A Tutorial of Adversarial Learning

Han Xiao

Department of Informatics
Technical University of Munich
Garching, D-85748
xiaoh@in.tum.de

Abstract. Just for fun, far from complete.

1 Problem Definition

We define \( x \in X \) as an instance, where \( X \) is the instance space. \( x \) is represented by a vector variable with \( n \) dimensions, namely \( x = (x_1, \ldots, x_n) \). \( x_i \) denotes the \( i \)th feature in instance \( x \). Each instance can belong to one of two classes: positive (malicious) or negative (innocent), which are denoted by \( x^+ \) and \( x^- \) respectively. Let training set \( S \subset X \) and test set \( T \subset X \) consist of both positive and negative instances. In practice, \( S \) is usually a finite set, whereas \( T = X \setminus S \) has infinite size.

We call a function \( C : X \mapsto \{-1, 1\} \) as a Boolean classifier, or a classifier for short. We refer to \( x^+ \) for which \( C(x) = 1 \) and \( x^- \) for which \( C(x) = -1 \). The task of a classifier is to learn from \( S \) a function \( C(x) \) that will correctly predict new instance \( x \in T \). Obviously, a well-performed classifier can secure the system by detecting malicious instances in advance (e.g. spam filtering and virus detection).

An adversary attempts to defeat the system by sending malicious instances without being detected. In fact, adversaries are actively disguising their behavior to evade detection. For instance, senders of junk email often add “good” words or sentences to cheat the spam filter for decreasing the likelihood of detection. Although these disguised instances are more indicative of innocent than malicious, they may also decrease the reward of adversary due to their ineffectiveness.

For adversary, some instances are more effective than others. We explain such differences on utility by an adversarial cost function \( A(x) : X \mapsto \mathbb{R}^+ \). Note, that \( A(x) \) is domain-dependent function. We assume that adversaries have a base instance \( x^a \) for which \( C(x^a) = 1 \) on hand. To evade detection, they are interested in finding an instance \( x^* \) that most similar to \( x^a \) but will be classified as \( x^- \). To measure the similarity between two instances, we first define an adversarial cost function as

\[
A(x) = \sum_{i=1}^{n} a_i |x_i - x^a_i|,
\]

where positive scalars \( a_i \) represent the relative cost of changing each feature, allowing that some features may be more important than others (from adversaries’ perspective of view). An illustrative example is depicted in Figure 1.

\( a_i \) is under an assumption that \( x^a \) is the best instance as far as the adversary knows. That is, any changes to \( x^a \) costs an utility loss.
Finally, the task of finding $x^*$ is formulated as

$$x^* = \arg \min_{x: C(x) = -1} A(x).$$

### 2 Actions of Classifier and Adversary

We first categorize adversaries by their attack.

**Reverse engineering** The adversary attempts to fully estimate the classifier $C(x)$’s decision boundary.

**Evasion** The adversary attempts to discover blind spots by querying a fixed or learning-based detector to find a low cost instance $x^*$ that the classifier does not filter.

Fully estimating the decision boundary is hard, especially in general convex [4]. Thus, we first restrict our discussion to the evasion problem. That is, the adversary directly searches for $x^*$ by sending test instances repetitively. A classifier returns the result of $C(x)$ for each instance $x$ it receives. In last section, we restrict ourselves to the binary result. More commonly, the type of result is summarized as follows:

**Nominal** $C(x) \in \mathbb{Z}_n$. The set $\mathbb{Z}_n$ is the finite set of integers modulo $n$. The result records a response as a set of categories that have no natural order to them. When $n$ is set to one, we have a binary result.

**Ordinal** $C(x) \in \mathbb{Z}_n$. For instance, a five-point rating scale $\mathbb{Z}_5$ can only take on the values 0, 1, 2, 3, 4, 5 where 5 is the highest score.

**Continuous** $C(x) \in \mathbb{R}$. For instance, in a document retrieving system the result indicates the relevance score of the given instance.

We assume the classifier can observe the number of instances in a certain time interval. Therefore, a salient arise of received instances can be immediately identified. This is not an onerous assumption in practice. A classifier can be categorized into two types by its defense against attack.
Passive The classifier makes itself temporally off-line to protect the system.
Active The classifier adapts the function $C(x)$ by considering $x^*$ as positive instance.

In some domains, being passively off-line can not bring benefit to the adversary (e.g. spam). Therefore, an adversary should submit test instances as few as possible to avoid the attack being noticed. To numerically measure the efficiency of an adversarial strategy, we count the number of test instances required to find $x^*$. Furthermore, an adversary is more effective if it yields smaller value of $A(x^*)$ than others.

The active classifier with a preemptively strategy has been studied in [2]. They construct a modified classifier designed to detect optimally modified instances, and use a cost-sensitive game theoretic approach to preemptively patch a classifier’s blind spots.

3 Evading the Linear Classifier

Linear classification has become one of the most promising learning techniques for large sparse data with a huge number of instances and features. A linear classifier generates a weight vector $w$ as the model. The decision function is

$$C(x) = \text{sgn}(w^T x).$$

In this section, we demonstrate an algorithm for evading linear classifier under adversarial cost function described in Section 1.

We begin with the case in which all features are continuous. Figure 2 shows an illustrative example of the target linear classifier. Intuitively, we’d like to search $x^*$ in a direction such that we can quickly reach the boundary (large $|w_i|$) with minimum cost (small $a_i$). Thus, we define a feature $f$ with highest weight-to-cost ratio as

$$f = \arg \max_{i \in \{1, \ldots, n\}} \frac{|w_i|}{a_i}.$$  \hspace{1cm} (1)

It can be shown that, in a continuous linear classifier, we can arrive at $x^*$ by changing only feature $f$ in $x^a$. Let $\hat{f}$ be the unit vector along dimension $f$, the instance of minimal adversarial cost is given by

$$x^* = x^a + t\hat{f},$$  \hspace{1cm} (2)

where $t$ is the step length.

3.1 IMAC Algorithm

We have shown that $x^*$ can be easily found by changing the value of $x_f$. By observing (1), we need to first approximate the value (or the relative value) of $w$ in order to determine $f$. Authors in [3] proposed an efficient algorithm for determining $f$ and finding Instance with Minimum Adversarial Cost (IMAC) $x^*$. Given one positive instance $x^+$ and a negative instance $x^-$, the complete process is stated as follows:

1. Finding a pair of instances $s^+, s^-$ such that $\exists i \forall j \neq i, s^+_j = s^-_j$ using $x^+$ and $x^-$. 

Fig. 2. Searching $x^*$ (denoted by a star) in a 2-dim. space with a linear decision boundary. The shaded area demonstrates the positive response of the classifier. Arrow represents the optimal searching direction. **(left)** The optimal searching direction is along $x_2$. **(right)** The optimal searching direction is along $x_1$.

2. Assessing the sign of $w_i$ and letting $w_i = \text{sgn}(w_i)$.
3. Searching a negative instance close to the linear decision boundary.
4. Searching in every other dimension $j \neq i$ and computing the relative value of weight $w_j$.
5. Computing $f$ by (1) and finding $x^*$ by (2).

To intuitively demonstrate each step of the algorithm, we depict an illustrative example on a 2-dimensional space in Figure 3. For a linear classifier with continuous features under linear cost functions, this algorithm requires at most polynomially many test queries for each step [3]. With a learned weight on hand, the adversary is now able to compute $f$ by (1) and find optimal $x^*$ by (2) using simple line search technique.

It is worth to highlight several points here. First of all, the motivation for doing the particular algorithm for continuous weights is that it gives a **guarantee** on the adversary’s relative cost. Secondly, the aim of this algorithm is not not to reverse engineer the entire classifier, but merely to find an instance of minimal adversarial cost. Finally, the algorithm is lite-weight in computation, which makes it easy to scale on high-dimensional data.

One may consider to build an approximated linear classifier by querying random test instances. However, this approach does not consider the fact that, the data space is biased in practice. That is, if one just pick random points in the space, then all of them might end up being positive, or all of them might end up being negative. In this case, the resulting test instances would be fairly uninformative about what the feature to choose, since the weights would be largely unknown.
(a) Initializing $x^+$ and $x^-$.  
(b) Searching $s^+$, $s^-$ to assess the sign of $w_1$.  
(c) Searching a negative instance close to the boundary.  
(d) Moving one unit away from the boundary.  
(e) Searching in another dimension.  
(f) Approximating $w$ via tangent rule.

Fig. 3. Algorithm proposed in [3] for approximating $w$.  
(a) We assume the adversary has a positive instance $x^+$ (denoted by a triangle) and a negative instance (denoted by a square) on hand but has no idea about the linear decision boundary. The shaded area represents the positive response of a linear classifier.  
(b) The algorithm starts with $x^+$ and changes feature values one at a time to match those of $x^-$. At some point, the class of instance must change. The previous value and the current value of intermediate instance are set to $s^-$ and $s^+$, respectively. This step requires at most $n$ test queries.  
(c) As $\forall j \neq i$, $s^+_j = s^+_j$, we perform binary search along the dimension $i$ to find a negative instance close to the decision boundary. Let $\epsilon$ be an approximation threshold, this step requires $O(\log(1/\epsilon + |s^+_i - s^-_i|))$ test queries. The dotted line represents the line search operation.  
(d) We set $w_i$ to $1$ or $-1$, and increase or decrease $x_i$ by $1$ to obtain a negative instance.  
(e) The algorithm proceeds by searching in every other directions $j \neq i$ using a line search. This consists of increasing or decreasing each $x_j$ exponentially until the class of $x$ changes, then bounding its exact value using a binary search.  
(f) We finally compute the approximated $w$ using tangent rule. The dashed line shows the learned weight, which is close enough to the ground-truth. The adversary is now able to compute $f$ by (1) and find optimal $x^*$ by (2).

3.2 Experimental results

We implement IMAC algorithm and evaluate its performance in terms of a) the number of queries for approximating the weight; b) the adversarial cost of $x^*$ given $x^n$. The number of queries for finding $x^*$ is not in our evaluation as it depends on the data. Our target is a linear classifier with the form of $C(x) = wx - T$, where $w$ and $T$ is randomly
generated. Note that the adversary does not have access to the original training data, but must come up with all of the queries itself. Thus, we use only a positive and a negative instance as the input of IMAC. To keep the setup as realistic as possible, we assume that the adversary guesses a feature set 10 times larger than $C(x)$ virtually used. The size of feature set varies from 1000 to 10000, each experiment is repeated for 10 times. Figure 4 shows the experimental result.

![chart](image.png)

**Fig. 4.** Result of IMAC algorithm on a random generated linear classifier. (a) Adversarial cost measure the distance between $x^*$ and $x^a$. Lower cost is better. (b) The number of queries used for determining the weights. Bars filled with black indicates the number of positive queries. In practice, it’s important to not only keep the number of total queries down, but also use as few positive queries as possible.

From Figure 4, we can observe that the performance of IMAC algorithm is stable with respect to the number of features. The algorithm does quite well at finding low cost instances. In terms of queries, the algorithm is reasonably efficient. Not surprisingly, fewer queries were required to sort through the smaller feature sets. Note, that in practice the adversary attempt to keep down not only the number of total queries but also the number of positive queries. The salient rising of positive queries may increase the likelihood of being detected.

For comparing with the naive “rebuilding” approach, we also train a perceptron linear classifier by randomly picking data points in space. The corresponding class labels are determined by querying $C(x)$. The training data also includes at least one positive instance: $x^a$. Unfortunately, when dimensions of feature increase, the perceptron fails to approximate the target classifier frequently due to most of the random training instances fall into a negative class. In fact, the perceptron is unable to find any optimal instance when feature dimensions is greater than 500.
Though IMAC algorithm performs well on evading linear classifier, one has to be careful when applying it to the classifier with a nonlinear decision boundary. An illustrative example of a common failure with nonlinear decision boundary is illustrated in Figure 5. In the next section, we introduce a particular algorithm for evading naive Bayes classifier.

Fig. 5. Two situations of a convex classifier. The shaded area denotes the positive response of classifier. (a) For convex positive, one can still find the optimal instance by changing one feature only. In this case, changing $x_1$ will lead to the optimum. (b) For convex negative, IMAC algorithm is not able to search the optimal cost instance. The approximate weights suggest that the optimal searching direction is along $x_2$. However, neither searching along $x_1$ nor $x_2$ is the wise direction in this case.

To be continue...

References