Generalized Oracle for Testing Machine Learning Computer Programs

Shin Nakajima

National Institute of Informatics
Tokyo, Japan

Abstract. Computation results of machine learning programs are not possible to be anticipated, because the results are sensitive to distribution of data in input dataset. Additionally, these computer programs sometimes adopt randomized algorithms for finding sub-optimal solutions or improving runtime efficiencies to reach solutions. The computation is probabilistic and the results vary from execution to execution even for a same input. The characteristics imply that no deterministic test oracle exists to check correctness of programs. This paper studies how a notion of oracles is elaborated so that these programs can be tested, and shows a systematic way of deriving testing properties from mathematical formulations of given machine learning problems.

1 Introduction

Software testing, debug testing, is a practical method for informal assurance on the quality and reliability of computer programs. It relies on *deterministic* test oracles with prescribed correct values usually given as design specifications. These oracles show the same behavior, for the same input data, returning the same checking result even for different executions of the target program.

Exact correct values of ML programs are not known in advance because they discern *unknown* valuable information from input dataset. Such programs are considered *non-testable* [6]. In addition, because ML problems are mostly NP-hard and intractable, ML programs may implement randomized algorithms for obtaining sub-optional solutions (e.g. [1]). This randomness makes computation results different from executions to executions. Deterministic oracles are not available for ML programs.

This paper studies an appropriate notion of oracles for testing ML programs. Section 2 studies characteristics of ML programs from testing views, and presents a general method to use metamorphic testing (MT) [3]. Section 3 and Section 4 present example cases of applying the proposed method to two machine learning problems, support vector machines (SVM) and neural networks (NN), which is followed by concluding discussions in Section 5.

2 Generalized Test Oracles

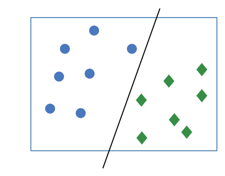
Deterministic test oracles for software testing rely on two key features. Given a specific test input data (X), a target program reaches a final state with a commutation result (Y). Firstly, those oracles determine whether the results are equal to prescribed correct values with respect to the input ($Y = C^X$). Secondly, the oracles behave the same for different executions of the target when the input is the same. ML programs violates assumptions for deterministic oracles; (1) correct values are not known in advance, (2) returned values are varied due to random values. We mainly consider the first aspect and then studies the second one for the case of a particular ML problem.

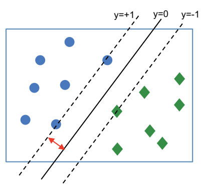
Pseudo oracles are using relative correct values instead of absolute correctness [6]. Such correct values are results of program executions. When two program versions (f_1 and f_2) exist, computation results of either one, say f_1, plays a role of *golden outputs*, correct values for the other (f_2). Alternatively, metamorphic testing (MT) [3] uses just one program, and considers two executions of f_1 with two different inputs (X_1 and X_2). The test inputs are related by a translation function T ($X_2 = T(X_1)$), and a certain relation R_T holds for the two execution outputs, $R_T(f_1(X_1), f_1(X_2))$. For a given translation T on the input test data, R_T is a *metamorphic relation* between the outputs. *Metamorphic testing* is a testing method that uses the relation R_T as a basis for pseudo oracles. We consider that some faults are in the program when R_T is violated, because R_T is so chosen that the two executions are believed to be the same. In simple cases, R_T is an identity ($f_1(X_1) = f_1(T(X_1))$).

The translation T and metamorphic relation (MR) M_T must respect functional requirements of the program. A systematic method to derive T and M_T from a problem description is needed so that identified pseudo oracles are effective to uncover faults with respect to the requirements.

Statistical machine learning is formulated as an optimization problem [1]. For a given machine learning task such as classifying a set of data, a problem is formulated as a mathematical function $F^C(\theta;\ \boldsymbol{x})$, where θ is a set of parameters to be determined. An ML problem is introducing a family of θ-indexed functions, $\{F^C(\theta;\ \boldsymbol{x})\}^{\theta}$, accompanied with *hyper-parameters* C defining an ML problem instance. Machine learning is determining parameter values θ^* such that an objective function $\mathcal{E}$, referring to $F^C(\theta;\ x)$, to be optimal. If the optimality refers to a minimization, then $\theta^* = arg\,min\ \mathcal{E}(\theta;\ \{\boldsymbol{x}^n\})$, where $\{\boldsymbol{x}^n\}$ is a dataset from which some valuable information is derived. An ML program is numerically solving to obtain θ^*. They are approximate because F^C are generally non-linear and so is $\mathcal{E}$. A *learnt* model defines a function $F(\theta^*;\ X)$ to calculate a result for a new data X. Note that hyper-parameters are constants in the optimization process. Thus, the parameters θ are dependent on C, θ^C. Obtaining optimal C is one of the major issues, but can be conducted only after software testing is passed.

From software testing views, ML computer programs can be characterized as follows. (1) An ML task is declaratively specified as an optimization problem, from which metamorphic properties are derived. (2) Learning is a process of obtaining (sub-) optimal parameters, which is solved by numerical searches. It implies that *correct* parameter values are not known in advance. (3) The learnt


(a) Hyperplane



(b) Support Vectors

Fig. 1. Support Vector Machines

result $F(\theta^*;\ X)$ with the optimal parameters is an *application program*, and its behavior is dependent on the parameter values θ^*.

From these observations, the optimization program is what should be tested, but without known absolute correct results. Thus, that program is a *quasi-testable* core with relative correctness criteria. As in existing work [5][7], MT can be a basis for pseudo oracle for testing ML programs. We will study two representative problems, because the information that is checked against the correctness criteria is dependent on ML tasks.

3 Support Vector Machines

A support vector machine (SVM) is a supervised machine learning classifier (e.g. Chapter 7 in [1]). Figure 1 (b) illustrates the concept of support vectors as opposed to a naive classifier in Figure 1 (a). The support vectors lie on the dotted hyperplanes parallel to the resultant separating hyperplane. *Margin*, a minimum gap between the support hyperplane and separating hyperplane, is so chosen to be maximum.

SVM is a constrained optimization problem. Below, $\langle \boldsymbol{x}^n,\ \ell^n \rangle$ are data points ($n = 1, \ldots,\ N$). $\boldsymbol{x}^n$ is a D-dimensional vector and ℓ^n is its label of either -1 or $+1$. $\boldsymbol{w}$ and b are two parameters defining hyperplanes.

$$arg\,\min_{\boldsymbol{w},b}\ \frac{1}{2}\|\ \boldsymbol{w}\ \|^2 \qquad \textbf{s.t.}\ \ell^n(\boldsymbol{w}^T{\cdot}\boldsymbol{x}^n\ +\ b) \geq 1$$

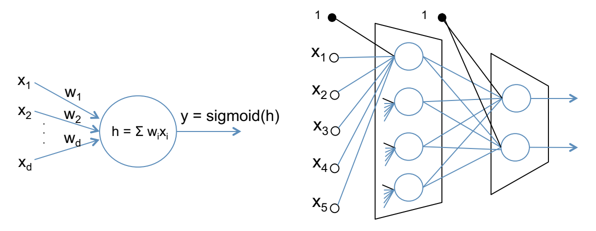
The problem is turned into a *dual* representation of a Lagrangian, $\mathcal{L}(\alpha_1, \ldots, \alpha_N)$, where α_n are Lagrange multipliers. It is actually a *soft* margin SVM and a hyper-parameter C is a measure allowing noise.

$$arg\,\max_{\alpha_n}\ \sum_{n=1}^{N}\alpha_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N}\alpha_n\alpha_m\ell^n\ell^m(\boldsymbol{x}^n{\cdot}\boldsymbol{x}^m)$$

$$\textbf{s.t.}\ \ 0 \leq \alpha_n \leq C\ (1 \leq n \leq N), \qquad \sum_{n=1}^{N}\alpha_n\ell^n\ =\ 0$$

The resultant multipliers constitute hyperplane parameters, where S is a set of indices of the support vectors (Figure 1(b)).

$$\boldsymbol{w}\ =\ \sum_{n\in S}\alpha_n\ell^n\boldsymbol{x}^n, \qquad b\ =\ \frac{1}{|\ S\ |}\sum_{m\in S}(\ell^m\ -\ \sum_{n\in S}\alpha_n\ell^n(\boldsymbol{x}^n{\cdot}\boldsymbol{x}^m))$$



(a) Perceptron (b) Two Layered Network

Fig. 2. Neural Network

Sequential minimal optimization (SMO) is a standard algorithm to solve SVM problem numerically. SMO decomposes the optimization problem into a sub-problem consisting of two data points, $W(a_1, a_2)$, consisting of two Lagrange multipliers with all the rest to be assumed as constants. $W(a_1, a_2)$ is analytically solvable, and the SMO solves it iteratively until solutions converge.

Programs are implementing the SMO algorithm, and are our test target. We derive metamorphic properties to test them from the Lagrangian $\mathcal{L}$ and functional behavior of the SMO algorithm. For example, interchanging indices of two data points, $\boldsymbol{x}^n \rightleftharpoons \boldsymbol{x}^m$, does not change $\mathcal{L}$, and multipliers are interchanged as well, $\alpha_n \rightleftharpoons \alpha_m$, The MR R^T is identity for this case. Another interesting property is *Reverse Labels*, $\langle \boldsymbol{x}^n, \ell^n \rangle \rightharpoonup \langle \boldsymbol{x}^n, -\ell^n \rangle$ for all n. Hyperplane parameters are changed accordingly, and thus $R^T = (\boldsymbol{w}^{(1)} = -\boldsymbol{w}^{(2)}) \wedge (b^{(1)} = -b^{(2)})$.

A metamorphic property *Reduce Margin* in [5] was effective for testing SVM programs. The property is a special instance of *Inclusive* property [7]. Its basic idea is adding a new data point to the input dataset for enabling *corner-case* testing. Because corner cases differ for different ML tasks, *Reduce Margin* takes into account the SVM characteristics that the problem is to determine separating hyperplanes with support vectors. Namely, we put newly added data points near the existing separating hyperplanes. [5] describes details about it.

4 Neural Networks

A neural network (NN) is a general framework for various machine learning tasks, regressions or classifiers (e.g. Chapter 5 in [1]). A perceptron (Figure 2 (a)) is a basic unit. Receiving a set of input signals $\{x_i\}$, it emits a signal y calculated by applying an activation function σ to a weighted sum of its input; $y = \sigma(\sum_{i=1}^{d} w_i x_i)$. An NN is a two layered network of perceptrons (Figure 2 (b)). All the input signals are fed into perceptrons in the hidden layer, and then all the signals from these perceptrons are input to perceptrons in the output layer. Let h and r be two activation functions. Given D-dimensional vector $\boldsymbol{x}$ and M perceptrons consisting of the hidden layer, signals ($k = 1, \ldots, R$) from the output layer is mathematically defined as below.

$$y_k(\boldsymbol{W};\ \boldsymbol{x}) = r(\ \sum_{j=0}^{M} v_{kj} h(\sum_{i=0}^{D} w_{ji} x_i)\ )$$

where v_{kj} and w_{ji} are weights, which are compactly written as $\boldsymbol{W}$.

For a set of input data $\{\langle \boldsymbol{x}^n,\ \boldsymbol{t}^n \rangle\}$ $(n = 1, \ldots,\ N)$, NN machine learning is an optimization problem using a loss function $\mathcal{E}$. Below, *cvecy* is a R-dimensional vector of the output signals.

$$\mathcal{E}(\boldsymbol{W}; \{\langle \boldsymbol{x}^n, \boldsymbol{t}^n \rangle\}) = \frac{1}{2}\sum_{n=1}^{N} \parallel \boldsymbol{t}^n - \boldsymbol{y}(\boldsymbol{W}; \boldsymbol{x}^n) \parallel^2, \qquad arg\min_{\boldsymbol{W}}\ \mathcal{E}(\boldsymbol{W}; \{\langle \boldsymbol{x}^n, \boldsymbol{t}^n \rangle\})$$

A naive algorithm to solve this optimization problem numerically is a steepest descent method (SD), calculating new weights until the values are converged; $\{\ \boldsymbol{W}^{(new)}\ =\ \boldsymbol{W}^{(old)}\ -\ \eta \nabla \mathcal{E}(\boldsymbol{W}^{old})\ \}$ where η (> 0) is a hyper-parameter, called a learning rate.

A standard learning algorithm employs the back propagation (BP) for calculating $\nabla \mathcal{E}(\boldsymbol{W}^{old})$ efficiently. Furthermore, NN training methods adopt several *learning tricks*. With the input normalization trick, all components x_i^n of $\boldsymbol{x}^n$ are pre-adjusted to follow a normal distribution of $Norm(0, 1)$ for each i. Stochastic gradient descent (SGD) is a randomized version of the SD. These techniques complicate the algorithm, and thus make programs difficult to test.

Interpretation of parameters θ in NN learning is quite different from the SVM case. In the latter, learning parameters are Lagrange multipliers to define a hyperplane, which is a good indicator to provide relative correctness criteria. Contrarily in NN learning, parameters θ are weights in $\boldsymbol{W}$. These are just numerical values and define no mathematical object. In addition, NN computer program executions are characterized by temporal behavior in view of convergence and/or optimality; programs may reach a stable state, but be trapped in local minimums. Therefore, such behavioral information is an important indicator, which must faithfully represent how $\boldsymbol{W}$ are changed as learning proceeds. However, a trace of loss function $\mathcal{E}$ or an accuracy for a given testing dataset, which is usually used, does not represent behavior faithfully. These pieces of information are influenced by slight changes in input training dataset.

Because the number of weights is very large in general, we cannot analyze the weights individually. Instead, we calculate statistical averages and deviations. Let $v_{kj}(e)$ be values at an epoch index e. A difference is that $\delta_{kj}(e) = v_{kj}(e) - v_{kj}(e-1)$. We, then, calculate statistical averages $(\mu(e))$ and deviations $(\sigma^2(e))$ of the differences. Both $\mu(e)$ and $\sigma^2(e)$ go to zero when solutions converge. These values constitute *indicator graph* with respect to epoch indices. We compared two graphs, one obtained from a probably correct program and another from a bug-injected one. The two show quite different behavior when viewed from these indicators. However, traces of $\mathcal{E}$ showed similar curves.

Because appropriate metamorphic properties are dependent on ML problems, we studied a problem of recognizing hand-written numerical numbers, a standard benchmark of MNIST dataset. Interchanging indices of two data points,

$\boldsymbol{x}_n \rightleftharpoons \boldsymbol{x}_m$, does not change $\mathcal{E}$. Indicator graphs are the same as well. *Additive* property [7], adding a constant to a particular attribute of all data points ($\boldsymbol{x}_i^n \rightharpoonup \boldsymbol{x}_i^n + b$), may have no impact on indicator graphs because the input normalization takes care of the pre-adjustment appropriately. However, *Multiplicative* property [7] ($\boldsymbol{x}_i^n \rightharpoonup a{\times}\boldsymbol{x}_i^n$) may change the shape of indicator graphs in that the convergence becomes slow.

Because indicator graphs represent temporal changes in statistical summaries of weight values, R^T may be considered a statistical approach, but is different from *statistical oracle* [4], which employs statistical hypothesis testing to refute the correctness of probabilistic programs. NN programs employ a notion of probability specifically for efficiency reasons. Statistical oracles are used for systems with inherent randamness.

5 Discussions and Concluding Remarks

Testing is an unknown-unknown task. We do not know whether a program has bugs or not, and thus may conduct testing in search of non-existing bugs. Nevertheless, a new way of testing is important for ML programs.

As machine learning results are sensitive to slight changes in input training dataset, changing distribution of data points in the dataset is a key issue for corner-case testing. This is similar to *Machine Teaching* [8], which is concerned with automatic generation of dataset for a given machine learning tasks. The generated dataset is biased in that ML process converges to given learning parameter values. Namely, the method is concerned with generating *well-biased* critical data points. Our future work includes studying a systematic method for generating such biased critical data points for corner-case testing.

References

1. C.M. Bishop : *Pattern Recognition and Machine Learning*, Springer-Verlag 2006.
2. K. Brinker : Incorporating Diversity in Active Learning with Support Vector Machines, In *Proc. 20th ICML*, 2003.
3. T.Y. Chen, S.C. Chung, and S.M. Yiu : Metamorphic Testing - A New Approach for Generating Next Test Cases, HKUST-CS98-01, The Hong Kong University of Science and Technology, 1998.
4. R. Guderlei, J. Mayer, C. Schneckenburger, and F. Fleischer : Testing Randomized Software by Means of Statistical Hypothesis Tests, In *Proc. SOQUA 2007*, pp.46-54, 2007.
5. S. Nakajima and H.N. Bui : Dataset Coverage for Testing Machine Learning Computer Programs, In *Proc. 23rd APSEC*, pp.297-304, 2016.
6. E.J. Weyuker : On Testing Non-testable Programs, *Computer Journal*, 25 (4), pp.465-470, 1982.
7. X. Xie, J.W.K. Ho, C. Murphy, G. Kaiser, B. Xu, and T.Y. Chen : Testing and Validating Machine Learning Classifiers by Metamorphic Testing, *J. Syst. Softw.*, 84(4), pp.544-558, 2011.
8. X. Zhu : Machine Teaching: An Inverse Problem to Machine Learning and an Approach Toward Optimal Education, In *Proc. 29th AAAI*, pp.4083-4087, 2015.