Rashomon Sets for Prototypical-Part Networks: Editing Interpretable Models in Real-Time

Supplementary Material

A. Sampling Unrestricted Last Layers from the Rashomon Set Without Explicitly Computing the Hessian

Let $\mathbf{z} \in \mathbb{R}^m$ denote a vector of prototype similarities, such that $h(\mathbf{z}) \in \mathbb{R}^t$ is the vector of predicted class probabilities for input $\mathbf{z}$. The Hessian matrix of h with respect to $\mathbf{z}$ for a standard, multi-class logistic regression problem is given (written in terms of block matrices) as:

$$\begin{aligned}\mathbf{H} &= -\begin{bmatrix} h_1(\mathbf{z})(1-h_1(\mathbf{z}))\mathbf{z}\mathbf{z}^T & -h_1(\mathbf{z})h_2(\mathbf{z})\mathbf{z}\mathbf{z}^T & \dots & -h_1(\mathbf{z})h_t(\mathbf{z})\mathbf{z}\mathbf{z}^T \\ -h_2(\mathbf{z})h_1(\mathbf{z})\mathbf{z}\mathbf{z}^T & h_2(\mathbf{z})(1-h_2(\mathbf{z}))\mathbf{z}\mathbf{z}^T & \dots & -h_2(\mathbf{z})h_t(\mathbf{z})\mathbf{z}\mathbf{z}^T \\ \vdots & \vdots & \ddots & \vdots \\ -h_t(\mathbf{z})h_1(\mathbf{z})\mathbf{z}\mathbf{z}^T & -h_t(\mathbf{z})h_2(\mathbf{z})\mathbf{z}\mathbf{z}^T & \dots & h_t(\mathbf{z})(1-h_t(\mathbf{z}))\mathbf{z}\mathbf{z}^T \end{bmatrix} \\ &= \left(\mathbf{\Lambda} - h(\mathbf{z})h(\mathbf{z})^T\right) \otimes \mathbf{z}\mathbf{z}^T,\end{aligned}$$

where

$$\mathbf{\Lambda}_{ij} := \mathbf{1}[i=j]h_i(\mathbf{z}).$$

In order to sample a member of a Rashomon set that falls along direction $\mathbf{d} \in \mathbb{R}^{mt}$, we need to compute a value τ such that

$$\tau = \sqrt{\frac{2\kappa(\theta - \ell(\mathbf{w}_h^*))}{\mathbf{d}^T\mathbf{H}\mathbf{d}}},$$

with each term defined as in Section 3.3 of the main paper. The key computational constraint here is the operation $\mathbf{d}^T\mathbf{H}\mathbf{d}$, which involves the large Hessian matrix $\mathbf{H} \in \mathbb{R}^{mt \times mt}$. However, the quantity $\mathbf{d}^T\mathbf{H}\mathbf{d}$ can be computed without explicitly storing $\mathbf{H}$.

Let $\text{mat} : \mathbb{R}^{mt} \to \mathbb{R}^{m \times t}$ denote an operation that reshapes a vector into a matrix, and let $\text{vec} : \mathbb{R}^{m \times t} \to \mathbb{R}^{mt}$ denote the inverse operaton, which reshapes a matrix into a vector such that $\text{vec}(\text{mat}(\mathbf{d})) = \mathbf{d}$. We can then leverage the property of the Kronecker product that $(\mathbf{A} \otimes \mathbf{B})\text{vec}(\mathbf{C}) = \text{vec}(\mathbf{B}\mathbf{C}\mathbf{A}^T)$ to compute:

$$\begin{aligned}\mathbf{d}^T\mathbf{H}\mathbf{d} &= \mathbf{d}^T\left(\left(\mathbf{\Lambda} - h(\mathbf{z})h(\mathbf{z})^T\right) \otimes \mathbf{z}\mathbf{z}^T\right)\mathbf{d} \\ &= \mathbf{d}^T\left(\left(\mathbf{\Lambda} - h(\mathbf{z})h(\mathbf{z})^T\right) \otimes \mathbf{z}\mathbf{z}^T\right)\text{vec}\left(\text{mat}\left(\mathbf{d}\right)\right) \\ &= \mathbf{d}^T\text{vec}\left(\underbrace{\mathbf{z}\mathbf{z}^T}_{m\times m}\underbrace{\text{mat}\left(\mathbf{d}\right)}_{m\times t}\underbrace{\left(\mathbf{\Lambda} - h(\mathbf{z})h(\mathbf{z})^T\right)^T}_{t\times t}\right).\end{aligned}$$

As highlighted by the shape annotations, this operation can be computed by storing matrices of no larger than $\max(m^2, t^2, mt)$.

B. Positive Connections Only

For both memory efficiency and conceptual simplicity, instead of learning the Rashomon set over all last layers, we might want to consider one with some parameter tying. In particular, we allow prototypes to connect only with their own class. We begin with notation: let $S_c := \{i : \psi(i) = c\}$ where c is a class and $\psi(i)$ is a function that returns the class associated with prototype i. The constrained last layer forms predictions as:

$$h_c^{lin}(\mathbf{x}) := \sum_{i \in S_c} w_i x_i \tag{3}$$

$$h(\mathbf{x}) = \text{softmax}(h^{lin}(\mathbf{x})). \tag{4}$$

The first partial derivative of cross entropy w.r.t. a parameter w_i is:

$$\begin{aligned}
\frac{\partial}{\partial w_i} CE(f(\mathbf{x}), \mathbf{y}) &= \frac{\partial}{\partial w_i} \left[\sum_{k=1}^{C} y_k \left(h_k^{lin}(\mathbf{x}) - \ln \left(1 + \sum_{k=1}^{C} \exp(h_k^{lin}(\mathbf{x})) \right) \right) \right] \\
&= y_{\psi(i)} \frac{\partial}{\partial \omega_i^{(+)}} h_{\psi(i)}^{lin}(\mathbf{x}) - \frac{\exp(h_{\psi(i)}^{lin}(\mathbf{x}))}{1 + \sum_{k=1}^{C} \exp(h_k^{lin}(\mathbf{x}))} \frac{\partial}{\partial \omega_i^{(+)}} h_{\psi(i)}^{lin}(\mathbf{x}) \\
&= y_{\psi(i)} x_i - \frac{\exp(h_{\psi(i)}^{lin}(\mathbf{x}))}{1 + \sum_{k=1}^{C} \exp(h_k^{lin}(\mathbf{x}))} x_i \\
&= (y_{\psi(i)} - h_{\psi(i)}(\mathbf{x})) x_i.
\end{aligned}$$

For the second derivative, we have:

$$\begin{aligned}
\frac{\partial}{\partial w_j} \frac{\partial}{\partial w_i} CE(f(\mathbf{x}), \mathbf{y}) &= \frac{\partial}{\partial w_j} (y_{\psi(i)} - h_{\psi(i)}(\mathbf{x})) x_i \\
&= -x_i \frac{\partial}{\partial w_j} h_{\psi(i)}(\mathbf{x}) \\
&= -x_i \sum_{k=1}^{C} \frac{\partial h_{\psi(i)}}{\partial h_k^{lin}} \frac{\partial h_k^{lin}}{\partial w_j} \\
&= -x_i \frac{\partial h_{\psi(i)}}{\partial h_{\psi(j)}^{lin}} \frac{\partial h_{\psi(j)}^{lin}}{\partial w_j} \\
&= -x_i h_{\psi(i)}(\mathbf{x})(\delta_{\psi(i)\psi(j)} - h_{\psi(j)}(\mathbf{x})) x_j \\
&= -x_i x_j h_{\psi(i)}(\mathbf{x})(\delta_{\psi(i)\psi(j)} - h_{\psi(j)}(\mathbf{x}))
\end{aligned}$$

Packaged together, we then have

$$\begin{aligned}
\mathbf{H} &:= \begin{bmatrix}
-x_1^2 h_{\psi(1)}(\mathbf{x})(\delta_{\psi(1)\psi(1)} - h_{\psi(1)}(\mathbf{x})) & -x_2 x_1 h_{\psi(1)}(\mathbf{x})(\delta_{\psi(2)\psi(1)} - h_{\psi(2)}(\mathbf{x})) & \dots \\
-x_1 x_2 h_{\psi(2)}(\mathbf{x})(\delta_{\psi(1)\psi(2)} - h_{\psi(1)}(\mathbf{x})) & -x_2^2 h_{\psi(2)}(\mathbf{x})(\delta_{\psi(2)\psi(2)} - h_{\psi(2)}(\mathbf{x})) & \dots \\
\multicolumn{3}{c}{\dots} \\
-x_1 x_d h_{\psi(d)}(\mathbf{x})(\delta_{\psi(1)\psi(d)} - h_{\psi(1)}(\mathbf{x})) & -x_2 x_d h_{\psi(d)}(\mathbf{x})(\delta_{\psi(2)\psi(d)} - h_{\psi(2)}(\mathbf{x})) & \dots
\end{bmatrix} \\
&= \begin{bmatrix}
h_{\psi(1)}(\mathbf{x})(\delta_{\psi(1)\psi(1)} - h_{\psi(1)}(\mathbf{x})) & h_{\psi(1)}(\mathbf{x})(\delta_{\psi(2)\psi(1)} - h_{\psi(2)}(\mathbf{x})) & \dots \\
h_{\psi(2)}(\mathbf{x})(\delta_{\psi(1)\psi(2)} - h_{\psi(1)}(\mathbf{x})) & h_{\psi(2)}(\mathbf{x})(\delta_{\psi(2)\psi(2)} - h_{\psi(2)}(\mathbf{x})) & \dots \\
\multicolumn{3}{c}{\dots} \\
h_{\psi(d)}(\mathbf{x})(\delta_{\psi(1)\psi(d)} - h_{\psi(1)}(\mathbf{x})) & h_{\psi(d)}(\mathbf{x})(\delta_{\psi(2)\psi(d)} - h_{\psi(2)}(\mathbf{x})) & \dots
\end{bmatrix} \odot -\mathbf{x}\mathbf{x}^T
\end{aligned}$$

where $\odot$ denotes a Hadamard product.

C. Sampling From a Rashomon Set After Requiring Prototypes

Let J denote a set of prototype indices we wish to require, such that we constrain $[\text{coef}(\mathbf{p}_j) \geq \alpha] \forall j \in J$ where $\text{coef}(\mathbf{p}_j)$ denotes the last layer coefficient associated with prototype $\mathbf{p}_j$ and $\alpha \in \mathbb{R}$ is the minimum acceptable coefficient. Given this information, there are a number of ways to formulate requiring prototypes as a convex optimization problem. One such form, which we use, is to optimize the following:

$$\min_{\mathbf{w}_h} \frac{1}{2(\theta - \bar{\ell}(\mathbf{w}_h^*))} (\mathbf{w}_h - \mathbf{w}_h^*)^T \mathbf{H} (\mathbf{w}_h - \mathbf{w}_h^*) \quad \text{s.t. } [\mathbf{e}_j^T \mathbf{w}_h \geq \alpha] \forall j \in J.$$

We then check whether the result for $(\mathbf{w}_h - \mathbf{w}_h^*)^T \mathbf{H} (\mathbf{w}_h - \mathbf{w}_h^*)$ satisfies the constraint of being < 1. If so, we have found weights within our Rashomon set approximation. If not, we have proved no such solution exists.

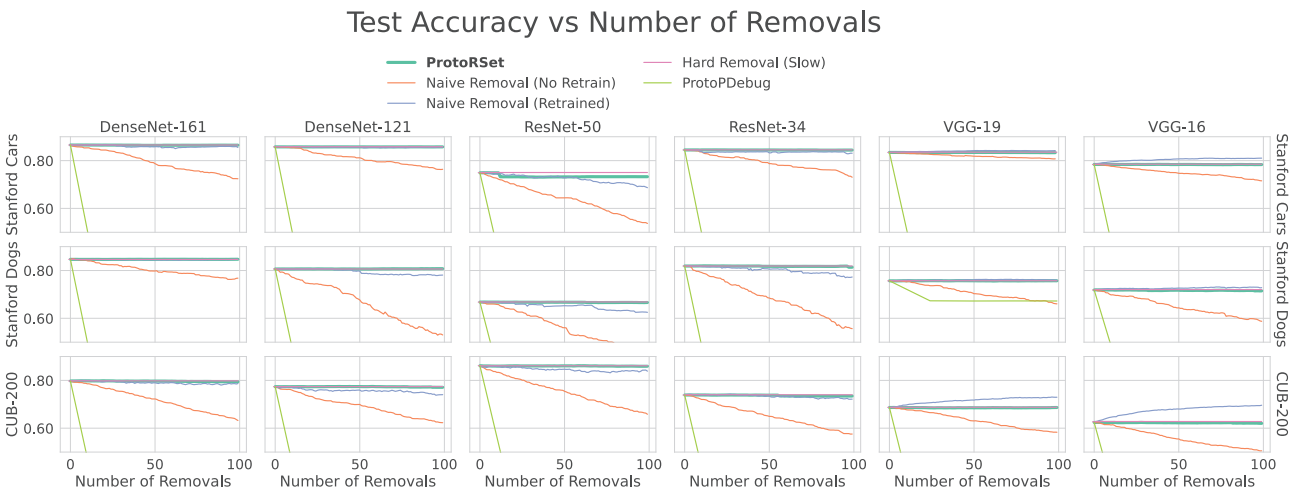

Figure 9. Change in test accuracy as random prototypes are removed. In all cases, we see that removing prototypes using ProtoRSet maintains or slightly improves the accuracy of the original model. With a reduced learning rate, we find that hard removal achieves comparable accuracy at the cost of an increased runtime (Figure 10).

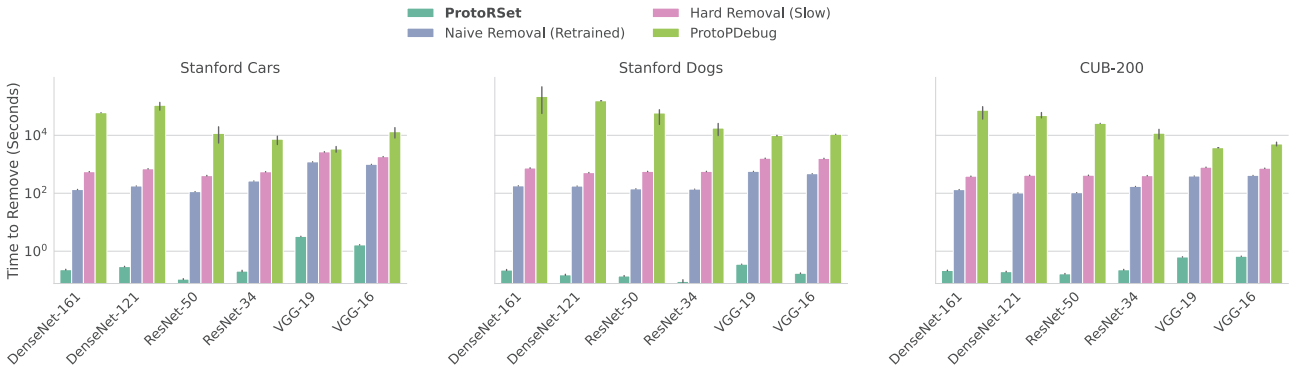

Figure 10. Time in seconds required to remove a single prototype, averaged over 100 iterations of removal. In all cases, ProtoRSet removes prototypes almost instantly. In contrast, removing a prototype then retraining the last layer can take orders of magnitude longer. Moreover, performing hard removal with a lower learning rate roughly quadruples the runtime of this method (note the logarithmic scale). We exclude naïve removal without retraining because it is simply updating a value in an array, and as such is nearly instantaneous.

D. Hard Removal Convergence

In our prototype removal experiments, one of the baseline methods considered is hard removal, where we strictly remove target prototypes and reoptimize all other last layer weights. Our primary experimental results showed that hard removal led to a slight decrease in accuracy, with a typical runtime of around 100 seconds. This result was found by optimizing hard removal using stochastic gradient descent with a learning rate of 0.001 and a maximum of 20,000 epochs, with early stopping when optimization failed to reduce loss by $\geq 1 \times 10^{-7}$.

We find that reducing the learning rate to 0.0001 led hard removal to match the accuracy of Proto-RSet (Figure 9), at the cost of multiplying the average runtime by a factor of 4.21 (Figure 10).

E. Training Details for Reference ProtoPNets

Proto-RSet is created by first training a reference ProtoPNet, then calculating a subset of the Rashomon set based on that reference model. Each reference ProtoPNet was trained using the Bayesian hyperparameter optimization framework described in [54]. We used cosine similarity for prototype comparisons. We trained each backbone using four training phases, as described in [7]: warm up, in which only the prototype layer and add-on layers (additional convolutional layers appended to the end of the backbone of a ProtoPNet) are optimized; joint, in which all backbone, prototype layer, and add-on layer parameters are optimized; project, in which prototypes are set to be exactly equal to their nearest neighbors in the latent space; and last-layer only, in which only the final linear layer is optimized.

We trained each model to minimize the following loss term:

$$\ell_{total} = CE + \lambda_{clst}\ell_{clst} + \lambda_{sep}\ell_{sep} + \lambda_{ortho}\ell_{ortho},$$

where each λ term is a hyperparameter coefficient, CE is the standard cross-entropy loss, ℓ_{clst} and ℓ_{sep} are the cluster and separation loss terms from [7] adapted for cosine similarity, and ℓ_{ortho} is orthogonality loss as defined in [11]. In particular, our adaptations of cluster and separation loss are defined as:

$$\ell_{clst} := \frac{1}{n}\sum_{i=1}^{n} \max_{j\in\{1,\ldots,m\}:\text{class}(\mathbf{p}_j)=y_i} g_j(f(\mathbf{X}_i))$$

$$\ell_{sep} := \frac{1}{n}\sum_{i=1}^{n} \max_{j\in\{1,\ldots,m\}:\text{class}(\mathbf{p}_j)\neq y_i} g_j(f(\mathbf{X}_i)),$$

where $\text{class}(\mathbf{p}_j)$ is a function that returns the class with which prototype $\mathbf{p}_j$ is associated. Note that, in our case, λ_{clst} is negative and λ_{sep} is positive. During last-layer only optimization, we additionally minimize the ℓ_1 norm of the final linear layer weights with a hyperparameter coefficient λ_{ℓ_1}. For each of our experiments, we performed Bayesian optimization with the prior distributions specified in Table 2 and used the model with the highest validation accuracy following projection as our reference ProtoPNet. We deviated from this selection criterion only for the user study; for that experiment, we selected a model with a large gap between train and validation accuracy (indicating overfitting to the induced confounding), and visually confirmed the use of confounded prototypes.

Hyperparameter Name	Distribution	Description
pre_project_phase_len	Integer Uniform; Min=3, Max=15	The number of warm and joint optimization epochs to run before the first prototype projection is performed
post_project_phases	Fixed value; 10	Number of times to perform projection and the subsequent last layer only and joint optimization epochs
phase_multiplier	Fixed value; 1	Amount to multiply each number of epochs by, away from a default of 10 epochs per training phase
lr_multiplier	Normal; Mean=1.0, Std=0.4	Amount to multiply all learning rates by, relative to the reference values in [54]
joint_lr_step_size	Integer Uniform; Min=2, Max=10	The number of training epochs to complete before each learning rate step, in which each learning rate is multiplied by 0.1
num_addon_layers	Integer Uniform; Min=0, Max=2	The number of additional convolutional layers to add between the backbone and the prototype layer
latent_dim_multiplier_exp	Integer Uniform; Min=-4, Max=1	If num_addon_layers is not 0, dimensionality of the embedding space will be multiplied by $2^{\text{latent\_dim\_multiplier}}$ relative to the original embedding dimension of the backbone
num_prototypes_per_class	Integer Uniform; Min=8, Max=16	The number of prototypes to assign to each class
cluster_coef	Normal; Mean=-0.8, Std=0.5	The value of λ_{clst}
separation_coef	Normal; Mean=0.08, Std=0.1	The value of λ_{sep}
l1_coef	Log Uniform; Min=0.00001, Max=0.001	The value of λ_{ℓ_1}
orthogonality_coef	Log Uniform; Min=0.00001, Max=0.001	The value of λ_{ortho}

Table 2. Prior distributions over hypeparameters for the Bayesian sweep used to train all reference ProtoPNets except for the one in the user study.

F. Hardware Details

All of our experiments were run on a large institutional compute cluster. We trained each reference ProtoPNet using a single NVIDIA RTX A5000 GPU with CUDA version 12.4, and ran other experiments using a single NVIDIA RTX A6000 GPU with CUDA version 12.4.

G. Confounding Details

To run our user study, we trained a reference ProtoPNet over a version of CUB200 in which a confounding patch has been added to each training image. For each training image coming from one of the first 100 classes, we add a color patch with width and height equal to $1/5$ those of the image to a random location in the image. The color of each patch is determined by the class of the training image; class 0 is set to the initial value fo the HSV color map in matplotlib, class 1 to the color $1/100^{\text{th}}$ further along this color map, and so on. A sample from each class with this confounding is shown in Figure 11. Images from the validation and test splits of the dataset were not altered.

Figure 11. One example image from each class in CUB200, with confounding patches added as in the user study. The first 100 classes receive random confounding patches, and the second 100 are unaltered.

H. User Study Details

In this section, we describe our user study in further detail.

H.1. Setup

We followed the procedure described in Appendix G to create a confounded version of the CUB200 training set, and fit a ProtoPNet with a VGG-19 backbone over this confounded set. We created this model using a Bayesian hyperparameter sweep, but rather than selecting the model with the highest validation accuracy, we selected the model with the largest gap between its train and validation accuracy, as this indicates overfitting to the confounding patches added to the training set. The hyperparameters used for the selected model are shown in Table 3. Each ProtoPDebug model trained in this experiment used these hyperparameters, and the prescribed "forbid loss" from [5] with a coefficient of 100 as in [5].

We visually examined the prototypes learned by this model to identify the minimal set of "gold standard" prototypes we expected users to remove. We found that, of 1509 prototypes, 27 were clearly confounded, with bounding boxes focused entirely on a confounding patch and global analyses illustrating that the three most similar training patches to each prototype were also confounding patches. Figure 12 shows all prototypes used by this model, as well as the 27 "gold standard" confounded prototypes we identified. Figure 13 shows a screenshot of the interface for our user study.

H.2. Recruitment and Task Description

Users were recruited from the crowd sourcing platform Prolific, and were tasked with removing prototypes that clearly focused on confounding patches. The complete, anonymized informed consent text shown to users – which provides instructions – was as follows:

> This research study is being conducted by Jon Donnelly, Hayden McTavish, and Stark Guo, doctoral student researchers; Dr. Alina Jade Barnett, a postdoctoral researcher; and their advisor, Dr. Cynthia Rudin, a faculty researcher; all at Duke University in Durham, NC. This research examines whether a novel tool can be used to remove obvious errors in an AI model. You will be asked to use the tool to remove obvious errors, a task that will take approximately 30 minutes and no longer than an hour. Your participation in this research study is voluntary.
>
> You may withdraw at any time, but you will only be paid if you remove at least 10% of color-patch prototypes and do not remove more than 2 non-color-patch prototypes. After completing the survey, you will be paid at a rate of $15/hour of work through the Prolific platform.
>
> In accordance with Prolific policies, we may reject your work if the task was not completed correctly, or the instructions were not followed. A bonus payment of $10 will be offered if all errors are identified and corrected within 30 minutes.
>
> There are no anticipated risks or benefits to participants for participating in this study. Your participation is anonymous as we will not collect any information that the researchers could identify you with.
>
> If you have any questions about this study, please contact Jon Donnelly at jon.donnelly@duke.edu and include the term "Prolific Participant Question" in your email subject line. For questions about your rights as a participant contact the Duke Campus Institutional Review Board at campusirb@duke.edu referencing protocol #2025-0185.
>
> If you consent to participate in this study please click the "⟩⟩" below to begin the survey.

A total of 51 Prolific users completed our survey. Of these, 20 submissions were rejected for either 1) failing to identify at least 10% of the gold standard confounded prototypes (i.e., identified 2 or fewer) or 2) removing more than 2 prototypes drawn from images with no confounding patch. A total of 4 users qualified for and received the $10 bonus payment.

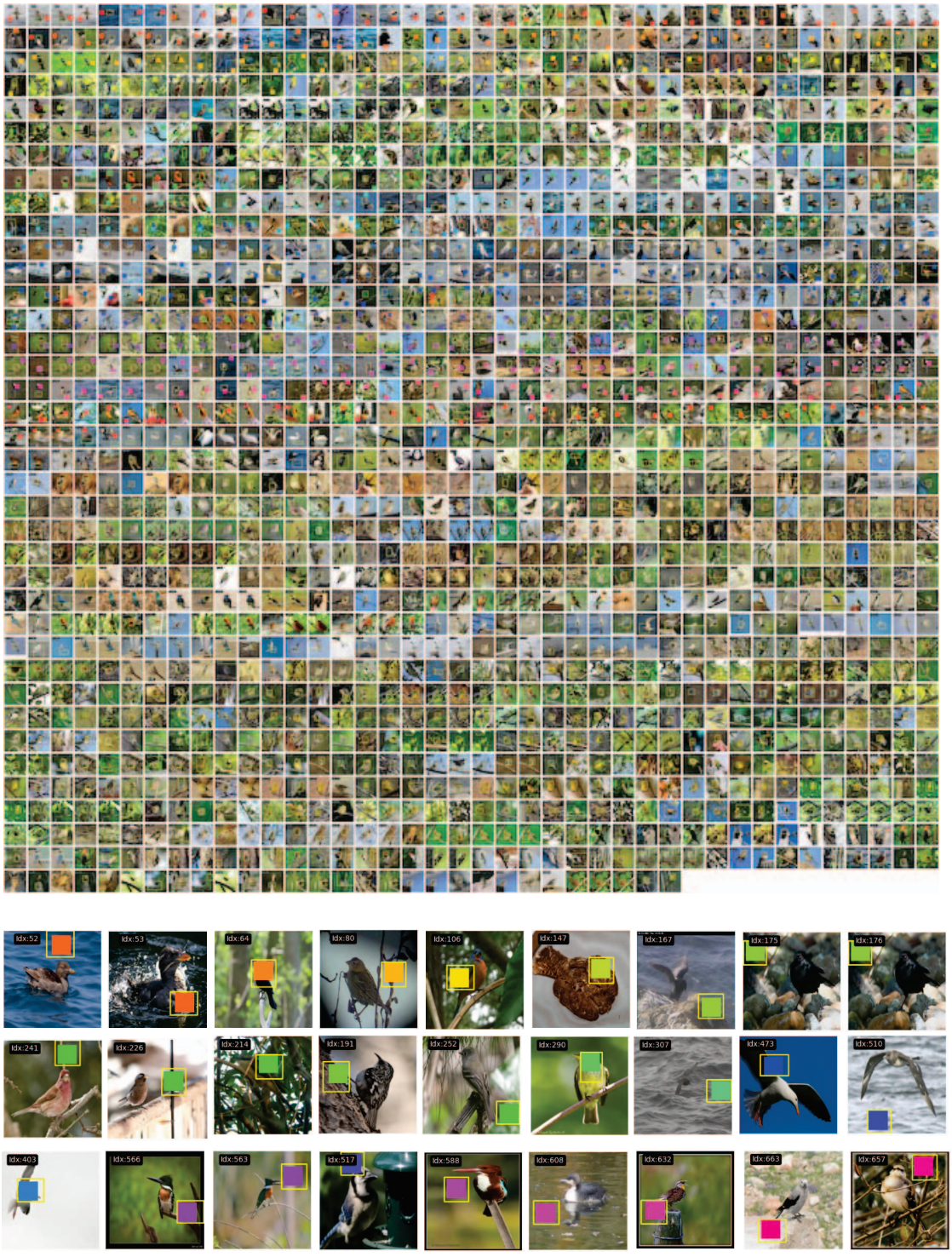

Figure 12. (**Top**) All 1509 prototypes used by the confounded model from the user study. (**Bottom**) The 27 clearly confounded prototypes we identified as the "gold standard" for users to remove.

Hyperparameter Name	Value	Description
pre_project_phase_len	11	The number of warm and joint optimization epochs to run before the first prototype projection is performed
post_project_phases	10	Number of times to perform projection and the subsequent last layer only and joint optimization epochs
phase_multiplier	1	Amount to multiply each number of epochs by, away from a default of 10 epochs per training phase
lr_multiplier	0.89	Amount to multiply all learning rates by, relative to the reference values in [54]
joint_lr_step_size	8	The number of training epochs to complete before each learning rate step, in which each learning rate is multiplied by 0.1
num_addon_layers	1	The number of additional convolutional layers to add between the backbone and the prototype layer
latent_dim_multiplier_exp	-4	If num_addon_layers is not 0, dimensionality of the embedding space will be multiplied by $2^{\text{latent\_dim\_multiplier}}$ relative to the original embedding dimension of the backbone
num_prototypes_per_class	14	The number of prototypes to assign to each class
cluster_coef	-1.2	The value of λ_{clst}
separation_coef	0.03	The value of λ_{sep}
l1_coef	0.00001	The value of λ_{ℓ_1}
orthogonality_coef	0.0004	The value of λ_{ortho}

Table 3. Hyperparameter values used to construct the confounded ProtoPNet for the user study.

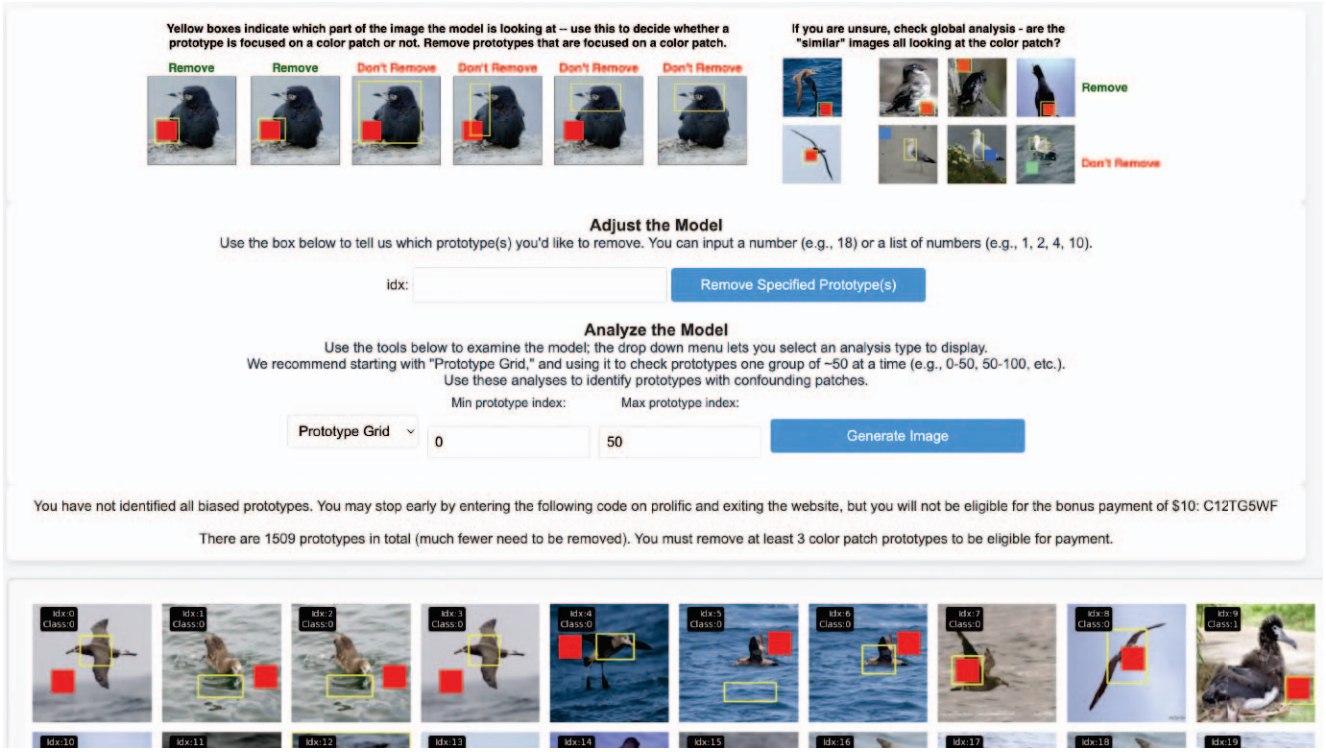

Figure 13. A screenshot of the interface for our user study. Users were tasked with identifying and removing confounded prototypes. The top right corner of the instructions shows two prototypes (left column) and the three most similar patches to each prototype (right three columns); users are instructed to remove the top prototype but not the bottom one because the top consistently focuses on confounding patches, while the bottom does not.

I. Proto-RSet Meets Feedback Better than ProtoPDebug

Here, we use results from the user study to highlight a key advantage in Proto-RSet over ProtoPDebug: Proto-RSet guarantees that user constraints are met when possible. We selected a random user who successfully identified all "gold standard" confounded prototypes (see Figure 12). With this user's feedback, we created two models: one using ProtoPDebug and one from Proto-RSet. We examined all prototypes used by each model. In both models, we identified every prototype for which the majority of the prototype's bounding box focused on a confounding color patch. Figure 14 shows every confounded prototype from the original reference model, the model produced by ProtoPDebug, and the model produced by Proto-RSet. Note that, for the original reference model, we visualize every prototype marked as confounded by the user plus those we identified as confounded. We mark "false positive" user feedback (prototypes that were removed, but do not focus on confounding patches) with a red "X," and do not count them toward the total number of confounded prototypes for the original model.

We found that a substantial portion of the prototypes used by ProtoPDebug – 6.3% of all prototypes – still used confounded prototypes despite user feedback. In contrast, only 0.5% of the prototypes used by Proto-RSet focused on confounding patches. Moreover, the 7 confounded prototypes that remain after applying Proto-RSet would not be present if the user had identified them; this is not guaranteed for the 21 confounded prototypes of ProtoPDebug.

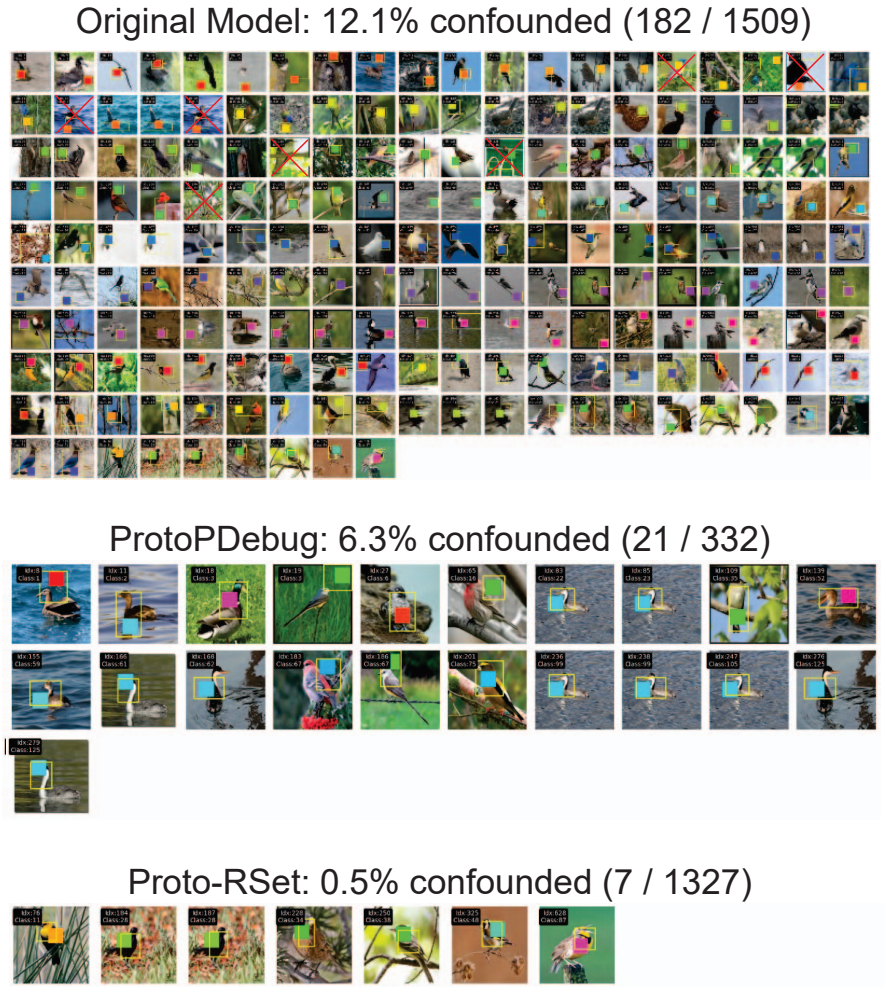

Figure 14. All confounded prototypes from each of a reference ProtoPNet, a model produced by ProtoPDebug, and a model produced by Proto-RSet. A prototype is considered confounded if the majority of the bounding box is covered by a confounding patch. Proto-RSet and ProtoPDebug were shown the feedback obtained from the same random user. A substantially larger proportion of the prototypes from ProtoPDebug are confounded than the prototypes from Proto-RSet.

J. Detailed Parameter Values for Proto-RSet

Here, we briefly describe the parameters used when computing the Proto-RSet's from each experimental section. Each Rashomon set was defined with respect to an ℓ_2 regularized cross entropy loss $\bar{\ell}(\mathbf{w}_h) = CE(\mathbf{w_h}) + \lambda\|\mathbf{w}_h\|_2$, with $\lambda = 0.0001$. In all of our experiments except for the user study, we used a Rashomon parameter of $\theta = 1.1\bar{\ell}(\mathbf{w}_h^*)$; for the user study, we set $\theta = 1.2\bar{\ell}(\mathbf{w}_h^*)$ to account for the fact that the training set was confounded, meaning training loss was a less accurate indicator of model performance. We estimated the empirical loss minimizer $\mathbf{w}_h^*$ using stochastic gradient descent with a learning rate of 1.0 for a maximum of $5,000$ epochs. We stopped this optimization early if loss decreased by no more than 10^{-7} between epochs.

K. Evaluating Proto-RSet's Ability to Require Prototypes

In this section, we evaluate the ability of Proto-RSet to require prototypes using the method described in the main paper. We match the experimental setup from the main prototype removal experiments; namely, we evaluate Proto-RSet over three fine-grained image classification datasets (CUB-200 [48], Stanford Cars [25], and Stanford Dogs [22]), with ProtoPNets trained on six distinct CNN backbones (VGG-16 and VGG-19 [44], ResNet-34 and ResNet-50 [17], and DenseNet-121 and DenseNet-161 [20]) considered in each case. For each dataset-backbone combination, we applied the Bayesian hyperparameter tuning regime of [54] for 72 GPU hours and used the best model found in terms of validation accuracy after projection as a reference ProtoPNet. For a full description of how these ProtoPNets were trained, see Appendix E.

In each of the following experiments, we start with a well-trained ProtoPNet and iteratively require that up to 100 random prototypes have coefficient greater than τ, where τ is the mean value of the non-zero entries in the reference ProtoPNet's final linear layer. If we find that no protoype can be required from the model while remaining in the Rashomon set, we stop this procedure early.

We consider two baselines for comparison in each of the following experiments: naïve prototype requirement, in which the correct-class last-layer coefficient for each required prototype is set to τ and no further adjustments are made to the model, and naïve prototype requirement with retraining, in which a similar requirement procedure is applied, but the last layer of the ProtoPNet is retrained (with all other parameters held constant) after prototype requirement. In the second baseline, we apply an ℓ_1 *reward* to coefficients corresponding to required prototypes to prevent the model from forgetting them and train the last layer until convergence, or for up to 5,000 epochs – whichever comes first. By ℓ_1 reward, we mean that this quantity is *subtracted* from the overall loss value, so that a larger value for these entries decreases loss.

Proto-RSet Produces Accurate Models. Figure 15 presents the test accuracy of each model produced in this experiment as a function of the number of prototypes required. We find that, across all six backbones and all three datasets, **Proto-RSet maintains test accuracy as constraints are added.** This stands in stark contrast to direct requirement of undesired prototypes, which dramatically reduces performance in all cases. We suspect this is because the ℓ_1 reward is unbounded – the model can continuously reduce loss by increasing the required coefficients at the cost of accuracy. While this is a flaw in the ℓ_1 reward term, it points to the difficulty of imposing a requirement constraint. The test accuracy of models produced by Proto-RSet is maintained **in all cases.**

Proto-RSet is Fast. Figure 16 presents the time necessary to require a prototype using Proto-RSet versus naïve prototype requirement with retraining. We observe that, across all backbones and datasets, Proto-RSet requires prototypes orders of magnitude faster than retraining a model. In contrast to prototype removal, we observe that the time necessary to require prototypes is heavily dependent on the reference ProtoPNet. Naïve requirement without retraining tends to be faster, but at the cost of substantial decreases in accuracy, as shown in Figure 15.

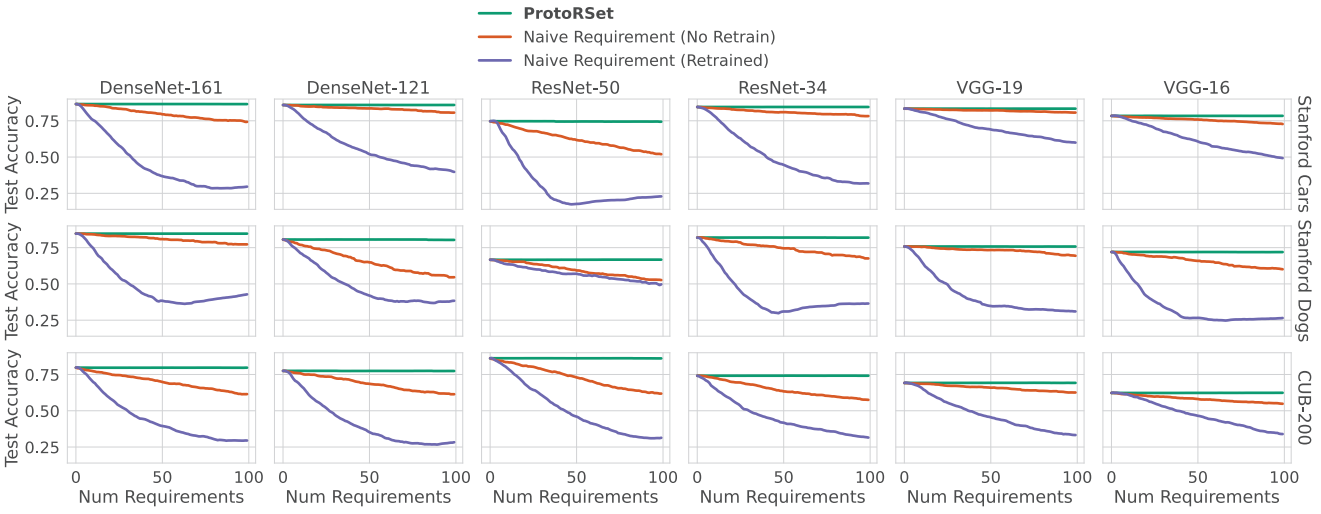

Figure 15. Change in test accuracy as random prototypes are required. In all cases, we see that requiring prototypes using ProtoRSet maintains or slightly improves the accuracy of the original model. Naïvely requiring prototypes either with or without retraining, on the other hand, dramatically reduces model performance.

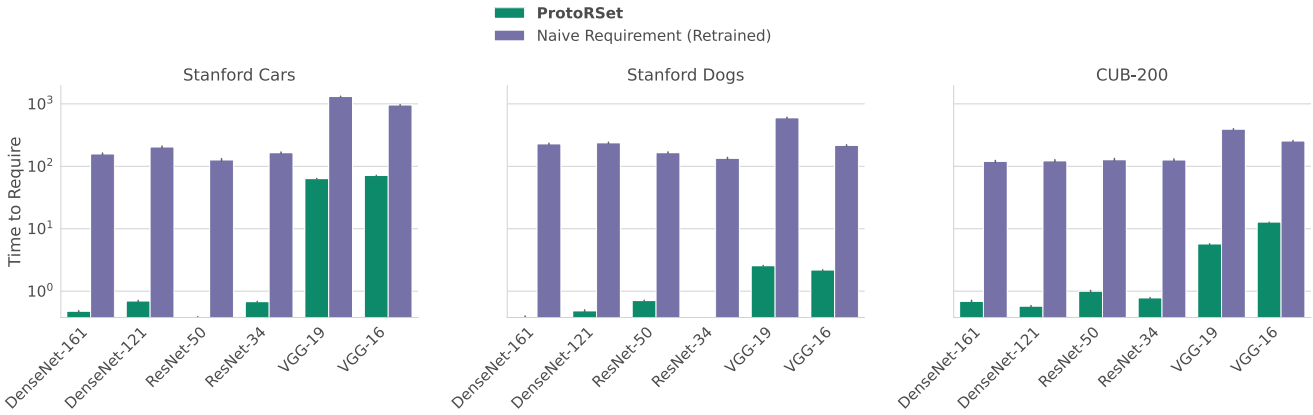

Figure 16. Time in seconds to produce a model meeting requirement constraints, averaged over 100 iterations of requirement. In all cases, ProtoRSet meets the prototype requirement constraint faster than the naïve method (requiring a prototype then retraining the last layer). We exclude naïve requirement without retraining from the chart because it is simply updating a value in an array, and as such is nearly instantaneous.

L. Evaluating the Sampling of Additional Prototypes

In this section, we evaluate whether the prototype sampling mechanism described in Subsection 3.4 allows users to impose additional constraints by revisiting the skin cancer classification case study from Subsection 4.2. In Subsection 4.2, we attempted to remove 10 of the original 21 prototypes used by the model, and found that we were unable to remove one of these prototypes without sacrificing accuracy.

We sampled 25 additional prototypes using the mechanism described in Subsection 3.4 using the same reference ProtoPNet, and analyzed the resulting model for any additional background or duplicate prototypes. We found that 9 out of the 25 additional prototypes focused primarily on the background. We removed all 10 of the original target prototypes and these 9 new ones, resulting in a model with 27 prototypes in total, as shown in Figure 17. **By sampling additional prototypes, Proto-RSet was able to meet user constraints that were not possible given the original set of prototypes.** This model achieved identical test accuracy to the original model. Recalling that accuracy dropped substantially when removing the 10 target prototypes without sampling alternatives, this demonstrates that sampling more prototypes increases the flexibility of Proto-RSet.

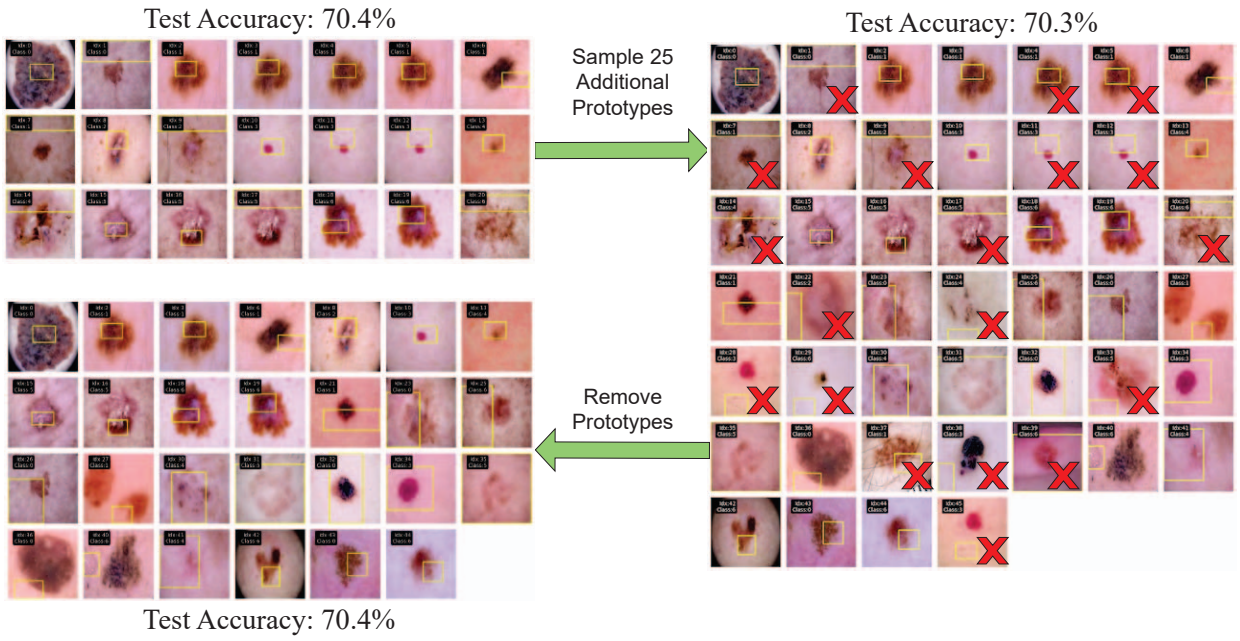

Figure 17. The process followed to remove all desired prototypes from a model for skin cancer classification. Starting from a reference ProtoPNet with 21 prototypes (Top Left), we first sampled 25 additional prototypes to produce a set of 46 candidate prototypes (Right). We removed 19 prototypes from this set of candidates, each of which is annotated with a red "X". Removing these prototypes using Proto-RSet resulted in a model with 27 non-confounded prototypes that matched the test accuracy of the original model.

M. Additional Visualizations of Model Refinement

In this section, we provide additional visualizations of model reasoning before and after user constraints are added. We consider the ResNet-50 based ProtoPNet trained for CUB-200 that we used for our experiments in the main body. The changes in prototype class-connection weight show that substantial changes to model reasoning are achieved when editing models using Proto-RSet.

Figure 18 presents the reasoning process of the model in classifying a Least Auklet before and after an arbitrary prototype is removed. In this case, we see that Proto-RSet substantially adjusted the weight assigned to prototypes of the same class as the removed prototype, and slightly adjusted the coefficient on prototypes from the class with the second highest logit – in this case, the Parakeet Auklet class. Notably, the weight assigned to prototype 12 increases from 5.014 to 9.029.

Figure 19 presents the reasoning process of the model in classifying a Black-footed Albatross before and after we require a large coefficient be assigned to an arbitrary prototype. We see that Proto-RSet substantially decreased the weight assigned to all other prototypes of the same class as the upweighted prototype, and moderately changed the coefficient on prototypes from the class with the second highest logit – in this case, the Sooty Albatross class. In particular, the weight assigned to prototype 6 was increased from 6.934 to 7.11, and the weight on prototype 5 was decreased from 6.87 to 6.786.

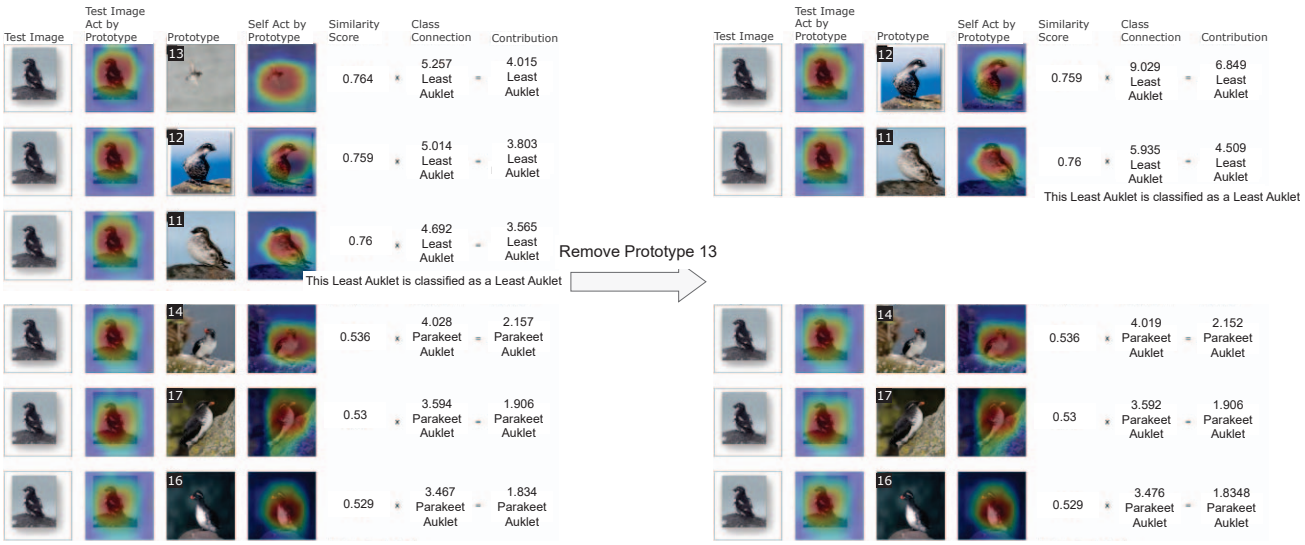

Figure 18. Reasoning process for a ProtoPNet classifying a Least Auklet before (Left) and after (Right) prototype 13 is removed using Proto-RSet. In each column, we present the reasoning for the predicted class (Top) and the class with the second highest logit (Bottom).

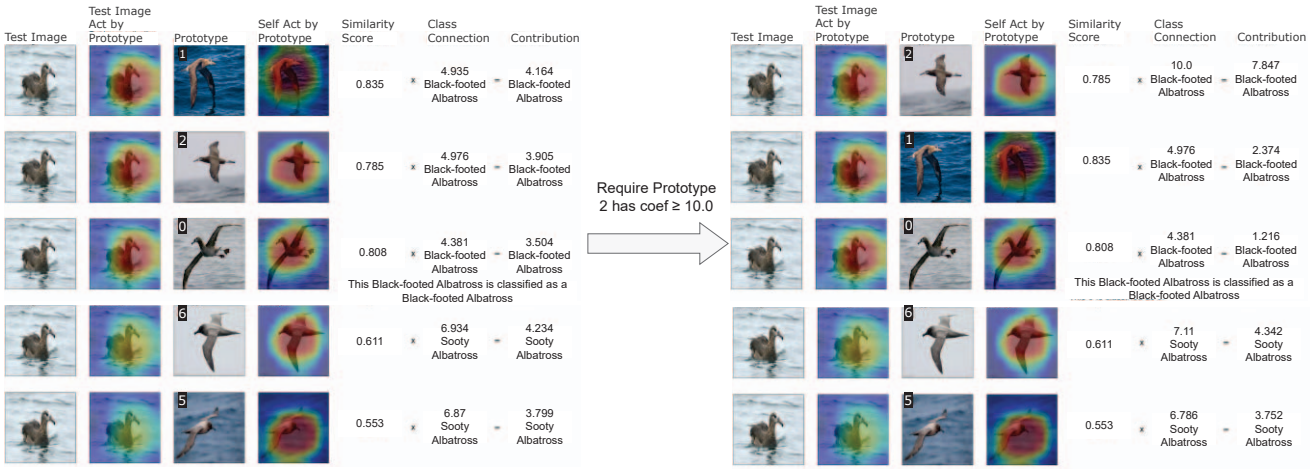

Figure 19. Reasoning process for a ProtoPNet classifying a Black-footed Albatross before (Left) and after (Right) the model is constrained such that prototype 2 has a coefficient of at least 10 using Proto-RSet. In each column, we present the reasoning for the predicted class (Top) and the class with the second highest logit (Bottom).