Training Set Debugging Using Trusted Items

Xuezhou Zhang and **Xiaojin Zhu** and **Stephen Wright**
{zhangxz1123, jerryzhu, swright}@cs.wisc.edu
Department of Computer Sciences, University of Wisconsin-Madison

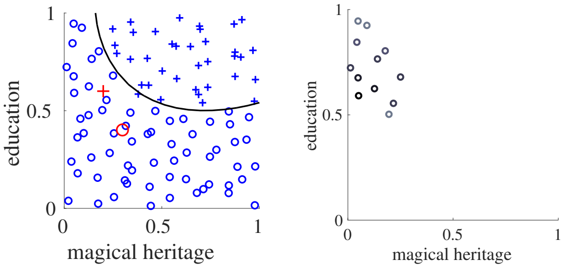
Abstract

Training set bugs are flaws in the data that adversely affect machine learning. The training set is usually too large for manual inspection, but one may have the resources to verify a few trusted items. The set of trusted items may not by itself be adequate for learning, so we propose an algorithm that uses these items to identify bugs in the training set and thus improves learning. Specifically, our approach seeks the smallest set of changes to the training set labels such that the model learned from this corrected training set predicts labels of the trusted items correctly. We flag the items whose labels are changed as potential bugs, whose labels can be checked for veracity by human experts. To find the bugs in this way is a challenging combinatorial bilevel optimization problem, but it can be relaxed into a continuous optimization problem. Experiments on toy and real data demonstrate that our approach can identify training set bugs effectively and suggest appropriate changes to the labels. Our algorithm is a step toward trustworthy machine learning.

1 Introduction

A good training set is essential for machine learning. The presence of bugs – mislabeled training items[1] – has adverse effects on learning (Brodley and Friedl 1999; Guruswami and Raghavendra 2009; Caramanis and Mannor 2008). Bugs can appear as outliers that are relatively easy to detect, or as systematic biases. Systematic bugs are much harder to detect because the data appear self-consistent.

We propose a novel algorithm DUTI (Debugging Using Trusted Items) which can detect both outlier and systematic training set bugs. In addition, it can propose fixes, namely the corrected label for the bugs. To do so, DUTI utilizes the knowledge of the machine learning algorithm and a small set of additional "trusted items". At its core, DUTI finds the smallest changes to the training set such that, when trained on the changed training set, the learned model agrees with the trusted items. The changes are then shown to a domain expert as suggested bug fixes. We will show how DUTI can be relaxed and solved efficiently using continuous optimization, and we demonstrate its debugging efficacy on multiple data sets. All code and data are published at `http://pages.cs.wisc.edu/~jerryzhu/DUTI`.



(a) training & trusted data (b) DUTI flagged bugs

Figure 1: Harry Potter Toy Example

To build intuition, consider a toy example *a la* Harry Potter in Figure 1a. Each blue point is a Hogwarts student whose magical heritage ranges from 0 (muggle-born) to 1 (pure-blood), and education ranges from 0 (failed school) to 1 (Hermione level). These blue points form a classification training set, where the class label is '+' (hired by Ministry of Magic after graduation) or 'o' (not hired). This training set shows historical bias against muggle-borns with high education. Kernel logistic regression trained on the data reflects this bias (black decision boundary). But suppose we know two more students and how they *should* be classified (the red points) – the assumption being that a fair decision is based simply on education ≥ 0.5. These two points are the trusted items. Simply training on the trusted items alone will be unsatisfactory – the boundary will not be flat at education$= 0.5$. Instead, DUTI can utilize the trusted items to flag potential bugs in the training set (Fig. 1b). Darker color represents higher confidence that a training item is a bug.

2 Training Set Debugging Formulation

DUTI needs three inputs:

1. The training set in the form of feature, label pairs $(X, Y) = \{(x_i, y_i)\}_{1:n}$. The labels Y can be continuous for regression or discrete for classification, and are potentially buggy.
2. Trusted items $(\tilde{X}, \tilde{Y}) = \{(\tilde{x}_i, \tilde{y}_i, c_i)\}_{1:m}$. These are items verified by domain experts typically at considerable ex-

[1] We focus on label bugs for simplicity, though our framework can be extended to feature bugs in a straightforward manner.

pense. The domain expert can optionally specify a confidence $c \geq 0$ for each trusted item. We assume $m \ll n$ so that the amount of trusted data is insufficient to train a good model. We do not assume the trusted items are iid.

3. The learning algorithm $\mathcal{A}$. In this work, we focus on regularized empirical risk minimizers

$$\mathcal{A}(X, Y) = \operatorname*{argmin}_{\theta \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \ell(x_i, y_i, \theta) + \lambda \Omega(\theta) \qquad (1)$$

with strongly convex and twice differentiable objective function, such that argmin returns a unique solution, and the Hessian matrix is always nonsingular.

Conceptually, DUTI solves the following optimization problem:

$$\operatorname*{argmin}_{Y'} \; Distance(Y', Y) \qquad (2)$$

$$\text{s.t. } Predictor = \mathcal{A}(X, Y') \qquad (3)$$

$$Predictor(\tilde{X}) = \tilde{Y} \wedge Predictor(X) = Y' \qquad (4)$$

That is, we find an alternative labeling Y' for the training data, as close as possible to the original labels Y, such that the model trained with the original feature vectors X and the alternative labels Y' correctly predicts the labels $\tilde{Y}$ of the trusted items $\tilde{X}$, and the alternative labeling is self-consistent. We call the training items for which $y'_i \neq y_i$ the *flagged bugs*; we give them to the domain expert for further inspection to see if they are true bugs.

Our next step is to relax it into a continuous single level optimization problem that can be solved efficiently.

2.1 Debugging for Regression

In regression, Y' and Y are both vectors in $\mathbb{R}^n$. We define $\delta \equiv Y' - Y$, and choose $\|\delta\|_1$ as the distance metric to encourage sparsity of δ. We will denote the predictor $\mathcal{A}(X, Y')$ as θ. Instead of requiring equalities as in (4), we relax them by the learner's surrogate loss function, and place them in the objective in the Lagrangian form:

$$\begin{aligned} \min_{\delta \in \mathbb{R}^n, \theta} \quad & \frac{1}{m} \sum_{i=1}^{m} c_i \ell(\tilde{x}_i, \tilde{y}_i, \theta) \\ & + \frac{1}{n} \sum_{i=1}^{n} \ell(x_i, y_i + \delta_i, \theta) + \gamma \frac{\|\delta\|_1}{n} \qquad (5) \\ \text{s.t.} \quad & \theta = \operatorname*{argmin}_{\beta \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \ell(x_i, y_i + \delta_i, \beta) + \lambda \Omega(\beta). \end{aligned}$$

where c_i's are the confidence levels assigned to each of the trusted items. This is a bilevel optimization problem. We now convert the lower level problem to a nonlinear constraint. Notice that since the lower problem is unconstrained and strongly convex, it can be replaced equivalently with its Karush–Kuhn–Tucker (KKT) condition:

$$g(\delta, \theta) \equiv \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(x_i, y_i + \delta_i, \theta) + \lambda \nabla_\theta \Omega(\theta) = 0. \quad (6)$$

Now, since g is continuously differentiable and $\frac{\partial g}{\partial \theta}$ is invertible, the solution to $g(\delta, \theta) = 0$ defines an implicit function $\theta(\delta)$. We can now replace θ with $\theta(\delta)$ in (5), and call the result $\mathcal{O}_\gamma(\delta)$. Using the fact $\frac{d\ell}{d\delta} = \frac{\partial \ell}{\partial \delta} + \frac{\partial \ell}{\partial \theta} \frac{\partial \theta}{\partial \delta}$, we can compute the gradient of $\mathcal{O}_\gamma(\delta)$ as

$$\begin{aligned} \nabla_\delta \mathcal{O}_\gamma \;=\; & \frac{1}{m} \sum_{i=1}^{m} c_i J^\top \nabla_\theta \ell(\tilde{x}_i, \tilde{y}_i, \theta)|_{\theta(\delta)} \qquad (7) \\ & + \frac{1}{n} \sum_{i=1}^{n} \frac{\partial \ell(x_i, y_i + \delta_i, \theta(\delta))}{\partial \delta_i} \mathbf{e}_i \\ & + \frac{1}{n} \sum_{i=1}^{n} J^\top \nabla_\theta \ell(x_i, y_i + \delta_i, \theta(\delta)) + \frac{\gamma}{n} \text{sgn}(\delta) \end{aligned}$$

where $\mathbf{e}_i$ is the ith canonical vector and J is defined by the implicit function theorem:

$$J \equiv \frac{\partial \theta}{\partial \delta} = - \begin{bmatrix} \frac{\partial g_1}{\partial \theta_1} & \cdots & \frac{\partial g_1}{\partial \theta_p} \\ \vdots & & \vdots \\ \frac{\partial g_p}{\partial \theta_1} & \cdots & \frac{\partial g_p}{\partial \theta_p} \end{bmatrix}^{-1} \begin{bmatrix} \frac{\partial g_1}{\partial \delta_1} & \cdots & \frac{\partial g_1}{\partial \delta_n} \\ \vdots & & \vdots \\ \frac{\partial g_p}{\partial \delta_1} & \cdots & \frac{\partial g_p}{\partial \delta_n} \end{bmatrix}.$$

With $\nabla_\delta \mathcal{O}_\gamma$ we then solve (5) with a gradient method.

2.2 Debugging for Classification

Let there be k classes. To avoid a combinatorial problem, we relax the optimization variable to the k-probability simplex Δ_k. Concretely, we first augment the learner $\mathcal{A}$ so that it takes *weighted* training items:

$$\theta = \operatorname*{argmin}_{\beta} \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} w_{ij} \ell(x_i, j, \beta) + \lambda \Omega(\beta) \qquad (8)$$

where $w_i \in \Delta_k$, that is, $w_{ij} \geq 0$ and $\sum_{j=1}^{k} w_{ij} = 1, \forall i, j$. The original learner (1) can be recovered with $w_i = \mathbf{e}_{y_i}, \forall i$.

We then represent the ith proposed new class label y'_i by $\delta_i \in \Delta_k$. Note that δ_i here represents the *new* set of labels, not a difference in labels. One way to measure the distance to the old label y_i is $1 - \delta_{i,y_i}$, namely the probability mass siphoned away from the old label. We thus obtain a bilevel optimization problem with continuous variables similar to (5):

$$\begin{aligned} \min_{\delta \in \Delta_k^n, \theta} \quad & \frac{1}{m} \sum_{i=1}^{m} c_i \ell(\tilde{x}_i, \tilde{y}_i, \theta) \qquad (9) \\ & + \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} \delta_{ij} \ell(x_i, j, \theta) + \frac{\gamma}{n} \sum_{i=1}^{n} (1 - \delta_{i,y_i}) \\ \text{s.t.} \quad & \theta = \operatorname*{argmin}_{\beta} \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{k} \delta_{ij} \ell(x_i, j, \beta) + \lambda \Omega(\beta). \end{aligned}$$

Finally, we go through similar steps as in regression: replacing the lower problem with its KKT condition; defining an implicit function $\theta(\delta)$; obtaining the objective $\mathcal{O}_\gamma(\delta)$ of (9); and computing its gradient $\nabla_\delta \mathcal{O}_\gamma$ using implicit function theorem. We then optimize $\mathcal{O}_\gamma(\delta)$ to solve for δ. The details can be found at the aforementioned website.

```
   Input  : Training set $(X, Y)$, trusted items $(\tilde{X}, \tilde{Y}, c)$,
            learner $\mathcal{A}$, examination budget $b \leq n$;
1  Initialize $t = 0$, $\delta^{(0)} = 0$ in regression or $Y$ in
   classification, $n_{flag} = 0$;
2  Initialize $\gamma^{(0)} = \max_i |\nabla_\delta \mathcal{O}_{\gamma=0}(\delta^{(0)})_i|$ in regression, or
   $\gamma^{(0)} = \max_i \nabla_\delta \mathcal{O}_{\gamma=0}(\delta^{(0)})_{i,y_i}$ in classification;
3  while $n_{flag} \leq b$ do
4      $t = t + 1$;
5      $\gamma^{(t)} = \gamma^{(t-1)}/2$;
6      $\delta^{(t)} = \text{argmin}\, \mathcal{O}_{\gamma^{(t)}}(\delta)$, initialized at $\delta^{(t-1)}$;
7      $F^{(t)} = \{i \mid \delta_i^{(t)} \neq 0\}$ in regression, or
8      $F^{(t)} = \{i \mid \text{argmax}_j(\delta_{ij}^{(t)}) \neq y_i\}$ in classification;
9      $n_{flag} = | \cup_{s=1}^{t} F^{(s)}|$;
10 end
   Output: $(\gamma_1, \delta^{(1)}), \cdots, (\gamma_t, \delta^{(t)})$;
```

Algorithm 1: DUTI

2.3 The DUTI Algorithm

We now present a unified DUTI algorithm for debugging both regression and classification, see Algorithm 1. As part of the input, the domain expert can specify an examination budget b, the preferred total number of flagged bugs that they are willing to check. Recall that the debugging formulations (5) and (9) have a sparsity parameter γ. DUTI automatically chooses a sequence of γ's, runs the corresponding debugging formulation, and flags potential bugs $F^{(t)}$ by line 7 or 8, respectively. $F^{(1)}, F^{(2)}, \ldots$ are not necessarily nested subsets. DUTI accumulates all previously flagged bugs by $\cup_{s=1}^{t} F^{(s)}$, and only stops if its size exceeds the examination budget b.

DUTI outputs the sequence of sparsity parameters γ and solutions δ, from which the F's can be recovered. This is helpful for the domain expert to investigate the flagged bugs. Specifically, DUTI's output induces a *ranking* over all flagged bugs. Bugs are ranked by the earliest time they appear in the sequence $F_1, F_2, \ldots$. When two bugs first appear at the same time, the tie is broken by comparing the deviation of δ from the initial value. Larger deviation = earlier order. Furthermore, the value δ suggests the fix, namely what regression output value or class label DUTI thinks the flagged bug should have had.

DUTI chooses the γ sequence as follows. It starts with the largest γ that returns a nontrivial solution δ, namely $\delta \neq \mathbf{0}$ in regression and $\delta \neq Y$ in classification. One can show that these are sufficient conditions for a nontrivial δ solution: In the regression setting, $\gamma^{(0)} = \max_i |\left[\nabla_\delta \mathcal{O}_{\gamma=0}(\mathbf{0})\right]_i|$, where $\nabla_\delta \mathcal{O}_{\gamma=0}$ is the gradient of the debugging objective with $\gamma = 0$, taken at $\delta = \mathbf{0}$. In the multiclass classification setting, $\gamma^{(0)} = \max_i \left[\nabla_\delta \mathcal{O}_{\gamma=0}(Y)\right]_{i,y_i}$, where $\nabla_\delta \mathcal{O}_{\gamma=0}$ is similarly the gradient of the debugging objective with $\gamma = 0$, taken at the initial value $\delta = Y$. The i, y_i-th entry of the gradient is the probability mass assigned to the original label y_i. A positive entry will result in optimization taking a gradient step, thus returning a nontrivial solution.

DUTI utilizes continuation method to speed up optimization. Specifically, each iteration t is initialized using the previous solution $\delta^{(t-1)}$. Moreover, DUTI uses a linear approximation $\theta^{(t-1)} + \frac{\partial \theta}{\partial \delta}(\delta^{(t)} - \delta^{(t-1)})|_{\delta^{(t-1)}}$ to initialize the computation of $\theta(\delta^{(t)})$.

3 Experiments

To the best of our knowledge, no machine learning debugging repository is publicly available. That is, no data set has provenance like "item i had the wrong label y; it should have label z" for research in debugging. Curating such a repository will be useful for trustworthy machine learning research. Software debugging repository in the programming language community, such as the Siemens test suite (Hutchins et al. 1994) and BugBench (Lu et al. 2005), are good examples to follow. In this paper, we had to simulate – as plausible as possible – true bugs on the real data sets.

3.1 Learner $\mathcal{A}$ Instantiation in DUTI

DUTI is a *family* of debuggers depending on the learner $\mathcal{A}$ (1). For illustration, in our experiments we let $\mathcal{A}$ be kernel ridge regression for regression and multiclass kernel logistic regression for classification, both with RBF kernels. We recall the relevant learner loss ℓ and regularizer Ω below; the derivation of the corresponding debugging objective $\mathcal{O}_\gamma$ and gradient $\nabla_\delta \mathcal{O}_\gamma$ is straightforward but tedious, and is left for a longer version. Extension to other learners can be done similarly.

Kernel Ridge Regression Let $(X, Y) = \{(x_i, y_i)\}_{1:n}$ be the training data, and $K = [k(x_i, x_j)]_{n \times n}$ be the kernel matrix. Denote by K_i the i-th column of K, and let $\alpha \in \mathbb{R}^n$ be the learning parameter. Kernel ridge regression can be written in the form of regularized ERM (1), where $\ell(x_i, y_i, \alpha) = (y_i - K_i \alpha)^2$ and $\Omega(\alpha) = \frac{1}{2}\alpha^\top K \alpha$.

Multiclass Kernel Logistic Regression with Ridge Regularization Let $\alpha \in \mathbb{R}_{n \times k}$ be the learning parameter, and denote by α_j the j-th column of α. Kernel logistic regression can be written in the form of weighted learner (8):

$$\begin{aligned}\theta = \underset{\alpha}{\text{argmin}} & -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k}\delta_{ij}K_i^\top \alpha_j \\ & + \frac{1}{n}\sum_{i=1}^{n}\log\left(\sum_{j=1}^{k}\exp(K_i^\top \alpha_j)\right) + \frac{\lambda}{2}\sum_{j=1}^{k}\alpha_j^\top K \alpha_j\end{aligned}$$

where $\ell(x_i, j, \alpha) = -K_i^\top \alpha_j + \log\left(\sum_{j=1}^{k}\exp(K_i^\top \alpha_j)\right)$ and $\Omega(\alpha) = \frac{1}{2}\sum_{j=1}^{k}\alpha_j^\top K \alpha_j$.

In all our experiments, the learner's hyperparameters are set by 10-fold cross validation on the original training data, and confidence levels on all trusted items c are set to 100.

3.2 Baseline Debugging Methods for Comparison

We compare DUTI with three baselines: influence function (INF) (Koh and Liang 2017), Nearest Neighbor (NN), and Label Noise Detection (LND) (Bhadra and Hein 2015).

Influence Function (Koh and Liang 2017) describes how perturbing a training point changes the prediction on a tested point. The influence of training labels on the trusted items can be written as

$$I = \frac{1}{m}\sum_{i=1}^{m} J^\top \nabla_\theta \ell(\tilde{x}_i, \tilde{y}_i, \theta)|_{\theta(\delta)} \tag{10}$$

This is in fact the first term of our objective gradient $\nabla_\delta \mathcal{O}_\gamma$ (assuming $c_i = 1, \forall i$). Therefore, one can view influence function as a first-order approximation to a simplified version of DUTI. Intuitively, a positive influence implies that decreasing the y value will reduce the loss of trusted items, while a negative influence implies that one should increase the corresponding y value to achieve the same result. In regression, INF prioritizes training points with larger absolute values of influence. In classification, each possible label of a training point will have an influence. INF will flag training points with positive influence on their original label, and prioritize ones with larger influence value. In suggesting fixes for each flagged training point, INF will suggest the label with the largest negative influence value.

Since INF is a first-order approximation to DUTI, we expect INF to be inferior for debugging due to nonlinearity between labels and predictions. This is confirmed by our experiments.

Nearest Neighbor NN is a simple heuristic: In regression, NN flags training points based on the Euclidean distance (after normalizing each feature dimension separately) to the closest trusted item. In classification, NN flags each training point whose label differs from the label of the closest trusted point, and prioritizes them by this distance. When asked for a suggested label fix, it recommends the label of closest trusted item.

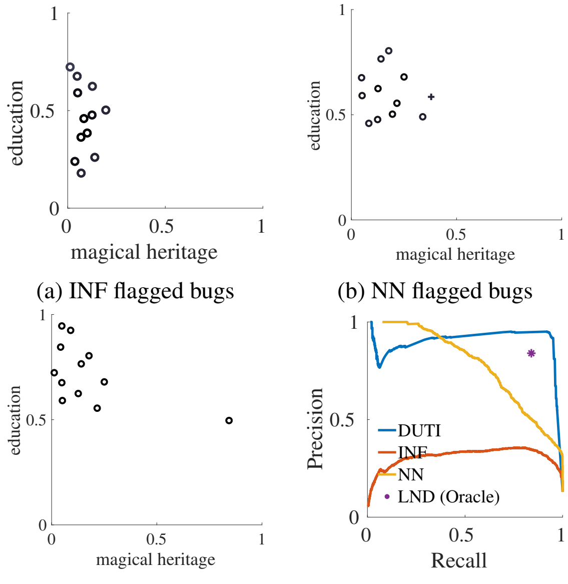
Label Noise Detection (Oracle) (Bhadra and Hein 2015) uses a Gaussian kernel density estimator as the consistency metric to define a combinatorial optimization problem:

$$\underset{\eta \in \{1,-1\}^n}{\text{argmax}} \; \eta^\top Q \eta. \tag{11}$$

where $Q_{ij} = y_i y_j K_h(x_i, x_j)$, $\eta_i = 1$ represents a correct original label, and $\eta_i = -1$ a bug. LND is only for binary labels. The algorithm finds the best relabeling η that maximize this mutual consistency metric. In the case that there are expert verified data, e.g. our trusted items, LND can incorporate them as constraints to reduce the search space. However, LND requires the user to provide the number of positive mislabeled items and the number of negative mislabeled items. In practice such information is usually unavailable, but in our experiments we provide LND with the ground truth numbers as a best-case scenario. For this reason we label this baseline LND (Oracle). Also note that LND applies only to binary classification problems, so we omit LND from baselines for our regression and multiclass classification experiments.

3.3 Toy Data: Harry Potter (Cont.)

We can now fully define the Harry Potter toy example. The true boundary is the line education$= 0.5$. A training point whose label disagrees with the true boundary is considered a bug. Thus the true bugs are all the "not hired" training points on the upper left. Figure 1b shows that DUTI is able to flag true bugs with a budget $b = 12$. Figure 2(a–c) show the top 12 training items flagged by INF, NN and LND, respectively. INF and NN flagged some training points below the true boundary, which they should not. LND, thanks to the oracle information, is able to flag mostly true bugs but also produced one false positive. By varying how many training points we ask each method (other than LND) to flag, we can produce a precision-recall (PR) curve with respect to true bugs for that method. LND is represented by a single point on the PR curve, because it flags only a fixed number of points due to the oracle information. Figure 2d shows the average PR curves from 100 random runs, in which the training data are randomly drawn with trusted items fixed. Overall, DUTI dominates the baseline methods.



(a) INF flagged bugs

(b) NN flagged bugs

(c) LND flagged bugs

(d) Average PR Curves

Figure 2: Harry Potter Toy Example, Continued

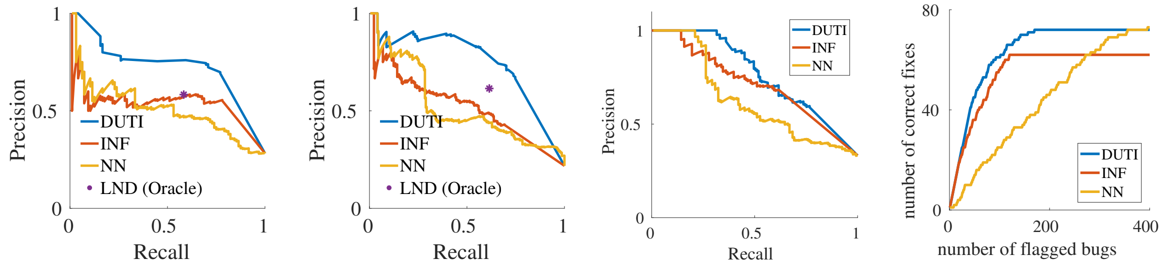
3.4 Real Data: German Loan Application

We study the UCI German Loan data set, which has been used in recent work on algorithmic fairness (Zemel et al. 2013; Feldman et al. 2015). Prior work suggested a systematic bias of declining applicants younger than 25. We now detail the way we simulate true bugs on German Loan. Throughout the learning and debugging process we remove the age attribute.

Step 1. The original data set consists of 1000 applicants with 190 young (age≤ 25) and 810 old (age> 25). We partition the dataset into three subsets A,B,C. A contains 20 random young and 20 random old applicants. B contains the remaining 170 young and another 170 random old applicants. C contains the remaining 620 old applicants.

Step 2. We use group C to train a loan decision classifier f^*, and use it as the ground truth concept.

Step 3. We relabel the applicants in group A using f^*, and



(a) German Loan PR Curves (b) Adult Income PR Curves (c) Digits PR Curves (d) Digits suggested fixes

Figure 3: Real Data Experiments

treat the relabeled group A as trusted items.

Step 4. Group B with the original UCI label is treated as the buggy training set. Whenever Y disagrees with $f^*(X)$, that training point is considered a bug. This methodology results in 96 bugs.

Figure 3a compares the PR curves of the four debugging methods. DUTI clearly dominates other methods.

3.5 Real Data: Adult Income

Another dataset often used in algorithmic fairness is UCI Adult Income (Kohavi 1996; Kamishima, Akaho, and Sakuma 2011). The task is to classify whether an individual receives an annual income of $\geq$ 50k. Prior work suggests that the data set contains systematic biases against females. In this experiment, we simulate true bugs based on such biases. Throughout the learning and debugging process we remove the gender attribute and 'wife & husband' values in the family relationship attribute.

Similar to the German Loan Dataset, we randomly subsample 3 disjoint subsets A,B,C. A contains 20 random male and 20 random female applicants. B contains 500 random males and 500 random female applicants. C contains 2000 random male applicants. We apply exactly the same steps 2,3,4 as in the German Loan data set. This process results in 218 bugs.

Figure 3b compares the PR curves of the four debugging methods and again DUTI dominates other methods.

3.6 Real Data: Handwritten Digits

In this section, we evaluate the debugging methods on a 10-class handwritten digit recognition problem (Mathworks 2017). The digit images are generated by applying random affine transformations to digit images created using different fonts. In multiclass classification, flagging the true bugs and suggesting the correct fixes are no longer equivalent, so we present evaluations on both criteria. Unlike the previous two experiments, we now simulate a contamination mechanism to generate buggy labels, while the original labels are considered clean.

Step 1. We randomly set aside 5000 data points, 500 per digit, to train a deepnet consisting of an autoencoder layer followed by a softmax layer, achieving a cross validation accuracy of 98%. Denote the trained neural net by f^*.

Step 2. Among the rest, we randomly sample 400 data points, 40 per digit, to be the training set X. We then blur the images heavily with a Gaussian filter, and classify on the blurred images with the trained deepnet. The intention is to simulate the generation of buggy labels by a human annotator with poor eyesight. These classifications will be used as the buggy labels of the training points, that is, $Y = f^*(\text{blur}(X))$. Meanwhile, the original labels for X are retained as the correct labels. A training image has a buggy label if Y and the original label disagree. This process gives rise to a total of 133 bugs. Note that X is always the clear images; the blurred images are only used to generate Y.

Step 3. Finally, among the rest, we randomly sample 160 data points, 16 per digit, to form the trusted items $\tilde{X}$, and use their original labels as trusted labels $\tilde{Y}$.

Figure 3c shows the PR curves that indicate whether each method flags the true bugs, regardless of whether their proposed fix is correct or not. For this easier task, DUTI and INF both excel, but DUTI has a slight advantage overall. Figure 3d plots the number of flagged bugs vs. the number of correct fixes. This is a harder task. Nonetheless, DUTI still dominates INF and NN, especially in the early stage. Specifically, DUTI successfully recovers more than half of the buggy labels (there were 133) within less than 200 attempts.

Table 1 visualizes selected buggy training points. The first row shows the original images, and the second row shows the wrong label they received from the blurred deepnet. The next three rows show the actions of DUTI, INF, and NN, respectively: A numerical entry indicates that the debugging method flagged this training image as a bug, and suggested that number as the fixed label. The entry "–" indicates a false negative: the debugging method missed the bug.

3.7 Regression Toy Examples

Finally, we demonstrate debugging for regression. We generate the clean data by uniformly sampling $n = 100$ points $x \sim U[0, 2]$, and generating their label as

$$y = \sin(2\pi x) + \epsilon_1, \text{ where } \epsilon_1 \sim N(0, 0.1). \qquad (12)$$

We then manually convert 24 of these points into systematic bugs by flipping the second peak from positive to negative; see Figure 4a. The dashed curve traces out $\sin(2\pi x)$ but is not used in the experiment otherwise. For trusted items we randomly pick $m = 3$ new points near one side of the flipped peak, and generate their $\tilde{y}$ values using $\tilde{y} = \sin(2\pi\tilde{x})$. We intentionally do not fully cover the bug region with the trusted items.

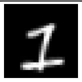
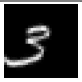
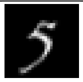
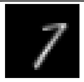
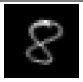
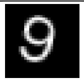
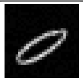
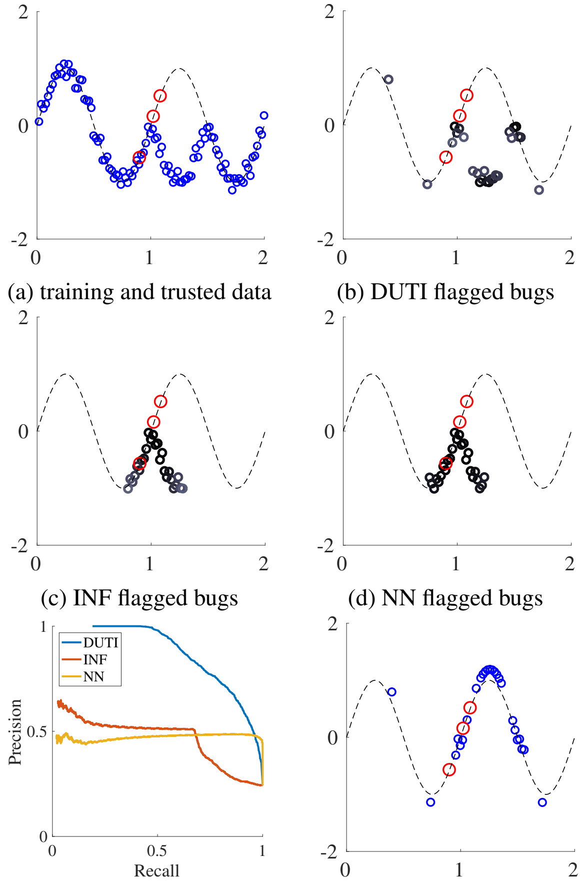
X										
$Y = f^*(\text{blur}(X))$	3	7	1	8	3	4	1	3	8	8
DUTI	1	2	3	7	5	6	7	8	–	0
INF	–	–	5	–	5	–	7	0	–	0
NN	1	–	5	7	8	6	4	0	–	5

Table 1: Selected images with buggy training labels, and the debugging actions on them



(a) training and trusted data

(b) DUTI flagged bugs

(c) INF flagged bugs

(d) NN flagged bugs

(e) Average PR Curves

(f) DUTI suggested fixes

Figure 4: Regression toy data with systematic bugs

Figure 4b,c,d plot the top 25 flagged bugs by DUTI, INF, and NN, respectively. Interestingly, DUTI successfully flags bugs throughout the flipped peak. In contrast, INF and NN are only able to flag points around the trusted items, and miss half of the bugs. We repeat the experiment for 100 times, each time with the training data randomly drawn according to (12). Figure 4e shows the average PR curves, where DUTI clearly dominates INF and NN. Figure 4f shows the suggested fixes on buggy points (i.e. $y_i + \delta_i$) generated by DUTI. Impressively, the fixes closely trace the sine curve, even though the learner is an RBF kernel regressor and knows nothing about sine function. This seems to be a smoothing effect of the self-consistent term, which is $Predictor(X) = Y'$ in the conceptual formulation (4) or $\frac{1}{n}\sum_{i=1}^{n} \ell(x_i, y_i + \delta_i, \theta)$ in (5).

4 Related Work

Debugging has been studied extensively in the programming language community (Shapiro 1983; Ball and Rajamani 2002). Only recently does the machine learning community start to investigate how to debug machine learning systems, e.g. (Cadamuro, Gilad-Bachrach, and Zhu 2016; Bhadra and Hein 2015). Our intuition is similar to that of (Cadamuro, Gilad-Bachrach, and Zhu 2016), who provide closed-form solutions to debugging Ordinary Least Squares. Our work allows one to debug any ERM learners satisfying mild conditions.

More generally, our work falls into the field of interpretable and trustworthy machine learning, where the central question is why a trained model makes certain predictions. One line of work explains the prediction by highlighting the responsible part of the test item. For example, (Selvaraju et al. 2016; Smilkov et al. 2017; Sundararajan, Taly, and Yan 2016) identify regions in a test image that have the greatest influence on the classification of that image. In other learning settings where features are less explanatory, (Koh and Liang 2017) propose to look instead at the most influential training items that affects the prediction on the target test item. Though the influence function is intuitive and easy to compute, we show in experiments that it achieves similar performance to the naive NN baseline, while DUTI provides better bug recommendations and has the advantage of producing exact fixes.

Our work is also related to data cleaning studied in data science and statistics. Earlier work (Jiang and Zhou 2004; Zhu, Wu, and Chen 2003; Brodley and Friedl 1996) applies an ensemble of classifiers to the training examples, and detects whether the class label assigned to each example is consistent with the output of these classifiers. The main problem with this approach is that the classifiers used to detect mislabeled examples are themselves constructed from contaminated training items. Another approach is to design robust statistics with a high break-point against incorrect training examples (Huber 2011). None of the above-mentioned methods can deal with systematic biases. Later, (Bhadra and Hein 2015; Valizadegan and Tan 2007) define a consistency metric and perturb the training labels to maximize the consistency metric. This approach results naturally in a combinatorial optimization problem. In particular, Bhadra and Hein's method is able to incorporate expert verified data as hard constraints in their optimization formulation. However, their methods requires information on the exact number of bugs which is often not available. DUTI does not request such information.

Along the line of incorporating outside information in training set debugging, (Ghosh et al. 2016) requires the learned model to satisfy certain Probabilistic Computation Tree Logic (PCTL). That is, the bugs are revealed by the learned model violating logical constraints rather than by the trusted items. This approach complements DUTI and the two can potentially be combined in the future. We note that in complex machine learning applications it can be more difficult for experts to encode domain knowledge in rigorous logical statements, whereas providing verified trusted items is often easier.

Our work is partly inspired by the need to identify historical unfairness in a training set. DUTI thus joins recent work (Hardt et al. 2016; Corbett-Davies et al. 2017; Zemel et al. 2013; Feldman et al. 2015) on algorithmic fairness. In our experiments, we demonstrated how DUTI can be used to identify historical discrimination, with the hope that such identification will improve fairness of machine learning systems.

5 Limitations of DUTI and Future Work

This paper contributed to trustworthy machine learning by proposing a novel training set debugging algorithm DUTI. As the experiments demonstrated, DUTI was able to detect and fix many bugs in diverse data sets. Like any method, DUTI has its limitations. We discuss three major ones.

Applicability: While a violated trusted item often indicates training set label bugs, it is not always the case. A domain expert needs to bear in mind other reasons that violate a trusted item while nothing is wrong with the training labels. Figure 5(a,c,e) presents three common cases. In all cases, the dashed line is the true decision boundary. The blue points represent the training set, which obey (stochastically in the third case) the true boundary. Therefore, there is no bug in the training labels *per se*. The red points are the trusted items, which also obey the true boundary. However, the solid curve is the boundary learned from the training set. In all three cases, some trusted items are indeed violated by the learned boundary: In (a), the true boundary is nonlinear but the hypothesis space is linear, resulting in underfitting and thus violating both trusted items. In (c), the trusted items are in a region of the feature space poorly covered by training data, thus unreliable extrapolation. This happens in covariate shift, for example. In (e), the underlying conditional distribution $P(Y \mid X)$ has high Bayes error (e.g. close to 0.5 near the true boundary), and the hypothesis space is too rich and ill-regularized, resulting in overfitting.

In all three cases, it is inappropriate to apply DUTI in the first place. In fact, blindly running DUTI will result in the "flagged bugs" in (b, d, f), respectively, and none are true bugs. Conversely, after the domain expert verifies that none of DUTI flagged items are bugs, she should suspect some of the above reasons in the machine learning pipeline.

Theoretical Guarantee: The trusted items need to be informative for DUTI to work. For example, in Figure 1(a) if the two trusted items were "hired" at $(1, 1)$ and "not hired" at $(1, 0)$, they would still be correct but would not have revealed the systematic bias in the data. In our real data experiments trusted items are i.i.d samples, but DUTI does not require this. Future work should study theoretical guarantees of (potentially non-iid) trusted items on debugging.

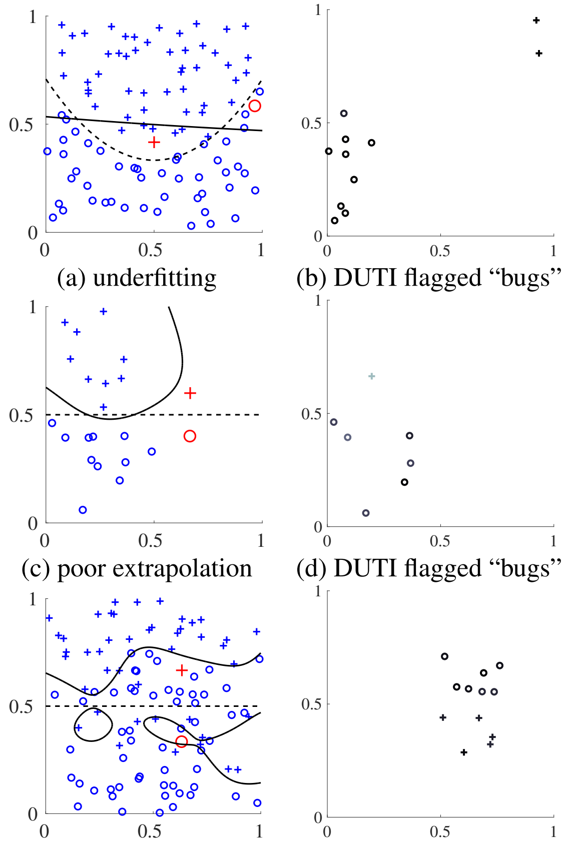


Figure 5: Trusted item violations not caused by label bugs

Scalability: The current implementation of DUTI has limited speed and scalability. At each step of optimization (5) or (9), it has to compute $\theta(\delta)$ which is equivalent to training the learner. Even with smart initialization, this repeated learning subroutine still takes the majority of time. For large data sets, one iteration can take minutes. DUTI currently can handle training set size n in the thousands. Future work is needed to yield a faster algorithm and implementation.

To conclude, in this work we designed DUTI, an efficient algorithm that helps the users to identify and correct training set bugs on any ERM learner, with the help of verified trusted items. Empirical experiments demonstrated that DUTI is able to tackle different types of realistic systematic bugs and outperforms other related methods. Future work will be dedicated to building a general theory of debugging and improving scalability through smart optimization.

Acknowledgment. This work is supported by NSF Awards 1704117, 1623605, 1561512, 1545481, 1447449, 1634597, and 1740707; AFOSR Award FA9550-13-1-0138; and Subcontract 3F-30222 from Argonne National Laboratory.

References

Ball, T., and Rajamani, S. K. 2002. The S LAM project: debugging system software via static analysis. *ACM SIGPLAN Notices* 37(1):1–3.

Bhadra, S., and Hein, M. 2015. Correction of noisy labels via mutual consistency check. *Neurocomputing* 160:34–52.

Brodley, C. E., and Friedl, M. A. 1996. Improving automated land cover mapping by identifying and eliminating mislabeled observations from training data. In *Geoscience and Remote Sensing Symposium, 1996. IGARSS'96.'Remote Sensing for a Sustainable Future.', International*, volume 2, 1379–1381. IEEE.

Brodley, C. E., and Friedl, M. A. 1999. Identifying mislabeled training data. *Journal of artificial intelligence research* 11:131–167.

Cadamuro, G.; Gilad-Bachrach, R.; and Zhu, X. 2016. Debugging machine learning models. *ICML Workshop on Reliable Machine Learning in the Wild.*

Caramanis, C., and Mannor, S. 2008. Learning in the limit with adversarial disturbances. In *COLT*, 467–478.

Corbett-Davies, S.; Pierson, E.; Feller, A.; Goel, S.; and Huq, A. 2017. Algorithmic decision making and the cost of fairness. *arXiv preprint arXiv:1701.08230.*

Feldman, M.; Friedler, S. A.; Moeller, J.; Scheidegger, C.; and Venkatasubramanian, S. 2015. Certifying and removing disparate impact. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 259–268. ACM.

Ghosh, S.; Lincoln, P.; Tiwari, A.; and Zhu, X. 2016. Trusted machine learning for probabilistic models. *ICML Workshop on Reliable Machine Learning in the Wild.*

Guruswami, V., and Raghavendra, P. 2009. Hardness of learning halfspaces with noise. *SIAM Journal on Computing* 39(2):742–765.

Hardt, M.; Price, E.; Srebro, N.; et al. 2016. Equality of opportunity in supervised learning. In *Advances in Neural Information Processing Systems*, 3315–3323.

Huber, P. J. 2011. Robust statistics. In *International Encyclopedia of Statistical Science*. Springer. 1248–1251.

Hutchins, M.; Foster, H.; Goradia, T.; and Ostrand, T. 1994. Experiments on the effectiveness of dataflow-and controlflow-based test adequacy criteria. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, 191–200. IEEE.

Jiang, Y., and Zhou, Z.-H. 2004. Editing training data for knn classifiers with neural network ensemble. *Advances in Neural Networks–ISNN 2004* 356–361.

Kamishima, T.; Akaho, S.; and Sakuma, J. 2011. Fairness-aware learning through regularization approach. In *Data Mining Workshops (ICDMW), 2011 IEEE 11th International Conference on*, 643–650. IEEE.

Koh, P. W., and Liang, P. 2017. Understanding black-box predictions via influence functions. In *The 34rd International Conference on Machine Learning (ICML).*

Kohavi, R. 1996. Scaling up the accuracy of naive-Bayes classifiers: A decision-tree hybrid. In *KDD*, volume 96, 202–207.

Lu, S.; Li, Z.; Qin, F.; Tan, L.; Zhou, P.; and Zhou, Y. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5.

Mathworks. 2017. Neural network toolbox: Training a deep neural network for digit classification. `https://www.mathworks.com/help/nnet/examples`.

Selvaraju, R. R.; Das, A.; Vedantam, R.; Cogswell, M.; Parikh, D.; and Batra, D. 2016. Grad-cam: Why did you say that? visual explanations from deep networks via gradient-based localization. *arXiv preprint arXiv:1610.02391.*

Shapiro, E. Y. 1983. *Algorithmic program debugging*. MIT press.

Smilkov, D.; Thorat, N.; Kim, B.; Viégas, F.; and Wattenberg, M. 2017. Smoothgrad: removing noise by adding noise. *arXiv preprint arXiv:1706.03825.*

Sundararajan, M.; Taly, A.; and Yan, Q. 2016. Gradients of counterfactuals. *arXiv preprint arXiv:1611.02639.*

Valizadegan, H., and Tan, P.-N. 2007. Kernel based detection of mislabeled training examples. In *Proceedings of the 2007 SIAM International Conference on Data Mining*, 309–319. SIAM.

Zemel, R.; Wu, Y.; Swersky, K.; Pitassi, T.; and Dwork, C. 2013. Learning fair representations. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 325–333.

Zhu, X.; Wu, X.; and Chen, Q. 2003. Eliminating class noise in large datasets. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 920–927.