Summary on A few Testing Machine learning papers

Presented by : Ji Gao

[1]Department of Computer Science, University of Virginia
https://qdata.github.io/deep2Read/

August 26, 2018

Overview

1. The challenge of verification and testing of machine learning
2. Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints
3. Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings
4. Understanding Black-box Predictions via Influence Functions
5. An Empirical Study of Bugs in Machine Learning Systems
6. What's your ML Test Score? A rubric for ML production systems
7. DeepXplore: Automated White-box Testing of Deep Learning Systems
8. A Family of Test Adequacy Criteria for Database-Driven Applications
9. On testing non-testable programs

Table

Table: Summarized papers

Name	Author	Conference
Software Testing: A research Travelogue (2000 - 2014)	Orso Alessandro (Georgia Tech) Gregg Rothermel (University of Nebraska-Lincoln)	Proceedings of the on Future of Software Engineering
A Few Useful Things to Know about Machine Learning	Pedro Domingos (University of Washington)	Communications of the ACM, 2012
DeepXplore: Automated Whitebox Testing of Deep Learning Systems	Kexin Pei, Yinzhi Cao, Junfeng Yang, Suman Jana (Columbia)	Arxiv
(Section [??]) Properties of Machine Learning Applications for Use in Metamorphic Testing.	Christian Murphy, Gail Kaiser, Lifeng Hu, Leon Wu (Columbia)	SEKE, 2008
(Section [??]) Whats your ML Test Score? A rubric for ML production systems	Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley (Google)	Reliable Machine Learning in the Wild-NIPS 2016 Workshop. 2016.
(Section [5]) An Empirical Study of Bugs in Machine Learning Systems	Ferdian Thung, Shaowei Wang, David Lo, Lingxiao Jiang (Sin-	2012 International Symposium on Software Reliability Engineering

Outline

The challenge of verification and testing of machine learning

- **Testing:** Evaluating the system in several conditions and observing its behavior, watching for defects.
- **Verification:** Producing a compelling argument that the system will not misbehave under a very broad range of circumstances.

The challenge of verification and testing of machine learning

- Machine learning practitioners have traditionally relied primarily on testing: Accuracy.
- No guarantee for upper bound.

Bring to adversarial samples

The challenge of verification and testing of machine learning

- Adversarial samples: Slightly perturbed samples that lead to an error
- Become a problem
- Testing can't solve this problem
- Even testing on adversarial samples can't reveal the robustness of the model
- Need stronger guarantee

- Several recent works:
  - *An abstraction-refinement approach to verification of artificial neural networks. Pulina, L., & Tacchella, A. (2010, July)*
  - *Safety Verification of Deep Neural Networks. Huang, X., Kwiatkowska, M., Wang, S., & Wu, M. (2016).*
  - *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. Katz, G., Barrett, C., Dill, D., Julian, K., & Kochenderfer, M. (2017). Computer Aided Verification 17*
- Doesn't seem very practical to me:
  Use SAT solver.
  Only works on very small networks.
  Need to have specific activation function and limit to certain structure.

Outline

Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints

Abstract: Language is increasingly being used to define rich visual recognition problems with supporting image collections sourced from the web. Structured prediction models are used in these tasks to take advantage of correlations between co-occurring labels and visual input but risk inadvertently encoding social biases found in web corpora. In this work, we study data and models associated with multilabel object classification and visual semantic role labeling. We find that (a) datasets for these tasks contain significant gender bias and (b) models trained on these datasets further amplify existing bias. For example, the activity cooking is over 33% more likely to involve females than males in a training set, and a trained model further amplifies the disparity to 68% at test time. We propose to inject corpus-level constraints for calibrating existing structured prediction models and design an algorithm based on Lagrangian relaxation for collective inference. Our method results in almost no performance loss for the underlying recognition task but decreases the magnitude of bias amplification by 47.5% and 40.5% for multilabel classification and visual semantic role labeling, respectively.

- Visual recognition tasks: Mining relationships in the image
- However, bias in the data makes learner learns stereotypes.
- Cooking in dataset: 33% man and 66% woman
- Cooking in a unlabeled data by Learned annotator: 16% man and 84% woman
- Stereotypes got amplified

- Bias score:

$$b(verb, man) = \frac{c(verb, man)}{c(verb, man) + c(verb, woman)}$$

- Bias Amplification:

$$\frac{1}{|O|} \sum_{o} (\hat{b}(o, g) - b^*(o, g))$$

$\hat{b}$ is the bias of learned annotator, and b^* is the bias of original data. o is the output, and g is gender.

Debiasing the model

Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints

- We can force the bias in a certain range, but it's hard.
- Form this as an optimization problem.
- Can be optimized together with the original optimization process.

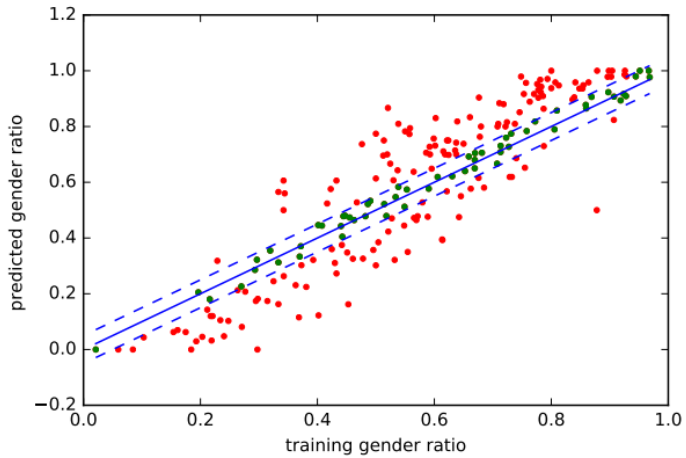
Debiasing the model

Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints

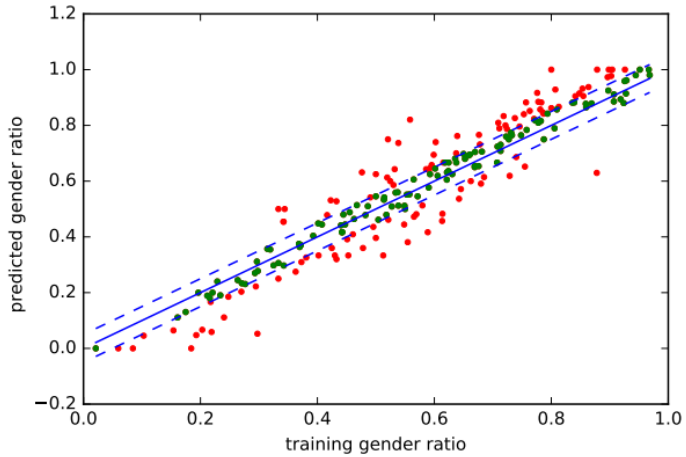
- Find $\arg\max_{y \in Y} f_\theta(y, i)$,$f_\theta(y, i)$ is a scoring function, in this case is $\log p(y|i, \theta)$
- Constraints $b^* - \gamma \leq \frac{\sum_i y^i_{v=v^*, r \in M}}{\sum_i y^i_{v=v^*, r \in M} + \sum_i y^i_{v=v^*, r \in W}} \leq b^* + \gamma$, i is the index of test sample
- Use combinatorial optimization tools to solve.

Experiment

Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints



(a) Bias analysis on imSitu vSRL without RBA



Experiment result

Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints

Method	Viol.	Amp. bias	Perf. (%)
vSRL: Development Set			
CRF	154	0.050	24.07
CRF + RBA	107	0.024	23.97
vSRL: Test Set			
CRF	149	0.042	24.14
CRF + RBA	102	0.025	24.01
MLC: Development Set			
CRF	40	0.032	45.27
CRF + RBA	24	0.022	45.19
MLC: Test Set			
CRF	38	0.040	45.40
CRF + RBA	16	0.021	45.38

Discussion

Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints

- Original method works properly, but still amplifies the stereotypes
- Possible to build a test set to exploit such bias?

Outline

Abstract

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

Abstract: The blind application of machine learning runs the risk of amplifying biases present in data. Such a danger is facing us with word embedding, a popular framework to represent text data as vectors which has been used in many machine learning and natural language processing tasks. We show that even word embeddings trained on Google News articles exhibit female/male gender stereotypes to a disturbing extent. This raises concerns because their widespread use, as we describe, often tends to amplify these biases. Geometrically, gender bias is first shown to be captured by a direction in the word embedding. Second, gender neutral words are shown to be linearly separable from gender definition words in the word embedding. Using these properties, we provide a methodology for modifying an embedding to remove gender stereotypes, such as the association between between the words receptionist and female, while maintaining desired associations such as between the words queen and female. We define metrics to quantify both direct and indirect gender biases in embeddings, and develop algorithms to "debias" the embedding. Using crowd-worker evaluation as well as standard benchmarks, we empirically demonstrate that our algorithms significantly reduce gender bias in embeddings while preserving the its useful properties such as the ability to cluster related concepts and to solve analogy tasks. The resulting embeddings can be used in applications without amplifying gender bias.

- Word Embedding trained include sexism result:
  $\overrightarrow{\text{man}} - \overrightarrow{\text{woman}} \approx \overrightarrow{\text{computer programmer}} - \overrightarrow{\text{homemaker}}$
- As widely used, word embedding can even amplify the gender stereotypes.
- Example: For a search algorithm use word embedding, if male names have a closer distance to the word "programmer" than female names, it would likely to return a male name first.

Goal of this paper

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

Major goals:

- Identify gender stereotypes.
- Debaising.

Related work

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

- Gender bias in language has been studied over decades in various fields.
- Several work on "fair" binary classification, including multiple ways to modify the prediction algorithm.
- Previous work dealing with gender bias by completely remove gender from the dataset.

Identify gender stereotypes in word embedding

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

- Step 1: Find out pair of words that are similar to the pair (she, he). Similar metric: Cosine similarity

$$S_{(a,b)}(x,y) = \cos(\vec{a} - \vec{b}, \vec{x} - \vec{y}) \text{ if } ||\vec{x} - \vec{y}|| \leq \delta$$

Extreme *she* occupations

1. homemaker	2. nurse	3. receptionist
4. librarian	5. socialite	6. hairdresser
7. nanny	8. bookkeeper	9. stylist
10. housekeeper	11. interior designer	12. guidance counselor

Extreme *he* occupations

1. maestro	2. skipper	3. protege
4. philosopher	5. captain	6. architect
7. financier	8. warrior	9. broadcaster
10. magician	11. figher pilot	12. boss

Identify gender stereotypes in word embedding

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

- Step 2: Use crowdsourcing to identify whether pair of words are stereotypes or not.

Gender stereotype *she-he* analogies.

sewing-carpentry	register-nurse-physician	housewife-shopkeeper
nurse-surgeon	interior designer-architect	softball-baseball
blond-burly	feminism-conservatism	cosmetics-pharmaceuticals
giggle-chuckle	vocalist-guitarist	petite-lanky
sassy-snappy	diva-superstar	charming-affable
volleyball-football	cupcakes-pizzas	hairdresser-barber

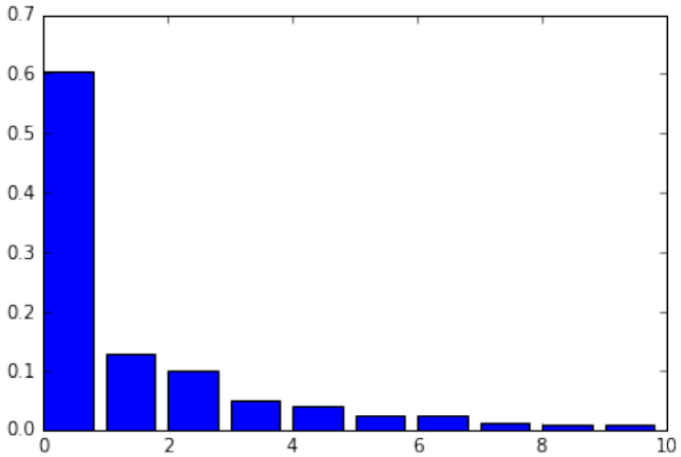
Gender appropriate *she-he* analogies.

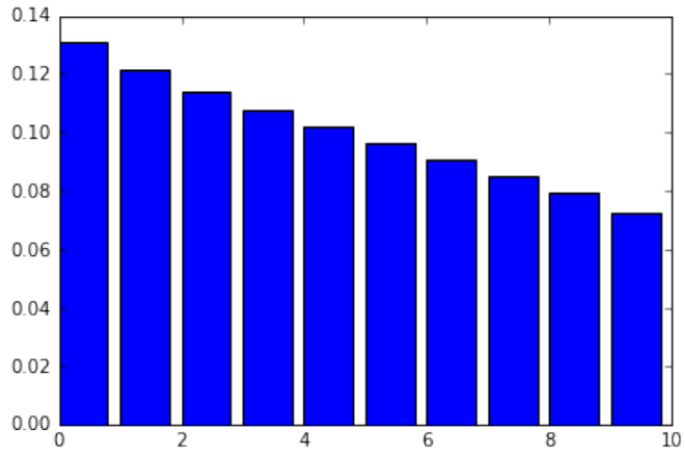
queen-king	sister-brother	mother-father
waitress-waiter	ovarian cancer-prostate cancer	convent-monastery

Identify gender direction

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

- Is there a universal gender vector exists?
- Aggregate different pair of words that reflect gender, i.e., he-she, mother-father, woman-man.
- Use PCA to capture the major direction.





Direct bias

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

- If a word should be gender-neutral, then the bias can be measured by

$$\text{DirectBias}_c = \frac{1}{|N|} \sum_{\omega \in N} |\cos(\vec{\omega}, g)|^c$$

g is the unit gender vector, and c as a hyperparameter.

Indirect bias

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

- What if two words which should have gender relationship to each other? The bias can be measured by projecting the embedding vector onto the gender space. Suppose $w = w_g + w_\perp$

$$\beta(w, v) = (w \cdot v - \frac{w_\perp \cdot v_\perp}{||w_\perp||_2 ||v_\perp||_2}) / w \cdot v$$

Debiasing

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

- Step 1: Identify the gender subspace
- Step 2: Define two methods: Hard de-biasing and Soft de-biasing.
- Hard debiasing: Neutralize all words to the gender space. Equalize the distance of pair of words outside the space.
- Soft debiasing: Tradeoff the previous method with similarity to the original embedding

Hard Debiasing:

Step 2a: Hard de-biasing (neutralize and equalize). Additional inputs: words to neutralize $N \subseteq W$, family of equality sets $\mathcal{E} = \{E_1, E_2, \ldots, E_m\}$ where each $E_i \subseteq W$. For each word $w \in N$, let $\vec{w}$ be re-embedded to

$$\vec{w} := (\vec{w} - \vec{w}_B)/\|\vec{w} - \vec{w}_B\|.$$

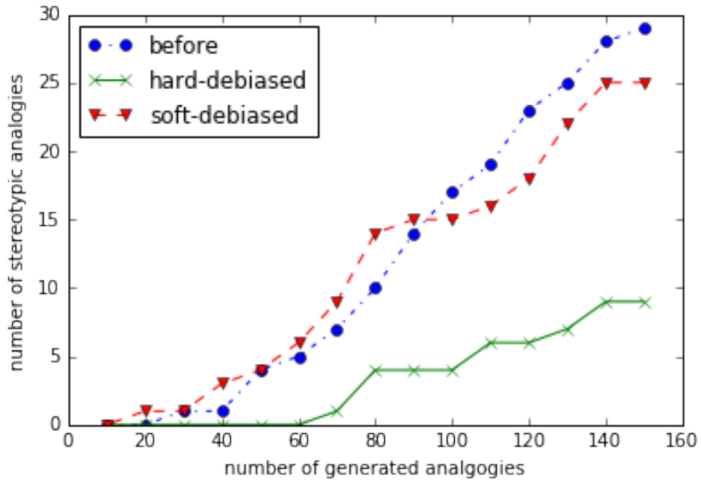
For each set $E \in \mathcal{E}$, let

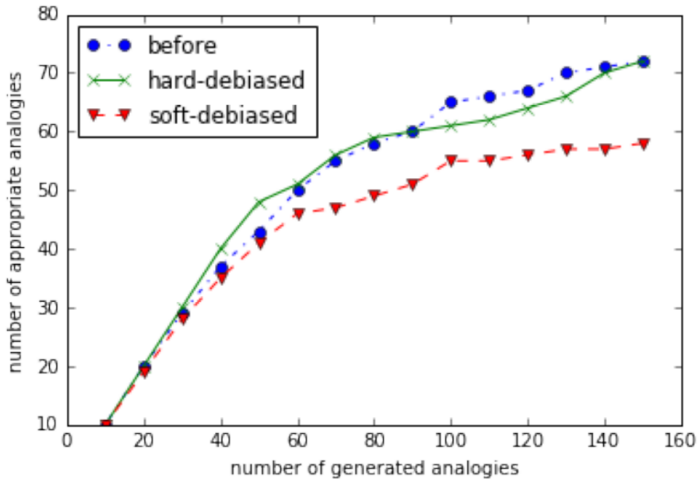
$$\begin{aligned} \mu &:= \sum_{w \in E} w/|E| \\ \nu &:= \mu - \mu_B \\ \text{For each } w \in E, \quad \vec{w} &:= \nu + \sqrt{1 - \|\nu\|^2} \frac{\vec{w}_B - \mu_B}{\|\vec{w}_B - \mu_B\|} \end{aligned}$$

Debiasing Result

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

Effectiveness: Is stereotypes successfully removed after the debiasing?





Debiasing Result

Man is to Computer Programmer as Woman is to Homemaker? Debiasing Word Embeddings

Accuarcy: Will debiasing affect the performance of the word embedding?

	RG	WS	analogy
Before	62.3	54.5	57.0
Hard-debiased	62.4	54.1	57.0
Soft-debiased	62.4	54.2	56.8

According to the result, the answer is no.

Outline

Understanding Black-box Predictions via Influence Functions

Abstract: How can we explain the predictions of a blackbox model? In this paper, we use influence functions a classic technique from robust statistics to trace a models prediction through the learning algorithm and back to its training data, thereby identifying training points most responsible for a given prediction. To scale up influence functions to modern machine learning settings, we develop a simple, efficient implementation that requires only oracle access to gradients and Hessian-vector products. We show that even on non-convex and non-differentiable models where the theory breaks down, approximations to influence functions can still provide valuable information. On linear models and convolutional neural networks, we demonstrate that influence functions are useful for multiple purposes: understanding model behavior, debugging models, detecting dataset errors, and even creating visually indistinguishable training-set attacks.

Goal of this papers

Understanding Black-box Predictions via Influence Functions

- How can we explain the predictions on machine learning models?
- Intuition: Understand what the consequence will be if change training samples
- Idea: Borrow the **Influence functions** idea from robust statistics

- Machine learning: Learn from $X \rightarrow Y$
- Suppose $L(z, \theta)$ is the loss function, θ stands for parameters, z stands for training point (x, y)
- total loss $\frac{1}{n} \sum_{i=1}^{n} L(z_i, \theta)$, $z_i..z_n$ are samples
- $\hat{\theta} = \arg\min_\theta \frac{1}{n} \sum_{i=1}^{n} L(z_i, \theta)$
- Goal: Figure out the change of θ and $L(\theta)$ if a single point z changed

Influence Function

Understanding Black-box Predictions via Influence Functions

- influence function: if one sample is upweighted, how would the prediction result change?
- That is $\hat{\theta}' = \arg\min_\theta \frac{1}{n}\sum_{i=1}^{n} L(z_i, \theta) + \epsilon L(z, \theta)$
- The result from paper (Cook and Weisberg, 1982) is:
- $\mathcal{I}_{up,params}(z) := \frac{d\hat{\theta}}{d\epsilon}|_{\epsilon=0} = -H_{\hat{\theta}}^{-1}\nabla_\theta L(z, \hat{\theta})$, Here $H_{\hat{\theta}}$ is the Hessian matrix: $H_{\hat{\theta}} = \frac{1}{n}\sum_{i=1}^{n} \nabla_\theta^2 L(z_i, \hat{\theta})$.

Using influence Function to estimate the effect of removing a sample

Understanding Black-box Predictions via Influence Functions

- If $\hat{\theta}' = \arg\min_\theta \frac{1}{n}\sum_{i=1}^{n} L(z_i, \theta) + \epsilon L(z, \theta)$, we have $\frac{d\hat{\theta}}{d\epsilon}|_{\epsilon=0} = -H_{\hat{\theta}}^{-1}\nabla_\theta L(z, \hat{\theta})$
- Removing a sample is equivalent to $\epsilon = -\frac{1}{n}$
- When n is large, $\frac{1}{n} \approx 0$
- Therefore, $\hat{\theta}' - \hat{\theta} \approx -\frac{1}{n}\frac{d\hat{\theta}}{d\epsilon}|_{\epsilon=0} = \frac{1}{n}H_{\hat{\theta}}^{-1}\nabla_\theta L(z, \hat{\theta})$
- Similarly, for any test sample z_{test}, the change of $L(z_{test}, \theta)$ can be approximated by:

$$
\begin{aligned}
L(z_{test}, \theta)' - L(z_{test}, \theta) &\approx -\frac{1}{n}\frac{dL(z_{test}, \theta)}{d\epsilon}|_{\epsilon=0} \\
&= -\frac{1}{n}\nabla_\theta L(z_{test}, \theta)\frac{d\hat{\theta}}{d\epsilon}|_{\epsilon=0} \\
&= \frac{1}{n}\nabla_\theta L(z_{test}, \theta)H_{\hat{\theta}}^{-1}\nabla_\theta L(z, \hat{\theta})
\end{aligned}
$$

Influence Function contd.

Understanding Black-box Predictions via Influence Functions

- For perturbation on a data point $z \to z_\delta$, here $x_\delta = x + \delta$
- Can be viewed as adding a new training point while deleting the old point.
- $\hat{\theta} = \arg\min_\theta \frac{1}{n} \sum_{i=1}^{n} L(z_i, \theta) + \epsilon L(z_\delta, \theta) - \epsilon L(z, \theta)$
- We have

$$\begin{aligned} \mathcal{I}_{pert,params}(z) =& \mathcal{I}_{up,params}(z_\delta) - \mathcal{I}_{up,params}(z) \\ =& - H_{\hat{\theta}}^{-1} \nabla_\theta (L(z_\delta, \hat{\theta}) - L(z, \hat{\theta})) \end{aligned} \tag{1}$$

- Assume it's continuous, then the preturbation can be approximated as $\hat{\theta}_{z_\delta, -z} - \hat{\theta} \approx -H_{\hat{\theta}}^{-1} [\nabla_x \nabla_\theta L(z_\delta, \hat{\theta})] \delta$

An logistic regression example

Understanding Black-box Predictions via Influence Functions

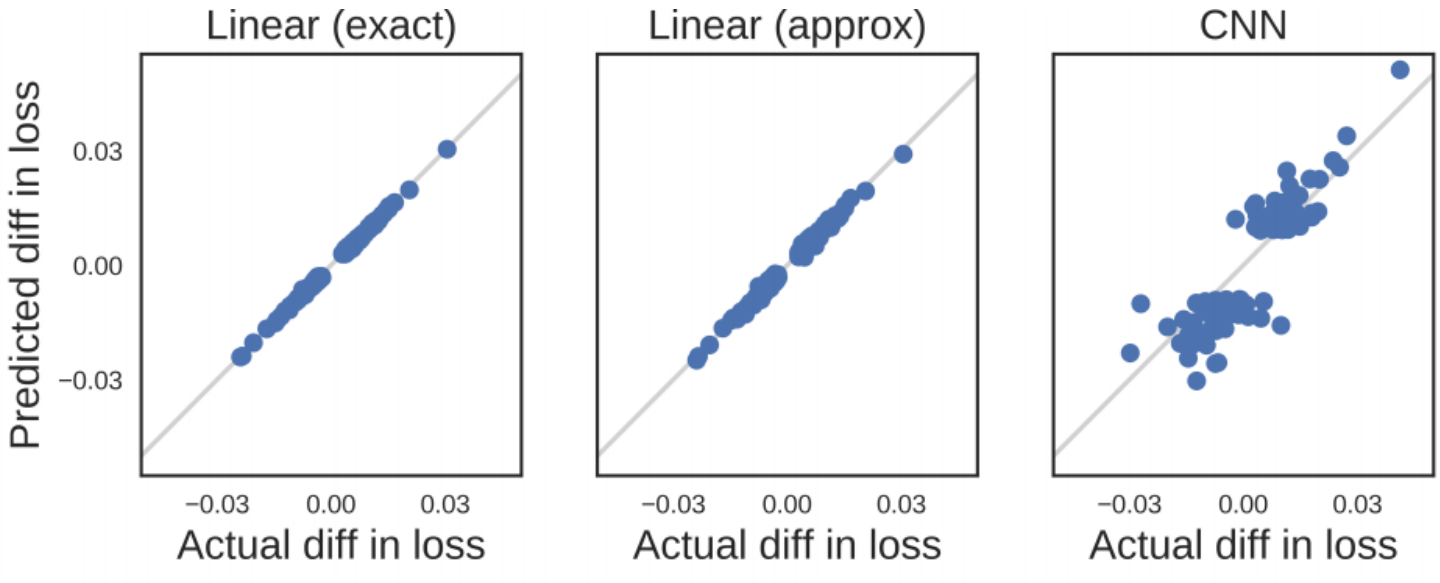
- Let $p(y|x) = \sigma(y\theta^T x)$. For a training point $z = (x, y)$, $L(z, \theta) = \log(1 + \exp(-y\theta^T x))$
- What's the influence on a test point?
  $\mathcal{I}_{up,loss}(z, z_{test}) = -y_{test} y \cdot \sigma(-y_{test}\theta^T x_{test}) \cdot (-y\theta^T x) \cdot x_{test}^T H_{\hat{\theta}}^{-1} x$
- $-y\theta^T x$ stands for error
- $H_{\hat{\theta}}^{-1}$ decides the impact, the size of the impact on the gradient direction.
- $x \cdot x_{test}$ represents the distance

Efficiently calculation of influence

Understanding Black-box Predictions via Influence Functions

- Calculation the hessian matrix takes $O(np^2 + p^3)$ time, and we also want to calculate it over all training points, which takes extra n times.
- Idea: Use Hessian-vector products(HVP) to approximate the hessian. That is, instead of $H_{\hat{\theta}}^{-1}$, calculate the product $H_{\hat{\theta}}^{-1} \nabla_\theta L(z, \hat{\theta})$ directly.
- Transform the matrix inversion to an optimization problem, and use conjugate gradient to solve it.
- In addition, do a stochastic estimation on the Hessian matrix.

- Compare the result of influence function and retraining, matches.



Non-convex case

- Previous result assume we can achieve the optimized solution. For Neural Network, it's not convex, so the method doesn't work?
- Use a convex approximation:
  $\tilde{L}(z,\theta) = L(z,\tilde{\theta}) + \nabla L(z,\tilde{\theta})^T(\theta - \tilde{\theta}) + \frac{1}{2}(\theta - \tilde{\theta})^T(H_{\tilde{\theta}} + \lambda I)(\theta - \tilde{\theta})$

Non-differentialable loss

Understanding Black-box Predictions via Influence Functions

- How to handle Non-differentialable loss?
- Idea: generate a smooth approximation.
- For example, Hingeloss $Hinge(s) = \max(0, 1 - s)$
- appriximated with $SmoothHinge(s) = tlog(1 + exp(\frac{1-s}{t}))$
- Result:

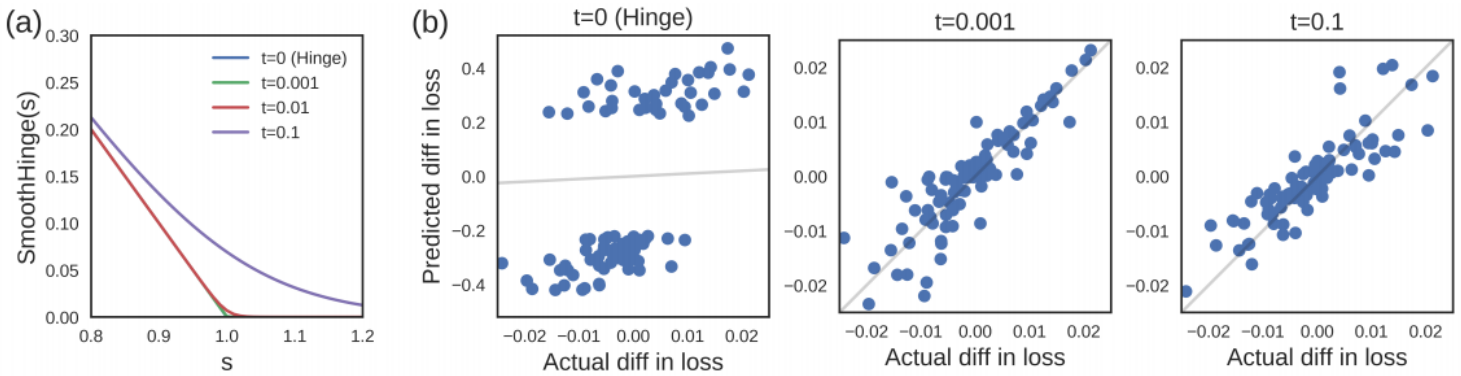


Figure 3. **Smooth approximations to the hinge loss. (a)** By varying t, we can approximate the hinge loss with arbitrary accuracy: the green and blue lines are overlaid on top of each other. **(b)** Using a random, wrongly-classified test point, we compared the predicted vs. actual differences in loss after leave-one-out retraining on the 100 most influential training points. A similar trend held for other test points. The SVM objective is to minimize $0.005 \|w\|_2^2 + \frac{1}{n} \sum_i \text{Hinge}(y_i w^\top x_i)$. **Left:** Influence functions were unable to accurately predict the change, overestimating its magnitude considerably. **Mid:** Using $\text{SmoothHinge}(\cdot, 0.001)$ let us accurately predict the change in the hinge loss after retraining. **Right:** Correlation remained high over a wide range of t, though it degrades when t is too large. When $t = 0.001$, Pearson's R = 0.95; when $t = 0.1$, Pearson's R = 0.91.

Case study 1: Understanding which sample is important to current prediction

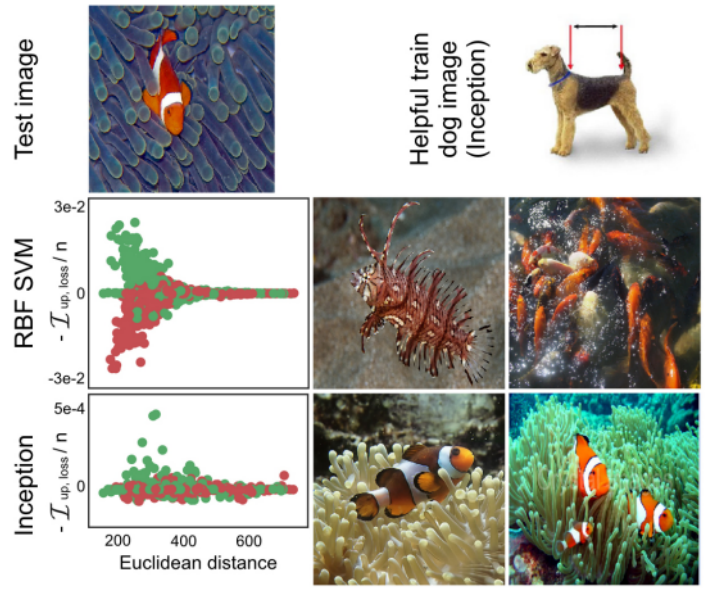


Figure 4. **Inception vs. RBF SVM. Bottom left:** $-\mathcal{I}_{\text{up,loss}}(z, z_{\text{test}})$ vs. $\|z - z_{\text{test}}\|_2^2$. Green dots are fish and red dots are dogs. **Bottom right:** The two most helpful training images, for each model, on the test. **Top right:** An image of a dog in the training set that helped the Inception model correctly classify the test image as a fish.

- If a train sample have a larger influence (positive $-\mathcal{I}_{up,loss}$), it's more helpful to the prediction.
- Result show in the graph.

Case study 2: Poisoning sample

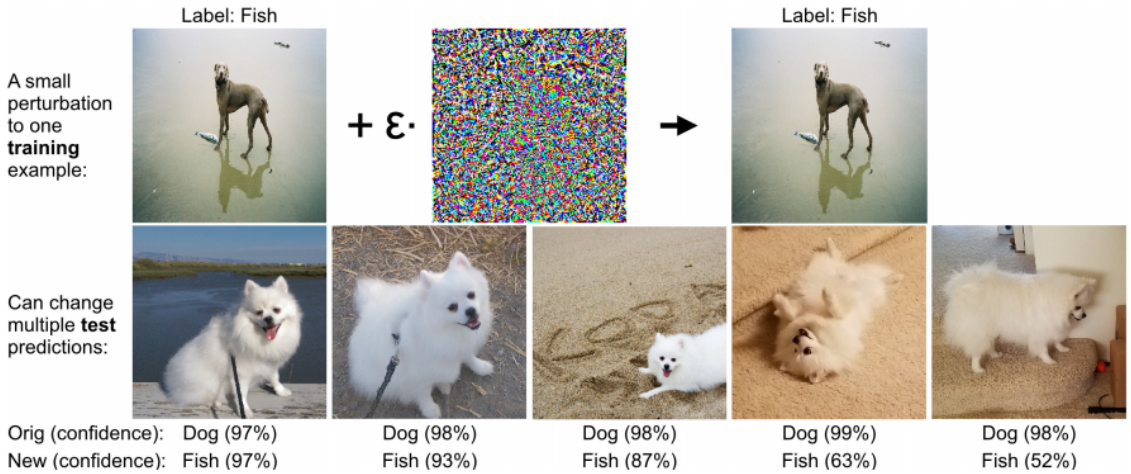


Figure 5. **Training-set attacks.** We targeted a set of 30 test images featuring the first author's dog in a variety of poses and backgrounds. By maximizing the average loss over these 30 images, we created a visually-imperceptible change to the particular training image (shown on top) that flipped predictions on 16 test images.

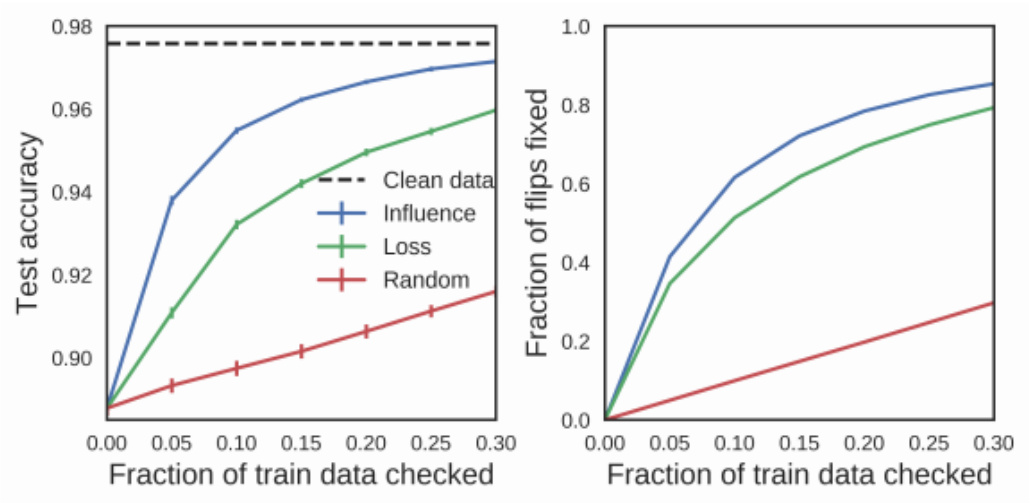
- For a typical test sample, modify the training sample following the rule of $z_i' := \prod(z_i' + \alpha \text{sign}(\mathcal{I}_{pert,loss}(z_i', z_{test}))$
- A slight modification on the training samples can change the prediction result!

Case study 3: Debugging domain mismatch

- Domain mismatch: Difference between the training set and testing set.
- Experiment: Predict diabetic patients to be readmitted using logistic regression with 127 features.
  Originally, 3 out of 24 children under age 10 in training set is readmitted. Manually remove 20 data from the set, so currently it's 3 out of 4. A bias is created between training and test set, and many children data in the test set is wrongly predicted in the test set.
- Without influence function: Check the learned θ on the feature children, not very abnormal: 14 features have a larger coefficient.
- Use influence function: All 4 children samples left have a very strong influence on those wrongly predicted samples, 30-40 times larger than others. Therefore there exists a domain mismatch.

Case study 4: Fixing mislabeled examples

- Idea: mislabeled example have a negative effect on the prediction.
- Experiment: Manually random flip 10% data.
- Baseline1: Check data points randomly
- Baseline2: Check those data points with highest loss.
- Method: Check those data points that would have a positive effect if removed using influence function.



Related Work

- Influence Function: In different statistics papers (Cook, 1977; Cook & Weisberg, 1980; 1982)
- Several previous papers in ML use influence functions to study model robustness and do fast cross-validation in kernel methods (Christmann & Steinwart (2004); Debruyne et al. (2008); Liu et al. (2014))
- Training set attack is equivalent to previous training set attack (Biggio et al. (2012)), and several poisoning attack papers (Mei & Zhu, 2015; Li et al., 2016).
- (Cadamuro et al. (2016)) consider the task of taking an incorrect test prediction and finding a small subset of training data such that changing the labels on this subset makes the prediction correct.

Outline

An Empirical Study of Bugs in Machine Learning Systems

Abstract: Many machine learning systems that include various data mining, information retrieval, and natural language processing code and libraries are used in real world applications. Search engines, internet advertising systems, product recommendation systems are sample users of these algorithm-intensive code and libraries. Machine learning code and toolkits have also been used in many recent studies on software mining and analytics that aim to automate various software engineering tasks. With the increasing number of important applications of machine learning systems, the reliability of such systems is also becoming increasingly important. A necessary step for ensuring reliability of such systems is to understand the features and characteristics of bugs occurred in the systems. A number of studies have investigated bugs and fixes in various software systems, but none focuses on machine learning systems. Machine learning systems are unique due to their algorithm-intensive nature and applications to potentially large-scale data, and thus deserve a special consideration. In this study, we fill the research gap by performing an empirical study on the bugs in machine learning systems. We analyze three systems, Apache Mahout, Lucene, and OpenNLP, which are data mining, information retrieval, and natural language processing tools respectively. We look into their bug databases and code repositories, analyze a sample set of bugs and corresponding fixes, and label the bugs into various categories. Our study finds that 22.6% of the bugs belong to the algorithm/method category, 15.6% of the bugs belong to the non-functional category, and 13% of the bugs belong to the assignment/initialization category. We also report the relationship between bug categories and bug severities, the time and effort needed to fix the bugs, and bug impacts. We highlight several bug categories that deserve attention in future research.

Goal of this paper

- Intuition: Bugs are prevalent in software systems. None of the previous work focus on Machine Learning systems
- Goal
- 1. understand the nature of the bugs in machine learning systems
- 2. help to deal with bugs

Analyze 3 machine learning systems/libraries.

- Apache Mahout: A data mining library
- Apache Lucene: A information retrieval library.
- Apache OpenNLP: A NLP toolkit

Contributions:

- first to analyze on machine learning systems
- manually categorize 500 bug reports
- investigate the relationship between bug categories and bug severities, bug fixing time and effort, and bug impact.

Categories: From paper *Defect categorization: making use of a decade of widely varying historical data* Categories

Table II
BUG CATEGORIES BASED ON CODE DEFECT TYPES IN [34]

Category	Definition
algorithm/method	The implementation of an algorithm/method does not follow the expected behavior.
assignment/initialization	Error in assigning variable values.
checking	Missing necessary checks that lead to an error or a wrong error message.
data	Wrong use of data structure.
external interface	Error in interfacing with other systems or users, such as using deprecated methods from other systems, required updates to own external interfaces for ease of usage, etc.
internal interface	Error in interfacing with another component of the same system, such as violating the contract of inheritance, wrong use of operations from other classes, etc.
logic	Incorrect expressions in conditional statements (e.g., if, while, etc.)
non-functional	Violations in non-functional requirements, such as improper variable or method names, wrong documentation to the implementation of a method, etc.
timing/optimization	Error that causes concurrency or performance issues, such as deadlock, high memory usage, etc.
configuration	Error in non-code (e.g., configuration files) that affects functionality.
others	Other bugs that do not fall into one of the above categories.

RQ1: How often bugs appear? RQ2: What kind of bug appear? RQ3: How severe are various kinds of bugs? RQ4: How long does it take to fix those bugs? RQ5: How much effort is needed to fix the bug? RQ6: How many files need to be fixed for various bugs?

RQ1: Bug frequency

Table III
BUG DENSITIES IN MACHINE LEARNING SYSTEMS

Application	Bug Count Per kLOC	Bug Count Per Year
Mahout	1.79 bugs/kLOC	73.36 bugs/year
Lucene	2.77 bugs/kLOC	144.76 bugs/year
OpenNLP	1.45 bugs/kLOC	95.68 bugs/year

Lucene have higher bug desity than Mahout and OpenNLP.

RQ2: Bug Types

Table IV
BUG TYPES

Type	Count	Percentage
algorithm/method	113	22.60%
non-functional	78	15.60%
assignment/initialization	65	13.00%
checking	57	11.40%
external interface	38	7.60%
internal interface	38	7.60%
data	28	5.60%
logic	27	5.40%
configuration	27	5.40%
timing/optimization	24	4.80%
others	5	1%

Algorithm $>$ non-functional $>$ assignment

RQ3: Bug severities

Table V
BUG COUNTS FOR VARIOUS BUG SEVERITIES

Type	Severity	Count	Proportion	Type	Severity	Count	Proportion
algorithm/method	Blocker	1	0.88%	internal interface	Blocker	1	2.63%
	Critical	3	2.65%		Critical	0	0.00%
	Major	70	61.95%		Major	22	57.89%
	Minor	33	29.20%		Minor	11	28.95%
	Trivial	6	5.31%		Trivial	4	10.53%
assignment/initialization	Blocker	2	3.08%	logic	Blocker	0	0.00%
	Critical	1	1.54%		Critical	0	0.00%
	Major	34	52.31%		Major	15	55.56%
	Minor	24	36.92%		Minor	8	29.63%
	Trivial	4	6.15%		Trivial	4	14.81%
checking	Blocker	2	3.51%	non-functional	Blocker	0	0.00%
	Critical	1	1.75%		Critical	0	0.00%
	Major	32	56.14%		Major	38	48.72%
	Minor	17	29.82%		Minor	29	37.18%
	Trivial	5	8.77%		Trivial	11	14.10%
configuration	Blocker	0	0.00%	timing/optimization	Blocker	0	0.00%
	Critical	0	0.00%		Critical	0	0.00%
	Major	19	70.37%		Major	19	79.17%
	Minor	6	22.22%		Minor	5	20.83%
	Trivial	2	7.41%		Trivial	0	0.00%
data	Blocker	0	0.00%	others	Blocker	0	0.00%
	Critical	1	3.57%		Critical	0	0.00%
	Major	15	53.57%		Major	3	60.00%
	Minor	12	42.86%		Minor	0	0.00%
	Trivial	0	0.00%		Trivial	2	40.00%
external interface	Blocker	1	2.63%				
	Critical	1	2.63%				
	Major	21	55.26%				
	Minor	12	31.58%				
	Trivial	3	7.89%				

RQ4: Bug-fixing duration

Table VI
BUG-FIXING DURATIONS IN TERMS OF DAYS

Type	Min	Max	Mean	Median
algorithm/method	0.0022	2433.7033	91.7238	3.8740
assignment/initialization	0.0003	160.9271	9.9160	0.5000
checking	0.0017	195.7766	17.1335	1.1175
configuration	0.0016	195.9011	22.3583	2.8032
data	0.0014	676.3825	40.8279	2.2666
external interface	0.0006	1700.5871	69.2463	0.4275
internal interface	0.0029	1688.5275	93.3543	2.4852
logic	0.0016	59.8305	6.8892	1.2537
non-functional	0.0006	1330.4142	47.8057	0.6949
timing/optimization	0.0017	569.7309	71.6649	3.3596
others	0.0005	1.3128	0.3344	0.0594

RQ5: Bug-fixing effort

Table X
BUG-FIXING EFFORT (DETAILED)

Type	Effort	Count	Proportion	Type	Effort	Count	Proportion
algorithm/method	1	71	62.83%	internal interface	1	25	65.79%
	2	17	15.04%		2	3	7.89%
	>2	25	22.12%		>2	10	26.32%
assignment/initialization	1	51	78.46%	logic	1	22	81.84%
	2	11	16.92%		2	4	14.81%
	>2	3	4.62%		>2	1	3.70%
checking	1	32	56.14%	non-functional	1	54	69.23%
	2	16	28.07%		2	14	17.95%
	>2	9	15.79%		>2	10	12.82%
configuration	1	16	59.26%	timing/optimization	1	6	25.00%
	2	7	25.93%		2	10	41.67%
	>2	4	14.81%		>2	8	33.33%
data	1	16	57.14%	others	1	5	100.00%
	2	8	28.57%		2	0	0.00%
	>2	4	14.29%		>2	0	0.00%
external interface	1	26	68.42%				
	2	5	13.16%				
	>2	7	18.42%				

RQ6: Impact on files

Table XI
NUMBERS OF IMPACTED FILES PER BUG

Type	Min	Max	Mean	Median
algorithm/method	1	117	11.7434	3
assignment/initialization	1	74	5.2459	2
checking	1	115	7.2982	2
configuration	1	40	5.2692	1
data	1	124	12.3571	3
external interface	1	92	13.3514	3
internal interface	1	52	7.6842	4.5
logic	1	15	2.7778	2
non-functional	1	676	16.7143	2
others	1	47	10.2000	1
timing/optimization	1	69	11.5833	5

Outline

What's your ML Test Score? A rubric for ML production systems

Abstract: Using machine learning in real-world production systems is complicated by a host of issues not found in small toy examples or even large offline research experiments. Testing and monitoring are key considerations for assessing the production-readiness of an ML system. But how much testing and monitoring is enough? We present an ML Test Score rubric based on a set of actionable tests to help quantify these issues.

- Real world machine learning systems might get problem not found in small **toy examples** or even **large offline experiments**.
- This paper list a set of actionable tests, and design a scoring system to measure how ready for production a given machine learning system is.

Scoring

For each test, one point is awarded for executing the test manually and documenting and distributing the results. A second point is awarded if there is a system in place to run that test automatically on a repeated basis. The final ML Test Score is computed by taking the minimum of the scores aggregated for every test.

- 0 points: More of a research project than a productionized system.
- 1-2 points: Not totally untested, but it is worth considering the possibility of serious holes in reliability.
- 3-4 points: Theres been first pass at basic productionization, but additional investment may be needed.
- 5-6 points: Reasonably tested, but its possible that more of those tests and procedures may be automated.
- 7-10 points: Strong levels of automated testing and monitoring, appropriate for missioncritical systems.
- 12+ points: Exceptional levels of automated testing and monitoring.

Related Work

Chris Murphy, Gail E Kaiser, and Marta Arias. An approach to software testing of machine learning applications. In SEKE, page 167. Citeseer, 2007.

Tests for features and data

Machine learning systems differ from traditional software-based systems in that the behavior of ML systems is not specified directly in code but is learned from data.

- Distribution: Test that the **distributions of each feature** match your expectations
- Correlation: Test the **relationship between each feature and the target**, and the pairwise correlations between individual signals.
- Calculation cost: Test the **cost of each feature**.
- Validity: Test that a model does not contain any features that have been manually determined as unsuitable for use.
- Privacy: Test that your system maintains **privacy** controls across its entire data pipeline.
- Develop time: Test the calendar time needed to develop and add a new feature to the production model.
- Code: Test all code that creates input features, both in training and serving.

Tests for Model Development

It can be all too tempting to rely on a single-number summary metric to judge performance, perhaps masking subtle areas of unreliability.

- Code: Test that every model specification undergoes a code review and is checked in to a repository.
- Metric: Test the relationship between offline proxy metrics and the actual impact metrics.
- Hyperparameters: Test the impact of each tunable hyperparameter.
- Staleness: Test the effect of model staleness.
- Test against a simpler model as a baseline.
- Test model quality on important data slices.
- Test the model for implicit bias.

Tests for ML Infrastructure

- Test the reproducibility of training. Use the same data to train twice.
- Unit test model specification code.
- Integration test the full ML pipeline.
- Test model quality before attempting to serve it.
- Test that a single example or training batch can be sent to the model, and changes to internal state can be observed from training through to prediction.
- Test models via a canary process before they enter production serving environments.
- Test how quickly and safely a model can be rolled back to a previous serving version.

Monitoring Tests for ML

- Test for upstream instability in features, both in training and serving
- Test that data invariants hold in training and serving inputs.
- Test that your training and serving features compute the same values.
- Test for model staleness.
- Test for NaNs or infinities appearing in your model during training or serving.
- Test for dramatic or slow-leak regressions in training speed, serving latency, throughput, or RAM usage.
- Test for regressions in prediction quality on served data.

Outline

DeepXplore: Automated White-box Testing of Deep Learning Systems

Abstract: ... We design, implement, and evaluate DeepXplore, the first white-box framework for systematically testing real-world DL systems. We address two main problems: (1) generating inputs that trigger different parts of a DL systems logic and (2) identifying incorrect behaviors of DL systems without manual effort. First, we introduce neuron coverage for systematically estimating the parts of DL system exercised by a set of test inputs. Next, we leverage multiple DL systems with similar functionality as cross-referencing oracles and thus avoid manual checking for erroneous behaviors. We demonstrate how finding inputs triggering differential behaviors while achieving high neuron coverage for DL algorithms can be represented as a joint optimization problem and solved efficiently using gradient-based optimization techniques ...

Goal of this paper

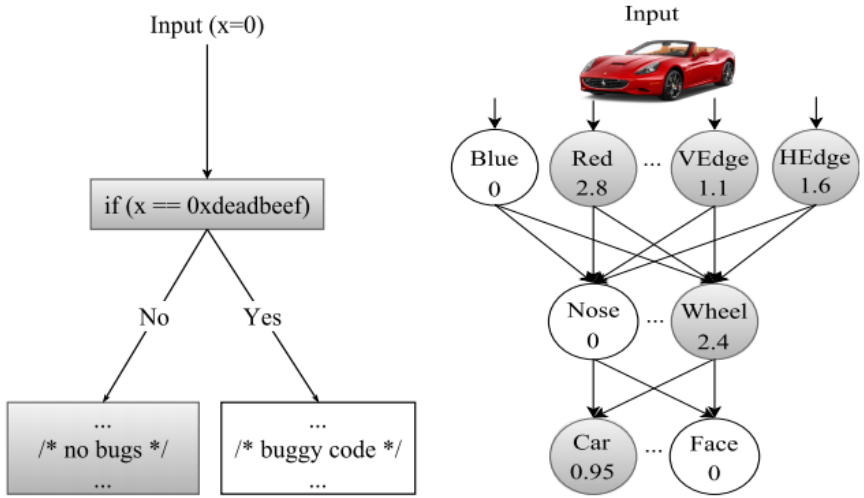
- It is important to ensure Deep Learning system works well.
- Problem: Traditional testing methods of Deep Learning have problems: Expensive labeling and Low neuron coverage coverage
- Automatic test case generation with high neuron coverage

Goal of this paper

DeepXplore: Automated White-box Testing of Deep Learning Systems

(a) Input 1 (b) Input 2 (darker version of 1)

Figure 1: An example erroneous behavior of Nvidia DAVE-2 self-driving car platform found by DeepXplore. The DNN-based self-driving car correctly decides to turn left for image (a); by contrast, the car decides to turn right and crashes into the guardrail for image (b), a slightly darker version of (a).



(a) A program with a rare branch (b) A DNN for detecting cars and faces

Figure 3: Comparison of program flows of a traditional program and a neural network. The nodes in gray denote the corresponding basic blocks or neurons that participated while processing an input.

Summary

DeepXplore: Automated White-box Testing of Deep Learning Systems

- Maximizing neuron coverage: For each step, select an inactivate neuron, and try to increase it's value.
- Labeling: Use differential testing. Train multiple target DNN together and find those test cases that is supported by all DNNs except one.

Metrics:

- Neuron coverage
- Execution time
- Retraining accuracy

Not very appealing as the major experiment result is on neuron coverage: Some metric defined by this paper.

Next step on this paper

- Use test case generated to improve the neural network:

  (In the paper) retraining

  Select out nodes that are not effective, remove/change them.
- The two things optimized together doesn't seems very compatible. It is possible that only optimize one thing will achieve better result.

Outline

A Family of Test Adequacy Criteria for Database-Driven Applications

Abstract: Although a software application always executes within a particular environment, current testing methods have largely ignored these environmental factors. Many applications execute in an environment that contains a database. In this paper, we propose a family of test adequacy criteria that can be used to assess the quality of test suites for database-driven applications. Our test adequacy criteria use dataflow information that is associated with the entities in a relational database. Furthermore, we develop a unique representation of a database-driven application that facilitates the enumeration of database interaction associations. These associations can reflect an applications definition and use of database entities at multiple levels of granularity. The usage of a tool to calculate intraprocedural database interaction as- sociations for two case study applications indicates that our adequacy criteria can be computed with an acceptable time and space overhead.

Goal

- Many software applications include databases.
- Current testing method ignore specific environment
- Goal of this paper is to design test accuracy criteria on database-driven approaches.

- Test cases should reveal faults of database applications.
- Specifically, this paper use database integrity to represent the faults:
  Validity:
  (1-v) it inserts a record into a database that does not reflect the real world
  (2-v) it fails to insert a record into the database when the status of the real world changes.
  Completeness:
  (1-c) it deletes a record from a database when this record still reflects the real world
  (2-c) the status of the real world changes and it fails to include this information as a record inside of a database.

- Traditional def-use test doesn't consider interaction with database.
- For the database case, should be doing an enumeration:

Definition 8. A test suite T for database interaction control flow graph $G_{DB} = (N_{DB}, E_{DB})$ satisfies the *all-database-DUs* test adequacy criterion if and only if for each association $\langle n_d, n_{use}, x \rangle$, where $x \in D(I)$ and $n_d, n_{use} \in N_{DB}$, there exists a test in T to create a complete path π_x in G_{DB} that covers the association. $\square$

- Also an algorithm to generate the new CFG graph including database interactions.

Future directions:

- Use this method as a guideline of automatic input generation.
- Perform static checking on such applications.

Outline

On testing non-testable programs

Abstract: A frequently invoked assumption in program testing is that there is an oracle (i.e. the tester or an external mechanism can accurately decide whether or not the outpugt produced by a program is correct). A program is non-testable if either an oracle does not exist or the tester must expend some extraordinary amount of time to determine whether or not the output is correct. The reasonableness of the oracle assumption is examined and the conclusion is reached that in many cases this is not a realistic assumption. The consequences of assuming the availability of an oracle are examined and alternatives investigated.

Oracle problem

On testing non-testable programs

- Non-testable programs:
  1. There's no oracle.
  2. The oracle is practically too difficult to find.
- Is it able to test without the oracle?

Testing without an oracle

On testing non-testable programs

- Produce a pseudo-oracle - dual coding
- Reduce the test set.
- Accept the plausible result