PAC Synthesis of Machine Learning Programs

Osbert Bastani
University of Pennsylvania
obastani@seas.upenn.edu

Abstract

We study the problem of synthesizing programs that include machine learning components such as deep neural networks (DNNs). We focus on statistical properties, which are properties expected to hold with high probability—e.g., that an image classification model correctly identifies people in images with high probability. We propose novel algorithms for sketching and synthesizing such programs by leveraging ideas from statistical learning theory to provide statistical soundness guarantees. We evaluate our approach on synthesizing list processing programs that include DNN components used to process image inputs, as well as case studies on image classification and on precision medicine. Our results demonstrate that our approach can be used to synthesize programs with probabilistic guarantees.

1 Introduction

Machine learning has recently become a powerful tool for solving challenging problems in artificial intelligence. As a consequence, there has been a great deal of interest in incorporating *machine learning components* such as deep neural networks (DNNs) into real-world systems, ranging from healthcare decision-making [1, 2, 3], to robotics perception and control [4, 5]. In these domains, there is often a need to ensure correctness properties of the overall system. To reason about such properties, we need to reason about properties of the incorporated machine learning components, focusing on *statistical properties* that should hold with high probability with respect to the distribution of inputs—e.g., we may want to ensure that the DNN detects 95% of pedestrians.

We propose a framework for synthesizing programs that incorporate machine learning components while satisfying statistical correctness properties. Our framework consists of two components. First, it includes a novel *statistical sketching* algorithm, which builds on the concept of sketching [6] to provide statistical guarantees. At a high level, it takes as input a *sketch* (i.e., a program with certain parts left unspecified), annotated with specifications encoding statistical properties that are expected to hold, as well as holes corresponding to real-valued thresholds for making decisions (e.g., the confidence level at which to label an image as containing a pedestrian or to diagnose a patient with a disease). Since statistical properties depend on the data distribution, it additionally takes as input a labeled dataset of training examples (separate from those used to train the DNNs). Then, our algorithm selects values to fill the holes in the sketch so all the given specifications are satisfied.

Second, our framework uses this sketching algorithm in conjunction with a syntax-guided synthesizer [7] to synthesize programs in a specific domain that provably satisfy statistical guarantees. Our strategy is to first synthesize a sketch whose specifications encode overall correctness, and then apply our sketching algorithm to fill the holes in the sketch so these specifications are satisfied.

We have implemented our approach as StatCoder. We evaluate its ability to synthesize list processing programs satisfying statistical properties, where the program inputs are images, and DNN components are used to classify objects in these images. We also perform three case studies of our sketching algorithm: one on ImageNet classification and another on a medical prediction task.

35th Conference on Neural Information Processing Systems (aiplans 2021), Sydney, Australia.

2 Overview

Here, we provide an overview of our approach; we give details on our sketch language in Section A, our sketching algorithm in Section C, and our synthesis algorithm in Section D.

Statistical sketching. We assume given a DNN component $f : \mathcal{X} \to [0, 1]$ that, given an image $x \in \mathcal{X}$, predicts whether x contains a person. In particular, $f(x)$ is a score indicating its confidence that x contains a person; higher score means more likely to contain a person. We do not assume the the scores are reliable—e.g., they may be overconfident. We assume that the ground truth label $y^* \in \mathcal{Y} = \{0, 1\}$ indicates whether x contains a person. For example, $f(x)$ may be the probability that an image contains a person according to a pretrained DNN such as ResNet [8]; then, the goal is to tailor this DNN to the current task in a way that provides correctness guarantees.

In particular, our goal is to choose a threshold $c \in [0, 1]$ such that the program returns that the given image x contains a person if $f(x)$ has confidence at least $1 - c$—i.e., $f(x) \geq 1 - c$, or equivalently, $1 - f(x) \leq c$. Furthermore, we want c to be correct in the following sense:

$$(y^* = 1) \Rightarrow (1 - f(x) \leq c)$$

That is, if the image contains a person (i.e., $y^* = 1$), then the classifier should say so (i.e., $1 - f(x) \leq c$). We do not require the converse—i.e., the program may incorrectly conclude that an image contains a person even if it does not. That is, we want soundness (i.e., no false negatives) but not necessarily completeness (i.e., no false positives). However, we cannot guarantee soundness for every x; instead, we guarantee it with high probability—i.e., we say c is *ϵ-approximately correct* if

$$\mathbb{P}_{p(x,y^*)}\big(y^* = 1 \Rightarrow 1 - f(x) \leq c\big) \geq 1 - \epsilon. \tag{1}$$

In adddition, our algorithm relies on training examples $\vec{z} = \{(x_1, y_1^*), ..., (x_n, y_n^*)\}$ to choose c, where $(x_i, y_i^*) \sim p$ are i.i.d. samples from $p(x, y^*)$. Thus, as with probably approximately correct (PAC) bounds from statistical learning theory [9, 10], we need to additional allow a possibility that our algorithm fails altogether due to the randomness in our training examples $\vec{z}$. In particular, consider an algorithm A that chooses $c = A(\vec{z})$; then, we say A is (ϵ, δ)-*PAC* if

$$\mathbb{P}_{p(\vec{z})}\big(A(\vec{z}) \text{ is } \epsilon\text{-approximately correct}\big) \geq 1 - \delta$$

where $p(\vec{z})$ is the distribution over the training examples $\vec{z}$, and $\delta \in \mathbb{R}_{>0}$ is another user-provided confidence level. Then, given the sketch, a value $\delta \in \mathbb{R}_{>0}$, and a dataset $\vec{z}$, our algorithm synthesizes a value of c to fill `??1` such that c is ϵ-approximately correct with probability at least $1 - \delta$.

Our sketching algorithm formulates the problem of synthesizing c as a binary classification problem. In particular, for each training example $(x, y^*) \in \mathcal{X} \times \mathcal{Y}$, it constructs the value

$$z = \begin{cases} 1 - f(x) & \text{if } y^* = 1 \\ -\infty & \text{otherwise.} \end{cases}$$

Note that for a given choice of c, the value $w = \mathbb{1}(z \leq c)$ indicates whether c is correct for (x, y^*)—either $y^* = 0$, in which case since $z = -\infty$ so the antecedent $y^* = 1$ in (1) trivially holds, or $y^* = 1$, in which case $z = 1 - f(x)$ and w indicates whether the consequent of (1) holds.

Thus, thinking of z as a random function of (x, y^*), our goal is to choose c such that $\mathbb{P}_{p(z)}(z \leq c) \geq 1 - \epsilon$. In other words, c can be thought of as a threshold in a binary classification problem, where the input is z and the desired label is always $w = 1$. The difference from typical binary classification is that we do not want to minimize the number of errors; instead, our goal is to minimize c subject to a constraint on the error rate—i.e., we want the most optimistic value of c that satisfies the PAC guarantee. In Section C, we provide an algorithm for solving this problem with PAC guarantees.

Synthesis algorithm. Next, suppose we want to synthesize a program that counts the number of people in a list of images $\ell = (x_1, ..., x_n)$. Intuitively, we can do so by writing a simple list processing program around our DNN for detecting people. In particular, letting

$$(\mathsf{predict}_{\mathsf{person}}\ x) = \mathbb{1}(1 - f(x) \leq ??)$$

be our DNN component, where the detection threshold has been left as a hole, then the sketch

$$\tilde{P}_{\text{ex}} = (\mathsf{fold}\ +\ (\mathsf{map}\ \mathsf{predict}_{\mathsf{person}}\ \ell)\ 0)$$

counts the number of people in ℓ. Given a few input-output examples along with the ground truth labels for each image, we can use a standard enumerative synthesizer to compute the sketch $\tilde{P}_{\text{ex}}$, assuming $\mathsf{predict}_{\mathsf{person}}$ returns the ground truth label. In particular, this sketch has a single hole in the DNN component $\mathsf{predict}_{\mathsf{person}}$ that remains to be filled.

Note that $\tilde{P}_{\text{ex}}$ evaluates correctly if $\mathsf{predict}_{\mathsf{person}}$ returns the ground truth label, but in general, it may make mistakes. Thus, the correctness property for the synthesized program P_{ex} needs to account for the possibility that $\mathsf{predict}_{\mathsf{person}}$ may return incorrectly. Mirroring the correctness property for a single prediction, suppose we want a program P_{ex} that conservatively overestimates the number of people in ℓ.[1] In particular, given confidence levels $\epsilon, \delta \in \mathbb{R}_{>0}$, we say a completion P_{ex} of $\tilde{P}_{\text{ex}}$ is *ϵ-approximately correct* if

$$\mathbb{P}_{p(\alpha)}([\![P_{\text{ex}}]\!]_\ell \geq y^*) \geq 1 - \epsilon,$$

where $\alpha = (\ell, y^*)$ is an example, and $[\![P]\!]_\ell$ denotes the output of running program P on input ℓ. Then, we say our synthesis algorithm is *(ϵ, δ)-probably approximately correct (PAC)* if

$$\mathbb{P}_{p(\vec{\alpha})}(A(\tilde{P}_{\text{ex}}, \vec{\alpha}) \text{ is } \epsilon\text{-approximately correct}) \geq 1 - \delta,$$

where $P_{\text{ex}} = A(\tilde{P}_{\text{ex}}, \vec{\alpha})$ is the program synthesized using our algorithm and training examples $\vec{\alpha}$.

Using our statistical sketching algorithm, we can provide (ϵ', δ')-PAC guarantees on $\mathsf{predict}_{\mathsf{person}}$ for any $\epsilon', \delta' \in \mathbb{R}_{>0}$; thus, the question is how to choose (i) the appropriate specification, (ii) the parameters of this specification, and (iii) the confidence levels ϵ', δ'. These choices depend on the specification that we want to ensure for the synthesized program P_{ex}. In our example, we can use the specification above—i.e., that $\mathsf{predict}_{\mathsf{person}}$ returns 1 with high probability if there is a person:

$$(\mathsf{predict}_{\mathsf{person}}\ x) = \mathbb{1}(1 - f(x) \leq ??)\ \{y^* = 1\}^{|}_{\epsilon'}.$$

In general, the specification on $\mathsf{predict}_{\mathsf{person}}$ may have additional parameters (in particular, for real-valued predictions, an error tolerance e).

Next, we need to choose ϵ', δ'. While there is only one hole, $\mathsf{predict}_{\mathsf{person}}$ is executed multiple times (assuming $\mathsf{length}(\ell) > 1$). We need to choose ϵ' and δ' so that with high probability, $\mathsf{predict}_{\mathsf{person}}$ is correct for *all* applications. For simplicity, we assume given an upper bound $N \in \mathbb{N}$ on the maximum possible length of ℓ (we discuss how we might remove this assumption in Section E). Given N, we take $\epsilon' = \epsilon/N$ and $\delta' = \delta/N$; then, we use our sketching algorithm to synthesize c to fill the hole in $\mathsf{predict}_{\mathsf{person}}$. By a union bound, for a given list ℓ, all applications of $\mathsf{predict}_{\mathsf{person}}$ are correct with probability at least $1 - \epsilon$, and this property holds with probability at least $1 - \delta$. Under this event, P_{ex} returns correctly—i.e., P_{ex} satisfies the desired (ϵ, δ)-PAC guarantee.

3 Evaluation

We provide partial experimental results here; see Section F for details and additional results.

Experimental setup. We consider synthesizing programs that operate over the predictions made by a state-of-the-art DNN for object detection. We assume given a DNN component $\hat{f}$ that given an image x, is designed to detect people and cars in x. We use a pretrained state-of-the-art object detector called Faster R-CNN [4] available in PyTorch [11], tailored to the COCO dataset [12], which is a dataset of real-world images containing people, cars, and other objects. There are multiple variants of Faster R-CNN; we use the most accurate one, X101-FPN with $3\times$ learning rate schedule.

We represent this DNN as a component $\hat{f} : \mathcal{X} \to \mathcal{Y} = \mathcal{D}^* \times \mathbb{R}$, where $\hat{f}(x) = (\hat{y}(x), \hat{p}(x))$ consists of a list of *detections* $d \in \hat{y}(x)$ along with a correctness score $\hat{p}(x)$ that the prediction is correct. Each detection $d \in \mathcal{D} = \mathbb{R}^2 \times \mathcal{Z}$ is itself a tuple $d = (b, z)$ including the position b and predicted category of the object. The ground truth label y^* for an image x is a list of detections $d \in y^*$. In general, we cannot expect to get a perfect match between the predicted bounding boxes and the ground truth ones.

[1] In Section D, our synthesis algorithm is presented for the case where it returns the correct answer or "unknown" with high probability, but as we discuss in Section E, it can easily be modified to return an overestimate of the correct answer.

Task	∅ Rate		Failure Rate	
	StatCoder	Naïve	StatCoder	Naïve
count number of people in x	0.054	0.901	0.124	0.003
check if x contains a person	0.054	0.901	0.124	0.003
count people near center of x	0.290	0.901	0.032	0.003
find people near a car	0.901	1.000	0.003	0.000
minimum distance from person to center of x	0.149	0.901	0.023	0.000
average	0.290	0.921	0.061	0.002

Table 1: We show results on synthesizing list processing programs over object detection. For each task, we show the "∅ Rate" (i.e., how often the program returns "unknown"), and the "Failure Rate" (i.e., how many errors the program makes). The desired PAC parameters are $\epsilon = \delta = 0.2$.

Typically, two detections d, d^* *match*, denoted $\|d - d^*\| \leq e$, where e is a specified error tolerance, if the distance between their centers satisfies $\|b - b^*\|_\infty \leq e$. Furthermore, we write $\|\hat{y}(x) - y^*\| \leq e$ if $|\hat{y}(x)| = |y^*|$ and there exists a one-to-one correspondence between $d \in \hat{f}(x)$ and $d^* \in y^*$ such that $\|d - d^*\| \leq e$. Then, we define $\mathsf{predict} : \mathcal{X} \to (\mathcal{Y} \cup \varnothing)$ by

$$(\mathsf{predict}\ x) = (\mathsf{if}\ \hat{p}(x) \geq ??_c\ \{\|\hat{y}(x) - y^*\| \leq ??_e\}^{\Rrightarrow}_{??_\epsilon}\ \mathsf{then}\ \hat{y}(x)\ \mathsf{else}\ \varnothing).$$

In other words, the specification says that a correct prediction is if the error tolerance is below a level $??_e$ to be specified. Thus, given e and ϵ to fill $??_e$ and $??_\epsilon$, respectively, our sketching algorithm synthesizes a threshold c to fill $??_c$ in a way that guarantees that this specification holds. Then, predict returns $\hat{y}(x)$ if the DNN is sufficiently confident in its prediction, and $\varnothing$ otherwise. For example, if a robot acting in the world is using results from the program to navigate, then it can act conservatively (e.g., safely come to a stop) when the program returns $\varnothing$ to ensure safety; alternatively, it might fall back on a more accurate but computationally expensive model to make predictions.

We use our synthesis algorithm in conjunction with this prediction component and a standard domain-specific language (DSL) of list processing programs, and PAC parameters $\epsilon = \delta = 0.2$. We use $n = 1000$ COCO validation set images for synthesis, and the remaining 1503 for evaluation.

We compare to a baseline that uses a naïve statistical estimator to synthesize c instead of statistical sketching. At a high level, it uses the more traditional method of chooses c to minimize the empirical error rate, whereas our algorithm minimizes c subject to a constraint on the error rate. We evaluate our approach on synthesizing five programs. For the program synthesized using each approach, we report the following metrics: (i) the $\varnothing$ rate—i.e., The rate at which $\bar{P}$ returns $\varnothing$, and (ii) the failure rate—i.e., the rate at which $\bar{P}$ makes mistakes.

Results. Results are shown in Table 1. Both StatCoder and the baseline always achieve the desired failure rate bound of $\epsilon = 0.2$. However, StatCoder outperforms the baseline by a large margin in terms of $\varnothing$ rate, since it uses a learning algorithm tailored to our setting. These results demonstrate the effectiveness of StatCoder at synthesizing accurate programs while satisfying a PAC guarantee.

4 Conclusion

We have proposed algorithms for synthesizing machine learning programs that come with PAC guarantees. Our technique leverages novel statistical learning bounds to achieve these guarantees. We have empirically demonstrated how our approach can be used to synthesize list processing programs that manipulate images using DNN components while satisfying PAC guarantees.

References

[1] Varun Gulshan, Lily Peng, Marc Coram, Martin C Stumpe, Derek Wu, Arunachalam Narayanaswamy, Subhashini Venugopalan, Kasumi Widner, Tom Madams, Jorge Cuadros, et al. Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *Jama*, 316(22):2402–2410, 2016.

[2] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *nature*, 542(7639):115–118, 2017.

[3] Matthieu Komorowski, Leo A Celi, Omar Badawi, Anthony C Gordon, and A Aldo Faisal. The artificial intelligence clinician learns optimal treatment strategies for sepsis in intensive care. *Nature medicine*, 24(11):1716–1720, 2018.

[4] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE transactions on pattern analysis and machine intelligence*, 39(6):1137–1149, 2016.

[5] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[6] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[7] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[9] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

[10] David Haussler, Michael Kearns, Nick Littlestone, and Manfred K Warmuth. Equivalence of models for polynomial learnability. *Information and Computation*, 95(2):129–161, 1991.

[11] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[12] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[13] Håkan LS Younes and Reid G Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *International Conference on Computer Aided Verification*, pages 223–235. Springer, 2002.

[14] Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *International Conference on Computer Aided Verification*, pages 202–215. Springer, 2004.

[15] Koushik Sen, Mahesh Viswanathan, and Gul Agha. On statistical model checking of stochastic systems. In *International Conference on Computer Aided Verification*, pages 266–280. Springer, 2005.

[16] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*, pages 409–426. Springer, 1994.

[17] Shai Ben-David, John Blitzer, Koby Crammer, and Fernando Pereira. Analysis of representations for domain adaptation. In *Advances in neural information processing systems*, pages 137–144, 2007.

[18] Joaquin Quionero-Candela, Masashi Sugiyama, Anton Schwaighofer, and Neil D Lawrence. *Dataset shift in machine learning*. The MIT Press, 2009.

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[20] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[22] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

[23] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

[24] Michael J Kearns, Umesh Virkumar Vazirani, and Umesh Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[25] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[26] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[27] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 527–536. JMLR. org, 2017.

[28] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122, 2014.

[29] Stephen E Kimmel, Benjamin French, Scott E Kasner, Julie A Johnson, Jeffrey L Anderson, Brian F Gage, Yves D Rosenberg, Charles S Eby, Rosemary A Madigan, Robert B McBane, et al. A pharmacogenetic versus a clinical algorithm for warfarin dosing. *New England Journal of Medicine*, 369(24):2283–2293, 2013.

[30] Hamsa Bastani and Mohsen Bayati. Online decision-making with high-dimensional covariates. *Operations Research*, 2015.

[31] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[32] Ziad Obermeyer, Brian Powers, Christine Vogeli, and Sendhil Mullainathan. Dissecting racial bias in an algorithm used to manage the health of populations. *Science*, 366(6464):447–453, 2019.

[33] Alexander L Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *International Conference on Machine Learning*, pages 1213–1222, 2017.

[34] Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. Houdini: Lifelong learning as program synthesis. In *Advances in Neural Information Processing Systems*, 2018.

[35] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. Learning to infer graphics programs from hand-drawn images. In *Advances in neural information processing systems*, pages 6059–6068, 2018.

[36] Halley Young, Osbert Bastani, and Mayur Naik. Learning neurosymbolic generative models via program synthesis. In *International Conference on Machine Learning*, 2019.

[37] Ameesh Shah, Eric Zhan, Jennifer J Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. Learning differentiable programs with admissible neural heuristics. In *Advances in neural information processing systems*, 2020.

[38] Aditya V Nori, Sherjil Ozair, Sriram K Rajamani, and Deepak Vijaykeerthy. Efficient synthesis of probabilistic programs. *ACM SIGPLAN Notices*, 50(6):208–217, 2015.

[39] Feras A Saad, Marco F Cusumano-Towner, Ulrich Schaechtle, Martin C Rinard, and Vikash K Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019.

[40] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054, 2018.

[41] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Advances in neural information processing systems*, pages 2494–2504, 2018.

[42] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 686–701, 2019.

[43] Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. Neurosymbolic reinforcement learning with formally verified exploration. In *Advances in neural information processing systems*, 2020.

[44] Samuel Drews, Aws Albarghouthi, and Loris D'Antoni. Efficient synthesis with probabilistic constraints. In *International Conference on Computer Aided Verification*, pages 278–296. Springer, 2019.

[45] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. Measuring neural net robustness with constraints. In *Advances in neural information processing systems*, pages 2613–2621, 2016.

[46] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[47] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer, 2017.

[48] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18. IEEE, 2018.

[49] Greg Anderson, Shankara Pailoor, Isil Dillig, and Swarat Chaudhuri. Optimization and abstraction: A synergistic approach for analyzing neural network robustness. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 731–744, 2019.

[50] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V Nori. Fairsquare: probabilistic verification of program fairness. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–30, 2017.

[51] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. Probabilistic verification of fairness properties via concentration. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[52] Radoslav Ivanov, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, pages 169–178, 2019.

[53] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 324–334, 2006.

[54] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. *ACM SIGPLAN Notices*, 47(6):169–180, 2012.

[55] Michael Carbin, Sasa Misailovic, and Martin C Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. *ACM SIGPLAN Notices*, 48(10):33–52, 2013.

[56] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C Rinard. Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels. *ACM Sigplan Notices*, 49(10):309–328, 2014.

[57] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 447–458, 2013.

[58] Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, Russ Tedrake, and John C Duchi. Scalable end-to-end autonomous vehicle testing via rare-event simulation. In *Advances in Neural Information Processing Systems*, pages 9827–9838, 2018.

[59] Tommaso Dreossi, Daniel J Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A Seshia. Verifai: A toolkit for the formal design and analysis of artificial intelligence-based systems. In *International Conference on Computer Aided Verification*, pages 432–442. Springer, 2019.

[60] Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78, 2019.

[61] Edward Kim, Divya Gopinath, Corina Pasareanu, and Sanjit A Seshia. A programmatic and semantic approach to explaining and debugging neural network based object detectors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11128–11137, 2020.

[62] Glenn Shafer and Vladimir Vovk. A tutorial on conformal prediction. *Journal of Machine Learning Research*, 9(Mar):371–421, 2008.

[63] Vineeth Balasubramanian, Shen-Shyang Ho, and Vladimir Vovk. *Conformal prediction for reliable machine learning: theory, adaptations and applications*. Newnes, 2014.

[64] Ryan J Tibshirani, Rina Foygel Barber, Emmanuel Candes, and Aaditya Ramdas. Conformal prediction under covariate shift. In *Advances in Neural Information Processing Systems*, pages 2530–2540, 2019.

[65] Yaniv Romano, Evan Patterson, and Emmanuel Candes. Conformalized quantile regression. In *Advances in Neural Information Processing Systems*, pages 3543–3553, 2019.

[66] Sangdon Park, Osbert Bastani, Nikolai Matni, and Insup Lee. Pac confidence sets for deep neural networks via calibrated prediction. In *International Conference on Learning Representations*, 2020.

[67] Anastasios Angelopoulos, Stephen Bates, Jitendra Malik, and Michael I Jordan. Uncertainty sets for image classifiers using conformal prediction. *arXiv preprint arXiv:2009.14193*, 2020.

[68] Danijel Kivaranovic, Kory D Johnson, and Hannes Leeb. Adaptive, distribution-free prediction intervals for deep networks. In *International Conference on Artificial Intelligence and Statistics*, pages 4346–4356, 2020.

[69] Sangdon Park, Shuo Li, Osbert Bastani, and Insup Lee. Pac confidence predictions for deep neural network classifiers. *arXiv preprint arXiv:2011.00716*, 2020.

$$
\begin{aligned}
P &::= c \mid x \mid f(P, ..., P) \\
&\quad \mid \phi(P, c)\ \{Q\}_{\epsilon}^{\omega} \mid \phi(P, ??)\ \{Q\}_{\epsilon}^{\omega} \mid \phi(P, c)\ \{Q\}_{??}^{\omega} \\
Q &::= c \mid x \mid y \mid f(Q, ..., Q)
\end{aligned}
$$

$$
\begin{aligned}
[\![c]\!]_{\alpha}^{*} &= c & [\![f(P, ..., P)]\!]_{\alpha}^{*} &= f([\![P]\!]_{\alpha}^{*}, ..., [\![P]\!]_{\alpha}^{*}) \\
[\![x]\!]_{\alpha}^{*} &= \alpha(x) & [\![f(Q, ..., Q)]\!]_{\alpha}^{*} &= f([\![Q]\!]_{\alpha}^{*}, ..., [\![Q]\!]_{\alpha}^{*}) \\
[\![y]\!]_{\alpha}^{*} &= \alpha(y) & [\![\phi(P, c)\ \{Q\}_{\epsilon}^{\omega}]\!]_{\alpha}^{*} &= [\![Q]\!]_{\alpha}^{*} \\
\\
[\![c]\!]_{\beta} &= c & [\![f(P, ..., P)]\!]_{\beta} &= f([\![P]\!]_{\beta}, ..., [\![P]\!]_{\beta}) \\
[\![x]\!]_{\beta} &= \beta(x) & [\![\phi(P, v)\ \{Q\}_{\epsilon}^{\omega}]\!]_{\beta} &= \mathbb{1}([\![P]\!]_{\beta} > c)
\end{aligned}
$$

Figure 1: Syntax (left), train semantics (right, top), and test semantics (right, bottom). The production rules in the syntax are implicitly universally quantified over constant values $c \in \mathcal{C}$, input variables $x \in \mathcal{X}$, *ground truth input variables* $y \in \mathcal{Y}$, components $f \in \mathcal{F}$ where $f : \mathcal{C}^k \to \mathcal{C}$, $\epsilon \in \mathbb{R}_{>0}$, and $\omega \in \{|, \Rightarrow\}$. The distinguished component $\phi \in \mathcal{F}$ is a function $\phi : \mathbb{R}^2 \to \mathbb{R}$ defined by $\phi(z, t) = \mathbb{1}(z \leq t)$.

A Sketch Language

In this section, we describe the syntax and semantics of our sketch language, as well as the desired correctness properties we expect that synthesized programs should satisfy.

Syntax. Our sketch language is shown in Figure 1. Intuitively, in the expression $\phi(P, c)\ \{Q\}_{\epsilon}^{\omega}$, Q is a specification that we want to ensure holds, P is a score (intuitively, it should indicate the likelihood that Q holds, but we make no assumptions about it), c is a threshold below which we consider Q to be satisfied, ϵ is the allowed failure probability, and ω indicates whether we want a *conditional guarantee* (i.e., $\omega = |$, the guarantee (1)) or *implication guarantee* (i.e., $\omega = \Rightarrow$, the guarantee (**??**)). We assume that P evaluates to a value in $\mathbb{R}$, $c \in \mathbb{R}$, and Q evaluates to a value in $\{0, 1\}$. Note that Q is itself a program; unlike programs P, it can use *ground truth inputs* y. Finally, either c and ϵ in this expression can be left as a hole $??$ (but not both simultaneously).

We say P is *complete* if it contains no holes and *partial* otherwise. We use $\mathcal{P}$ to denote the space of programs, $\bar{\mathcal{P}} \subseteq \mathcal{P}$ to denote the space of complete programs, and $\bar{P} \in \bar{\mathcal{P}}$ to denote a complete program. For $P \in \mathcal{P}$, we use $\Phi(P)$ to denote the expressions $\phi(P', c)\ \{Q\}_{\epsilon}^{\omega}$ in P (including cases where c or ϵ is a hole), $\Phi_{??}^{c}(P) \subseteq \Phi(P)$ to denote the expressions $\phi(P', ??)\ \{Q\}_{\epsilon}^{\omega}$ in P, $\Phi_{??}^{\epsilon}(P) \subseteq \Phi(P)$ to denote the expressions $\phi(P', c)\ \{Q\}_{??}^{\omega}$ in P, and $\Phi_{??}(P) = \Phi_{??}^{c}(P) \cup \Phi_{??}^{\epsilon}(P)$.

Semantics. We define two semantics for programs P, shown in Figure 1:

- **Train semantics:** Given a *training valuation* $\alpha \in \mathcal{A}$, where $\alpha : \mathcal{X} \cup \mathcal{Y} \to \mathcal{C}$ maps both inputs and ground truth inputs y to values, the *train semantics* $[\![\cdot]\!]_{\alpha}^{*}$ evaluate Q instead of $\phi(P, c)$. Since they ignore ϕ, they can be applied to both partial and complete programs.
- **Test semantics:** Given a *test valuation* $\beta \in \mathcal{B}$, where $\beta : \mathcal{X} \to \mathcal{C}$ maps inputs to values, the *test semantics* $[\![\cdot]\!]_{\beta}$ evaluate $\phi(P, c)$ instead of Q. They only apply to complete programs.

Correctness properties. We define what it means for a complete program to be correct—i.e., satisfies its specifications. We begin with correctness of a single specification.

Definition A.1. Given a distribution $p(\alpha)$ over test valuations $\alpha \in \mathcal{A}$, $\phi(\bar{P}, c)\ \{Q\}_{\epsilon}^{|}$ is *approximately sound* if it satisfies the *conditional guarantee*[2]

$$\mathbb{P}_{p(\alpha)}\big([\![\phi(\bar{P}, c)\ \{Q\}_{\epsilon}^{\omega}]\!]_{\alpha} \;\big|\; [\![\phi(\bar{P}, c)\ \{Q\}_{\epsilon}^{\omega}]\!]_{\alpha}^{*}\big) \geq 1 - \epsilon,$$

and $\{Q\}_{\epsilon}^{\Rightarrow}$ is *approximately sound* if it satisfies the *implication guarantee*

$$\mathbb{P}_{p(\alpha)}\big([\![\phi(\bar{P}, c)\ \{Q\}_{\epsilon}^{\omega}]\!]_{\alpha}^{*} \Rightarrow [\![\phi(\bar{P}, c)\ \{Q\}_{\epsilon}^{\omega}]\!]_{\alpha}\big) \geq 1 - \epsilon.$$

This property can be thought of as probabilistic soundness; it says that we should have $\phi(\bar{P}, c) \Rightarrow Q$ with high probability, which means that $\phi(\bar{P}, c)$ is a sound overapproximation of Q.

Definition A.2. A complete program $\bar{P}$ is *approximately correct* (denoted $\bar{P} \in \bar{\mathcal{P}}^{*}$) if every expression $\phi(\bar{P}', c)\ \{Q\}_{\epsilon}^{\omega}$ in $\bar{P}$ is approximately sound.

[2]Note that since α includes valuations of $x \in \mathcal{X}$, we can use it in conjunction both train semantics and test semantics.

B Statistical Verification

We describe our algorithm for verifying a complete program. Our algorithm (Algorithm 1) takes as input a complete program $\bar{P}$, training valuations $\vec{\alpha} = (\alpha_1, ..., \alpha_n)$, where $\alpha_1, ..., \alpha_n \sim p$ are i.i.d. samples, and a confidence level $\delta \in \mathbb{R}_{>0}$, and outputs a value $A(\bar{P}, \vec{\alpha}) \in \{0, 1\}$ indicating whether $\bar{P}$ is approximately complete, which is correct with probability at least $1 - \delta$ with respect to $p(\vec{\alpha})$.

Our algorithm is based on *statistical verification* [13, 14, 15]. These algorithms leverage *concentration inequalities* from probability theory to provide high-probability correctness guarantees. Concentration inequalities are theorems that provide rigorous bounds on the rate of convergence of statistical estimators. For instance, consider a Bernoulli distribution $p = \text{Bernoulli}(\mu)$ with unknown mean μ. Given samples $z_1, ..., z_n \sim p$, Hoeffding's inequality [16] says that the empirical mean $\hat{\mu}(\vec{z}) = n^{-1} \sum_{i=1}^{n} z_i$ converges to μ:

$$\mathbb{P}_{p(\vec{z})}\big(|\hat{\mu}(\vec{z}) - \mu| \leq \epsilon\big) \geq 1 - \delta \text{ where } \delta = 2e^{-2n\epsilon^2}, \tag{2}$$

i.e., $\hat{\mu}(\vec{z})$ is a good approximation of μ with high probability.

In our setting, given training valuation α and a specification $E = \phi(\bar{P}', c)\ \{Q\}^{\omega}_{\epsilon}$ in $\bar{P}$, we let $z_\alpha = [\![E]\!]_\alpha$ and $z^*_\alpha = [\![E]\!]^*_\alpha$. Then, ϵ-approximate soundness of E is equivalent to $\mu = \mathbb{P}_{p(\alpha)}(z_\alpha \mid z^*_\alpha) \geq 1 - \epsilon$ if $\omega = |$, or $\mu = \mathbb{P}_{p(\alpha)}(z^*_\alpha \Rightarrow z_\alpha) \geq 1 - \epsilon$ if $\omega = \Rightarrow$. That is, ϵ-approximate soundness is equivalent to $\mu \geq 1 - \epsilon$, where μ is the mean of a Bernoulli random variable z_α that is a function of a random variable α with distribution $p(\alpha \mid z^*_\alpha)$ (if $\omega = |$) or the mean of a Bernoulli random variable $z^*_\alpha \Rightarrow z_\alpha$ that is a function of α with distribution $p(\alpha)$ (if $\omega = \Rightarrow$). However, z_α is potentially a complicated function of α and $p(\alpha)$ is unknown, so μ is hard to compute directly. Instead, given i.i.d. samples $\alpha_1, ..., \alpha_n \sim p(\alpha)$, we can construct the samples $z_{\alpha_1}, ..., z_{\alpha_n}$ and $z^*_{\alpha_1}, ..., z^*_{\alpha_n}$ and use them estimate μ:

$$\hat{\mu}(\vec{\alpha}) = \begin{cases} \frac{\sum_{i=1}^{n} z_{\alpha_i} \wedge z^*_{\alpha_i}}{\sum_{i=1}^{n} z^*_{\alpha_i}} & \text{if } \omega = | \\ \sum_{i=1}^{n} z^*_{\alpha_i} \Rightarrow z_{\alpha_i} & \text{if } \omega = \Rightarrow . \end{cases}$$

Then, we can use (2) to bound the error of $\hat{\mu}(\vec{\alpha})$—e.g., if $\hat{\mu}(\vec{\alpha}) \geq 1 - \frac{\epsilon}{2}$ and $|\hat{\mu}(\vec{\alpha}) - \mu| \leq \frac{\epsilon}{2}$ with probability at least $1 - \delta$, then $\mu \geq 1 - \epsilon$ with probability at least $1 - \delta$. However, this approach is inefficient since Hoeffding's inequality is not tight for our setting. Instead, our verification algorithm (Section B.2) leverages a concentration inequality tailored to our setting (Section B.1). Finally, we disucss how our approach can be used in the context of runtime monitoring (Section B.3).

B.1 A Concentration Bound

Problem formulation. Consider a Bernoulli distribution $p = \text{Bernoulli}(\mu)$ with unknown mean $\mu \in [0, 1]$. Given $\epsilon \in \mathbb{R}_{>0}$, our goal is to determine whether $\mu \geq 1 - \epsilon$. For instance, a sample $z \sim p$ may indicate a desired outcome (e.g., a correctly classified input), in which case μ is the correctness rate and ϵ is a desired bound on the error rate; then, our goal is to check whether the μ meets the desired error bound. More precisely, we want to compute $\psi \in \{0, 1\}$ such that

$$\psi \Rightarrow (\mu \geq 1 - \epsilon). \tag{3}$$

That is, ψ is a sound overapproximation of the property $\mu \geq 1 - \epsilon$ (i.e., $\psi = 1$ implies $\mu \geq 1 - \epsilon$).

To compute such a ψ, we are given a training set of examples $\vec{z} = (z_1, ..., z_n) \in \{0, 1\}^n$, where $z_1, ..., z_n \sim p$ are n i.i.d. samples from p. An *estimator* is a mapping $\hat{\psi} : \mathbb{R}^n \to \mathbb{R}$. We say such an estimator is *approximately correct* if it satisfies the condition (3)—i.e., $\hat{\psi}(\vec{z}) \Rightarrow (\mu \geq 1 - \epsilon)$.

In general, we cannot guarantee $\hat{\psi}(\vec{z})$ is approximately correct due to the randomness in the training examples $\vec{z}$.[3] Thus, we allow a probability $\delta \in \mathbb{R}_{>0}$ that $\hat{\psi}(\vec{z})$ is not approximately correct.

Definition B.1. Given $\epsilon, \delta \in \mathbb{R}_{>0}$, $\hat{\psi}$ is (ϵ, δ)-*PAC* if $\mathbb{P}_{p(\vec{z})}\big(\hat{\psi}(\vec{z}) \Rightarrow (\mu \geq 1 - \epsilon)\big) \geq 1 - \delta$.

In other words, $\hat{\psi}(\vec{z})$ is approximately correct with probability at least $1 - \delta$ according to the randomness in $p(\vec{z})$. Our goal is to construct an (ϵ, δ)-PAC estimator $\hat{\psi}(\vec{z})$.

[3] Note that $\hat{\psi}$ is a deterministic function; the randomness of $\hat{\psi}(\vec{z})$ is entirely due to the randomness in the training data $\vec{z}$.

Algorithm 1 Use statistical verification to check if P is approximately correct.

procedure VERIFY($\bar{P}, \vec{\alpha}, \delta$)
  $m \leftarrow |\Phi(P)|$
  for $\phi(\bar{P}', c)\ \{Q\}^{\omega}_{\epsilon} \in \Phi(\bar{P})$ **do**
    Compute $\vec{z}_{\vec{\alpha}}$ according to (5)
    Compute $\hat{\psi}(\vec{z}_{\vec{\alpha}})$ according to (4) with $(\epsilon, \delta/m)$
    if $\neg\hat{\psi}(\vec{z}_{\vec{\alpha}})$ **then**
      return false
    end if
  end for
  return true
end procedure

Estimator. Given $\epsilon, \delta \in \mathbb{R}_{>0}$, consider the estimator

$$\hat{\psi}(\vec{z}) = \mathbb{1}(L(\vec{z}) \leq k) \quad \text{where} \quad k = \max\left\{ h \in \mathbb{N} \;\middle|\; \sum_{i=0}^{h} \binom{n}{i} \epsilon^i (1-\epsilon)^{n-i} \leq \delta \right\} \tag{4}$$

and where $L(\vec{z}) = \sum_{z \in \vec{z}}^{n} (1 - z)$. Intuitively, $L(\vec{z})$ counts the number of errors, so we conclude the desired property holds as long as $L(\vec{z})$ is below a threshold k. This threshold is chosen so $\hat{\psi}$ is (ϵ, δ)-PAC—in particular, δ upper bounds the CDF of the binomial distribution evaluated at k.

To compute the solution k in (4), we start with $h = 0$ and increment it until it no longer satisfies the condition. To ensure numerical stability, this computation is performed using logarithms. Note that k does not exist if the set inside the maximum in (4) is empty; in this case, we choose $\hat{\psi}(\vec{z}) = 0$, which trivially satisfies the PAC property. We have the following; see Section H.1 for a proof:

Theorem B.2. *The estimator $\hat{\psi}$ is (ϵ, δ)-PAC.*

B.2 Verification Algorithm

Problem formulation. A verification algorithm $A : \bar{\mathcal{P}} \times \mathcal{A}^n \to \{0, 1\}$ takes as input a complete program $\bar{P} \in \bar{\mathcal{P}}$, and a set of test valuations $\vec{\alpha} = (\alpha_1, ..., \alpha_n) \in \mathcal{A}^n$, where $\alpha_1, ..., \alpha_n \sim p$ are i.i.d. samples from a distribution $p(\alpha)$. For example, $p(\alpha)$ may be the distribution of input images to an image classifier encountered while running in production, that have been manually labeled using crowdsourcing. Then, $A(\bar{P}, \vec{\alpha}) \in \{0, 1\}$ should indicate whether $\bar{P}$ is approximately sound—i.e., whether every expression $\phi(\bar{P}', c)\ \{Q\}^{\omega}_{\epsilon} \in \Phi(\bar{P})$ is approximately sound. We say that A is *sound* if $A(\bar{P}, \vec{\alpha}) \Rightarrow \bar{P} \in \bar{\mathcal{P}}^*$. As before, we cannot guarantee that A is sound; instead, given $\delta \in \mathbb{R}_{>0}$, we want this property to hold with probability at least $1 - \delta$ according to $p(\vec{\alpha})$.

Definition B.3. A verification algorithm $A : \bar{\mathcal{P}} \times \mathcal{A}^n \to \{0, 1\}$ is δ-probably approximately sound if for all $\bar{P} \in \bar{\mathcal{P}}$, $\mathbb{P}_{p(\vec{\alpha})}\big(A(\bar{P}, \vec{\alpha}) \Rightarrow \bar{P} \in \bar{\mathcal{P}}^*\big) \geq 1 - \delta$.

Algorithm. Our verification algorithm is shown in Algorithm 1. It check approximate correctness of $\bar{P}$ by checking approximate soundness of each $\phi(\bar{P}', c)\ \{Q\}^{\omega}_{\epsilon} \in \Phi(\bar{P})$. It does so by allocating a δ/m probability of failure for each expression, where $m = |\Phi(\bar{P})|$ is the number of such expressions.

Next, we describe how our algorithm checks approximate soundness for a single expression $\phi(\bar{P}', c)\ \{Q\}^{\omega}_{\epsilon}$. Given a single test valuation $\alpha \sim p$, consider the indicators

$$z_\alpha = [\![\phi(\bar{P}', c)\ \{Q\}^{\omega}_{\epsilon}]\!]_\alpha \qquad \text{and} \qquad z^*_\alpha = [\![\phi(\bar{P}', c)\ \{Q\}^{\omega}_{\epsilon}]\!]^*_\alpha.$$

That is, z_α indicates whether $\phi(\bar{P}', c)$ holds, and z^*_α indicates whether Q holds. Then, $\phi(\bar{P}', c)\ \{Q\}$ is approximately sound if and only if

$$\mathbb{P}_{p(\alpha)}(z_\alpha \mid z^*_\alpha) \geq 1 - \epsilon \quad \text{if } \omega = | \qquad \text{or} \qquad \mathbb{P}_{p(\alpha)}(z^*_\alpha \Rightarrow z_\alpha) \geq 1 - \epsilon \quad \text{if } \omega = \Rightarrow .$$

Next, note that $z_\alpha \in \{0, 1\}$ is a Bernoulli random variable with mean $\mu = \mathbb{P}_{p(\alpha)}(z_\alpha \mid z^*_\alpha)$ (if $\omega = |$) or $\mu = \mathbb{P}_{p(\alpha)}(z^*_\alpha \Rightarrow z_\alpha)$ (if $\omega = \Rightarrow$). Thus, given $\vec{\alpha} = (\alpha_1, ..., \alpha_n)$, where $\alpha_1, ..., \alpha_n \sim p$ are i.i.d.

samples,

$$\vec{z}_{\vec{\alpha}} = \begin{cases} \{z_\alpha \mid \alpha \in \vec{\alpha} \wedge z^*_\alpha\} & \text{if } \omega = | \\ \{z^*_\alpha \Rightarrow z_\alpha \mid \alpha \in \vec{\alpha}\} & \text{if } \omega = \Rightarrow \end{cases} \tag{5}$$

is a vector of i.i.d. samples from Bernoulli(μ). Then, the estimator $\hat{\psi}(\vec{z}_{\vec{\alpha}})$ in (4) with parameters $(\epsilon, \delta/m)$ indicates whether $\mu \geq 1 - \epsilon$ with high probability—i.e., if $\hat{\psi}(\vec{z}_{\vec{\alpha}}) = 1$, then $\mu \geq 1 - \epsilon$ holds with probability at least $1 - \delta/m$ according to $p(\vec{\alpha})$. The following guarantee follows from Theorem B.2 by a union bound over expressions in $\Phi(\bar{P})$:

Theorem B.4. *Algorithm 1 is δ-probably approximately sound.*

B.3 Runtime Monitoring

One challenge is that the specifications considered by our framework depend on the distribution of the data. As a consequence, if this distribution changes, then our correctness guarantees may no longer hold. This potential failure mode, called *distribution shift* [17, 18], is a major challenge for machine learning components. A key feature of our framework is that it can be used not only to sketch or verify the program before it is deployed, but also to continuously re-sketch the program based on feedback obtained in production to account for potential distribution shift. The primary requirement for using this approach is the need for feedback—i.e., continuing to collect labeled examples in production. In some settings, this kind of feedback is naturally available; otherwise, a solution is to manually label a small fraction of examples—e.g., using crowdsourcing [19].

Given ground truth labels for the input examples encountered in production, our verification algorithm can be straightforwardly adapted to the runtime setting. In particular, our system collects examples during execution; once it collects at least N examples, it re-runs verification or sketching. It can do so after every subsequent example, or every K examples. Finally, we may want to discard an examples after T steps, both for computational efficiency and to account for the fact that the data distribution may be shifting over time so older examples are less representative. Here, $K, N, T \in \mathbb{N}$ are hyperparameters. Finally, we note that our statistical sketching algorithm can similarly be adapted to the runtime setting.

C Statistical Sketching

Next, we describe our algorithm for synthesizing values c and ϵ to fill holes in a given sketch. Our algorithm, shown in Figure 2, takes as input a sketch P, training valuations $\vec{\alpha} = (\alpha_1, ..., \alpha_n)$, where $\alpha_1, ..., \alpha_n \sim p$ are i.i.d. samples, and a confidence level $\delta \in \mathbb{R}_{>0}$, and outputs a complete program $A(\bar{P}, \vec{\alpha}) \in \bar{\mathcal{P}}^*$ that is approximately correct with probability at least $1 - \delta$ with respect to $p(\vec{\alpha})$.

Our algorithm synthesizes c and ϵ in a bottom-up fashion, so that all subtrees of the current expression are complete. Our sketching algorithm uses probabilistic bounds in conjunction with the given samples $\vec{\alpha}$ to provide guarantees. Intuitively, since we are estimating parameters from data, our problem is a statistical learning problem [9], so we can leverage techniques from statistical learning theory to provide guarantees on the synthesized sketch.

For synthesizing c—i.e., an expression $E = \phi(P, c)\ \{Q\}^{\omega}_{\epsilon}$. Letting $z_\alpha = [\![P]\!]_\alpha \in \mathbb{R}$ and $z^*_\alpha = [\![E]\!]$, then c is ϵ-approximately correct if $z_\alpha \leq c$ conditioned on $z^*_\alpha = 1$ (if $\omega = |$) or whenever $z^*_\alpha = 1$ (if $\omega = \Rightarrow$) with probability at least $1 - \epsilon$ with respect to $p(\alpha)$. In either case, synthesizing c is equivalent to a binary classification problem with labels z^*_α, with a one-dimensional hypothesis space $c \in \mathbb{R}$ and a one-dimensional feature space $z_\alpha \in \mathbb{R}$. Furthermore, this problem is simple—c is a linear classifier. Thus, we could use standard learning theory results to provide guarantees.

However, we can obtain sharper guarantees using a learning theory bound specialized to our setting. We build on a bound based on [10] (Section C.1) tailored to the *realizable* setting, where there exists a classifier that makes zero mistakes. Our setting is realizable, since $c = \infty$ always makes zero mistakes. The main difference is that their bound always chooses a classifier that makes zero mistakes, which can be overly conservative. We prove a novel generalization bound that allows for some number k of mistakes that is a function of ϵ, δ, and n.

Synthesizing a value ϵ is a bit different, since we are not classifying examples that depend on a single α, but examples that depend on $\vec{\alpha}$. Thus, we can formulate it as a learning problem where the

examples are $\vec{\alpha}$; however, this approach is complicated due to the need to figure out how to divide our given samples $\vec{\alpha}$ into multiple sub-examples $\vec{\alpha}_1, ...\vec{\alpha}_n$. Instead, we use an approach based on Hoeffding's inequality [16] (Section C.2) to infer ϵ. In particular, Hoeffding's inequality gives us a lower bound on the correctness rate $\mathbb{P}_{p(\alpha)}(z_\alpha \mid z^*_\alpha) \geq 1 - \epsilon$ (if $\omega = |$) or $\mathbb{P}_{p(\alpha)}(z^*_\alpha \Rightarrow z_\alpha) \geq 1 - \epsilon$ (if $\omega = \Rightarrow$), and we can simply use this ϵ.

Finally, our sketching algorithm uses the above two approaches to synthesize c and ϵ (Section C.3).

C.1 A Learning Theory Bound

Problem formulation. We consider a unary classification problem with one-dimensional feature and hypothesis spaces. In particular, given a probability distribution $p(z)$ over $z \in \mathbb{R}$ (the feature), the goal is to select the smallest possible threshold $t \in \mathbb{R}$ (the hypothesis) such that

$$\mathbb{P}_{p(z)}(z \leq t) \geq 1 - \epsilon \tag{6}$$

for a given $\epsilon \in \mathbb{R}_{>0}$. That is, we want the smallest possible t such that $z \in (-\infty, t]$ with probability at least $1 - \epsilon$ according to $p(z)$. We denote the subset of t that satisfies (6) by

$$\mathcal{T}_\epsilon = \left\{ t \in \mathbb{R} \mid \mathbb{P}_{p(z)}(z \leq t) \geq 1 - \epsilon \right\}.$$

To compute such a t, we are given a training set of examples $\vec{z} = (z_1, ..., z_n) \in \mathbb{R}^n$, where $z_1, ..., z_n \sim p$ are n i.i.d. samples from p. An *estimator* $\hat{t}$ is a mapping $\hat{t} : \mathbb{R}^n \to \mathbb{R}$. Then, the constraint (6) is $\hat{t}(\vec{z}) \in \mathcal{T}_\epsilon$; we say such a $\hat{t}$ is *ϵ-approximately correct*—i.e., it is correct for "most" samples $z \sim p$.

In general, we are unable to guarantee that $\hat{t}$ is approximately correct due to the randomness in the training examples $\vec{z}$. Thus, we additionally allow for a small probability $\delta \in \mathbb{R}_{>0}$ that $\hat{t}$ is not approximately correct.

Definition C.1. Given $\epsilon, \delta \in \mathbb{R}_{>0}$, $\hat{t}$ is (ϵ, δ)-*PAC* if $\mathbb{P}_{p(\vec{z})}(\hat{t}(\vec{z}) \in \mathcal{T}_\epsilon) \geq 1 - \delta$.

That is, $\hat{t}(\vec{z})$ is approximately correct with probability at least $1 - \delta$ according to $p(\vec{z})$. Our goal is to construct an (ϵ, δ)-PAC estimator $\hat{t}(\vec{z})$ that tries to minimize $\hat{t}(\vec{z})$.

Estimator. Given $\epsilon, \delta \in \mathbb{R}_{>0}$, consider the estimator

$$\hat{t}(\vec{z}) = \inf_{t \in \mathbb{R}} \left\{ t \in \mathbb{R} \;\middle|\; L(t; \vec{z}) \leq k \right\} + \gamma(\vec{z}) \quad \text{where} \quad k = \max \left\{ h \in \mathbb{N} \;\middle|\; \sum_{i=0}^{h} \binom{n}{i} \epsilon^i (1 - \epsilon)^{n-i} \leq \delta \right\} \tag{7}$$

where the *empirical loss* is $L(t; \vec{z}) = \sum_{z \in \vec{z}} \mathbb{1}(z > t)$, and where $\gamma(\vec{z}) > 0$ is an arbitrary positive function. Intuitively, the empirical loss counts the number of mistakes that t makes on the training data—i.e., $z \in \vec{z}$ such that $z \notin (-\infty, t]$. To compute the solution k in (7), we start with $h = 0$ and increment it until it no longer satisfies the condition. To ensure numerical stability, this computation is performed using logarithms. Note that k does not exist if the set inside the maximum in (7) is empty; in this case, we choose $\hat{\psi}(\vec{z}) = 0$, which trivially satisfies the PAC property. To compute $\hat{t}(\vec{z})$, we sort the training examples $z_1, ..., z_n$ by magnitude, so $z_1 \geq z_2 \geq ... \geq z_n$. Finally, z_{k+1} solves the minimization problem in (7), so $\hat{t}(\vec{z}) = z_{k+1} + \gamma(\vec{z})$. If k does not exist, then we choose $\hat{t}(\vec{z}) = \infty$, which trivially satisfies the PAC property. We have the following; see Section H.2 for a proof:

Theorem C.2. *The estimator $\hat{t}(\vec{z})$ in (7) is (ϵ, δ)-PAC.*

C.2 A Concentration Bound

Problem formulation. Consider a Bernoulli distribution $p = \text{Bernoulli}(\mu)$ with unknown mean $\mu \in [0, 1]$. Our goal is to compute a lower bound $\nu \in [0, 1]$ of μ—i.e., $\mu \geq \nu$. For example, if μ is the error rate of a classifier, then ν is a lower bound on this rate. To compute ν, we are given a training set $\vec{z} = (z_1, ..., z_n) \in \{0, 1\}^n$, where $z_1, ..., z_n \sim p$ are n i.i.d. samples from p. An *estimator* is a mapping $\hat{\nu} : \mathbb{R}^n \to \mathbb{R}$. We say $\hat{\nu}$ is *correct* if it satisfies $\mu \geq \hat{\nu}(\vec{z})$. We are unable to guarantee that $\hat{\nu}(\vec{z})$ is correct due to the randomness in the training examples $\vec{z}$. Thus, we additionally allow for a small probability $\delta \in \mathbb{R}_{>0}$ that $\hat{\psi}(\vec{z})$ is not correct—i.e., it is *probably correct (PC)*.

Algorithm 2 Use learning theory to sketch $\bar{P}$ that is approximately correct.

procedure SKETCH($P, \vec{\alpha}, \delta$)
  $m \leftarrow |\Phi_{??}(P)|$
  for $E \in$ BottomUp($\Phi_{??}(P)$) **do**
    if $E = \phi(\bar{P}', ??)\ \{Q\}^{\omega}_{\epsilon}$ **then**
      Compute $\vec{z}_{\vec{\alpha}}$ according to (9)
      Compute $\hat{t}(\vec{z}_{\vec{\alpha}})$ according to (7) with $(\epsilon, \delta/m)$
      Fill the hole ?? with $\hat{t}(\vec{z}_{\alpha})$
    else if $E = \phi(\bar{P}', c)\ \{Q\}^{\omega}_{??}$ **then**
      Compute $\vec{z}_{\vec{\alpha}}$ according to (9)
      Compute $\hat{\nu}(\vec{z}_{\vec{\alpha}})$ according to (8) with δ/m
      Fill the hole ?? with $1 - \hat{\nu}(\vec{z}_{\vec{\alpha}})$
    end if
  end for
  return true
end procedure

Definition C.3. Given $\delta \in \mathbb{R}_{>0}$, $\hat{\nu}$ is δ-*PC* if $\mathbb{P}_{p(\vec{z})}\big(\mu \geq \hat{\nu}(\vec{z})\big) \geq 1 - \delta$.

In other words, $\hat{\nu}(\vec{z})$ is correct with probability at least $1 - \delta$ according to the randomness in $p(\vec{z})$. Our goal is to construct an δ-PC estimator $\hat{\nu}(\vec{z})$.

Estimator. Given $\delta \in \mathbb{R}_{>0}$, consider the estimator

$$\hat{\nu}(\vec{z}) = \hat{\mu}(\vec{z}) - \sqrt{\frac{\log(1/\delta)}{2n}}, \tag{8}$$

where $\hat{\mu}(\vec{z}) = n^{-1} \sum_{z \in \vec{z}} z$ is an estimate of μ based on the samples $\vec{z}$; we take $\hat{\nu}(\vec{z}) = 0$ if (8) is negative. Intuitively, the second term in $\hat{\nu}(\vec{z})$ is a correction to $\hat{\mu}(\vec{z})$ to ensure it is (ϵ, δ)-PC, based on Hoeffding's inequality [16]. We have the following; see Section H.3 for a proof:

Theorem C.4. *The estimator $\hat{\nu}$ is δ-PC.*

C.3 Sketching Algorithm

Problem formulation. A sketching algorithm $A : \mathcal{P} \times \mathcal{A}^n \to \bar{\mathcal{P}}$ takes as input a partial program $P \in \mathcal{P}$, together with a set of test valuations $\vec{\alpha} = (\alpha_1, ..., \alpha_n) \in \mathcal{A}^n$, where $\alpha_1, ..., \alpha_n \sim p$ are i.i.d. samples from an underlying distribution $p(\alpha)$. Then, $\bar{P} = A(P, \vec{\alpha})$ should be a complete program that is approximately correct by filling each hole in expressions $\phi(P', ??)\ \{Q\}^{\omega}_{\epsilon} \in \Phi^{c}_{??}(P)$ with a value $c \in \mathbb{R}$ and each hole in expressions $\phi(P', c)\ \{Q\}^{\omega}_{??} \in \phi^{\epsilon}_{??}(P)$ with a value $\epsilon \in \mathbb{R}_{>0}$. We assume that every expression in $\Phi(P)$ has a hole—i.e., $\Phi(P) = \Phi_{??}(P)$; otherwise, we cannot guarantee that the existing thresholds in these expressions are approximately sound.

Definition C.5. A partial program $P \in \mathcal{P}$ is a *full sketch*, denoted $P \in \mathcal{P}^0$, if $\Phi_{??}(P) = \Phi(P)$.

Then, we say A is *correct* if $A(P, \vec{\alpha}) \in \bar{\mathcal{P}}^*$. We cannot guarantee this property; instead, given $\delta \in \mathbb{R}_{>0}$, we want it to hold with probability at least $1 - \delta$ according to $p(\vec{\alpha})$.

Definition C.6. A sketching algorithm $A : \mathcal{P}^0 \times \mathcal{A}^n \to \bar{\mathcal{P}}$ is δ-*probably approximately correct (PAC)* if for all $P \in \mathcal{P}^0$, we have $\mathbb{P}_{p(\vec{\alpha})}\big(A(P, \vec{\alpha}) \in \bar{\mathcal{P}}^*\big) \geq 1 - \delta$.

Note that this definition does not include ϵ since these values are provide in the given sketch.

Algorithm. Our sketching algorithm is shown in Algorithm 2. At a high level, it fills each hole so that the resulting expressions $\phi(\bar{P}', c)\ \{Q\}^{\omega}_{\epsilon}$ are all approximately sound. The order in which these expressions are processed is important; a expression cannot be processed until all its descendants have been processed. This order ensures that $\bar{P}'$ is complete, so it can be evaluated. In Algorithm 2, the function BottomUp ensures that the expressions in $\Phi_{??}(P)$ is processed in such an order. The algorithm allocates a δ/m probability of failure for each expression, where $m = |\Phi_{??}(P)|$.

Synthesizing c. We describe how our algorithm synthesizes a threshold c for an expression $E = \phi(\bar{P}', ??)\ \{Q\}^{\omega}_{\epsilon}$. Given a single test valuation $\alpha \sim p$, consider the values

$$z_{\alpha} = [\![\bar{P}']\!]_{\alpha} \qquad \text{and} \qquad z^{*}_{\alpha} = [\![\phi(\bar{P}', ??)\ \{Q\}^{\omega}_{\epsilon}]\!]^{*}_{\alpha}$$

Given $c \in \mathbb{R}$, it follows by definition of $[\![\cdot]\!]_\alpha$ that

$$[\![\phi(\bar{P}', c)\ \{Q\}^{\omega}_{\epsilon}]\!]_\alpha = \mathbb{1}(z_\alpha \leq c).$$

Thus, E is approximately sound for some $c \in \mathbb{R}$ if and only if

$$\mathbb{P}_{p(\alpha)}(z_\alpha \leq c \mid z^*_\alpha) \geq 1-\epsilon \quad \text{if } \omega = | \qquad \text{or} \qquad \mathbb{P}_{p(\alpha)}(z^*_\alpha \Rightarrow z_\alpha \leq c) \geq 1-\epsilon \quad \text{if } \omega = \Rightarrow .$$

Given $\vec{\alpha} = (\alpha_1, ..., \alpha_n)$, where $\alpha_1, ..., \alpha_n \sim p$ i.i.d.,

$$\vec{z}_{\vec{\alpha}} = \begin{cases} \{z_\alpha \mid \alpha \in \vec{\alpha} \wedge z^*_\alpha\} & \text{if } \omega = | \\ \{z^*_\alpha \Rightarrow z_\alpha \mid \alpha \in \vec{\alpha}\} & \text{if } \omega = \Rightarrow \end{cases} \tag{9}$$

is a vector of i.i.d. samples. The estimator $\hat{t}(\vec{z}_{\vec{\alpha}})$ in (7) with parameters $(\epsilon, \delta/m)$ ensures approximate soundness with high probability—i.e.,

$$\mathbb{P}_{p(\alpha)}\big(z_\alpha \leq \hat{t}(\vec{z}_{\vec{\alpha}}) \mid z^*_\alpha\big) \geq 1-\epsilon \quad \text{if } \omega = | \quad \text{or} \quad \mathbb{P}_{p(\alpha)}\big(z^*_\alpha \Rightarrow z_\alpha \leq \hat{t}(\vec{z}_{\vec{\alpha}})\big) \geq 1-\epsilon \quad \text{if } \omega = \Rightarrow .$$

holds with probability at least $1-\delta/m$ according to $p(\vec{\alpha})$.

Synthesizing ϵ. We describe how our algorithm synthesizes a confidence level ϵ for an expression $E = \phi(\bar{P}', c)\ \{Q\}^{\omega}_{??}$. Given a single test valuation $\alpha \sim p$, consider the values

$$z_\alpha = [\![\phi(\bar{P}', c)\ \{Q\}^{\omega}_{??}]\!]_\alpha \qquad \text{and} \qquad z^*_\alpha = [\![\phi(\bar{P}', c)\ \{Q\}^{\omega}_{??}]\!]^*_\alpha.$$

Note that we compute these values even though the ϵ is a hole, since $[\![\cdot]\!]_\alpha$ and $[\![\cdot]\!]^*_\alpha$ do not depend on ϵ. Also, note that unlike the case of synthesizing c, where $z_\alpha \in \mathbb{R}$ is a score, in this case, $z_\alpha \in \{0, 1\}$ is a binary value. Given $\epsilon \in \mathbb{R}_{>0}$, E is ϵ-approximately sound for ϵ if and only if

$$\mathbb{P}_{p(\alpha)}(z_\alpha \mid z^*_\alpha) \geq 1-\epsilon \quad \text{if } \omega = | \qquad \text{or} \qquad \mathbb{P}_{p(\alpha)}(z^*_\alpha \Rightarrow z_\alpha) \geq 1-\epsilon \quad \text{if } \omega = \Rightarrow .$$

Given $\vec{\alpha} = (\alpha_1, ..., \alpha_n)$, where $\alpha_1, ..., \alpha_n \sim p$ are i.i.d. samples, $\vec{z}_{\vec{\alpha}}$ defined in (9) is a vector of i.i.d. samples from Bernoulli(μ). Then, the estimator $\hat{\nu}(\vec{z}_{\vec{\alpha}})$ in (8) with parameter δ/m is a lower bound on μ with high probability—i.e.,

$$\mathbb{P}_{p(\alpha)}(z_\alpha \mid z^*_\alpha) \geq \hat{\nu}(\vec{z}_{\vec{\alpha}}) \quad \text{if } \omega = | \qquad \text{or} \qquad \mathbb{P}_{p(\alpha)}(z^*_\alpha \Rightarrow z_\alpha) \geq \hat{\nu}(\vec{z}_{\vec{\alpha}}) \quad \text{if } \omega = \Rightarrow .$$

holds with probability at least $1-\delta/m$ according to $p(\vec{\alpha})$. Thus, it suffices to choose $1-\epsilon = \hat{\nu}(\vec{z}_{\vec{\alpha}})$.

The following guarantee follows from Theorems C.2 & C.4 by a union bound over $\Phi(\bar{P})$:

Theorem C.7. *Algorithm 2 is δ-PAC.*

D Synthesis Algorithm

We now describe a syntax-guided synthesizer that uses our sketching algorithm to identify programs with machine learning components while satisfying a desired error guarantee. In general, to design such a synthesizer, we need to design a space of specifications along with a domain-specific language (DSL) of programs. For clarity, we focus on a specific set of design choices; as we discuss in Section E, our approach straightforwardly generalizes in several ways. We consider the following choices:

- **Specifications:** We consider specifications $\tilde{\psi} = (\psi, \epsilon, e)$, consisting of both a traditional part ψ indicating the logical property that the train semantics of the program should satisfy (provided either as a logical formula or input-output examples), and a statistical part (ϵ, e) indicating that the program should have error at most e with probability at least $1-\epsilon$ with respect to $p(\alpha)$, or else return $\varnothing$.
- **DSL:** We consider a DSL (shown in Figure 2) of list processing programs where the inputs are images of integers. Our DSL includes components designed to predict the integer represented by a given image. These components return the predicted value if its confidence is above a certain threshold, and return $\varnothing$ otherwise. Values $\varnothing$ are propagated as $\varnothing$ by all components in our DSL—i.e., if any input to a function is $\varnothing$, then its output is also $\varnothing$.

$$
\begin{aligned}
P_\tau &::= \mathsf{input}^1_\tau \mid \cdots \mid \mathsf{input}^{k}_\tau \\
&\mid (P_{\tau' \to \tau}\ P_{\tau'}) \\
&\mid (\mathsf{fold}\ P_{\tau' \to \tau \to \tau}\ P_{\mathsf{list}(\tau')}\ P_\tau) \\
P_{\mathsf{list}(\tau)} &::= (\mathsf{map}\ P_{\tau' \to \tau}\ P_{\mathsf{list}(\tau')}) \\
&\mid (\mathsf{filter}\ P_{\tau \to \mathsf{bool}}\ P_{\mathsf{list}(\tau)}) \\
&\mid (\mathsf{slice}\ P_{\mathsf{list}(\tau)}\ P_{\mathsf{int}}\ P_{\mathsf{int}}) \\
P_{\mathsf{int}} &::= (\mathsf{length}\ P_{\mathsf{list}(\tau)})
\end{aligned}
\qquad
\begin{aligned}
P_{\sigma \to \sigma \to \sigma} &::= + \mid - \\
P_{\mathsf{int} \to \mathsf{int} \to \mathsf{bool}} &::= \ \leq \mid = \mid \geq \\
P_{\mathsf{float} \to \mathsf{float} \to \mathsf{bool}} &::= \mathsf{cond}\text{-}\leq \mid \mathsf{cond}\text{-}\geq \\
P_{\mathsf{image} \to \sigma} &::= \mathsf{predict}_\sigma \\
P_{\mathsf{image} \to \mathsf{image}} &::= \mathsf{cond\text{-}flip}
\end{aligned}
$$

$$
\begin{aligned}
(\mathsf{predict}_{\mathsf{int}}\ x) &= (\mathsf{if}\ \hat{p}(x, \hat{f}(x)) \geq ??_c\ \{\hat{f}(x) = y^*\}^{\Rightarrow}_{??_\epsilon}\ \mathsf{then}\ \hat{f}(x)\ \mathsf{else}\ \varnothing) \\
(\mathsf{predict}_{\mathsf{float}}\ x) &= (\mathsf{if}\ \hat{p}(x, \hat{f}(x)) \geq ??_c\ \{|\hat{f}(x) - y^*| \leq ??_e\}^{\Rightarrow}_{??_\epsilon}\ \mathsf{then}\ \hat{f}(x)\ \mathsf{else}\ \varnothing) \\
(\mathsf{cond\text{-}flip}\ x) &= (\mathsf{if}\ \hat{p}_{\mathsf{flip}}(x, \hat{f}_{\mathsf{flip}}(x)) \geq ??_c\ \{\hat{f}_{\mathsf{flip}}(x) = y^*_{\mathsf{flip}}\}^{\Rightarrow}_{??_\epsilon}\ \mathsf{then}\ (\mathsf{cond\text{-}flip}_0\ x)\ \mathsf{else}\ \varnothing) \\
(\mathsf{cond\text{-}flip}_0\ x) &= (\mathsf{if}\ \hat{f}_{\mathsf{flip}}(x)\ \mathsf{then}\ \mathsf{flip}(x)\ \mathsf{else}\ x) \\
(\mathsf{cond}\text{-}\leq\ y_1\ y_2) &= (\mathsf{if}\ |y_1 - y_2| \geq ??_c\ \{y_1^* \leq y_2^*\}^{\Rightarrow}_{??_\epsilon}\ \mathsf{then}\ y_1 \geq y_2\ \mathsf{else}\ \varnothing) \\
(\mathsf{cond}\text{-}\geq\ y_1\ y_2) &= (\mathsf{if}\ |y_1 - y_2| \geq ??_c\ \{y_1^* \geq y_2^*\}^{\Rightarrow}_{??_\epsilon}\ \mathsf{then}\ y_1 \geq y_2\ \mathsf{else}\ \varnothing)
\end{aligned}
$$

Figure 2: This figure shows our domain-specific language (DSL) of list processing programs over images of inputs. The top half shows the production rules; these rules are implicitly universally quantified over the type variables τ and σ, where $\tau ::= \mathsf{bool} \mid \mathsf{int} \mid \mathsf{float} \mid \mathsf{image} \mid \mathsf{list}(\tau) \mid \tau \to \tau$ and $\sigma ::= \mathsf{int} \mid \mathsf{float} \mid \mathsf{image}$. The bottom half shows the semantics of functions in our language that have statistical specifications.

Algorithm 3 Use learning theory to synthesize $\bar{P}$ that is approximately correct.

procedure SYNTHESIZE($\vec{\alpha}, \psi, \epsilon, e, N, \delta$)
  $\tilde{P} \leftarrow \mathsf{SynthesizePartialSketch}(\psi)$
  $\vec{\alpha}_{\text{synth}}, \vec{\alpha}_{\text{sketch}} \leftarrow \mathsf{Split}(\vec{\alpha})$
  $P \leftarrow \arg\max_{P' \in \mathsf{FillAll}(\tilde{P}, \epsilon, e)} \mathsf{Score}(\mathsf{Sketch}(P', \vec{\alpha}_{\text{synth}}, \delta))$
  return $\mathsf{Sketch}(P, \vec{\alpha}_{\text{sketch}}, \delta)$
end procedure

For clarity, we refer to specifications $\tilde{\psi}$ as *task specifications* and specifications on DSL components as *component specifications*. As a running example, consider the program in Figure 3. This program predicts the value x of the image input^1 (as an integer) and values ℓ of the images in the list input^2 (as real values), and then sums the values in ℓ that are greater than equal to x. It contains three components that have component specifications: the two machine learning components $\mathsf{predict}_{\mathsf{int}}$ and $\mathsf{predict}_{\mathsf{float}}$, along with the inequality $\mathsf{cond}\text{-}\leq$. The first two component specifications ensure that the corresponding machine learning model returns correctly (or $\varnothing$) with high probability. For the last one, note that in the expression $y_1 \leq y_2$, the inputs y_1 and y_2 may have a small amount of prediction error, so if they are to close together (i.e., $|y_1 - y_2| \leq c$ for some $c \in \mathbb{R}_{\leq 0}$), then $y_1 \leq y_2$ might be incorrect. Thus, to ensure $\leq$ returns correctly, $\mathsf{cond}\text{-}\leq$ returns $\varnothing$ if $|y_1 - y_2| \leq c$.

Finally, note that we use $\omega = \Rightarrow$, indicating that our goal is to synthesize $\bar{P}$ such that the the overall success rate is bounded—i.e., $\mathbb{P}_{p(\alpha)}\big([\![\bar{P}]\!]_\alpha = \varnothing \vee |[\![\bar{P}]\!]_\alpha - [\![\bar{P}]\!]^*_\alpha| > e\big) \geq 1 - \epsilon$. We could use $\omega = |$ here if we instead wanted to bound the probability of failure conditioned on $[\![\bar{P}]\!]_\alpha \neq \varnothing$.

Given labeled training examples $\vec{\alpha}$, a task specification $\tilde{\psi}$, a maximum list length N, and a confidence level δ, our algorithm shown in Algorithm 3 synthesizes a complete program $\bar{P}$ that satisfies $\tilde{\psi}$ with probability at least $1 - \delta$. At a high level, this algorithm proceeds in three steps:

- **Step 1:** First, our algorithm uses the logical specification ψ to identify a sketch $\tilde{P}$ whose train semantics is consistent with ψ. Note that the train semantics for sketches in our DSL in Figure 2 are well-defined even when the holes left unfilled. We refer to $\tilde{P}$ as a *partial sketch*, since it has additional holes that cannot be filled by our sketching algorithm.
- **Step 2:** While our algorithm uses our sketching algorithm described in Algorithm 2 to fill holes $??_c$ in $\tilde{P}$, it must first fill the holes $??_\epsilon$ and $??_e$ (described below), which cannot be handled by this algorithm. To this end, it analyzes the program to identify constraints on

Task Specification	$\tilde{\psi}_{\text{ex}} = \big(\psi = \{[1,2,3] \mapsto 3,\ [2,4,2] \mapsto 4\},\ \epsilon = 0.05,\ e = 6,\ N = 3,\ \delta = 0.05\big)$
Partial Sketch	$\tilde{P}_{\text{ex}} = (\mathsf{fold}\ +\ (\mathsf{filter}\ (\underbrace{\mathsf{cond}\text{-}\leq}_{f_1}\ \underbrace{(\mathsf{predict}_{\mathsf{int}}\ \mathsf{input}^1)}_{f_2})\ (\mathsf{map}\ \underbrace{\mathsf{predict}_{\mathsf{float}}}_{f_3}\ \mathsf{input}^2)\ 0))$
Components with Holes	$f_1 = (\lambda y_1\ (\lambda y_2\ (\mathsf{if}\ \lvert y_1 - y_2\rvert \geq ??_c\ \{y_1^* \leq y_2^*\}^{\Rightarrow}_{??_\epsilon}\ \mathsf{then}\ y_1 \leq y_2\ \mathsf{else}\ \varnothing)))$ $f_2 = (\lambda x\ (\mathsf{if}\ \hat{p}(x, \hat{f}(x)) \geq ??_c\ \{\hat{f}(x) = y^*\}^{\Rightarrow}_{??_\epsilon}\ \mathsf{then}\ \hat{f}(x)\ \mathsf{else}\ \varnothing))$ $f_3 = (\lambda x\ (\mathsf{if}\ \hat{p}(x, \hat{f}(x)) \geq ??_c\ \{\lvert\hat{f}(x) - y^*\rvert \leq ??_e\}^{\Rightarrow}_{??_\epsilon}\ \mathsf{then}\ \hat{f}(x)\ \mathsf{else}\ \varnothing))$

Figure 3: Example of a task in our list processing domain. Given $\tilde{\psi}_{\text{ex}}$, the goal is to synthesize a program $\bar{P}$ whose train semantics satisfies ψ, and whose test semantics satisfy $\mathbb{P}_{p(\alpha)}\big([\![\bar{P}]\!]_\alpha = \varnothing \vee \lvert[\![\bar{P}]\!]_\alpha - [\![\bar{P}]\!]^*_\alpha\rvert \leq e\big) \geq 1 - \epsilon$.

the values of ϵ and e that can be assigned to each hole $??_\epsilon$ and $??_e$, respectively and satisfy the desired task specification (ϵ, e). Given candidate values $\vec{\epsilon}$ and $\vec{e}$, it constructs the sketch $P = \mathsf{Fill}(\tilde{P}, \vec{\epsilon}, \vec{e})$, and evaluates the success rate $\mathsf{Score}(P)$ (i.e., how often $[\![P]\!]_\alpha \neq \varnothing$). It chooses the sketch P that maximizes this objective over a finite set of choices of $\vec{\epsilon}$ and $\vec{e}$.

- **Step 3:** Finally, it uses a held-out set of labeled examples $\vec{\alpha}_{\text{sketch}}$ in conjunction with our sketching algorithm in Algorithm 2 to synthesize We use a held-out set since Theorem C.7 only holds if the examples $\vec{\alpha}_{\text{sketch}}$ are not used to construct the sketch P.

In Figure 3, we show the partial sketch $\tilde{P}_{\text{ex}}$ along with two analyses which are used to help compute the search space over $\vec{\epsilon}$ and $\vec{e}$. Below, we describe our DSL and synthesis algorithm in more detail.

D.1 Domain-Specific Language

Our DSL is summarized in Figure 2. To be precise, this figure shows sketches in our language; filling holes in these sketches produces a program in our language. At a high level, the language consists of standard list processing operators such as map, filter, and fold, along with a set of functions that can be applied to individual integers, real numbers, or images.

Machine learning components. Our DSL has three machine learning components: $\mathsf{predict}_{\mathsf{int}}$, $\mathsf{predict}_{\mathsf{float}}$, and cond-flip. The first two predict the value in a given image. They are identical except for their component specification; whereas the integer predictions must be exactly correct, the real-valued predictions are allowed to have bounded error. We describe these specifications below. This difference gives the user flexibility in terms of what kind of guarantees they want to provide.

The third machine learning component checks if the input image is flipped along the vertical axis. We include it to demonstrate how our approach can combine multiple machine learning components. It only returns an image if it is confident about its prediction; otherwise, it returns $\varnothing$.

Component specifications. Intuitively, there are two kinds of component specifications in our language: (i) require that the output is exactly correct, and (ii) require that the error of the output is bounded. There are four components in (i): $\mathsf{predict}_{\mathsf{int}}$, cond-flip, cond-$\leq$, and cond-$\geq$. The first two are straightforward—they consist of a machine learning component, and return the predicted value if the prediction confidence is a threshold to be synthesized, and return $\varnothing$ otherwise.

The latter two are result from challenges handling inequalities on real-valued predictions. In particular, real-valued predictions (i.e., by $\mathsf{predict}_{\mathsf{float}}$) can be wrong by a bounded amount, yet the return value of $\leq$ and $\geq$ is a Boolean value that must be exactly correct. Thus, these components include a component specification indicating that their output must be correct with high probability. Note that the scoring function used in the condition is $|y_1 - y_2|$; intuitively, if the inputs y_1 and y_2 are far apart (i.e., $|y_1 - y_2|$ is large), then the predicted result is less likely to be an error.

The $\mathsf{predict}_{\mathsf{float}}$ component is the only one in (ii). The only difference from $\mathsf{predict}_{\mathsf{int}}$ is that it only requires that the prediction is correct to within some bounded amount of error—i.e., $|\hat{f}(x) - y^*| \leq e$, for some $e \in \mathbb{R}_{\geq 0}$. Note that e is left as a hole to be filled.

Holes. Our language has three kinds of holes. The first two are holes $??_c$ and $??_\epsilon$; these are in our sketch DSL in Figure 2. Note that in that DSL, each component specification could only have either

c or ϵ as a hole, but here we allow both to be left as holes; our algorithm searches over choices of ϵ to fill holes $??_\epsilon$, and uses our sketching algorithm in Algorithm 2 to fill holes $??_c$. The third kind of hole is the hole $??_e$ in the component specification $|\hat{f}(x) - y^*| \leq ??_e$ for $\mathsf{predict}_{\mathsf{float}}$, which indicates the magnitude of error allowed by the prediction of that component. As with $??_\epsilon$ holes, the $??_e$ holes are filled by our algorithm before our sketching algorithm is applied. Intuitively, $??_\epsilon$ (resp., $??_e$) holes must be filled in a way that satisfies the overall ϵ failure probability guarantee (resp., e error guarantee) in the user-provided task specification $\tilde{\psi}$.

D.2 Synthesis Algorithm

Our algorithm (Algorithm 3) takes as input labeled training examples $\vec{\alpha}$, a task specification $\tilde{\psi} = (\psi, \epsilon, e)$, and $\delta \in \mathbb{R}_{>0}$, and returns a program $\bar{P}$ that satisfies $\tilde{\psi}$ with probability $\geq 1 - \delta$.

Step 1: Syntax-guided synthesis. Our algorithm first synthesizes a partial sketch $\tilde{P}$ in our DSL whose train semantics satisfies ψ—i.e., $\mathsf{UNSAT}_{\alpha,y}\big(y = [\![\tilde{P}]\!]^*_\alpha \wedge \neg\psi(\alpha, y)\big)$. Importantly, note that $[\![\tilde{P}]\!]^*_\alpha$ is well-defined even though there are holes in $\tilde{P}$. We can compute $\tilde{P}$ using any standard synthesizer.

Step 2: Sketching ϵ and e. Next, our algorithm fills the holes $??_\epsilon$ in $\tilde{P}$ with values $\vec{\epsilon}$ and holes $??_e$ with values $\vec{e}$ to obtain a sketch $P = \textsc{Fill}(\tilde{P}, \vec{\epsilon}, \vec{e})$. Since P only has holes $??_\epsilon$, we can use Algorithm 2 to fill these holes in a way that guarantees correctness for the given values $\vec{\epsilon}$ and $\vec{e}$—i.e.,

$$\mathbb{P}_{p(\alpha)}\big(|[\![\bar{P}]\!]_\alpha - [\![P]\!]^*_\alpha| \leq e\big) \geq 1 - \epsilon, \tag{10}$$

where $\bar{P}$ is a completion of P where the holes $??_c$ in P have been filled with values $\vec{c}$. We need to use $\bar{P}$ since the test semantics are not well-defined for sketches P. In particular, we need to choose values $\vec{\epsilon}$ and $\vec{e}$ that ensure that (10) holds for *all* possible completions $\bar{P}$ of P.

Furthermore, we not only want to choose $\vec{\epsilon}$ and $\vec{e}$ to ensure correctness, but also to maximize a quantitative property of $\bar{P}$. In particular, we want to choose it in a way that maximizes the probability that P does not return $\varnothing$—i.e., maximize the score

$$\mathsf{Score}(P) = \mathbb{P}_{p(\alpha)}\big([\![\bar{P}]\!]_\alpha \neq \varnothing\big)$$

Note that the score depends critically on the choice of thresholds $\vec{c}$ used to fill holes $??_c$ in P. Thus, given a set of candidate choices $\vec{\epsilon}$ and $\vec{e}$, our algorithm constructs the corresponding sketch $P' = \mathsf{Fill}(\tilde{P}, \vec{\epsilon}, \vec{e})$, uses our sketching algorithm to fill the holes $??_c$ in P' to obtain $\bar{P}' = \mathsf{Sketch}(P', \vec{\alpha}, \delta)$, and finally scores $\bar{P}'$. Then, our algorithm chooses P' with the highest score. In Algorithm 3, we let $\mathsf{FillAll}(\tilde{P}, \epsilon, e)$ denote the set of all sketches P' constructed from candidates $\vec{\epsilon}$ and $\vec{e}$.

One important detail is that Algorithm 2 requires that P is a straight-line program—i.e., it cannot handle loops. For now, we assume that we are given a bound $N \in \mathbb{N}$ on the maximum length of any input list. Then, we can unroll list operations such as map, filter, and fold into straight-line code. Algorithm 3 uses this strategy to apply Algorithm 2 to sketches P. We describe how we can remove the assumption that we have an upper bound N in Section E.

Step 3: Sketching c. Finally, we use Algorithm 2 to choose values $\vec{c}$ to fill holes $??_c$ in the highest scoring sketch P from the previous step, and return the result $\bar{P} = \mathsf{Sketch}(P, \vec{\alpha}_{\mathsf{sketch}}, \delta)$. Importantly, in the previous step, P is chosen based on a subset $\vec{\alpha}_{\mathsf{synth}}$ of the training examples $\vec{\alpha}$, whereas in this step, $\bar{P}$ is constructed based on a disjoint subset $\vec{\alpha}_{\mathsf{sketch}}$. We choose these two subsets to be of equal size since Algorithm 2 is sensitive to the number of examples in $\vec{\alpha}$. This strategy ensures that P does not depend on the random variable $\vec{\alpha}_{\mathsf{sketch}}$, thereby ensuring that Theorem C.7 holds.

D.3 Search Space Over $\vec{\epsilon}$ and $\vec{e}$

Here, we describe how we choose candidates $\vec{\epsilon}$ and $\vec{e}$ in Step 2 so that the candidate sketches $P' = \mathsf{Fill}(\tilde{P}, \vec{\epsilon}, \vec{e})$ satisfy (10). At a high level, for $\vec{\epsilon}$, for each component f of $\tilde{P}$ with an $??_\epsilon$ hole, we compute $[\![\tilde{P}]\!]^\#_f$, which is the number of times f occurs in the unrolled version of $\tilde{P}$; then, we consider $\vec{\epsilon} = (\epsilon_{f_1}, ..., \epsilon_{f_d})$ such that $\sum_f \epsilon_f \leq \epsilon$. For $\vec{e}$, for each component f of $\tilde{P}$ with an $??_e$ hole, we compute $[\![\tilde{P}]\!]^{\mathsf{err}} : \vec{e} \mapsto e'$, which is a linear function mapping $\vec{e}$ to an upper bound e' on the error of the output; then, we consider $\vec{e}$ such that $[\![\tilde{P}]\!]^{\mathsf{err}}(\vec{e}) \leq e$. We provide details below.

$$
\begin{aligned}
[\![(F\ L)]\!]_f^\# &= [\![F]\!]_f^\# + [\![L]\!]_f^\# \\
[\![(\mathsf{fold}\ F\ L\ B)]\!]_f^\# &= N \cdot [\![F]\!]_f^\# + [\![L]\!]_f^\# + [\![B]\!]_f^\# \\
[\![(\mathsf{map}\ F\ L)]\!]_f^\# &= N \cdot [\![F]\!]_f^\# + [\![L]\!]_f^\# \\
[\![(\mathsf{filter}\ F\ L)]\!]_f^\# &= N \cdot [\![F]\!]_f^\# + [\![L]\!]_f^\# \\
[\![(\mathsf{slice}\ L\ I_1\ I_2)]\!]_f^\# &= [\![L]\!]_f^\# + [\![I_1]\!]_f^\# + [\![I_2]\!]_f^\# \\
[\![(\mathsf{length}\ L)]\!]_f^\# &= [\![L]\!]_f^\# \\
[\![f']\!]_f^\# &= \mathbb{1}(f' = f) \\
[\![\mathsf{input}_\tau^i]\!]_f^\# &= 0
\end{aligned}
\qquad
\begin{aligned}
[\![(F\ L)]\!]^{\mathsf{err}} &= [\![F]\!]^{\mathsf{err}}([\![L]\!]^{\mathsf{err}}) \\
[\![(\mathsf{fold}\ F\ L\ B)]\!]^{\mathsf{err}} &= \max_{n \in \{0,1,\ldots,N\}} ([\![F]\!]^{\mathsf{err}})^n([\![L]\!]^{\mathsf{err}}, [\![B]\!]^{\mathsf{err}}) \\
[\![(\mathsf{map}\ F\ L)]\!]^{\mathsf{err}} &= [\![F]\!]^{\mathsf{err}}([\![L]\!]^{\mathsf{err}}) \\
[\![(\mathsf{filter}\ F\ L)]\!]^{\mathsf{err}} &= [\![L]\!]^{\mathsf{err}} \\
[\![(\mathsf{slice}\ L\ I_1\ I_2)]\!]^{\mathsf{err}} &= [\![L]\!]^{\mathsf{err}} \\
[\![(\mathsf{length}\ L)]\!]^{\mathsf{err}} &= 0 \\
[\![f]\!]^{\mathsf{err}} &= \begin{cases} \lambda\eta.e_f & \text{if } f = \mathsf{predict}_{\mathsf{float}} \\ \lambda\eta.\lambda\eta'.\eta + \eta' & \text{if } f \in \{+, -\} \\ \lambda\eta.\eta & \text{otherwise} \end{cases} \\
[\![\mathsf{input}_\tau^i]\!]^{\mathsf{err}} &= 0
\end{aligned}
$$

Figure 4: Rules Algorithm 3 uses to compute the search space over $\vec{\epsilon}$ (left) and $\vec{e}$ (right). In the rule of $[\![\cdot]\!]^{\mathsf{err}}$ for fold, $f^n(\ell, b) = f(\ell, f^{n-1}(\ell, b))$ (and $f^0(\ell, b) = b$) is the function f iterated n times in its second argument. The definitions of η, η', and $\eta + \eta'$ in the rule for $[\![f]\!]^{\mathsf{err}}$ are given in Section D.3.

Search space over $\vec{\epsilon}$. First, we describe our search space over parameter values $\vec{\epsilon}$ used to fill holes $??_\epsilon$ so that the overall failure rate is at most ϵ. Note that here, $\vec{\epsilon} = (\epsilon_{f_1}, ..., \epsilon_{f_k})$, where $\mathcal{F}_{\tilde{P}} = \{f_1, ..., f_k\}$ are subexpressions of $\tilde{P}$ of the form $\mathsf{predict}_{\mathsf{int}}$, $\mathsf{predict}_{\mathsf{float}}$, $\mathsf{cond\text{-}flip}$, $\mathsf{cond\text{-}}\leq$, or $\mathsf{cond\text{-}}\geq$, since each of these subexpressions contains exactly one hole of the form $??_\epsilon$.

Intuitively, we can ensure correctness via a union bound—i.e., if the sum of the ϵ_f is bounded by ϵ, then the overall failure probability is also bounded by ϵ. The key caveat is that to apply Algorithm 2, we need to unroll the sketch $P = \mathsf{Fill}(\tilde{P}, \vec{\epsilon}, \vec{e})$. Thus, we need to count a value ϵ_f multiple times if the corresponding subexpression f occurs multiple times in the unrolled version of P.

In particular, the rules $[\![P]\!]_{f'}^\#$ shown in Figure 4 are designed to count the number of occurrences of the subexpression f' in the *unrolled* version of P. Note that in these rules, f' refers to a *specific* subexpression, and $\mathbb{1}(f = f')$ refers to whether f is that specific subexpression; multiple uses of the same construct (e.g., a program with two uses of $\mathsf{predict}_{\mathsf{int}}$) are counted separately. These rules are straightforward; for instance, when unrolling the fold operator, the expressions for the list L and the initial value B are included exactly once, whereas the function expression F occurs N times. Then, to ensure that the failure probability is at most ϵ, it suffices for $\vec{\epsilon}$ to satisfy

$$\sum_{f \in \mathcal{F}_{\tilde{P}}} [\![\tilde{P}]\!]_f^\# \cdot \epsilon_f \leq \epsilon. \tag{11}$$

Now, let $\Delta_{\mathcal{F}_{\tilde{P}}} = \{\vec{x} \in \mathbb{R}^{|\mathcal{F}_{\tilde{P}}|} \mid \forall f\ .\ 0 \leq x_f \leq 1 \wedge \sum_{f \in \mathcal{F}_{\tilde{P}}} x_f = 1\}$ be the regular simplex in $\mathbb{R}^{|\mathcal{F}_{\tilde{P}}|}$. Now, given any $\vec{x} \in \Delta_{\mathcal{F}_{\tilde{P}}}$, letting $\epsilon_f = x_f \cdot \epsilon / [\![\tilde{P}]\!]_f^\#$, then (11) is satisfied. In our algorithm, we search over a finite set of points from $\Delta_{\mathcal{F}_{\tilde{P}}}$, and construct the corresponding set of values $\vec{\epsilon}$.

In Figure 6, the rule for filter applies $f_1 = \mathsf{cond\text{-}}\leq$ and $f_2 = \mathsf{predict}_{\mathsf{int}}$ each $N = 3$ times (where N is the given bound on the list length), so we have $[\![\tilde{P}_{\text{ex}}]\!]_{f_1}^\# = [\![\tilde{P}_{\text{ex}}]\!]_{f_2}^\# = 3$. Similarly, map applies $f_3 = \mathsf{predict}_{\mathsf{float}}$ a total of $N = 3$ times, so $[\![\tilde{P}_{\text{ex}}]\!]_{f_1}^\# = 3$. As an example of a point in our search space, taking $\vec{x} = (1/3, 1/3, 1/3)$ yields $\vec{\epsilon} = (1/9, 1/9, 1/9)$.

Search space over $\vec{e}$. Next, we describe our search space over parameter values $\vec{e}$ used to fill holes $??_e$ so the overall error is at most e. Similar to before, $\vec{e} = (e_{f_1}, ..., e_{f_h})$, but this time $\mathcal{G}_{\tilde{P}} = \{f_1, ..., f_h\}$ are subexpressions of $\tilde{P}$ of the form $\mathsf{predict}_{\mathsf{float}}$, which each contain exactly one hole of the form $??_e$. In this case, we define an analysis that bounds the overall error of the output of $\bar{P} = \mathsf{Fill}(P, \vec{\epsilon}, \vec{e})$ for any $\vec{\epsilon}$ as a function of $\vec{e}$. More precisely, $[\![P]\!]^{\mathsf{err}}$ satisfies the following property:

$$\left\| [\![\mathsf{Fill}(P, \vec{\epsilon}, \vec{e})]\!]_\alpha - [\![P]\!]_\alpha^* \right\|_\infty \leq [\![P]\!]^{\mathsf{err}}(\vec{e}) \tag{12}$$

for all $\vec{\epsilon}$ and $\vec{e}$, and for all α such that all component specifications in $\mathsf{Fill}(P, \vec{\epsilon}, \vec{e}$ hold for $\vec{\alpha}$. In other words, (12) bounds the error of the output for examples α such that predictions fall within the desired error bounds (failures happen with probability at most ϵ according to our choices of $\vec{\epsilon}$).

Note that (12) uses the L_∞ norm. For scalar outputs, we have $\|x - x'\|_\infty = |x - x'|$. For list outputs, for the L_∞ norm to be well-defined, we need to ensure that $x = [\![\mathsf{Fill}(P, \vec{\epsilon}, \vec{e})]\!]_\alpha$ and $x' = [\![\bar{P}]\!]_\alpha^*$

are of the same length (at least, when all component specifications are satisfied). In particular, the only potential case where x and x' have unequal lengths is if $\bar{P}$ contains a filter operator. We focus on filtering real-valued lists; filtering integer-valued lists is similar (and there are no operations to filter list-valued lists or image-valued lists). In the real-valued case, the filter function must be either cond-$\leq$ and cond-$\geq$. Assuming the component specifications on cond-$\leq$ and cond-$\geq$ are satisfied, then their (Boolean) outputs are guaranteed to be equal, so the outputs of the filter operator have equal length under train and test semantics. Thus, $\|x - x'\|_\infty$ is well-defined.

Given $[\![P]\!]^{\mathsf{err}}$, our goal is to compute $\vec{e}$ satisfying

$$[\![P]\!]^{\mathsf{err}}(\vec{e}) \leq e. \tag{13}$$

As with $\vec{\epsilon}$, we can construct a candidate $\vec{e}$ for any point in $x \in \Delta_{\mathcal{G}_{\tilde{P}}}$ by taking $e_f = x_f \cdot e/a_f$, where $[\![\tilde{P}]\!]^{\mathsf{err}} = \sum_{f \in \mathcal{G}_{\tilde{P}}} a_f \cdot e_f$. In Figure 3, we have $[\![\tilde{P}]\!]^{\mathsf{err}} = 3 \cdot e_{f_3}$, so there is a single candidate $e_{f_3} = e/3$.

Next, we describe the rules $[\![P]\!]^{\mathsf{err}}$, which are shown in Figure 4 (right). They compute an symbolic expression of the form $\eta = \sum_{f \in \mathcal{G}_{\tilde{P}}} a_f \cdot e_f \in \mathcal{E}_{\tilde{P}}$, where $a_f \in \mathbb{R}_{\geq 0}$ and e_f is a symbol. Given $\vec{e}$, an expression η can be evaluated by substituting $\vec{e}$ for the symbols e_f in η. Now, the rule for function application assumes given a function abstraction $[\![F]\!]^{\mathsf{err}} : \mathcal{E}_{\tilde{P}} \to \mathcal{E}_{\tilde{P}}$. In particular, $[\![F]\!]^{\mathsf{err}}$ is the identity function except for $\mathsf{predict}_{\mathsf{float}}$, $+$, and $-$. The case $\mathsf{predict}_{\mathsf{float}}$ follows since we have assumed that the component specification holes, and the component specification for $f = \mathsf{predict}_{\mathsf{float}}$ says exactly that $|[\![\bar{f}]\!]_\alpha - [\![f]\!]^*_\alpha| \leq e_f$ for any completion $\bar{f}$ of f. For $+$ and $-$, letting $\eta = \sum_{f \in \mathcal{G}_{\tilde{P}}} a_f \cdot e_f$ and $\eta' = \sum_{f \in \mathcal{G}_{\tilde{P}}} a'_f \cdot e_f$, we define $\eta + \eta' = \sum_{f \in \mathcal{G}_{\tilde{P}}} (a_f + a'_f) \cdot e_f$. The rule for map follows since we are using the L_∞ norm, so the bound is applied elementwise. The remaining rules are straightforward.

In Figure 3, the rule for $\mathsf{predict}_{\mathsf{float}}$ returns e_{f_3}, so the rule for map returns $3 \cdot e_{f_3}$ (since the given bound on the list length is $N = 3$). The remaining rules propagate this value, so $[\![\tilde{P}_{\mathsf{ex}}]\!]^{\mathsf{err}} = 3 \cdot e_{f_3}$.

Finally, the fact that $[\![\cdot]\!]^{\mathsf{err}}$ is a linear function follows by structural induction. Additional components (e.g., multiplication) can result in nonlinear expressions, but a similar approach applies.

Overall search space. Our overall search space consists of pairs $\vec{\epsilon}$ and $\vec{e}$ such that $\vec{\epsilon}$ satisfies (11) and $\vec{e}$ satisfies (13); given such a pair, $\mathsf{FillAll}(\tilde{P}, \epsilon, e)$ includes the program $P = \mathsf{Fill}(\tilde{P}, \vec{\epsilon}, \vec{e})$. Together, (11) and (13) ensure the desired property (10). In particular, for any completion $\bar{P}$ of P, (13) ensures that $|[\![\bar{P}]\!]_\alpha - [\![P]\!]^*_\alpha| \leq e$ as long as α satisfies all the component specifications, and (11) ensures that α satisfies the component specifications with probability at least $1 - \epsilon$ over $p(\alpha)$.

E Discussion

Generality. In Section D, we described a synthesizer tailored to the language in Figure 2. Our approach generalizes straightforwardly in several ways. First, we note that the $\mathsf{predict}_{\mathsf{int}}$ and $\mathsf{predict}_{\mathsf{float}}$ machine learning components are not specific to images of integers, and represent general classification and regression problems, respectively. Furthermore, we can also include additional list processing components as long as we provide the abstract semantics $[\![\cdot]\!]^\#$ and $[\![\cdot]\!]^{\mathsf{err}}$. Thus, our algorithm can be viewed as a general algorithm for synthesizing list processing programs with DNNs for classification and regression, where the specification is that with high probability, the program should return the either the correct answer (within some given error tolerance) or $\varnothing$.

We can also modify the specification in certain ways; for instance, we can ignore certain kinds of errors by modifying the annotations on $\mathsf{predict}_{\mathsf{int}}$ and $\mathsf{predict}_{\mathsf{float}}$. For instance, to allow for one-sided errors in regression problems (e.g., it is fine to say "person" when there isn't one but not vice versa), we can simply drop the absolute values from the task specification ψ and from the annotations on $\mathsf{predict}_{\mathsf{float}}$. For this case, the algorithm for allocating errors e works as is, but in general, it may need to be modified to ensure the annotations imply the specification.

Bound on examples. In Section D, we assumed given a bound N on the maximum length of any list observed during program execution. Intuitively, we can circumvent this assumption by computing a high probability bound N; the error probability can be included in the user-provided allowable error rate ϵ. In particular, let $[\![\tilde{P}]\!]^{\mathsf{len}}_\alpha$ denote the maximum list length observed while executing $\tilde{P}$ on input

DSL Variant	Task	∅ Rate			Failure Rate		
		StatCoder	No Search	$k = 0$	StatCoder	No Search	$k = 0$
int	sum $x \in \ell$	0.000	0.000	0.177	0.018	0.018	0.001
	max $x \in \ell$	0.000	0.000	0.177	0.008	0.008	0.001
	sum $x \in \ell$ that are $\leq k$	0.001	0.022	0.206	0.016	0.010	0.001
	max first k elements $x \in \ell$	0.000	0.008	0.195	0.007	0.007	0.000
	count $x \in \ell$ that are $\leq k$	0.001	0.022	0.206	0.000	0.000	0.000
average	–	0.000	0.010	0.192	0.010	0.009	0.001
float	sum $x \in \ell$	0.000	0.000	0.000	0.001	0.001	0.001
	max $x \in \ell$	0.000	0.000	0.000	0.000	0.000	0.000
	sum $x \in \ell$ that are $\leq k$	0.000	1.000	1.000	0.010	0.000	0.000
	max first k elements $x \in \ell$	0.000	0.005	0.177	0.000	0.000	0.000
	count $x \in \ell$ that are $\leq k$	0.000	1.000	1.000	0.000	0.000	0.000
average	–	0.000	0.401	0.435	0.002	0.000	0.000
flip	sum $x \in \ell$	0.015	0.016	0.230	0.012	0.012	0.001
	max $x \in \ell$	0.015	0.016	0.230	0.006	0.006	0.001
	sum $x \in \ell$ that are $\leq k$	0.025	0.085	0.265	0.012	0.004	0.001
	max first k elements $x \in \ell$	0.063	0.046	0.258	0.005	0.004	0.000
	count $x \in \ell$ that are $\leq k$	0.025	0.085	0.265	0.000	0.000	0.000
average	–	0.029	0.050	0.250	0.007	0.005	0.001
fast	sum $x \in \ell$	0.033	0.033	0.706	0.026	0.026	0.000
	max $x \in \ell$	0.033	0.033	0.706	0.008	0.008	0.000
	sum $x \in \ell$ that are $\leq k$	0.039	0.127	0.755	0.023	0.005	0.000
	max first k elements $x \in \ell$	0.035	0.061	1.000	0.010	0.007	0.000
	count $x \in \ell$ that are $\leq k$	0.039	0.127	0.755	0.000	0.000	0.000
average	–	0.036	0.076	0.784	0.013	0.009	0.000
overall	–	0.016	0.134	0.415	0.008	0.006	0.000

Table 2: We show results on synthesizing list processing programs, for both our approach (StatCoder) and the baseline that does not search over $\vec{\epsilon}$ and $\vec{e}$ ("No Search"). For each DSL variant and each task, we show the "∅ Rate" $\mathbb{P}_{p(\alpha)}(\llbracket \bar{P} \rrbracket_\alpha = \varnothing)$, and the "Failure Rate" $\mathbb{P}_{p(\alpha)}(\llbracket \bar{P} \rrbracket_\alpha \neq \varnothing \wedge |\llbracket \bar{P} \rrbracket_\alpha - \llbracket \bar{P} \rrbracket^*_\alpha = \varnothing|) > e$.

α. Then, suppose we can obtain N such that

$$\mathbb{P}_{p(\alpha)}\big(\llbracket \tilde{P} \rrbracket^{\mathsf{len}}_\alpha \leq N\big) \geq 1 - \frac{\epsilon}{2}.$$

Now, if we synthesize a completion $\bar{P}$ of $\tilde{P}$ with overall error rate $\leq \epsilon/2$, then by a union bound, the total error rate is $\leq \epsilon$. Finally, to obtain such an N, we can use the specification

$$\llbracket \tilde{P} \rrbracket^{\mathsf{len}}_\alpha \leq ?? \; \{\mathsf{true}\}^{\Rightarrow}_{\epsilon/2}.$$

Letting c be the synthesized value used to fill the hole, the specification says that $\llbracket \tilde{P} \rrbracket^{\mathsf{len}}_\alpha \leq c$ with probability at least $\epsilon/2$ according to $p(\alpha)$, which is exactly the desired condition on N; thus, we can take $N = c$. Note that since the specification is true, we can use either $|$ or $\Rightarrow$.

F Evaluation

We describe our evaluation on synthesizing list processing programs, as well as on three case studies: (i) a state-of-the-art image classifier, (ii) a random forest trained to predict Warfarin drug dosage, and (iii) object detection.

F.1 Synthesizing List Processing Programs with Image Classification

Experimental setup. We evaluate our synthesis algorithm on our list processing domain in Section D. Inputs are lists of MNIST digits [20]. We use a convolutional DNN (two convolutional layers followed by two fully connected layers, with ReLU activations) [21] to predict the integer in an image, trained on the MNIST training set; it achieves 99.2% accuracy. We also train a single layer DNN, which is 4.04× faster but only 98.5% accurate. Finally, for inputs with the flip component, with consider input images flipped along their horizontal axis. We train a DNN to predict whether a given image is flipped; it achieves 99.6% accuracy.

(b) (c)

Figure 5: For list processing programs, we show $\varnothing$ rate (black) and failure rate (red) as a function of (a) ϵ, (b) δ, and (c) e, on average for (a,b) "Int" programs and (c) "Float" programs. Defaults are $\epsilon = \delta = 0.05$ and $e = 6.0$.

For the synthesizer, we use a standard enumerative synthesizer that returns the smallest program in terms of depth (but chooses arbitrarily among equal depth programs). We give it 5 labeled input-output examples as a specification ψ, along with the type of the function to be synthesized [22, 23]. For the search space over each $\vec{\epsilon}$ and $\vec{e}$, we consider values $\vec{x}_0 \in \{1, 3, 5\}^d$, where $d = |\mathcal{F}_{\tilde{P}}|$ or $d = |\mathcal{G}_{\tilde{P}}|$, and then take $\vec{x} = \vec{x}_0 / \|\vec{x}_0\|_1$ to normalize it to Δ^d. We also compare to (i) a baseline "No Search", which only considers a single $\vec{x}_0 = (1, ..., 1)$, and (ii) a baseline "$k = 0$", which uses a variant of our generalization bound that uses either $k = 0$ (or $k = \varnothing$, if there are insufficient samples); this strategy captures the guarantees provided by traditional generalization bounds from statistical learning theory [10, 24, 25]. We use our algorithm with parameters $\epsilon = \delta = 0.05$, $e = 6$, and $N = 3$. We use 2500 MNIST test set images for each α_{synth} and α_{sketch}, and the remaining 5000 for evaluation. Next, we consider four variants of our DSL:

- **Int:** Restrict to components with integer type and omit the cond-flip component
- **Float:** Same as "int", but include components with real types
- **Flip:** Same as "int", but include the flip component
- **Fast:** Same as "int", but use the fast neural network.

For each variant, we consider five list processing tasks, which are designed to exercise different kinds of components. These programs all take as input a list ℓ of images $x \in \ell$; in addition, several of them take as input a second image k that encodes some information relevant to task. Then, they output an integer or real value (as specified by ψ). The tasks are shared across the different DSL variants, but specific programs change based on the available components.

Results. We show results in Table 2. For the program $\bar{P}$ synthesized using each our approach StatCoder and our baseline that does not search over $\vec{\epsilon}$ and $\vec{e}$, we show the following metrics:

- $\varnothing$ **Rate:** The rate at which $\bar{P}$ returns $\varnothing$—i.e., $\mathbb{P}_{p(\alpha)}(\llbracket \bar{P} \rrbracket_\alpha = \varnothing)$.
- **Failure Rate:** The rate at which $\bar{P}$ makes mistakes—i.e.,

$$\mathbb{P}_{p(\alpha)}\big(\llbracket \bar{P} \rrbracket_\alpha \neq \varnothing \wedge |\llbracket \bar{P} \rrbracket_\alpha - \llbracket \bar{P} \rrbracket_\alpha^* = \varnothing|\big) > e.$$

As can be seen, both StatCoder and the baseline always achieve the desired failure rate bound of $\epsilon = 0.05$. Furthermore, by searching over candidates $\vec{\epsilon}$ and $\vec{e}$, StatCoder substantially outperforms the baseline, achieving an $8\times$ reduction in $\varnothing$ rate on average. For simpler programs (i.e., sum and max), the two perform similarly since there is only a single hole, so the search space only contains one candidate. However, for larger programs, the search improves performance by up to an order of magnitude. There is a single case where the baseline performs better (the fourth program in the "flip" DSL), due to random chance since the dataset α_{sketch} used to synthesize the final program $\bar{P}$ from $\tilde{P}$ differs from the dataset α_{synth} used to choose $\vec{\epsilon}$ and $\vec{e}$. StatCoder outperforms the "$k = 0$" baseline by an even larger margin, due to the fact that the generalization bound is overly conservative; these results demonstrate the importance of using a generalization bound specialized to our setting rather than a more traditional generalization bound that minimizes the empirical risk.

Next, in Figure 5, we show how these results vary as a function of the specification parameters ϵ, δ, and e. As can be seen, ϵ has the largest effect on $\varnothing$ and failure rates, followed by e; as expected, δ has almost no effect since the dependence of our bound on δ is logarithmic.

Finally, we note that the failure rates for the "fast" DSL are very low. Thus, we could use our technique to chain together the fast program with the slow one, along the same lines as discussed in our case study in Section F.3; we estimate that doing so results in a $3\times$ speedup on average.

Task	∅ Rate			Failure Rate		
	StatCoder	No Search	$k=0$	StatCoder	No Search	$k=0$
count the number of people in x	0.054	0.054	0.901	0.124	0.124	0.003
check if x contains a person	0.054	0.054	0.901	0.124	0.124	0.003
count people near the center of x	0.290	0.290	0.901	0.032	0.032	0.003
find people near a car	0.901	0.901	1.000	0.003	0.003	0.000
minimum distance from a person to the center of x	0.149	0.149	0.901	0.023	0.023	0.000
average	0.290	0.290	0.921	0.061	0.061	0.002

Table 3: We show results on synthesizing list processing programs over object detection, for our approach StatCoder. For each DSL variant and each task, we show the "∅ Rate" $\mathbb{P}_{p(\alpha)}(\llbracket \bar{P} \rrbracket_\alpha = \varnothing)$, and the "Failure Rate" $\mathbb{P}_{p(\alpha)}(\llbracket \bar{P} \rrbracket_\alpha \neq \varnothing \wedge |\llbracket \bar{P} \rrbracket_\alpha - \llbracket \bar{P} \rrbracket^*_\alpha = \varnothing|) > e$. Parameters are $\epsilon = \delta = 0.2$ and $e = 20.0$.

F.2 Synthesizing List Processing Programs with Object Detection

Experimental setup. Next, we consider synthesizing programs that operate over the predictions made by a state-of-the-art DNN for object detection. We assume given a DNN component $\hat{f}$ that given an image x, is designed to detect people and cars in x. We use a pretrained state-of-the-art object detector called Faster R-CNN [4] available in PyTorch [11], tailored to the COCO dataset [12], which is a dataset of real-world images containing people, cars, and other objects. There are multiple variants of Faster R-CNN; we use the most accurate one, X101-FPN with $3\times$ learning rate schedule.

We represent this DNN as a component $\hat{f} : \mathcal{X} \to \mathcal{Y} = \mathcal{D}^* \times \mathbb{R}$, where $\hat{f}(x) = (\hat{y}(x), \hat{p}(x))$ consists of a list of *detections* $d \in \hat{y}(x)$ along with a correctness score $\hat{p}(x)$ that the prediction is correct. Each detection $d \in \mathcal{D} = \mathbb{R}^2 \times \mathcal{Z}$ is itself a tuple $d = (b, z)$ including the position b and predicted category of the object. The ground truth label y^* for an image x is a list of detections $d \in y^*$. In general, we cannot expect to get a perfect match between the predicted bounding boxes and the ground truth ones. Typically, two detections d, d^* *match*, denoted $\|d - d^*\| \leq e$, where e is a specified error tolerance, if the distance between their centers satisfies $\|b - b^*\|_\infty \leq e$. Furthermore, we write $\|\hat{y}(x) - y^*\| \leq e$ if $|\hat{y}(x)| = |y^*|$ and there exists a one-to-one correspondence between $d \in \hat{f}(x)$ and $d^* \in y^*$ such that $\|d - d^*\| \leq e$. Then, we define $\mathsf{predict} : \mathcal{X} \to (\mathcal{Y} \cup \varnothing)$ by

$$(\mathsf{predict}\ x) = (\mathsf{if}\ \hat{p}(x) \geq ??_c\ \{\|\hat{y}(x) - y^*\| \leq ??_e\}^{\Rrightarrow}_{??_\epsilon}\ \mathsf{then}\ \hat{y}(x)\ \mathsf{else}\ \varnothing).$$

In other words, the specification says that a correct prediction is if the error tolerance is below a level $??_e$ to be specified. Thus, given e and ϵ to fill $??_e$ and $??_\epsilon$, respectively, our sketching algorithm synthesizes a threshold c to fill $??_c$ in a way that guarantees that this specification holds. Then, predict returns $\hat{y}(x)$ if the DNN is sufficiently confident in its prediction, and $\varnothing$ otherwise.

We can use this component in conjunction with our synthesis algorithm in the same way that it uses $\mathsf{predict}_{\mathsf{float}}$. In particular, we define the abstract semantics

$$\llbracket(\mathsf{predict}\ x)\rrbracket^{\mathsf{err}} = \lambda\eta.e_{\mathsf{predict}}.$$

These semantics enable it to select the error tolerance e to fill $??_e$. The remainder of the synthesis algorithm proceeds as in Section F.1. We use parameters $\epsilon = \delta = 0.2$, $e = 20.0$, and $N = 3$, and use $n = 1000$ COCO validation set images for each α_{synth} and α_{sketch} and the remaining 1503 for evaluation. We use larger ϵ and δ since the accuracy of the object detector is significantly lower than that of the image classifier, so the $\varnothing$ rates are very high for smaller choices.

We evaluate our approach on synthesizing five programs, which include additional list processing components: (i) $(\mathsf{product}\ L\ L')$, which returns the list of all pairs (x, x') such that $x \in \llbracket L \rrbracket$ and $x' \in \llbracket L' \rrbracket$, (ii) $(\mathsf{compose}\ f\ f')$, which returns the composition $\lambda x.f(f'(x))$, (iii) $(\mathrm{is}_{z'}\ D)$, which returns $\mathbb{1}(z = z')$, where $\llbracket D \rrbracket = (b, z)$ is a detection and $z' \in \mathcal{Z}$ is an object category, and (iv) $(\mathsf{distance}\ D\ D')$, which returns the distance $\|b - b'\|_\infty$ between two detections $\llbracket D \rrbracket = (b, z)$ and $\llbracket D' \rrbracket = (b', z')$. Their abstract semantics are straightforward: for $\llbracket\cdot\rrbracket^\#$, they each evaluate each of their arguments once, and for $\llbracket\cdot\rrbracket^{\mathsf{err}}$, the only one that propagates errors is distance, for which

$$\llbracket(\mathsf{distance}\ D\ D')\rrbracket^{\mathsf{err}} = \llbracket D \rrbracket^{\mathsf{err}} + \llbracket D' \rrbracket^{\mathsf{err}}.$$

Results. We provide results in Table 3. The trends are similar to Section F.1; the main difference is that search does not help in this case, likely because there is only a single machine learning component

```
def is_person(x, y_true=None):
    if 1.0 - f(x) <= ??1 {y_true} [|, 0.05]:
        return True
    else:
        return False

def is_person_fast(x):
    if 1.0 - f_fast(x) <= ??2 {is_person(x)} [|, 0.05]:
        return is_person(x)
    else:
        return False
```

```
def monitor_correctness(x):
    if np.random.uniform() <= 0.99:
        return
    passert 1.0 - f_fast(x) <= ??2 {is_person(x)} [|, 0.05]

def monitor_speed(x):
    passert 1.0 - f_fast(x) > ??2 {true} [|, ??3]
```

Figure 6: A program used to predict whether an image x contains a person. Specifications are shown in green; curly brackets is the specification and square brackets is the value of ϵ. The corresponding inequality with a hole in blue. Holes with the same number are filled with the same value.

(b) (c)

(e) (f)

(h) (i)

(j)

Figure 7: For ResNet alone, we show the recall (red), desired lower bound on the recall (blue), and precision (black) as a function of (a) ϵ, (b) δ, (c) $y \in \mathcal{Y}$, and (j) the number of synthesis examples n; the defaults are $\epsilon = \delta = 0.05$, $n = 25,000$, and $y =$ "car", except in (c) we use $\epsilon = 0.1$ to facilitate the comparison with (f). We show the same values for ResNet+AlexNet as a function of (d) ϵ, (e) δ, and (f) $y \in \mathcal{Y}$. For ResNet+AlexNet (black) compared to AlexNet alone (green), we show the running time as a function of (g) ϵ, (h) δ, and (i) $y \in \mathcal{Y}$; we omit ResNet alone since its running time (82.6 minutes) is significantly above the scale.

so optimizing the allocation does not significantly affect performance. Finally, we can chain these programs with a faster object detector to reduce running time; see Section F.5.

F.3 Case Study 1: ImageNet Image Classification

Correctness. Consider program shown in Figure 6, which classifies images as "person" (returns true) or "not person" (returns false). The function `is_person` takes as input an image x, and optionally the ground truth label y^* (which is only used during sketching). The specification in `is_person` says that the program should return true with high probability if the image is of a person (i.e., $y^* = 1$). The predicate $\mathbb{1}(1 - f(x) \leq c)$ is shown in blue, where the value of c has been left as a hole `??1`, the specification $y^* = 1$ is shown in green in the curly braces, and the value $\epsilon = 0.05$ is shown in green in the square braces. We perform a case study in the context of this program (though for labels other than "person"). We consider the ImageNet dataset [19], a large image classification benchmark with over one million images in 1000 categories, including various different animals and inanimate objects. We consider the ResNet-152 DNN architecture [8], a state-of-the-art image classification model trained on ImageNet that achieves about 88% accuracy overall. For both architectures, we use the implementation in PyTorch [11].

To use our system, we split the ImageNet validation set consisting of 50,000 held-out images into (at most) 25,000 for synthesis (i.e., the *synthesis set*) and 25,000 for validation. Because ImageNet has so many labels, each object category has very few examples in the validation dataset (50 on average). Thus, we group the labels into larger, coarse-grained categories, focusing on ones that correspond to many fine-grained ImageNet labels. We consider "dog" (130 labels, 6,500 images) "bird" (59 labels, 2,950 images), "insect" (27 labels, 1,350 images), "car" (21 labels, 1050 images), "snake" (17 labels, 850 images), and "cat" (13 labels, 650 images). The default one we use is "car"; this category contains vehicles such as passenger cars, bikes, busses, trolleys, etc. For the scoring function, given a coarse-grained category $Y \subseteq \mathcal{Y}$, we use the sum of the fine-grained label probabilities—i.e., $f(x) = \sum_{y \in Y} p(x, y)$, where $p(x, y)$ is the predicted probability of label y according to ResNet-152.

Then, we use our sketching algorithm to synthesize c to fill `??1`. We show results in the first and fourth rows of Figure 7. Note that the red curves ideally equal the blue curves, but are slightly

conservative to account for synthesis being based on finitely many samples. The value of ϵ has the biggest effect on performance, since it directly governs recall; as ϵ grows, recall drops (as desired) and precision substantially improves. In contrast, the performance does not vary significantly with δ. These trends match sample complexity guarantees from learning theory relevant to our setting of $n = O(\log(1/\delta)/\epsilon)$ [24, 25]. Next, as n grows larger, recall can more closely match the desired maximum, allowing precision to improve dramatically (the non-monotone effect is most likely due to random chance). Finally, the dependence on the target label is also governed by the number of synthesis images in each category.

Improving speed. Next, we describe how our framework can be used to compose f with a second DNN f_{fast}, which is much faster than f but has lower accuracy. Intuitively, we want to use f_{fast} when we can guarantee its prediction is correct with high probability, and use f otherwise. This approach has been used to reduce running time [26, 27]; our framework can be used to do so while providing rigorous accuracy guarantees.

The code for this approach is shown in `is_person_fast` in Figure 6. As before, the idea is to compute a threshold c' such that the prediction $f_{\text{fast}}(x) \geq 1 - c'$ is correct with high probability. There are two differences. First, if we conclude that there might be a person in the image according to f_{fast}, then we return the prediction according to f (instead of true). While f_{fast} is guaranteed to detect 95% images with people with high probability, it may have more false positives than f; calling f after f_{fast} reduces these false positives. Second, the correctness guarantee is with respect to the prediction $\hat{y} = \mathbb{1}(f(x) \geq 1 - c)$ rather than y^*. We could use y^*, but there is no need—if $\hat{y}$ is incorrect, then it is not helpful for f_{fast} to predict correctly since it falls back on $\hat{y}$.

For f_{fast}, we use AlexNet, which achieves about 57% accuracy overall; in particular, we use $f_{\text{fast}}(x) = \sum_{y \in Y} p_{\text{fast}}(x, y)$, where $p_{\text{fast}}(x, y)$ is the predicted probability of label y according to AlexNet. Then, we conclude that x (may) have label y if $f_{\text{fast}}(x) \geq 1 - c'$, where c' is synthesized by our algorithm. We obtain results on an Nvidia GeForce RTX 2080 Ti GPU. We show results on the second and third rows of Figure 7. All results shown are for the combined predictions (i.e., using both AlexNet and ResNet), and are estimated on the validation set. For running time, we omit results for ResNet since its running time is 82.6 minutes, which is more than $4\times$ the running time of our combined model. For the "dog" category, our approach reduces running time $6\times$ from 82.6 minutes to 13.8 minutes without any sacrifice in precision or recall.

Thus, our approach significantly reduces running time while achieving the desired error rate. Furthermore, comparing to Figure 7 (d), the precision does not significantly decrease across most labels. It does suffer for the labels "car" and "snake". Intuitively, for these labels, there are relatively few examples in the synthesis set, so the synthesis algorithm needs to choose more conservative thresholds. Since the fast program has two thresholds whereas the original program only has one, it is more conservative in the latter case. This difference is reflected in the fact that Figure 7 (e) has higher recall than (d), especially for "car" and "snake".

Importantly, these results rely on the fact that we are tailoring our predictions to a single category—i.e., our system enables the user to tailor the predictions of pretrained DNN models such as ResNet and AlexNet to their desired task. For instance, it can focus on predicting cars rather than achieving good performance on all 1000 ImageNet categories.

Runtime monitoring. As described in Section B.3, our framework can monitor the synthesized program at runtime, which is useful since PAC guarantees are specific to the data distribution $p(x, y^*)$. Thus, if the program is executed on data from a different distribution, called *distribution shift* [17, 18], then our guarantees may not hold. Monitoring requires us to obtain ground truth labels y^* for inputs x encountered at run time; then, we use these ground truth labels to estimate the failure rate of the model and ensure it is below the desired value ϵ.

We show how we can monitor the correctness of `is_person_fast`. In this case, we can easily obtain ground truth labels since the specification for `??2` can be obtained by evaluating $f(x)$. We want to avoid running f on every input since this would defeat the purpose of using a fast DNN; instead, we might run it once every N iterations for some large N. The function `monitor_correctness` implements this check, generating a ground truth label once every $N = 100$ iterations on average. Note that we formulate the check as a probabilistic assertion [28]—i.e.,

$$\textsf{passert } 1 - f_{\text{fast}}(x) \leq c' \ \{1 - f(x) \leq c\}^{|}_{0.05},$$

```
def predict_warfarin_dose(x, y_true=None):
    y = argmax([(ys, f(x, y)) for y in ['low', 'med', 'high']])
    if y == 'low' and f(x, 'low') >= ??1 {y_true != 'high'} [|, 0.05]
        return 'low'
    if y == 'high' and f(x, 'high') >= ??2 {y_true != 'low'} [|, 0.05]
        return 'high'
    return 'med'
```

```
def monitor_correctness(x):
    y = argmax([(ys, f(x, y)) for y in ['low', 'med', 'high']])
y_true = obtain_result(x)
    if y == 'low':
 passert f(x, 'low') >= ??1 {y_true != 'high'} [|, 0.05]
    if y == 'high':
        passert f(x, 'high') >= ??2 {y_true != 'low'} [|, 0.05]
```

Figure 8: A program that predicts the Warfarin dose for a patient with covariates x. Specifications are shown in green; curly brackets is the specification and square brackets is the value of ϵ. The corresponding inequality with a hole in blue. Holes with the same number are filled with the same value.

(b) (c)

(e) (f)

Figure 9: We show the error rate (top) and accuracy (bottom) for our program (black), the random forest (red), always predicting "medium" (green) as a function of (a,d) ϵ, (b,e) δ, and (c,f) the number of synthesis examples n; for the top plots, we also show the desired upper bound on the error rate (blue).

which has the semantics

$$\mathbb{P}_{p(x,y^*)}\big(1 - f_{\text{fast}}(x) \leq c' \mid 1 - f(x) \leq c\big),$$

which is the specification in `is_person_fast`. When our framework synthesizes a value c' to fill `??2` in `is_person_fast`, it uses the same value to fill `??2` in `monitor_correctness`. Then, at run time, it accumulates pairs $(x, \hat{y})$, where $\hat{y} = \mathbb{1}(f(x) \leq c)$, in calls to `monitor_correctness` and uses them to check whether the probabilistic assertion in that function is true.

To evaluate whether monitoring can detect shifts, we select two subsets of the "car" category: (i) bikes, including motor bikes, and (ii) passenger cars, excluding busses, trucks, etc., with 6 fine-grained labels each. Then, we consider a shift from the car category to the bike category—i.e., if we imagine that bikes were instead labeled as cars, would the recall of our program continue to be above the desired threshold. First, we check whether it proves correctness when the data distribution does *not* shift—i.e., using the test images labeled "passenger car". We run our verification algorithm on this property using the test set images labeled As expected, our verification algorithm correctly concludes that both the recall and the running time are within the expected bounds. Then, we check whether it proves correctness when the data distribution shifts—i.e., using the test images labeled "bike". In this case, our verification algorithm concludes that recall is incorrect, but running time is correct. Indeed, the average running time is now lower—intuitively, f_{fast} is incorrectly rejecting many "car" images, which reduces recall (undesired) as well as running time (desired).

As a side note, our framework can also be used to monitor quantitative properties. For instance, we can keep monitor how frequently the branch $f_{\text{fast}}(x) > c$ is taken—i.e., avoiding the need to evaluate $f(x)$. In Figure 6, `monitor_running_time` includes a probabilistic assertion

$$\textsf{passert}\ \ 1 - f_{\text{fast}}(x) > c' \ \ \{\text{true}\}^{|}_{\epsilon}$$

to perform this check. This assertion says that $1 - f_{\text{fast}}(x) > c'$ with probability at least $1 - \epsilon$—i.e., the faster branch in `is_person_fast` should be taken at least $1 - \epsilon$ fraction of the time according to $p(x, y^*)$. We might not know what is a reasonable value of ϵ—i.e., the rate at which f_{fast} predicts there is a person in the image. Thus, we leave it as a hole `??3`. Given training examples $\vec{z}$, our framework can be used to synthesize a value of ϵ to fill this hole.

F.4 Case Study 2: Precision Medicine

Warfarin dosing task. Next, we consider a task from precision medicine. In particular, we consider a random forest trained to predict dosing level for the Warfarin drug based on individual covariates such as genetic biomarkers [29]. Personalized dosing can improve patient outcomes, but significant errors can lead to adverse events if not quickly corrected. The ideal dosage is a real-valued label. The goal is to train a model to predict this dosage as a decision support tool for physicians. For simplicity,

we build on an approach that converts the problem into a classification problem by discretizing this value into labels $\mathcal{Y} = \{\text{high}, \text{medium}, \text{low}\}$ dose [30]. Then, the goal is to maximize accuracy while ensuring that very few patients for whom a high dose is predicted but should have been assigned a low dose, and vice versa.

Experimental setup. We split the dataset (5,528 examples) into training (1,658 examples), synthesis (2,764 examples), and test (1,106 examples) sets. Then, we use scikit-learn [31] to train a random forest $f : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ with 100 trees on the training set, where $f(x, y) \in \mathbb{R}$ is the probability assigned to label $y \in \mathcal{Y}$, and use f in conjunction with the program shown in Figure 9. This program includes two thresholds c_{low} and c_{high}, and only assigns a low dose to a patient with covariates x if $f(x) \geq 1 - c_{\text{low}}$, and similarly for a high dose—i.e., it only assigns the riskier outcomes when f is sufficiently confident in its prediction. Importantly, the specification on c_{low} refers not to the error rate on predictions for patients for whom $y = \text{low}$, but for whom $y = \text{high}$—i.e., we want to choose c_{low} to ensure precision specifically on patients for whom $y = \text{high}$, and conversely for c_{high}. We use our synthesis algorithm to synthesize values of c_{low} and c_{high} that satisfy these specifications.

Correctness. Figure 9 shows the results of our approach (black) compared to directly predicting the highest probability label according to the random forest f (red), always predicting "medium" (green), and the desired error rate (blue), as a function of the maximum error rate ϵ, the maximum failure probability δ, and the number of synthesis examples n. The top plots show the error rate, which is the maximum of the rate at which patients with $y = \text{low}$ are assigned a high dose, and the rate at which patients $y = \text{high}$ are assigned a low dose; this value should be below the blue line. The bottom plot shows the overall accuracy of the program—i.e., how often its predicted dose equals the ground truth dose. All values are estimated on the held-out test set. As before, ϵ has the largest impact on performance since it directly controls the error rate; however, once it hits $\epsilon = 0.06$, performance levels off since its accuracy now equals that of f, and the program never assigns a dose not predicted by f. Performance is flat as a function of δ. Finally, performance increases quickly as n goes from 500 to 1000, but plateaus thereafter, again once accuracy equals that of f.

Runtime monitoring. In the case of Warfarin dosing, the doctor administers an initial dose to the patient (possibly the predicted dose, depending on the doctor's judgement), and gradually adjusts it based on the patient response. Thus, we eventually observe the ground truth dose that should have been recommended, which we can use to monitor our program. This process is achieved by the `monitor_correctness` subroutine; here, `obtain_result` returns the true dose eventually observed for a patient with covariates x. We evaluate whether our runtime monitoring can detect shifts in the data distribution that lead to a reduction in performance. We consider a shift in terms of the ethnicity of the patients, which has recently been identified as an important challenge in algorithmic healthcare [32]. In particular, we consider a model trained using non-Hispanic White patients (2,969 examples), which we refer to as the "majority patients", and test it on Black, Hispanic, and Asian patients (2,559 examples), which we refer to as the "minority patients".

First, we check if it proves correctness when the data distribution does not shift—i.e., we train the random forest, and synthesize and verify the program on majority patients. As expected, it successfully verifies correctness. Next, we check if it proves correctness when there is a shift—i.e., we train the random forest and synthesize the program on majority patients, but verify the program on minority patients. As expected, it rejects the program as incorrect.

Finally, recall that whether verification is successful depends on how many test examples are provided; thus, we also evaluate how many test examples are needed in this setting. To make sure we have enough examples, we use all examples in this case. Then, we find that for 2,000 test examples, our verification algorithm successfully proves correctness, but for 500, 1,000, or 1,500 test examples, it fails. Intuitively, the number of test examples needed to verify correctness needs to be more than the number use to synthesize the parameters, or else the synthesized thresholds will be more precise (i.e., closer to their "optimal" value) and the verification algorithm will not have enough data to validate them. In this case, we use 1,000 synthesis examples, so about $2\times$ as many test examples are needed to verify correctness.

F.5 Case Study 3: Object Detection

Object detection. We assume given a DNN component f that given an image x, is designed to detect people in x. Our formulation of object detection in this section is slightly different than the

```
def detect_ppl(x):
    y_hat = f(x)
    return [d.box for d in y_hat if check_det(x, d)]

def check_det(x, d, d_true=None)
    return d.score > ??1 {IOU(d.box, d_true) >= 0.5} [|, 0.05]

def detect_ppl_fast(x, y_true=None):
    y_hat = f_fast(x)
    no_ppl_score = 1.0 - max([d.score for d in y_hat])
    if no_ppl_score > ??2 {len(detect_ppl(x)) != 0} [|, 0.05]:
        return []
    else:
        return detect_ppl(x)

def monitor_correctness(x):
    if np.random.uniform() <= 0.99:
        return
    y_hat = f_fast(x)
    no_ppl_score = 1.0 - max([d.score for d in y_hat])
    passert no_ppl_score > ??1 { len(detect_ppl(x)) != 0} [??3]
def monitor_speed(x):
    y_hat = f_fast(x)
    no_ppl_score = 1.0 - max([d.score for d in y_hat])
    passert no_ppl_score > ??1 {true} [??3]
```

Figure 10: A program used to detect people in a given image x. Specifications are shown in green; curly brackets is the specification and square brackets is the value of ϵ. The corresponding inequality with a hole in blue. Holes with the same number are filled with the same value.

(b) (c)

(e) (f)

(h) (i)

Figure 11: For the slow model alone (top) and slow+fast model (middle), we show recall (red), the desired lower bound on recall (blue), and precision (black) as a function of (a,d) ϵ, (b,e) δ, and (c,f) the object category y. For slow+fast (black), slow alone (red), and fast alone (green), we show running time as a function of (g) ϵ, (h) δ, and (i) the object category y.

previous setup. In particular, $d \in f(x)$ is a list of *detections*, which is a pair $d = (b, p)$ including a *bounding box* $b \in \mathbb{R}^4$ that encodes the center, width, and height of a rectangular region of x, and a value $p \in [0, 1]$ that is the predicted probability that b exists. In addition, the ground truth label y^* for an image x is a list of bounding boxes $b \in y^*$. In general, we cannot expect to get a perfect match between the predicted bounding boxes and the ground truth ones. Typically, two bounding boxes $b,'$ *match* if have significant overlap—in particular, their intersection-over-union satisfies $\text{IOU}(b, b') \geq \rho$ for some threshold $\rho \in [0, 1]$; we use a standard choice of $\rho = 0.5$. We denote that b and b' match in this sense by $b \cong b'$. Finally, b approximately matches a bounding box in y^* it $b \cong b'$ for some $b' \in y^*$, which we denote by $b \,\tilde{\in}\, y^*$.

Experimental setup. We use a pretrained state-of-the-art object detector called Faster R-CNN [4] available in PyTorch [11], tailored to the COCO dataset [12]. There are multiple variants of Faster R-CNN; we use the most accurate one, termed X101-FPN with $3\times$ learning rate schedule. For each predicted bounding box, this model additionally outputs a predicted object category (e.g., "person"), as well as the size of the bounding box ("small", "medium", and "large"). For most of our evaluation, we use "person" and "large". We specify alternative choices when we used them; in particular, we additionally consider 6 of the 91 object categories: "person" (10777 bounding boxes), "car" (1918 bounding boxes), "truck" (414 bounding boxes), "motorcycle" (367 bounding boxes), "bike" (314 bounding boxes), and "bus" (283 bounding boxes). We split the COCO validation set into 2000 synthesis images and 3000 test images.

Correctness. Our goal is to detect a majority of people. In particular, we consider synthesizing a threshold c and selecting all bounding boxes with probability above c—i.e.,

$$f(x, c) = \{b \mid (b, p) \in f(x) \wedge p \geq c\}.$$

This task is more challenging to specify than our examples so far since $f(x)$ is a structured output. In particular, we are not reasoning about whether $f(x)$ is correct with high probability with respect to $p(x, y^*)$, but whether bounding boxes $(b, p) \in f(x)$ are correct. Thus, we need a distribution $p(b \mid x)$ over bounding boxes b in an image x. Given such a distribution, our goal is to choose c so

$$\mathbb{P}_{p(x,y^*),p(b|x)}\left(b \,\tilde{\in}\, f(x,c) \mid b \,\tilde{\in}\, y^*\right) \geq 1-\epsilon, \tag{14}$$

where $p(x, y^*)$ is the data distribution. Intuitively, this property says that $f(x, c)$ contains at least a $1 - \epsilon$ fraction of ground truth bounding boxes. A reasonable choice for $p(b \mid x)$ is the uniform distribution over $f(x, 0)$—i.e., the set of all bounding boxes predicted by f. One issue is when a ground truth bounding box $b \in y^*$ is completely missing from $f(x, 0)$; in this case, b would not occur in $p(b \mid x)$, so (14) would not count it as an error even though it is missing from $f(x, c)$ for any c. To handle this case, we simply add $(b, 0)$ to $f(x)$ during synthesis for such bounding boxes b—i.e., f predicts b occurs with probability zero.

The program for achieving this goal is shown in the subroutine `detect_ppl` in Figure 10. We use our algorithm in conjunction with the synthesis examples to synthesize the parameter `??1` for this program, using the default values $\epsilon = \delta = 0.05$ and the object category "person". In Figure 11, we show the recall (red), desired lower bound on recall (blue), and precision (black) as a function of (a) ϵ, (b) δ, and (c) the object category y. The trends are largely similar to before—e.g., performance varies significantly with ϵ and the object category, but not very much with δ. For (c), we use $\epsilon = 0.1$ to facilitate comparison to our fast program described below.

Improving speed. We use a similar approach to improve speed as before—i.e., given a fast object detector f_{fast}, we want to use it to check the image, and only send it to the slow object detector f if necessary. A challenge compared to image classification is that the object detection model does not operate at the level of bounding boxes, which is the level at which we defined correctness, but at the level of images. Thus, we cannot decide whether we want to run the slow model independently for each detection $d \in f_{\text{fast}}(x)$; instead, we have to make such a decision for an image x as a whole. Intuitively, we check whether the fast model returns *any* detections in the given image x. To this end, we compute the maximum score p across all detections $(b, p) \in f_{\text{fast}}(x)$—i.e.,

$$\tilde{f}_{\text{fast}}(x) = \max_{(b,p)\in f_{\text{fast}}(x)} p.$$

Then, we want to guarantee that $y^* = \varnothing$ if this score is below some threshold that ensures that $y^* = \varnothing$; this property is equivalent to its contrapositive

$$(y^* \neq \varnothing) \Rightarrow (1 - \tilde{f}_{\text{fast}}(x) \leq c), \tag{15}$$

where the right-hand side of the implication is equivalent to $f_{\text{fast}}(x) \geq 1 - c$—i.e., the score is above the threshold $1 - c$. As before, we cannot ensure this property holds with probability one, so instead we use the high-probability variant

$$\mathbb{P}_{p(x,y^*)}(1 - \tilde{f}_{\text{fast}}(x) \leq c \mid y^* \neq \varnothing) \geq 1 - \epsilon.$$

This approach is shown in the `detect_ppl_fast` subroutine in Figure 10. We note that this approach does not provide guarantees as strong as the ones for image classification—in particular, there is a chance that the false negative images x of f_{fast} (i.e., x does not satisfy (15)) will contain larger numbers of ground truth bounding boxes compared to true positive images. Then, the recall at the level of bounding boxes may be less than $1 - 2\epsilon$. However, we find that it works well in practice; intuitively, f_{fast} is more likely to have false negative images that contain *fewer* ground truth bounding boxes.

For f_{fast}, we use a variant of Faster R-CNN termed R50-FPN with $3\times$ learning rate schedule, which is the fastest variant available. Then, we synthesize the parameters of `??1` and `??2` in Figure 10 using the synthesis examples. As before, all results are run on an Nvidia GeForce RTX 2080 Ti GPU. In Figure 11, we show the recall (red), desired lower bound on recall (blue), and precision (black) of our approach as a function of (d) ϵ, (e) δ, and (f) the object category y. Similarly, we show the running time (on the entire test set) of the combined program slow+fast (black), fast alone (green), and slow alone (red). As can be seen, our approach reduces running time by more than $2\times$ except in the case of "person" (28% reduction) and "truck" (45% reduction). The person speedup is relatively small because so many of the images in the COCO dataset contain people. Compared to the image classification setting, we obtain a smaller speedup since the gap between the fast and

slow models is not as large, and also because we can only avoid using the slow model for images that contain zero detections. Furthermore, comparing Figure 11 (c) and (f) (i.e., slow alone vs. slow+fast, respectively), for categories "car" and "truck", we suffer no loss in precision, though we suffer a small loss in precision for the others.

Finally, we note that in Figure 11 (e), for $\delta = 0.15$ and $\delta = 0.2$, the estimated recall falls slightly below the desired lower bound on recall. This result is most likely due to random chance, either because of randomness in the synthesis set or because these values are estimates based on a random test set. In particular, 0.15 is a fairly high failure probability (note that the results across δ are correlated, since we are using the same synthesis and test sets across all δ).

Runtime monitoring. We use runtime monitors to check that our program meets the desired bounds both in terms of error rate (the subroutine `monitor_correctness` in Figure 10) and running time (the subroutine `monitor_speed` in Figure 10). These approaches are the same as for image classification—the correctness monitor checks that the error rate (i.e., $f_{\text{fast}}(x)$ concludes there are no detections but $f(x) \neq \varnothing$) is below the desired rate ϵ, and the running time monitor checks that f is not called too often (i.e., $f_{\text{fast}}(x)$ concludes there are no detections sufficiently frequently).

To evaluate these monitors, we consider a shift from the default "large" bounding boxes we use to "small" and "medium". Intuitively, the smaller bounding boxes correspond to objects farther in the background, which are harder to detect but also tend to be less important (e.g., an autonomous car may not care as much about detecting far-away pedestrians). The trends are as before. First, we find that the monitors correctly prove correctness when there is no shift. Second, we find that the running time does not increase due to the shift, so the running time monitor continues to prove correctness. Finally, our correctness monitor rejects correctness for the shift to "small" bounding boxes; interestingly, it proves correctness for "medium" bounding boxes, which suggests that our synthesized program generalizes to this case.

G Related Work

Synthesizing machine learning programs. There has been work on synthesizing programs that include DNN components [33, 34, 35, 36, 37] and on synthesizing probabilistic programs [38, 39]; however, they do not provide guarantees on the synthesized program. There has been work on synthesizing control policies that satisfy provable guarantees [40, 41, 42, 43]; however, they focus on the setting where the learner can interact with the environment, and are not applicable to our supervised learning setting. Finally, there has been work on synthesizing programs with probabilistic constraints [44], but requires that the search space of programs has finite VC dimension.

Verified machine learning. There has been recent interest in verifying machine learning programs—e.g., verifying robustness [45, 46, 47, 48, 49], fairness [50, 51], and safety [46, 52]. More broadly, there has been work verifying systems such as approximate computing [53, 54, 55, 56] and probabilistic programming [57, 28]. The most closely related work is [58, 59, 60, 61], which verify semantic properties of machine learning models by sampling synthetic inputs from a user-specified space. In contrast, our focus is on synthesizing machine learning programs.

Statistical verification. There has been work leveraging statistical bounds to verify stochastic systems [13, 14, 15], probabilistic programs [57, 28], and machine learning programs [51]. Our verification algorithm in Section B relies on bounds similar to the ones used in these approaches [13]. To the best of our knowledge, we are the first to focus on synthesis; in contrast to verification, our approach relies on bounds from learning theory to provide correctness guarantees.

Conformal prediction. There has been work on *conformal prediction* [62, 63, 64, 65], including applications of these ideas to deep learning [66, 67, 68, 69], which aim to use statistical techniques to provide guarantees on the predictions of machine learning models. In particular, they provide confidence sets of outputs that contain the true label with high probability. Our techniques are inspired by these approaches, extending them to a general framework of synthesizing machine learning programs that satisfy provable guarantees.

H Proofs

H.1 Proof of Theorem B.2

It suffices to show that if $\mu < 1 - \epsilon$, then $\mathbb{P}_{p(\vec{z})}(\hat{\psi}(\vec{z})) < \delta$. First, note that since $z_1, ..., z_n$ are i.i.d. Bernoulli random variables with mean μ, then $1 - z_1, ..., 1 - z_n$ are i.i.d. Bernoulli random variables with mean $\nu = 1 - \mu$. Their sum $L(\vec{z})$ is a binomial random variable—i.e., $L(\vec{z}) \sim \text{Binomial}(n, \nu)$. Also, note that the condition $\mu < 1 - \epsilon$ is equivalent to $\nu > \epsilon$. Thus, we have

$$\begin{aligned}
\mathbb{P}_{p(\vec{z})}(\hat{\psi}(\vec{z})) &= \sum_{i=0}^{k} \binom{n}{i} \nu^i (1-\nu)^{n-i} \\
&< \sum_{i=0}^{k} \binom{n}{i} \epsilon^i (1-\epsilon)^{n-i} \\
&\leq \delta,
\end{aligned}$$

where the first inequality follows by standard properties of the CDF of the Binomial distribution. The claim follows. □

H.2 Proof of Theorem C.2

First, define

$$t_\epsilon^0 = \inf_{t \in \mathbb{R}} \mathcal{T}_\epsilon.$$

Intuitively, $t_\epsilon^0 \in \mathbb{R}$ is the threshold that determines whether t is ϵ-approximately correct. In particular, it is clear that $t \in \mathcal{T}_\epsilon$ for all $t > t_\epsilon^0$ and $t \notin \mathcal{T}_\epsilon$ for all $t < t_\epsilon^0$; in general, $t_\epsilon^0 \in \mathcal{T}_\epsilon$ may or may not hold. Thus, it suffices to show

$$\mathbb{P}_{p(\vec{z})}(\hat{t}(\vec{z}) \leq t_\epsilon^0) < \delta.$$

To this end, note that the constraint $L(t; \vec{z}) \leq k$ in (7) implies

$$\sum_{z \in \vec{z}} \mathbb{1}(z > \hat{t}(\vec{z}) - \gamma(\vec{z})) \leq k.$$

Thus, on event $\hat{t}(\vec{z}) \leq t_\epsilon^0$, we have $\hat{t}(\vec{z}) - \gamma(\vec{z}) \leq t_\epsilon^0 - \gamma(\vec{z})$, so

$$k \leq \sum_{z \in \vec{z}} \mathbb{1}(z > \hat{t}(\vec{z}) - \gamma(\vec{z})) \leq \sum_{z \in \vec{z}} \mathbb{1}(z > t_\epsilon^0 - \gamma(\vec{z})).$$

As a consequence, we have

$$\mathbb{P}_{p(\vec{z})}(\hat{t}(\vec{z}) \leq t_\epsilon^0) \leq \mathbb{P}_{p(\vec{z})}\left(\sum_{z \in \vec{z}} \mathbb{1}(z > t_\epsilon^0 - \gamma(\vec{z})) \geq k\right).$$

Next, since $t_\epsilon^0 - \gamma(\vec{z}) < t_\epsilon^0$, we have $t_\epsilon^0 - \gamma(\vec{z}) \notin \mathcal{T}_\epsilon$—i.e.,

$$\epsilon < \mathbb{P}_{p(z)}(z > t_\epsilon^0 - \gamma(\vec{z})) = \mathbb{E}_{p(z)}(\mathbb{1}(z > t_\epsilon^0 - \gamma(\vec{z}))).$$

In other words, the random variables $\mathbb{1}(z > t_\epsilon^0 - \gamma(\vec{z}))$ for $z \in \vec{z}$ are i.i.d. Bernoulli random variables with mean $\nu > \epsilon$. Thus, we have

$$\begin{aligned}
\mathbb{P}_{p(\vec{z})}\left(\sum_{z \in \vec{z}} \mathbb{1}(z > t_\epsilon^0 - \gamma(\vec{z})) \geq k\right) &= \sum_{i=0}^{k} \mathbb{P}_{p(\vec{z})}\left(\sum_{z \in \vec{z}} \mathbb{1}(z > t_\epsilon^0 + \gamma(\vec{z})) = i\right) \\
&= \sum_{i=0}^{k} \text{Binomial}(i; n, \nu) \\
&< \sum_{i=0}^{k} \text{Binomial}(i; n, \epsilon) \\
&= \sum_{i=0}^{k} \binom{n}{i} \epsilon^i (1-\epsilon)^{n-i} \\
&\leq \delta,
\end{aligned}$$

where the first inequality follows by standard properties of the CDF of the Binomial distribution. The claim follows. $\square$

H.3 Proof of Theorem C.4

First, we have the following classical inequality [16]:

Theorem H.1. *(Hoeffding's inequality) We have*

$$\mathbb{P}_{p(\vec{z})}\Big(\mu - \hat{\mu}(\vec{z}) \geq t\Big) \leq e^{-2nt^2}.$$

Now, letting $t = \sqrt{\frac{\log(1/\delta)}{2n}}$, we have

$$\mathbb{P}_{p(\vec{z})}\Big(\mu \geq \hat{\nu}(\vec{z})\Big) \leq \mathbb{P}_{p(\vec{z})}\Big(\mu - \hat{\mu}(\vec{z}) \geq t\Big) \leq e^{-2nt^2} \leq \delta,$$

where the second-to-last inequality follows from Theorem H.1. The claim follows. $\square$