyna, M., Nsoesie, E., Clifford, G.: Rethinking algorithm performance metrics for artificial intelligence in diagnostic medicine. JAMA (2022)
20. Saha, S., Knandelwal, S., et alia.: MedTric: A clinically applicable metric for evaluation of multi-label computational diagnostic systems. PLOS One (2022)

21. WHO: Consolidated guidelines on tuberculosis (module 2, screening), page 26. World Health Organization (2021)
22. WHO: Generating evidence for artificial intelligence-based medical devices: A framework for training, validation and evaluation. World Health Organization (2021)
23. Wiens, J., Goldenberg, A., et alia.: Do no harm: a roadmap for responsible machine learning for health care. Nature Medicine (2019)

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 Fig 5. 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, $m_k = median({s_i}^0)$ 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 Fig 5 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).

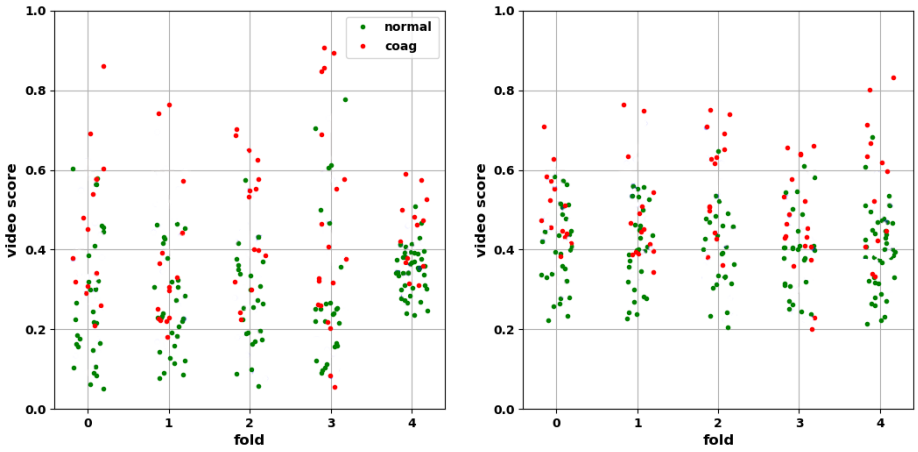

Fig. 5. Effect of z-scale alignment. y-axis = scores. Green = controls, red = coagulated. **Left:** Raw output scores per fold: Each fold's model requires a different threshold for a given specificity. **Right:** mapped scores per fold: One threshold gives the same specificity for all folds.

Appendix B: Python function defs

(i) 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 left-handed, using just the points to the right (or left) of the median.

```
import numpy as np
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
```

(ii) 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.')
      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

#----------------------

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(f'rh stdDev=0 on {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(f'lh stdDev=0 on np.sum(t0>0) cases < median.')
      adjX[inds] = t0 + p['canonicalMedian']
   # optional: could clip adjX at 0 and 1.

   return adjX
```

(iii) 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
```