CS 763: Security and Privacy in Data Science **Fall 2019**

Presentation 3: Data Poisoning

Presenters: Somya Arora & Zi Wang *Scribes: Pierre Petrella & Miru Park*

3.1 How secure are our classifiers?

3.1.1 Introduction

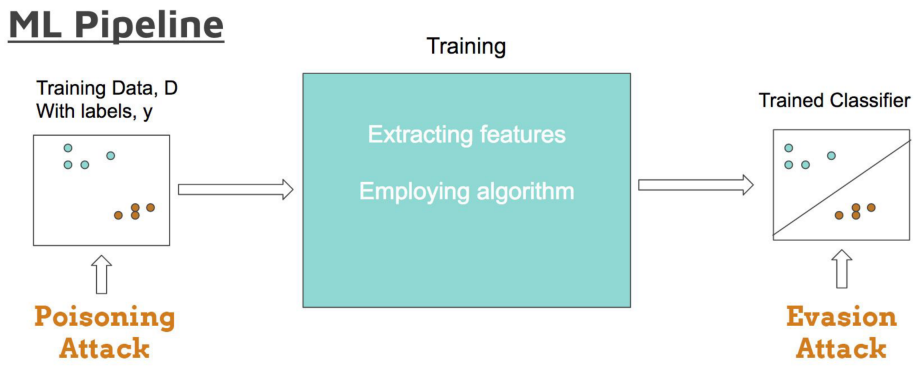
This is a summary based on the presentation given by Zi and Somya. They introduced a method with which supervised machine learning algorithms could be exploited by an"intelligent adversary". The method presented is known as poisoning attack. The name is derived from the fact that the adversary need only craft and inject one or a few malicious training examples to maximize the loss function of the machine learning algorithms in question, leading to higher classification error. The presentation was based on two reasonably recent papers: paper1 and paper2

3.1.2 Potential Threats of ML Pipeline Poisoning

Machine learning systems have become a necessity in this new data era as data analysis on such big data sets can only be managed by automated processes. It is more precisely used for image recognition from distinguishing cats from dogs, red lights from green lights to facial recognition.

Unfortunately these machine learning systems can be compromised. They are now becoming the weakest part of the security chain and consequently of the whole system. If no precautions are taken, this weakness can be used as a weapon by the attackers.

3.1.3 General Attack on a ML Pipeline



3.2 Targeted Clean Labor Poisoning Attacks on Neural Networks

3.2.1 Assumptions

The attacker has:

1. No knowledge of the training data
2. No control over the target instance during test time
3. No control over labelling of data for training
4. Knowledge of the model and it's parameters

These restriction are quite strict on the attacker. This implies that the data poisoning requires minimal intrusion which makes it difficult to detect for the ML model.

3.2.2 Properties

Clean Labels

There are various types of poisoning attacks. In this paper, we are focusing on clean labels as opposed to poisoning that involve tampering with the labels. Clean labels allows to poison a training set with minimal intrusion as the poisoned image can simply be uploaded online and wait to be used by a ML model.

Targeted

This type of attack is built to affect one image specifically and not tamper with the other ones. This allows the poisoning to happen without the Users noticing that the model was tampered with. the Degradation of the model should be unnoticeable.

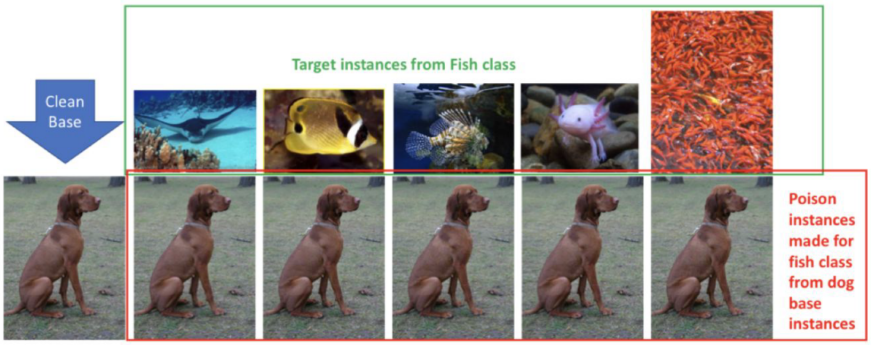
Good Success Rate

If the poisoning is affecting the last layer of the ML model, we can assume that the poisoning will have a success rate of 100% most of the time.

3.2.3 Poisoning Attacks Example

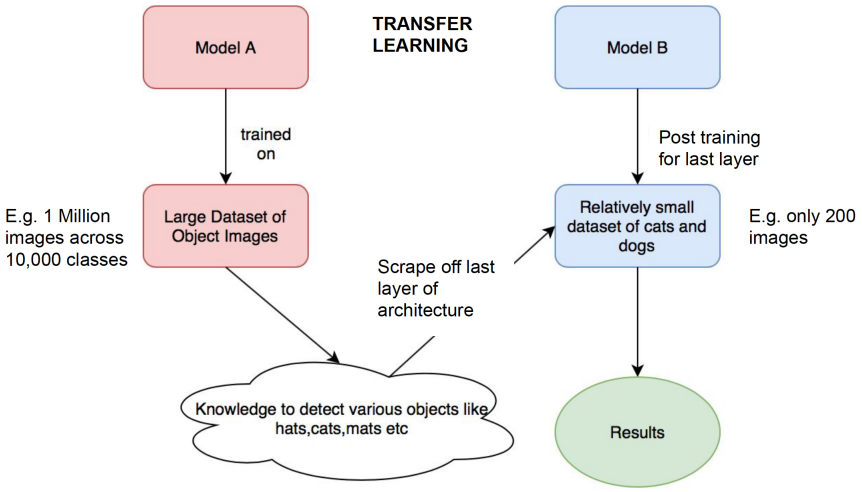
We will be considering the classifier that sorts between fish and dogs. The goal of the attacker will be to chose a specific picture of a fish and trick the ML model to think it is a dog. To do so, the the adversary will insert a poisoned instance of a dog with fish features. This poisoned picture, which looks like a dog will be labeled accordingly and therefore trick the ML model.

Here are examples of poisoned dog images associated with the fish images they are targeting. We can see that the difference between the original and poisoned dog is indistinguishable to the human eye.

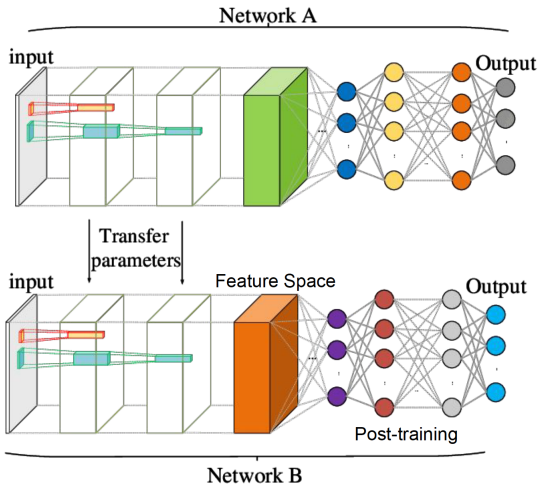


3.2.4 Transfer Learning

On way to train a new ML model is to use an existing working model that has been trained on a larger data set (Model A) and get rid of the last couple of layers. We then reconstruct the last layers using a smaller data set to fine tune the model to distinguish cats and dogs for instance (Model B).



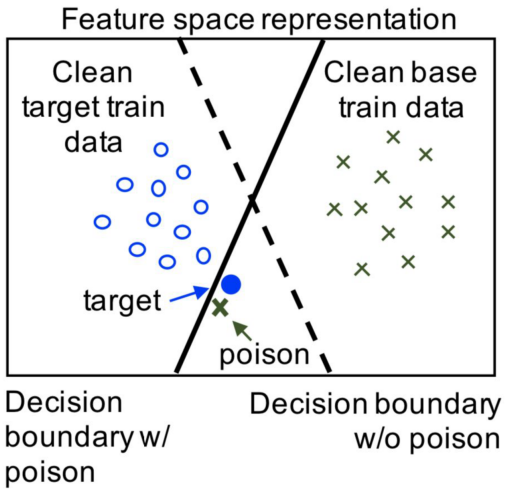
These last layers are very susceptible to change making it an idea target for data poisoning. Transfer learning is an effective way to train a new model without requiring all the resources necessary to train a ML model from scratch. Unfortunately if the data set used to train the last layer has been tampered with classification accuracy can be easily tampered with. Here is another representation of Transfer learning.



- Everything till penultimate layer is borrowed - static knowledge
- Knowledge is used to convert input to feature space : x to f(x)
- Post training is done on a smaller dataset - poison it!
- Attack happens in post training

3.2.5 Finding the Poisoning

The poisoning image must look like a dog but must have all the qualities of it's target (the fish). This will move the decision boundary from the initial doted line to the full line (see diagram). Now any pictures close enough to the targets characteristics will be on the"dog" side of the decision line and therefore be seen as a dog instead of a fish.



In order to create such a poisonous image we must find an image to satisfy this equation with x the image we want to find, t the target, b the initial"dog" image that we will modify and β the importance we give to the similarity between the original image and the poisonous one:

$$\mathbf{p} = \underset{\mathbf{x}}{\operatorname{argmin}} \ \|f(\mathbf{x}) - f(\mathbf{t})\|_2^2 + \beta \|\mathbf{x} - \mathbf{b}\|_2^2$$

Make the poison instance move towards the target instance in feature space

Make the poison appear like a base class instance to a human labeller

3.2.6 Approach

The protocol to follow to poison the data set is the following:

1. Choose a target instance to misclassify
2. Choose a base instance & make imperceptible changes to it to get a poison
3. Inject poison into training data and let model be trained on poisoned dataset

3.2.7 Algorithm

The algorithm to find the image consists of a forward backward iterative splitting procedure to find poison iteratively. This is achieved by iterating these 2 steps:

1. Minimize distance to the target instance in feature space
2. Minimize distance from the base instance in input space

Algorithm 1: Poisoning Example Generation

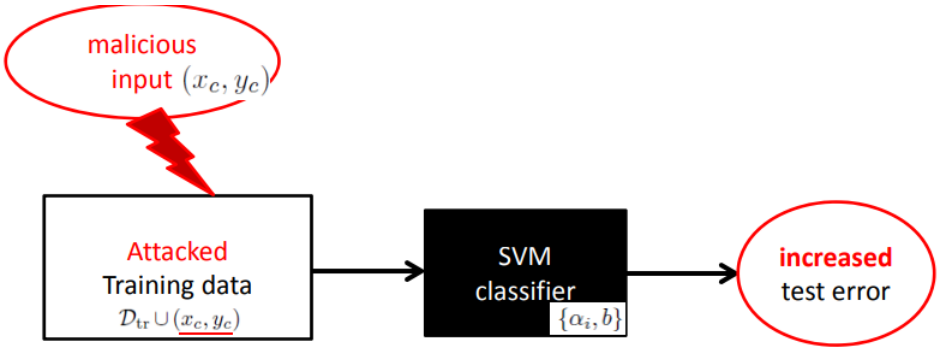
Input: target instance t, base instance b, learning rate λ
Output: final attack point
Initialize x: $x_0 \longleftarrow b$
Define: $L_p(x) = \|f(x) - f(t)\|^2$
for $i = 1$ *to maxIters* **do**
  Forward step: $\hat{x}_i = x_i - 1 - \lambda \nabla_x L_p(x_{i-1})$
  Backward step: $x_i = (\hat{x}_i + \lambda\beta b)/(1 + \beta\lambda)$
end

3.3 Poisoning Attack Against Support Vector Machines(SVM)

3.3.1 Overview of Data Poisoning Against SVM

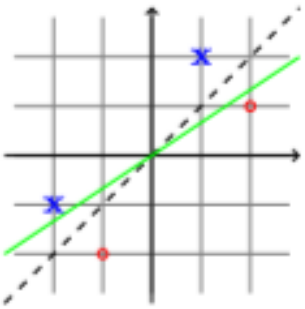
Poisoning attack against Support Vector Machines is an instance of causative attacks. Causative attack is an attack which takes advantage of the common assumption that machine learning algorithms receive well-behaved data. However that is not always the case. An intelligent attacker can tamper with optimal solutions to the Support Vector Machine by injecting a specific and well crafted attack example. Injection of such point into the training data is called Poisoning Attack.

The attacking scheme is rather simple (See picture below). The point (x_c, y_c) is the desired attack point that will enable the attacker to tamper with optimal solutions to the SVM, D_{tr} is the training data, and $\{\alpha_i, b\}$ is the solution to the SVM. As can be seen in the figure, the attack leverages access to the training data. One way to inject some malicious input would be to simply upload the malicious data online or create a set of fake accounts online.



3.3.2 Refresher on Support Vector Machine

We begin our discussion with a brief overview of Support Vector Machine. For simplicity, let us first assume that our data is linearly separable and our goal is to build a classifier (linear) to predict unobserved instances' labels. See below.



The goal of SVM is to find the best decision boundary; in the picture above, we consider the dotted line to be the better than the green one. More precisely, consider the classifiers in Figure 3.1. Our goal is to find the best linear decision boundary (consider a hyperplane) with the maximum margin M. We first discuss how to do this with respect to the case depicted on the left half of the figure (3.1). Suppose that our data consists of N pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$ where $x_i \in R^m$ and $y_i \in \{-1, 1\}$. Define a hyperplane as follows.

$$\{x : f(x) = x^T\beta + \beta_0 = 0\}$$

Since our data is separable, we can compute the following function $f(x) = x^T\beta + \beta_0$, where $y_i f(x_i) \geq 0$. In other words, we can find the hyperplane that gives us the largest margin between the data points for our

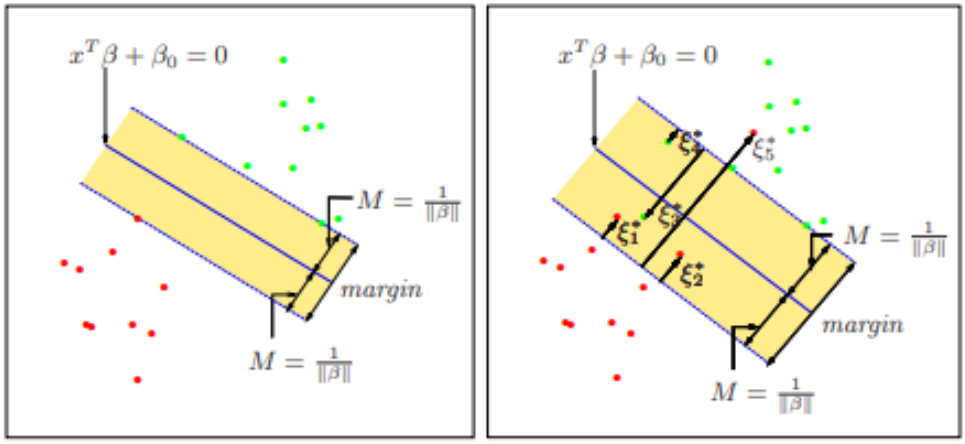


Figure 3.1: Support Vector Machines. The left box shows the case in which data is linearly separable (Hard-Margin). The right box shows the case in which data is not linearly separable (Soft-Margin)

label 1 and -1. More formally, here is the optimization problem that embodies our goal.

$$\begin{aligned} &\min_{\beta,\beta_0,\|\beta\|=1} \|B\| \\ &\text{subject to } y_i(x_i^T\beta + \beta_0) \geq M, i = 1, \ldots, N \end{aligned} \tag{3.1}$$

The above optimization problem is equivalent to:

$$\begin{aligned} &\min \|\beta\| \\ &\text{subject to } y_i(x_i^T\beta + \beta_0) \geq 1, \text{ for } i = 1, \ldots, N \end{aligned} \tag{3.2}$$

Now we extend our idea to the case in which the data may not be perfectly linearly separable as depicted in the right half of figure (3.1). In this case, we can still aim to maximize the margin M but allow for some points to be on the wrong side of the margin. We introduce new *slack* variables $\zeta = (\zeta_1, \ldots, \zeta_N)$. One way to integrate these slack variables into our previous optimization problem 3.2 is:

$$\begin{aligned} &y_i(x_i^T\beta + \beta_0) \geq M(1 - \zeta_i) \\ &\sum_{i=1}^{N} \zeta_i \leq C \end{aligned} \tag{3.3}$$

where C is some constant and ζ_i are non-negative.

The slack variables ζ_i's express how much (x_j, y_j) fails to be separated; for any point (x_i, y_i) that satisfies $y_i(\beta^T x_i) > 0$, the corresponding $\zeta_i = 0$. Now, we would like the total sum of margin violations to be as small as possible and thus will add some penalty term or a penalty constant C to discourage large violations. Combining the above constraints, we have our final optimization problem to solve for optimal solutions to the Support Vector Machine:

$$\begin{aligned} &\min_{\beta,\beta_0} \frac{1}{2}\|\beta\|^2 + C\sum_{i=1}^{N} \zeta_i \\ &\text{subject to } \zeta \geq 0, y_i(x_i^T\beta + \beta_0) \geq 1 - \zeta_i, \text{ for } i = 1, \ldots, N \end{aligned} \tag{3.4}$$

Remark: If the reader wants to actually derive the optimization problem 3.4 it is easier to first take $M = \frac{1}{\|\beta\|}$ and see that the optimization problem 3.2 is equivalent to:

$$\begin{aligned} &\min\|\beta\| \\ \text{subject to } \ &y_i(x_i^T\beta + \beta_0) \geq 1 - \zeta_i, \text{ for } i = 1, \ldots, N \\ &\zeta_i \geq 0, \sum_{i=1}^{N} \zeta_i \leq C \end{aligned}$$

3.3.3 Quadratic Programming (QP)

For some readers, the optimization problem 3.4 may have seemed familiar. The reason is because it belongs to a special and well known family of optimization problems known as quadratic (convex) programming (QP). In other words, we are trying to minimize a convex function subject to linear inequality constraints. Before we solve our QP-problem, first we define what a standard QP-Problem is and then we introduce a very powerful theorem.

Definition 3.1. *A standard QP-Problem is the following:*

$$\begin{aligned} &\min_{u \in R^L} \frac{1}{2} u^T Q u + p^T u \\ &\textit{subject to } a^T u \geq c \end{aligned}$$

Theorem 3.2. *For a feasible convex QP-problem in primal form,*

$$\begin{aligned} &\min_{u \in R^L} \frac{1}{2} u^T Q u + p^T u \\ &\textit{subject to } a_m^T u \geq c_m, \textit{ for } m = 1, \ldots, N \end{aligned}$$

define the Lagrange function

$$L(u, \alpha) = \frac{1}{2} u^T Q u + p^T u + \sum_{m=1}^{M} \alpha_m (c_m - a_m^T u)$$

The solution μ^* is optimal for the primal if and only if (u^*, α^*) is a solution to the dual optimization problem

$$\max_{\alpha \geq 0} \min_{u} L(u, \alpha)$$

The optimal (u^*, α^*) satisfies the Karush-Kuhn-Tucker (KKT) conditions:

$$a_m^T u^* \geq c_m, \alpha_m \geq 0 \tag{3.5}$$

$$\alpha_m^* (a_m^T u^* - c_m) = 0 \tag{3.6}$$

$$\nabla_u L(u^*, \alpha^*) = 0 \tag{3.7}$$

With this theorem, we can now solve our original optimization problem 3.4. Let μ be the Lagrange multiplier for ζ. Given the primal problem 3.4, the Lagrange function is

$$L_p(\beta_0, \beta, \zeta, \alpha, \mu) = \frac{1}{2}\|\beta\|^2 + C \sum_{i=1}^{N} \zeta_i - \sum_{i=1}^{N} \alpha_i [y_i(x_i^T \beta + \beta_0) - (1 - \zeta_i)] - \sum_{i=1}^{N} \mu_i \zeta_i \tag{3.8}$$

where $\alpha_i \geq 0$ are the Lagrange multipliers for $y_i(\beta^T x_i + \beta_0) \geq 1 - \zeta_i$ and $\mu_i \geq 0$ are the Lagrange multipliers for $\zeta_i \geq 0$. The third KKT condition 3.7 of the theorem 3.2 also tells that for optimal solutions, $\frac{\partial_L}{\partial_{\zeta_i}} = 0$. This means nothing but

$$C - \alpha_i - \mu_i = 0$$

which gives us μ in terms of C and α. This allows us to simplify our Lagrange dual problem to be

$$\max_{\alpha,\mu \geq 0} \min_{\beta_0,\beta,\zeta} L(\beta_0, \beta, \zeta, \alpha)$$

where L is:

$$L = \frac{1}{2}\|\beta\| + C\sum_{i=1}^{N}\zeta_i + \sum_{i=1}^{N}\alpha_i(1 - \zeta_i - y_i(\beta^T x_i + \beta_0)) - \sum_{i=1}^{N}(C - \alpha_i)\zeta_i = \frac{1}{2}\|\beta\| + \sum_{i=1}^{N}(1 - y_i(\beta^T x_i + \beta_0)) \quad (3.9)$$

L of the form 3.9 is called the Lagrange Dual problem. As the theorem suggests we first minimize L with respect to β and β_0 then we maximize with respect to α.

$$\frac{\partial L}{\partial \beta_0} = -\sum_{i=1}^{N}\alpha_i y_i$$
$$\frac{\partial L}{\partial \beta} = \beta - \sum_{i=1}^{N}\alpha_i y_i x_i$$

Setting these to 0 we get

$$\begin{aligned} 0 &= \sum_{i=1}^{N}\alpha_i y_i \\ \beta &= \sum_{i=1}^{N}\alpha_i y_i x_i \end{aligned} \quad (3.10)$$

Again, we use the third KKT condition 3.7 of Theorem 3.2 and plug them back into the Lagrangian given at 3.9 to get

$$L(\alpha) = -\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j \alpha_i \alpha_j x_i^T x_j + \sum_{i=1}^{N}\alpha_i \quad (3.11)$$

Finally, we have the final portion of the Lagrange dual problem.

$$\begin{aligned} &\max_{\alpha \geq 0} L(\alpha) \\ &\text{subject to} \sum_{i=1}^{N} y_i \alpha_i = 0 \end{aligned} \quad (3.12)$$

However, there is one problem. Since we will not solve this QP by hand (a keen reader may try this), we need our QP to be in a specific format so that a standard QP-solver program may solve it (many programming languages including Python, MATLAB, and Julia have their own QP-solvers available). This only requires that our problem is a minimization problem and not a maximization problem. This is an easy fix as can be seen next.

$$\begin{aligned} &\min_{\alpha \geq 0} \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} y_i y_j \alpha_i \alpha j x_i^T x_j - \sum_{i=1}^{N}\alpha_i \\ &\text{subject to} \sum_{i=1}^{N} y_i \alpha_i = 0 \end{aligned} \quad (3.13)$$

Note that 3.13 is the same as 3.12.

Finally, with the construction of appropriate Q_D, A_D we have the following standard QP-Problem that is equivalent to 3.13. Computation of Q_D and A_D are left to the readers as an exercise.

$$\begin{aligned} &\min_{\alpha} \frac{1}{2}\alpha^T Q_D \alpha - 1_N^T \alpha \\ &\text{subject to } A_D \alpha \geq 0_{N+2} \end{aligned} \tag{3.14}$$

Let us now suppose we have solved 3.14. Let α^* be the optimal α. We can compute the optimal β by substituting into 3.10. Thus the optimal weights β^* are:

$$\beta^* = \sum_{i=1}^{N} \alpha_i^* y_i x_i$$

At this point, you may notice that we still have not computed β_0^*. This can be easily computed; this is given by the second KKT condition 3.6 of theorem 3.2. Thus, we have

$$y_s(\beta^{*T} x_s + \beta_0^*) = 1$$

where the subscript s stands for support vectors i.e. $\alpha_s^* > 0$ and this gives us

$$\beta_0^* = y_s - \sum_{i=1}^{N} y_i \alpha_i^* x_n^T x_s$$

3.3.4 Poisoning Attack

As mentioned in our earliest discussion of poisoning attacks, an intelligent adversary can construct some malicious data point which will only need to be injected to the training data set at training time of the model. Consider this scenario. Suppose that our Support Vector Machine is used for classifying computer virus or malicious code. One way to collect training data for our algorithm would be to purposefully expose our machine to spam emails or a specific website that transfers either of the aforementioned items. If the attacker knows this, he or she may intentionally place a well crafted piece of malicious code onto the website that will only need to wait until it is collected at data collection time.

Before we begin our discussion of poisoning attack, we assume that the attacker knows:

1. The learning algorithm
2. The original training data

Let D_{tr} be training data set and $D_{val} = \{x_k, y_k\}_{k=1}^m$ be validation data set. Attacker's goal is to find a point (x_c, y_c) such that the hinge loss $\sum_{k=1}^m (1 - y_c f_{x_c}(x_c))_+ = L(x_c)$ is maximized on D_{val} by Support Vector Machine trained on $D_{tr} \cup (x_c, y_c)$. Generally, $L(x_c)$ is a non-convex function and gradient ascent algorithm can be used to identify a reasonably good local maxima to increase our support vector machine's test error. The general idea of gradient ascent algorithm (with respect to our goal) is to train our SVM on a prescribed training set, update our attacking point (x_c) with an upward gradient $\nabla L(x_c)$, and then to add the updated attacking point to our training set and repeat this process until we see "convergence" (we will see what this means in the next section but for those who are familiar with the term, we essentially want to meet the Cauchy-convergence criteria). Here is the updating procedure, where t is some arbitrary step size.

$$x_c^i = x_c^{i-1} + t\nabla L(x_c)$$

The complete derivation of the gradient $\nabla L(x_c)$ is beyond the scope of this summary. However, we give a high-level idea of the derivation process. Nonetheless, the full derivation process can be seen in the original paper.

As can be seen in the iteration step of the gradient ascent algorithm shown above, the most important ingredient that will help us find the final attack point (after completion of the gradient ascent algorithm) is $\nabla L(x_c) = u$ (for convenience, let us now call this u). Thus, our problem now can be considered as: find u that will maximize $L(x_c)$. Let some validation set $D_{val} = \{x_k, y_k\}_{k=1}^m$ be given. Define

$$L(x_c) = \sum_{k=1}^{m}(1 - y_c f_{x_c}(x_c))_+ = \sum_{k=1}^{m}(-g_k)_+$$

Then we have

$$g_k = \sum_{j \neq c} Q_{kj}\alpha_j(x_c) + Q_{kc}(x_c)\alpha_c(x_c) + y_k\beta_0(x_c) - 1$$

where $Q_{ab} = y_a y_b x_a^T x_b$ as suggested in 3.14. As expected, in order to maximize our hinge loss function we differentiate g_k with respect to our gradient u and set it equal to zero. Using the product rule, this gives us

$$\frac{\partial g_k}{\partial u} = Q_{ks}\frac{\partial \alpha}{\partial u} + \frac{\partial Q_{kc}}{\partial u}\alpha_c + y_k\frac{\partial \beta_0}{\partial u} = 0 \tag{3.15}$$

The equation 3.15 may look overwhelming mainly because we may not know $\frac{\partial \alpha}{\partial u}$. However, this need not be true if we recall theorem 3.2 we introduced (and the KKT conditions 3.5 - 3.7) in the previous section at the page 3-8 and borrow some ideas from Gert Cauwenberghs and Tomaso Poggio. Essentially, Cauwenberghs' and Poggio's paper explores the idea of efficient "online learning" of SVM. In other words, they show how to learn an SVM "incrementally" with $(k+1)$ training examples given that it was previously trained with only k examples. This is very much the same goal we would like to achieve, except the new point to be added is an attacking point after each iteration.

Lastly, we make note of the most important aspect of their observation on this method of learning with respect to our goal: the preservation of KKT conditions on all previously seen training examples, while adding a new example to the solution. As can be seen from theorem 3.2, this is just saying that adding small changes to our attacking point x_c (i.e. taking a "step" in the direction of u) should "maintain" the optimal SVM solution with respect to previously seen training data (or very infinitesimal change).

In summary, Cauwenberghs' and Poggio's work allows us to predict how the solution to SVM should "change" with respect to changes in our attacking point x_c (thus giving us insight on how to compute u) and this allows for tractable computation of each partial derivative involved in equation 3.15. Computation of the α's and u's are crucial in the scheme of poisoning attack.

3.3.5 The Algorithm

In this section, we introduce the algorithm to compute the attacking point. As will be shown, the highlight of the algorithm is really the computation of the"poisonous" gradient $u = \nabla L(x_c)$, which will cause the loss

function to increase (hence the name "gradient ascent").

Algorithm 2: Find attack point

Input: Training Data (D_{tr}), validation data, attack point label, initial attack point (x_c^0, y_c), step size (t)
Output: final attack point

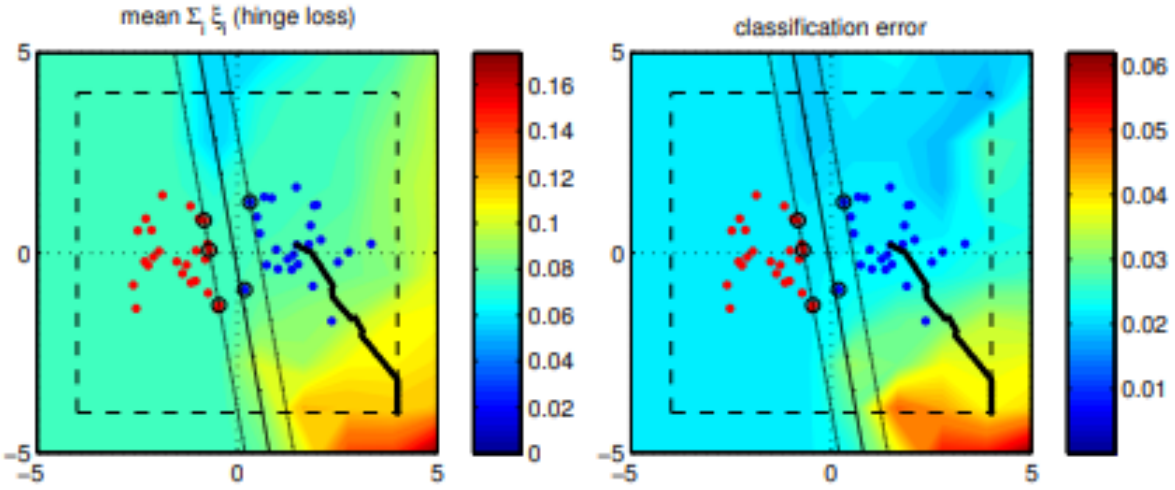
1. $\{\alpha, \beta_0\} \longleftarrow$ solve SVM on D_{tr}
2. pick initial attacking point x_c^0
3. $p \longleftarrow 0$

while $L(x_c^p) - L(x_c^{p-1}) > \epsilon$ **do**
  Recompute $\{\alpha, \beta_0\}$ on $D_{tr} \cup \{x_c^p, y_c\}$
  Compute $\frac{\partial L}{\partial u}$ on D_{val}.
  Align u in the direction of $\frac{\partial L}{\partial u}$ and normalize u
  $x_c^{p+1} \longleftarrow x_c^p + tu$
end
return: $x_c = x_c^p$

With respect to the algorithm above, please keep the following in mind.

1. Adversary knows the training data used by the learner.
2. initial attack point is picked from a region sufficiently deep within the attacking class's margin.
3. $L(x_c^j)$ is the loss function with respect to the attacking point at each iteration, $j = 1, \ldots, m$.

Here is the trajectory of the attacking point as the algorithm proceeds. (This experiment was done on artificial data)



- Heatmap background is the error surface
- Dashed lines (the inner square) is the range of attack points

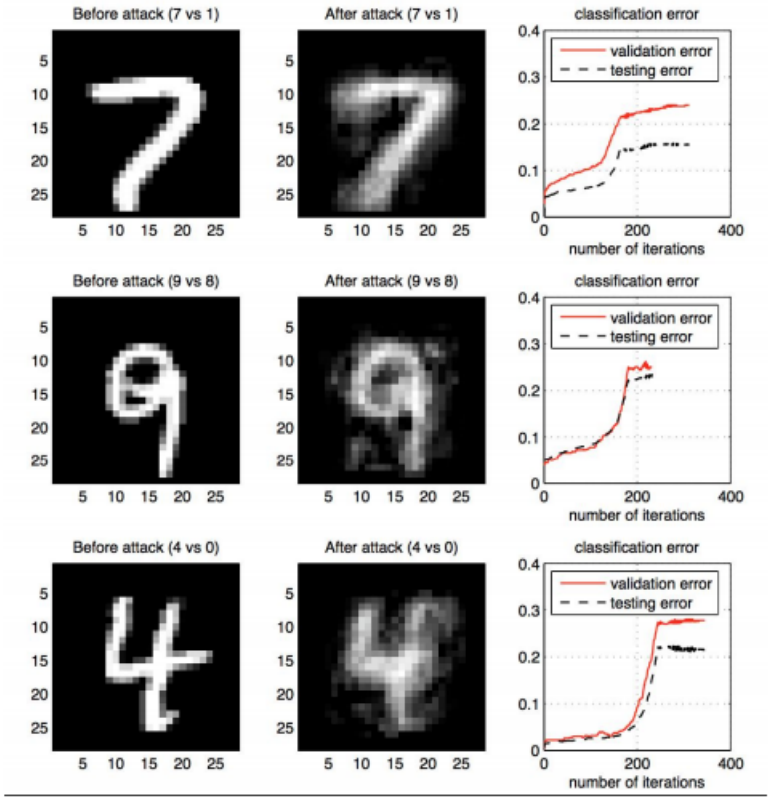
- Thick black line is the trajectory of the attack point toward the local maximum
- Normal black line is the decision boundary
- Mini black circles are the support vectors

3.3.6 Final Result and Evaluation

We end our discussion with some empirical data on the effectiveness of poisoning attack against SVM. Below is the summary of the results of the experiment done on the MNIST dataset. As can be seen, the experiment was done on three pairs of digits:

1. 7 vs 1
2. 9 vs 8
3. 4 vs 0

There is a stark contrast between the digits before the attack and after the attack as can be seen in the first two columns of the figure. The last column of the figure shows the increasing error over the course of the attack.



3.4 References

B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In International Conference on Machine Learning (ICML), pages 1467–1474, 2012.

T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction

S. Ben-David and S.Shalev-Schwartz. Understanding Machine Learning: From Theory to Algorithm

Wu, Cheng-Ju(2017) Poisoning Attacks Against Support Vector Machines[PowerPoint slide 14]. Retrieved from https://people.eecs.berkeley.edu/ roydong/fa17$_$files/EECS290O$_$IEOR290-Student$_$Presentations-Wu-01.pdf

G. Cauwenberghs and T.Poggio. Incremental and Decremental Support Vector Machine Learning