Introduction to Machine Learning

Kyunghyun Cho

Courant Institute of Mathematical Sciences and
Center for Data Science,
New York University

December 29, 2016

Abstract

This is a lecture note for the course CSCI-UA.0473-001 (Intro to Machine Learning) at the Department of Computer Science, Courant Institute of Mathematical Sciences at New York University. The content of the lecture note was selected to fit a single 12-week-long course (3 hours a week) and to mainly serve undergraduate students majoring in computer science. Many existing materials in machine learning therefore had to be omitted.

For a more complete coverage of machine learning (with math!), the following text books are recommended in addition to this lecture note:

- "Pattern Recognition and Machine Learning" by Chris Bishop [1]
- "Machine Learning: a probabilistic perspective" by Kevin Murphy [2]
- "A Course in Machine Learning" by Hal Daumé[1]

For practical exercises, Python scripts based on numpy and scipy are available at `https://github.com/nyu-dl/Intro_to_ML_Lecture_Note/tree/master/notebook`. They are under heavy development and subject to frequent changes over the course. I recommend you to check back frequently. Again, these are not exhaustive, and for a more complete coverage on machine learning practice, I recommend the following book:

- "Introduction to Machine Learning with Python" by Andreas Müller and Sarah Guido

Note that those sections marked with $\star$ are optional and will only be covered should time permits.

[1] Available at `http://ciml.info/`.

Notations

Throughout this lecture note, I will use the following notational conventions:

- A bold-faced lower-case alphabet is used for a vector: $\mathbf{x}$
- A bold-faced upper-case alphabet is used for a matrix: $\mathbf{W}$
- A lower-case alphabet is often used for a scalar: x, η
- A lower-case alphabet is also used for denoting a function
-

Chapter 1

Classification

1.1 Supervised Learning

In supervised learning, our goal is to build or find a machine M that takes as input a multi-dimensional vector $\mathbf{x} \in \mathbb{R}^d$ and outputs a response vector $\mathbf{y} \in \mathbb{R}^{d'}$. That is,

$$M : \mathbb{R}^d \to \mathbb{R}^{d'}.$$

Of course this cannot be done out of blue, and we first assume that there exists a reference design M^* of such a machine. We then refine our goal as to build or find a machine M that imitates the reference machine M^* as closely as possible. In other words, we want to make sure that for any given $\mathbf{x}$, the outputs of M and M^* coincide, i.e.,

$$M(\mathbf{x}) = M^*(\mathbf{x}), \text{ for all } \mathbf{x} \in \mathbb{R}^d. \tag{1.1}$$

This is still not enough for us to find M, because there are infinitely many possible M's through which we must search. We must hence decide on our *hypothesis set* H of potential machines. This is an important decision, as it directly influences the difficulty in finding such a machine. When your hypothesis set is not constructed well, there may not be a machine that satisfies the criterion above.

We can state the same thing in a slightly different way. First, let us assume a function D that takes as input the output of M^*, a machine M and an input vector $\mathbf{x}$, and returns how much they differ from each other, i.e.,

$$D : \mathbb{R}^{d'} \times H \times \mathbb{R}^{d'} \to \mathbb{R}_+,$$

where $\mathbb{R}_+$ is a set of non-negative real numbers. As usual in our everyday life, the smaller the output of D the more similar the outputs of M and M^*. An example of such a function would be

$$D(y, M, \mathbf{x}) = \begin{cases} 0, & \text{if } y = M(\mathbf{x}) \\ 1, & \text{otherwise} \end{cases}.$$

It is certainly possible to tailor this distance function, or a *per-example cost* function, for a specific target task. For instance, consider an intrusion detection system M which takes as input a video frame of a store front and returns a binary indicator, instead of a real number, whether there is a thief in front of the store (0: no and 1: yes). When there is no thief ($M^*(\mathbf{x}) = 0$), it does not cost you anything when M agrees with M^*, but you must pay \$10 for security dispatch if M predicted 1. When there is a thief in front of your store ($M^*(\mathbf{x}) = 1$), you will lose \$100 if the alarm fails to detect the thief ($M(\mathbf{x}) = 0$) but will not lose any if the alarm went off. In this case, we may define the per-example cost function as

$$D(y, M, \mathbf{x}) = \begin{cases} 0, & \text{if } y = M(\mathbf{x}) \\ -10, & \text{if } y = 0 \text{ and } M(\mathbf{x}) = 1 \\ -100, & \text{if } y = 1 \text{ and } M(\mathbf{x}) = 0 \end{cases}.$$

Note that this distance is asymmetric.

Given a distance function D, we can now state the supervised learning problem as finding a machine M, with in a given hypothesis set H, that minimizes its distance from the reference machine M^* for any given input. That is,

$$\arg\min_{M \in H} \int_{\mathbb{R}^d} D(M^*(\mathbf{x}), M, \mathbf{x}) \mathrm{d}\mathbf{x}. \tag{1.2}$$

You may have noticed that these two conditions in Eqs. (1.1)–(1.2) are not equivalent. If a machine M satisfies the first condition, the second conditions is naturally satisfied. The other way around however does not necessarily hold. Even then, we prefer the second condition as our ultimate goal to satisfy in machine learning. This is because we often cannot guarantee that M^* is included in the hypothesis set H. The first condition simple becomes impossible to satisfy when $M^* \notin H$, but the second condition gets us a machine M that is *close* enough to the reference machine M^*. We prefer to have a suboptimal solution rather than having no solution.

The formulation in Eq. (1.2) is however not satisfactory. Why? Because not every point $\mathbf{x}$ in the input space $\mathbb{R}^d$ is born equal. Let us consider the previous example of a video-based intrusion detection system again. Because the camera will be installed in a fixed location pointing toward the store front, video frames will generally look similar to each other, and will only form a very small subset of all possible video frames, unless some exotic event happens. In this case, we would only care whether our alarm M works well for those frames showing the store front and people entering or leaving the store. Whether the distance between the reference machine and my alarm is small for a video frame showing the earth from the outer space would not matter at all.

And, here comes probability. We will denote by $p_X(\mathbf{x})$ the probability (density) assigned to the input $\mathbf{x}$ under the distribution X, and assume that this probability reflects how likely the input $\mathbf{x}$ is. We want to emphasize the impact of the distance D on likely inputs (high $p_X(\mathbf{x})$) while ignoring the impact on unlikely inputs (low $p_X(\mathbf{x})$). In other words, we weight each per-example cost with the probability of the corresponding example. Then the problem of supervised learning becomes

$$\arg\min_{M \in H} \int_{\mathbb{R}^d} p_X(\mathbf{x}) D(M^*(\mathbf{x}), M, \mathbf{x}) \mathrm{d}\mathbf{x} = \arg\min_{M \in H} \mathbb{E}_{\mathbf{x} \sim X} \left[D(M^*(\mathbf{x}), M, \mathbf{x}) \right]. \tag{1.3}$$

Are we done yet? No, we still need to consider one more hidden cost in order to make the description of supervised learning more complete. This hidden cost comes from the operational cost, or *complexity*, of each machine M in the hypothesis set H. It is reasonable to think that some machines are cheaper or more desirable to use than some others are. Let us denote this cost of a machine by $C(M)$, where $C : H \to \mathbb{R}_+$. Our goal is now slightly more complicated in that we want to find a machine that minimizes both the cost in Eq. (1.3) and its operational cost. So, at the end, we get

$$\arg\min_{M \in H} \underbrace{\mathbb{E}_{\mathbf{x} \sim X} \left[D(M^*(\mathbf{x}), M, \mathbf{x}) \right] + \lambda C(M)}_{\text{Expected Cost}}, \tag{1.4}$$

where $\lambda \in \mathbb{R}_+$ is a coefficient that trades off the importance between the expected distance (between M^* and M) and the operational cost of M.

In summary, supervised learning is a problem of finding a machine M such that has bot the low expectation of the distance between the outputs of M^* and M over the input distribution and the low operational cost.

In Reality It is unfortunately impossible to solve the minimization problem in Eq. (1.4) in reality. There are so many reasons behind this, but the most important reason is the input distribution p_X or lack thereof. We can decide on a distance function D ourselves based on our goal. We can decide ourselves a hypothesis set H ourselves based on our requirements and constraints. All good, but p_X is not controllable in general, as it reflects how the world is, and the world does not care about our own requirements nor constraints.

Let's take the previous example of video-based intrusion system. Our reference machine M^* is a security expert who looks at a video frame (and a person within it) and determines whether that person is an intruder. We may decide to search over any arbitrary set of neural networks to minimize the expected loss. We have however absolutely no idea what the precise probability $p(\mathbf{x})$ of any video frame. Instead, we only observe $\mathbf{x}$'s which was randomly sampled from the input distribution by the surrounding environment. We have no access to the input distribution itself, but what comes out of it.

We only get to observe a *finite* number of such samples $\mathbf{x}$'s, with which we must approximate the expected cost in Eq. (1.4). This approximation method, that is approximation based on a finite set of samples from a probability distribution, is called a *Monte Carlo method*. Let us assume that we have observed N such samples: $\left\{\mathbf{x}^1, \ldots, \mathbf{x}^N\right\}$. Then we can approximate the expected cost by

$$\underbrace{\mathbb{E}_{\mathbf{x} \sim X} \left[D(M^*(\mathbf{x}), M, \mathbf{x}) \right] + \lambda C(M)}_{\text{Expected Cost}} = \underbrace{\frac{1}{N} \sum_{n=1}^{N} D(M^*(\mathbf{x}^n), M, \mathbf{x}^n) + \lambda C(M)}_{\text{Empirical Cost}} + \varepsilon, \tag{1.5}$$

where ε is an approximation error. We will call this cost, computed using a finite set of input vectors, an *empirical cost*.

Inference We have so far talked about what is a correct way to find a machine M for our purpose. We concluded that we want to find M by minimizing the empirical cost in Eq. (1.5). This is a good start, but let's discuss why we want to do this first. There may be many reasons, but often a major complication is the expense of running the reference machine M^* or the limited access to the reference machine M^*. Let us hence make it more realistic by assuming that we will have access to M^* only once at the very beginning together with a set of input examples. In other words, we are given

$$D_{\text{tra}} = \left\{ (\mathbf{x}^1, M^*(\mathbf{x}^1)), \ldots, (\mathbf{x}^N, M^*(\mathbf{x}^N)) \right\},$$

to which we refer as a *training set*. Once this set is available, we can find M that minimizes the empirical cost from Eq. (1.5) without ever having to query the reference machine M^*.

Now let us think of what we would do when there is a *new* input $\mathbf{x} \notin D_{\text{tra}}$. The most obvious thing is to use $\hat{M}$ that minimizes the empirical cost, i.e., $\hat{M}(\mathbf{x})$. Is there any other way? Another way is to use all the models in the hypothesis set, instead of using only one model. Obviously, not all models were born equal, and we cannot give all of them the same chance in making a decision. Preferably we give a higher weight to the machine that has a lower empirical cost, and also we want the weights to sum to 1 so that they reflect a properly normalized proportion. Thus, let us (arbitrarily) define, as an example, the weight of each model as:

$$\omega(M) = \frac{1}{Z} \exp\left(-J(M, D_{\text{tra}})\right),$$

where J corresponds to the empirical cost, and

$$Z = \sum_{M \in H} \exp\left(-J(M, D_{\text{tra}})\right)$$

is a normalization constant.

With all the models and their corresponding weights, I can now think of many strategies to *infer* what the output of M^* given the new input $\mathbf{x}$. Indeed, the first approach we just talked about corresponds to simply taking the output of the model that has the highest weight. Perhaps, I can take the weighted average of the outputs of all the machines:

$$\sum_{M \in H} \omega(M) M(\mathbf{x}),$$

which is equivalent to $\mathbb{E}\left[M(\mathbf{x})\right]$ under our arbitrary construction of the weights.[1] We can similarly check the variance of the prediction. Perhaps I want to inspect a set of outputs from the top-K machines according to the weights.

We will mainly focus on the former approach, which is often called *maximum a posteriori* (MAP), in this course. However, in a few of the lectures, we will also consider the latter approach in the framework of *Bayesian* modelling.

[1] *Is it really arbitrary, though?*

1.2 Perceptron

Let us examine how this concept of supervised learning is used in practice by considering a binary classification task. Binary classification is a task in which an input vector $\mathbf{x} \in \mathbb{R}^d$ is classified into one of two classes, negative (0) and positive (1). In other words, a machine M takes as input a d-dimensional vector and outputs one of two values.

Hypothesis Set In perceptron, a hypothesis set is defined as

$$H = \left\{ M | M(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \tilde{\mathbf{x}}), \mathbf{w} \in \mathbb{R}^{d+1} \right\},$$

where $\tilde{\mathbf{x}} = [\mathbf{x}; 1]$ denotes concatenating 1 at the end of the input vector $\mathbf{x}$,[2] and

$$\text{sign}(x) = \begin{cases} 0, & \text{if } x \leq 0, \\ 1, & \text{otherwise} \end{cases} . \tag{1.6}$$

In this section, we simply assume that each and every machine in this hypothesis set has a constant operational cost c, i.e., $C(M) = c$ for all $M \in H$.

Distance rewrite!

Given an input $\mathbf{x}$, we now define a distance between M^* and M. In particular, we will use the following distance function:[3]

$$D(M^*(\mathbf{x}), M, \mathbf{x}) = -\underbrace{(M^*(\mathbf{x}) - M(\mathbf{x}))}_{\text{(a)}} \underbrace{\left(\mathbf{w}^\top \tilde{\mathbf{x}}\right)}_{\text{(b)}}. \tag{1.7}$$

The term (a) states that the distance between the predictions of the reference and our machines is 0 as long as their predictions coincide. When it is not, the term (a) will be 1 if $M^*(\mathbf{x}) = 1$ and -1 if $M^*(\mathbf{x}) = 0$.

When it is not, the term (a) will be 1, which is when the term (b) comes into play. The dot product in (b) computes how well the weight vector $\mathbf{w}$ and the input vector $\mathbf{x}$ aligns with each other.[4] When they are positively aligned (pointing in the same direction), this term will be positive, making the output of the machine 1. When they are negative aligned (pointing in opposite directions), it will be negative with the output of the machine M 0.

Considering both (a) and (b), we see that the smallest value D can take is 0 ,when the prediction is correct,[5] and otherwise, positive. When the term (a) is 1, $\mathbf{w}^\top \tilde{\mathbf{x}}$ is negative, because $M(\mathbf{x})$ was 0, and the overall distance becomes positive (note the

[2] Why do we augment the original input vector $\mathbf{x}$ with an extra 1? What is an example in which this extra 1 is necessary? This is left to you as a **homework assignment**.

[3] There is a problem with this distance function. What is it? This is left to you as a **homework assignment**.

[4] $\mathbf{w}^\top \tilde{\mathbf{x}} = \sum_{i=1}^{d+1} w_i x_i$.

[5] There is one more case. What is it?

negative sign at the front.) When the term (b) is -1, $\mathbf{w}^\top \tilde{\mathbf{x}}$ is positive, because $M(\mathbf{x})$ was 1, in which case the distance is again positive.

What should we do in order to decrease this distance, if it is non-zero? We want to make the weight vector $\mathbf{w}$ to be aligned more positively with $M^*(\mathbf{x})$, if the term (a) is 1, which can be done by moving $\mathbf{w}$ toward $\tilde{\mathbf{x}}$. In other words, the distance D shrinks if we add a bit of $\tilde{\mathbf{x}}$ to $\mathbf{w}$, i.e., $\mathbf{w} \leftarrow \mathbf{w} + \eta\tilde{\mathbf{w}}$. If the term (a) is -1, we should instead push $\mathbf{w}$ so that it will *negatively* align with $\tilde{\mathbf{x}}$, i.e., $\mathbf{w} \leftarrow \mathbf{w} - \eta\tilde{\mathbf{w}}$. These two cases can be unified by

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \left(M^*(\mathbf{x}) - M(\mathbf{x})\right)\tilde{\mathbf{x}}, \tag{1.8}$$

where η is often called a *step size* or *learning rate*. We can repeat this update until the term (a) in Eq. (1.7) becomes 0.

Learning As discussed earlier, we assume that we make only a finite number of queries to the reference machine M^* using a set of inputs drawn from the unknown input distribution $p(\mathbf{x})$. This results in our training dataset:

$$D_{\text{tra}} = \left\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\right\},$$

where we begin to use a short hand $y_n = M^*(\mathbf{x}_n)$.

With this dataset, our goal now is to find $M \in H$ that has the least empirical cost in Eq. (1.5) with the distance function defined in Eq. (1.7). Combining these two, we get

$$J(\mathbf{w}, D_{\text{tra}}) = -\frac{1}{N}\sum_{n=1}^{N} \left(y_n - M(\mathbf{x}_n)\right)\left(\mathbf{w}^\top \tilde{\mathbf{x}_n}\right). \tag{1.9}$$

We will again resort to an iterative method for minimizing this empirical cost function, as we have done with a single input vector above. What we will do is to collect all those input vectors on which M (or equivalently $\mathbf{w}$) has made mistakes. This is equivalent to considering only those input vectors where $y_n - M(\mathbf{x}_n) \neq 0$. Then, we collect all the *update directions*, computed using Eq. (1.8), and move the weight vector $\mathbf{w}$ toward its average. That is,

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \frac{1}{N}\sum_{n=1}^{N} \left(y_n - M(\mathbf{x}_n)\right)\tilde{\mathbf{x}}. \tag{1.10}$$

We apply this rule repeatedly until the empirical cost in Eq. (1.9) does not improve (i.e., decrease).

This learning rule is known as a *perceptron learning rule*, which was proposed by Rosenblatt in 1950's [3], and has a nice property that it will find a correct M in the sense that the empirical cost is at its minimum (0), *if* such M is in H. In other words, if our hypothesis set H is good and includes a reference machine M^*, this perceptron learning rule will eventually find an equally good machine M. It is important to note that there may be many such M, and the perceptron learning rule will find one of them.

1.3 Logistic Regression

The perceptron is not entirely satisfactory for a number of reasons. One of them is that it does not provide a well-calibrated measure of the degree to which a given input is either negative or positive. That is, we want to know not whether it is negative or positive but rather how likely it is negative. It is then natural to build a machine that will output the probability $p(C|\mathbf{x})$, where $C \in \{-1, 1\}$.

Hypothesis Set To do so, let us first modify the definition of a machine M. M now takes as input a vector $\mathbf{x} \in \mathbb{R}^d$ and returns a probability $p(C|\mathbf{x}) \in [0, 1]$ rather than $\{0, 1\}$. We only need to change just one thing from the perceptron, that is

$$M(\mathbf{x}) = \sigma(\mathbf{w}^\top \tilde{\mathbf{x}}), \tag{1.11}$$

where σ is a sigmoid function defined as

$$\sigma(a) = \frac{1}{1 + \exp(-a)}$$

and is bounded by 0 from below and by 1 from above. Suddenly this machine does not return the prediction, but the probability of the prediction being positive (1). That is,

$$p(C = 1|\mathbf{x}) = M(\mathbf{x}).$$

Naturally, our hypothesis set is now

$$H = \left\{ M | M(\mathbf{x}) = \sigma(\mathbf{w}^\top \tilde{\mathbf{x}}), \mathbf{w} \in \mathbb{R}^{d+1} \right\},$$

where $\tilde{\mathbf{x}} = [\mathbf{x}; 1]$ denotes concatenating 1 at the end of the input vector as before.

Distance The distance is not trivial to define in this case, because the things we want to measure the distance between are not directly comparable. One is an element in a discrete set (0 or 1), and the other is a probability. It is helpful now to think instead about *how often* a machine M will agree with the reference machine M^*, if we randomly sample the prediction given its output distribution $p(C|\mathbf{x})$. This is exactly equivalent to $p(C = M^*(\mathbf{x})|\mathbf{x})$. In this sense, the distance between the reference machine M^* and our machine M given an input vector $\mathbf{x}$ is smaller than this frequency of M being correct is larger, and vice versa. Therefore, we define the distance as the negative log-probability of the M's output being correct:

$$\begin{aligned} D(M^*(\mathbf{x}), M, \mathbf{x}) &= -\log p(C = M^*(\mathbf{x})|\mathbf{x}) \\ &= -\left(M^*(\mathbf{x}) \log M(\mathbf{x}) + (1 - M^*(\mathbf{x})) \log(1 - M(\mathbf{x}))\right), \end{aligned} \tag{1.12}$$

where $p(C = 1|\mathbf{x}) = M(\mathbf{x})$. The latter equality comes from the definition of *Bernoulli distribution*.[6]

[6] give a brief def of Bernoulli distribution

With this definition of our distance, how do we adjust $\mathbf{w}$ of M to lower it? In the case of perceptron, we were able to manually come up with an *algorithm* by looking at the perceptron distance in Eq. (1.7). It is however not too trivial with this logistic regression distance.[7] Thus, we now turn to Calculus, and use the fact that the *gradient* of a function points to the direction toward which its output increases (at least locally).

The gradient of the above logistic regression distance function with respect to the weight vector $\mathbf{w}$ is[8]

$$\nabla_{\mathbf{w}} D(M^*(\mathbf{x}), M, \mathbf{x}) = -(M^*(\mathbf{x}) - M(\mathbf{x}))\tilde{\mathbf{x}}. \tag{1.13}$$

When we move the weight vector ever so slightly in the opposite direction, the logistic regression distance in Eq. (1.12) will decrease. That is,

$$\mathbf{w} \leftarrow \mathbf{w} + \eta\,(M^*(\mathbf{x}) - M(\mathbf{x}))\,\tilde{\mathbf{x}}, \tag{1.14}$$

where $\eta \in \mathbb{R}$ is a small scalar and called either a *step size* or *learning rate*.

Coincidence? Is it surprising to realize that this rule for logistic regression is identical to the perceptron rule in Eq. (1.8)? Let us see what this logistic regression rule, or equivalent to perceptron learning rule, does by focusing on the update term (the second term in the learning rule). The first multiplicative factor $(M^*(\mathbf{x}) - M(\mathbf{x}))$ can be written as

$$M^*(\mathbf{x}) - M(\mathbf{x}) = \underbrace{\overline{\text{sign}}(M^*(\mathbf{x}) - M(\mathbf{x}))}_{\text{(a)}} \underbrace{|M^*(\mathbf{x}) - M(\mathbf{x})|}_{\text{(b)}}, \tag{1.15}$$

where

$$\overline{\text{sign}}(x) = \begin{cases} -1, & \text{if } x \leq 0, \\ 1, & \text{otherwise} \end{cases}$$

is a symmetric sign function (compare it to the asymmetric sign function in Eq. (1.6).)

The term (a) effectively tell us *in which way* the machine is wrong about the input $\mathbf{x}$. Is M saying it is likely to be 0 when the reference machine says 1, or is M saying it is likely to be 1 when the reference machine says 0? In the former case, we want the weight vector $\mathbf{w}$ to align more with $\mathbf{x}$, and thus the positive sign. In the latter case, we want the opposite, and hence the negative sign. The second term (b) tells us *how much* the machine is wrong about the input $\mathbf{x}$. This term ignores in which direction the machine is wrong, since it is computed by the term (a), but entirely focuses on how *far* the model's prediction is from that of the reference machine. If the prediction is close to that of the reference machine, we only want the weight vector to move ever so slowly.

The second term in both the logistic regression and perceptron rules is the input vector, augmented with an extra 1. This term is there, because the prediction by a machine M is made based on how well the weight vector and the input vector align with each other, which is computed as a dot product between these two vectors.

[7] It may be trivial to some who have great mathematical intuition.

[8] The step-by-step derivation of this is left to you as a **homework assignment**.

In other words, it is not a coincidence, but only natural that they are equivalent, as both of them effectively solve the same problem of binary classification. In the case of perceptron, we have reached its learning rule based on our observation of the process, while in the case of logistic regression, we relied on a more mechanical process of using gradient to find the steepest ascent direction. The latter approach is often more desirable, as it allows us to apply the same procedure (update the machine following the steepest descent direction,) however with a constraint that the empirical cost be differentiable (almost everywhere[9]) with respect to the machine's parameters. We will thus largely stick to this kind of gradient-based optimization to minimize any distance function from here on.

The only major difference between the learning rules for the logistic regression and perceptron is whether the term (b) in Eq. (1.15) is discrete (in the case of perceptron) or continuous (in the case of logistic regression.) More specifically, the term (b) in the perceptron learning rule is either 0 or 1.

Learning With the distance function defined, we can write a full empirical cost as

$$J(\mathbf{w}, D_{\text{tra}}) = -\frac{1}{N}\sum_{n=1}^{N} y_n \log M(\mathbf{x}_n) + (1-y_n)\log(1-M(\mathbf{x}_n)).$$

We assume that the operational cost, or complexity, of each M can be ignored. Similarly to what we have done for minimizing (decreasing) the distance between M and M^* given a single input vector, we will use the gradient of the whole empirical cost function to decide how we change the weight vector $\mathbf{w}$. The gradient is

$$\nabla_{\mathbf{w}} J = -\frac{1}{N}\sum_{n=1}^{N} \nabla_{\mathbf{w}} D(y_n, M, \mathbf{x}_n),$$

which is simply the average of the gradients of the distances from Eq. (1.13) over the whole training set.

The fact that the empirical cost function is (twice) differentiable with respect to the weight vector allows us to use any advanced gradient-based optimization algorithm. Perhaps even more importantly, we can use automatic differentiation to compute the gradient of the empirical cost function *automatically*.[10]

[9] We will see why the cost function does not have to be differentiable everywhere later in the course.

[10] Some of widely-used software packages that implement automatic differentiation include

- Autograd for Python: `https://github.com/HIPS/autograd`
- Autograd for Torch: `https://github.com/twitter/torch-autograd`
- Theano: `http://deeplearning.net/software/theano/`
- TensorFlow: `https://www.tensorflow.org/`

Throughout this course, we will use Autograd for Python for demonstration.

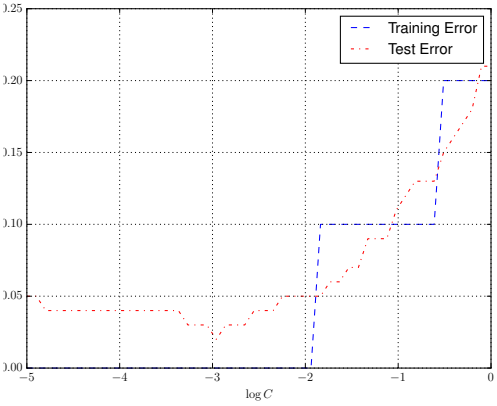


Figure 1.1: Training and test errors with respect to the weight decay coefficient. Notice that the test error grows back even when the training error is 0 as the weight decay coefficient decreases.

1.4 Overfitting, Regularization and Complexity

1.4.1 Overfitting: Generalization Error

1.4.2 How Wrong is Wrong?

1.5 Support Vector Machines

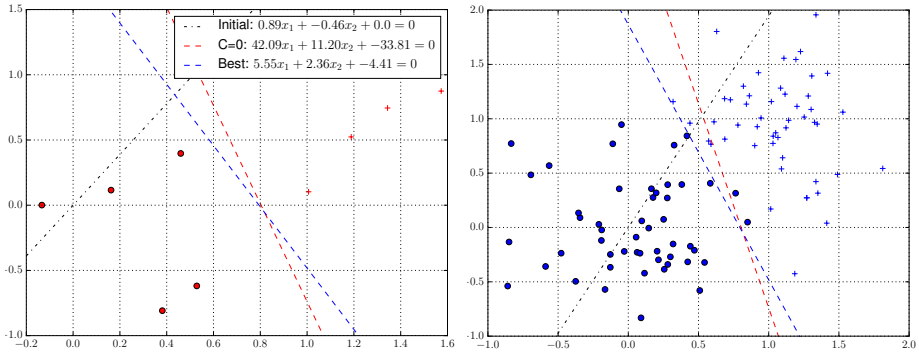
1.5.1 Classification as Scoring

Let us use f as a shorthand for denoting the dot product between the weight vector $\mathbf{w}$ and an input vector $\mathbf{x}$ augmented with an extra 1, that is $f(\mathbf{x};\mathbf{w}) = \mathbf{w}^\top \tilde{\mathbf{x}}$. Instead of $y \in \{0,1\}$ as a set of labels (negative and positive), we will switch to $y \in \{-1,1\}$ to make later equations less cluttered. Now, let us define a score function[11] that takes as input an input vector $\mathbf{x}$, the correct label y (returned by a reference machine M^*) and the weight vector (or you can say the machine M itself):

$$s(y,\mathbf{x};M) = y\mathbf{w}^\top \tilde{\mathbf{x}}. \tag{1.16}$$

Given any machine that performs binary classification, such as perceptron and logistic regression, this score function tells us whether a given input vector $\mathbf{x}$ is correctly classified. If the score is larger than 0, it was correctly classified. Otherwise, the score

[11] Note that the term "score" has a number of different meanings. For instance, in statistics, the score is defined as a gradient of the TODO



(a) Training Set (b) Test Set

Figure 1.2: Both solutions of logistic regression perfectly fit the training set regardless of whether weight decay regularization was used. However, it is clear that the model with the optimal weight decay coefficient (blue line) classifies the test set better.

would be smaller than 0. In other words, the score function defines a *decision boundary* of the machine M, which is defined as a set of points at which the score is 0, i.e.,

$$B(M) = \{\mathbf{x} | s(M(\mathbf{x}), \mathbf{x}; M) = 0\}.$$

When the score function s is defined as a linear function of the input vector $\mathbf{x}$ as in Eq. (1.16), the decision boundary corresponds to a linear hyperplane. In such a case, we call the machine a *linear classifier*.

With this definition of a score function s, the problem of classification is equivalent to finding the weight vector $\mathbf{w}$, or the machine M, that positively scores each pair of an input vector and the corresponding label. In other words, our empirical cost function for classification is now

$$J(M, D_{\text{tra}}) = \frac{1}{|D_{\text{tra}}|} \sum_{(y,\mathbf{x}) \in D_{\text{tra}}} \underbrace{\overline{\text{sign}}(-s(y, \mathbf{x}; M))}_{D_{\text{0-1}}=\text{0-1 Loss}}. \tag{1.17}$$

Log Loss: Logistic Regression The distance[12] function of logistic regression in Eq. (1.12) can be re-written as

$$D_{\log}(y, \mathbf{x}; M) = \frac{1}{\log 2} \log(1 + \exp(-s(y, \mathbf{x}; M))), \tag{1.18}$$

where $y \in \{-1, 1\}$ instead of $\{0, 1\}$, and the score function s is defined in Eq. (1.16).[13] This loss function is usually referred to as *log loss*.

[12] From here on, I will use both *distance* and *loss* to mean the same thing. This is done to make terminologies a bit more in line with how others use.

[13] **Homework assignment**: show that Eq. (1.12) and Eq. (1.18) are equivalent up to a constant multiplication for binary logistic regression.

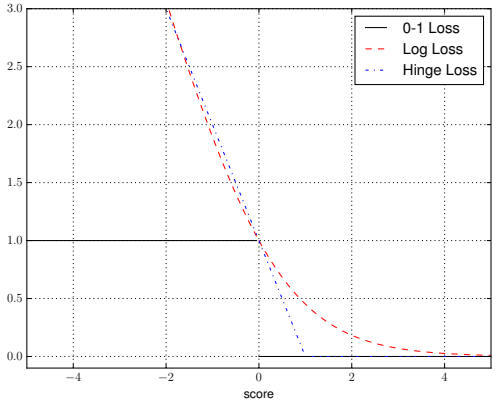


Figure 1.3: The figure plots three major loss functions–0-1 loss, log loss and hinge loss– with respect to the output of a score function s. Note that the log loss and hinge loss are upper-bound to the 0-1 loss.

How is this log loss related to the 0-1 loss from Eq. (1.17)? As shown in Fig. 1.3, the log loss is an upper-bound of the 0-1 loss. That is,

$$D_{\log}(y,\mathbf{x};M) \geq D_{0\text{-}1}(y,\mathbf{x};M) \text{ for all } s(y,\mathbf{x};M) \in \mathbb{R}.$$

1.5.2 Hinge Loss and Support Vector Machines

1.6 Nonlinear Classification

1.6.1 Radial Basis Function Networks

1.6.2 k-Nearest Neighbours

1.6.3 Kernel Support Vector Machines

1.7 Decision Tree*

1.8 Ensemble Methods*

Chapter 2

Regression

We have so far considered a problem of classification, where the output of a machine M is constrained to be a finite set of discrete labels/classes. In this section, we consider a *regression* problem in which case the machine outputs an element from an infinite set.

2.1 Linear Regression

2.1.1 Linear Regression

2.1.2 Regularization and Prior Distributions

2.2 Bayesian Linear Regression and Gaussian Process Regression

2.2.1 Bayesian Approach to Machine Learning

2.2.2 Gaussian Process Regression*

Chapter 3

Dimensionality Reduction

3.1 Unsupervised Learning: Problem Setup

Unsupervised learning is a weird, but fascinating problem. Unlike supervised learning in which a machine was defined as a transformation of an input vector $\mathbf{x}$, a machine M in unsupervised learning *generates* an input vector.

3.1.1 Naive Bayes Classifier

3.2 Dimensionality Reduction: Problem Setup

3.3 Principal Component Anslysis

3.3.1 Maximum Variance Criterion

3.3.2 Minimum Reconstruction Criterion

3.3.3 Probabilistic Principal Component Analysis

Expectation-Maximization Algorithm

3.4 Other Dimensionality Reduction Techniques*

Chapter 4

Clustering

4.1 Problem Setup

4.1.1 Clustering vs. Dimensionality Reduction

4.2 k-Means Clustering

4.3 Mixture of Gaussians*

4.4 Other Clustering Methods*

Chapter 5

Structured Output Prediction and Reinforcement Learning

5.1 Time-Series Modelling*

Bibliography

[1] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[2] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[3] F. Rosenblatt. *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Report (Cornell Aeronautical Laboratory). Spartan Books, 1962.