Introduction to Machine Learning

Kyunghyun Cho

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

March 22, 2017

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 [**?**]
- "Machine Learning: a probabilistic perspective" by Kevin Murphy [**?**]
- "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 not be covered during regular lectures. They will likely not be filled in by the end of the first round of the lectures either..

[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

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

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

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

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:

$$\sum_{M \in H} \omega(M) M(\mathbf{x}), \tag{1.6}$$

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

$$\text{sign}(x) = \begin{cases} 0, & \text{if } x \leq 0, \\ 1, & \text{otherwise} \end{cases} . \tag{1.7}$$

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

$$D(M^*(\mathbf{x}), M, \mathbf{x}) = -\underbrace{(M^*(\mathbf{x}) - M(\mathbf{x}))}_{\text{(a)}} \underbrace{\left(\mathbf{w}^\top \tilde{\mathbf{x}}\right)}_{\text{(b)}}. \tag{1.8}$$

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 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.

[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?

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

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \left(M^*(\mathbf{x}) - M(\mathbf{x})\right)\tilde{\mathbf{x}}, \tag{1.9}$$

where η is often called a *step size* or *learning rate*. We can repeat this update until the term (a) in Eq. (1.8) 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. (1.8). Combining these two, we get

$$J(\mathbf{w}, D_{\text{tra}}) = -\frac{1}{N}\sum_{n=1}^{N} \left(y_n - M(\mathbf{x}_n)\right)\left(\mathbf{w}^\top \tilde{\mathbf{x}_n}\right). \tag{1.10}$$

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. (1.9), and move the weight vector $\mathbf{w}$ toward its average. That is,

$$\mathbf{w} \leftarrow \mathbf{w} + \eta \frac{1}{N}\sum_{n=1}^{N} \left(y_n - M(\mathbf{x}_n)\right)\tilde{\mathbf{x}}. \tag{1.11}$$

We apply this rule repeatedly until the empirical cost in Eq. (1.10) does not improve (i.e., decrease).

This learning rule is known as a *perceptron learning rule*, which was proposed by Rosenblatt in 1950's [**?**], 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

$$M(\mathbf{x}) = \sigma(\mathbf{w}^\top \tilde{\mathbf{x}}), \tag{1.12}$$

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:

$$\begin{aligned} D(M^*(\mathbf{x}), M, \mathbf{x}) &= -\log p(C = M^*(\mathbf{x})|\mathbf{x}) \\ &= -\left(M^*(\mathbf{x}) \log M(\mathbf{x}) + (1 - M^*(\mathbf{x})) \log(1 - M(\mathbf{x}))\right), \end{aligned} \tag{1.13}$$

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

[6] A binary, or Bernoulli, random variable X may take one of two values c_0 and c_1 (often 0 and 1). The probability of X being c_1 is defined as a scalar $p \in [0, 1]$, and the probability of X being c_0 as $1 - p$. When $c_0 = 0$ and $c_1 = 1$, we can write the probability of X as

$$p(X) = p^X (1 - p)^{(1 - X)},$$

and its logarithm is

$$\log p(X) = X \log p + (1 - X) \log(1 - p).$$

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. (1.8). 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]

$$\nabla_{\mathbf{w}} D(M^*(\mathbf{x}), M, \mathbf{x}) = -(M^*(\mathbf{x}) - M(\mathbf{x}))\tilde{\mathbf{x}}. \tag{1.14}$$

When we move the weight vector ever so slightly in the opposite direction, the logistic regression distance in Eq. (1.32) will decrease. That is,

$$\mathbf{w} \leftarrow \mathbf{w} + \eta\,(M^*(\mathbf{x}) - M(\mathbf{x}))\,\tilde{\mathbf{x}}, \tag{1.15}$$

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. (1.9)? 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

$$M^*(\mathbf{x}) - M(\mathbf{x}) = \underbrace{\overline{\text{sign}}(M^*(\mathbf{x}) - M(\mathbf{x}))}_{\text{(a)}} \underbrace{|M^*(\mathbf{x}) - M(\mathbf{x})|}_{\text{(b)}}, \tag{1.16}$$

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. (1.7).)

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. (1.16) 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. In the case of logistic regression, on the other hand, the term (b) corresponds to what degree the logistic regression's output is close to the true output.

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. (1.14) 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] Any further discussion on advanced optimization algorithms is out of this course's scope, and I recommend you the following courses:

[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.

- DS-GA 3001.03: Optimization and Computational Linear Algebra for Data Science
- CSCI-GA.2420 NUMERICAL METHODS I
- CSCI-GA.2421 NUMERICAL METHODS II

1.4 One Classifier, Many Loss Functions

1.4.1 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):

$$s(y,\mathbf{x};M) = y\mathbf{w}^\top \tilde{\mathbf{x}}. \tag{1.17}$$

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 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. (1.17), the decision boundary corresponds to a linear hyperplane. In such a case, we call the machine a *linear classifier*, and if the reference machine M^* is a linear classifier, we call this problem of classification *linear separable*.

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

$$J(M, D_{\text{tra}}) = \frac{1}{|D_{\text{tra}}|} \sum_{(y,\mathbf{x}) \in D_{\text{tra}}} \underbrace{\overline{\text{sign}}(-s(y,\mathbf{x};M))}_{D_{\text{0-1}}=\text{0-1 Loss}}. \tag{1.18}$$

[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 log-probability, that is

$$\frac{\partial \log p(x)}{\partial x}.$$

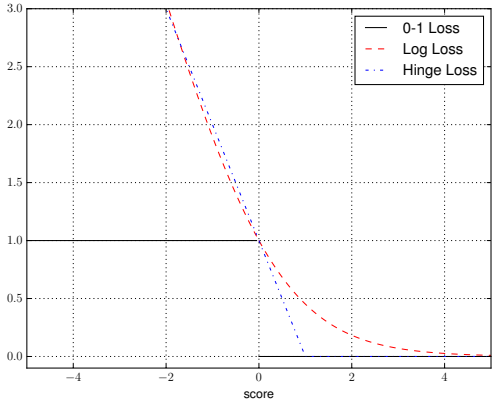


Figure 1.1: The figure plots three major loss functions–0-1 loss, log loss and hinge loss– with respect to the output of a score function s. Note that the log loss and hinge loss are upper-bound to the 0-1 loss.

Log Loss: Logistic Regression The distance[12] function of logistic regression in Eq. (1.32) can be re-written as

$$D_{\log}(y,\mathbf{x};M) = \frac{1}{\log 2}\log(1+\exp(-s(y,\mathbf{x};M))), \tag{1.19}$$

where $y \in \{-1,1\}$ instead of $\{0,1\}$, and the score function s is defined in Eq. (1.17).[13] This loss function is usually referred to as *log loss*.

How is this log loss related to the 0-1 loss from Eq. (1.18)? As shown in Fig. 1.1, the log loss is an upper-bound of the 0-1 loss. That is,

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

By minimizing this upper-bound, we can indirectly minimize the 0-1 loss. Of course, minimizing the upper-bound does not necessarily minimize the 0-1 loss, but we can be certain that the 0-1 loss, or true classification loss, is lower than how far we have minimized the log loss.

1.4.2 Support Vector Machines: Max-Margin Classifier

Hinge Loss One potential issue with the log loss in Eq. (1.19) is that it is never 0:

$$D_{\log}(y,\mathbf{x};M) > 0.$$

Why is this an issue? Because it means that the machine M *wastes* its modelling capacity on pushing those examples as far away from the decision boundary as possible even if they were already correctly classified. This is unlike the 0–1 loss which ignores any correctly classified example.

[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. (1.32) and Eq. (1.19) are equivalent up to a constant multiplication for binary logistic regression.

Let us introduce another loss function, called *hinge loss*, which is defined as

$$D_{\text{hinge}}(y, \mathbf{x}; M) = \max(0, 1 - s(y, \mathbf{x}; M)).$$

Similarly to the log loss, the hinge loss is always greater than or equal to the zero-one loss, as can be seen from Fig. 1.1. We minimize this hinge loss and consequently minimize the 0–1 loss.[14]

What does this imply? It implies that minimizing the empirical cost function that is the sum of hinge losses will find a solution in which all the examples are at least a unit-distance (1) away from the decision boundary ($s(y, \mathbf{x}; M) = 0$). Once any example is further than a unit-distance away from *and* on the correct side of the decision boundary, there will not be any penalty, i.e., zero loss. This is contrary to the log loss which penalizes even correctly classified examples unless they are infinitely far away from and on the correct side of the boundary.

Max-Margin Classifier It is time for a thought experiment. We have only two unique training examples; one positive example $\tilde{\mathbf{x}}^+$ and one negative example $\tilde{\mathbf{x}}^-$. We can draw a line l between these two points. Any linear classifier perfectly classifies these two examples into their correct classes as long as the decision boundary, or *separating hyperplane*, meets the line l connecting the two examples. Because we are in the real space, there are infinitely many such classifiers. Among these infinitely many classifiers, which one should we use? Should the intersection between the separating hyperplane and the connecting line l be close to either one of those examples? Or, should the intersection be as far from both points as possible? An intuitive answer is "yes" to the latter: we want the intersection to be as far away from both points as possible.

Let us define a margin γ as the distance between the decision boundary ($\mathbf{w}^\top \tilde{\mathbf{x}} = 0$) and the nearest training example $\tilde{\mathbf{x}}$, of course, (loosely) assuming that the decision boundary classifies most of, if not all, the training examples correctly. This assumption is necessary to ensure that there are at least one correctly classified example on each side of the decision boundary. The above thought experiment now corresponds to an idea of finding a classifier that has the largest margin, i.e., a *max-margin classifier*.

The distance to the nearest positive and negative examples can be respectively written down as

$$d^+ = \frac{\mathbf{w}^\top \tilde{\mathbf{x}}^+}{\|\mathbf{w}\|},$$
$$d^- = -\frac{\mathbf{w}^\top \tilde{\mathbf{x}}^-}{\|\mathbf{w}\|},$$

[14] One major difference between this hinge loss and the log loss is that the former is not differentiable everywhere. Does it mean that we cannot use a gradient-based optimization algorithm for finding a solution that minimizes the empirical cost function based on the hinge loss? If not, what can we do about it? The answer is left to you as a **homework assignment**.

Then, the margin can be defined in terms of these two distances as

$$\gamma = \frac{1}{2}(d^+ + d^-) \tag{1.20}$$

$$= \frac{C}{\|\mathbf{w}\|}, \tag{1.21}$$

where C is the unnormalized distance to the positive and negative examples from the decision boundary. These two examples are equi-distance C away from the decision boundary, because our thought experiment earlier suggests that the decision boundary with the maximum margin should be as far away from both of these examples as possible.[15]

Eq. (1.20) states that the margin γ is *inversely proportional* to the norm of the weight vector $\|\mathbf{w}\|$. In other words, we should minimize the norm of the weight vector if we want to maximize the margin.

Support vector machines Now let us put together the hinge loss based empirical cost function and the principle of maximum margin into one optimization problem:

$$J_{\text{svm}}(M, D_{\text{tra}}) = \frac{C}{2} \underbrace{\|\mathbf{w}\|^2}_{\text{(a)}} + \underbrace{\frac{1}{|D_{\text{tra}}|} \sum_{(y,x)\in D_{\text{tra}}} D_{\text{hinge}}(y, \mathbf{x}; M)}_{\text{(b)}}, \tag{1.22}$$

where $C/2$ can be thought of as a regularization coefficient. This is a cost function for so-called support vector machines [**?**].

This classifier is called a *support vector* machine, because at its minimum, the weight vector can be fully described by a small set of training examples which are often referred to as support vectors. Let us derive it quickly here:

$$\frac{\partial J_{\text{svm}}}{\partial \mathbf{w}} = C\mathbf{w} - \frac{1}{|D_{\text{tra}}|} \sum_{(y,x)\in D_{\text{tra}}} \mathbb{I}(y\mathbf{w}^\top \mathbf{x} \leq 1) y\mathbf{x} = 0$$

$$\Leftrightarrow \mathbf{w} = \frac{1}{C|D_{\text{tra}}|} \sum_{(y,x)\in D_{\text{misclas}}} y\mathbf{x},$$

where D_{miscla} is a set of misclassified, or barely classified, training examples, and

$$\mathbb{I}(a) = \begin{cases} 1, & \text{if } a \text{ is true} \\ 0, & \text{otherwise} \end{cases}.$$

Often, $|D_{\text{miscla}}| \ll |D_{\text{tra}}|$, and thus, a support vector machine is categorized into a family of sparse classifiers.

[15] **Homework assignment**: Derive Eq. (1.20), and explain in words the derivation.

1.5 Overfitting, Regularization and Complexity

1.5.1 Overfitting: Generalization Error

At the very beginning of this course, we have talked about two cost functions; (1) expected cost in Eq. (1.4) and (2) empirical cost in Eq. (1.5).[16] We loosely stated that we find a machine that minimizes the empirical cost $\tilde{R}$ because we cannot compute the expected cost R, somehow hoping that the approximation error ε (from Eq. (1.5)) would be small. Let's discuss this a bit more in detail here.

Let us define the generalization error as a difference between the empirical and expected cost functions given a reference machine M^*, a machine M and a training set D_{tra}:

$$G(M^*, M, D_{\text{tra}}) = R(M^*, M) - \tilde{R}(M^*, M, D_{\text{tra}}). \tag{1.23}$$

We can further define its expectation as

$$\bar{G}(M^*, M) = R(M^*, M) - \mathbb{E}_{D\sim P}\left[\tilde{R}(M^*, M, D)\right], \tag{1.24}$$

where P is the data distribution.

When this generalization error is large, it means that we are hugely overestimating how good the machine M is compared to the reference machine M^*. Although M is not really good, i.e., the expected cost R is high, M is good on the training set D_{tra}, i.e., the empirical cost $\tilde{R}$ is low. This is precisely the situation we want to avoid: we do not want to brag our machine is good when it is in fact a horrible approximation to the reference machine M^*. In this embarrassing situation, we say that the machine M is *overfitting* to the training data.

Unlike how I said earlier, the goal of supervised learning, or machine learning in general, is therefore to search for a machine in a hypothesis set not only to minimize the empirical cost function but also to minimize the generalization error. In other words, we want to find a machine that simultaneously minimizes the empirical cost function and avoids overfitting.

Statistical learning theory, a major subfield of machine learning, largely focuses on computing the upper-bound of the generalization error. The bound, which is often probabilistic, is often a function of, for instance, the dimensionality of an input vector $\mathbf{x}$ and a hypothesis set. This allows us to understand how well we should expect our learning setting to work, in terms of generalization error, even *without* testing it on actual data. Awesome, but we will skip this in this course, as the upper-bound is often too loose, and rough sample-based approximation of the generalization error works better in practice.[17]

1.5.2 Validation

In practice, the generalization error itself is rarely of interest. It is rather the expected cost function R of a machine M that interests us, because we will eventually pick one

[16] Note that the complexity, or operational cost, of a machine M is often *not* included in either of the cost functions, but this is not a problem to include them as long as both cost functions have them.

[17] *Well, the better answer is that it involves too much math..*

M that has the least expected cost.[18] But, again, we cannot really compute the expected cost function and must resort to an approximate method. As done for training, we again use a set D_{val} of samples from the data distribution to estimate the expected cost, as in Eq. (1.5), that is[19]

$$\tilde{R}_{\text{val}} = \frac{1}{N'} \sum_{(y,\mathbf{x}) \in D_{\text{val}}} D(y, M, \mathbf{x}) + \lambda C(M).$$

In order to avoid any bias in estimating the expected cost function, via this validation cost, we must use a validation set D_{val} *independent* from the training set D_{tra}. We ensure this in practice by splitting a given training set into two disjoint subsets, or partitions, and use them as training and validation sets.

This is how we will estimate the expected cost of a trained model M. How are we going to use it? Let us consider having more than one hypothesis set H_l for $m = l, \ldots, L$. Given a training set D_{tra} and one of hypothesis sets H_l, we will find a machine $M^l \in H_l$ that minimizes the empirical cost function:

$$M^l = \underset{M \in H_l}{\arg\min}\, \tilde{R}(M, D_{\text{tra}}).$$

We now have a set of trained models $\left\{M^1, \ldots, M^L\right\}$, and we must choose one of them as our final solution. In doing so, we use the validation cost computed using a *separate* validation set D_{val} which approximates the expected cost. We choose the one with the smallest validation cost among the L trained models:

$$\hat{M} = \underset{M^l | l=1,\ldots,L}{\arg\min}\, \tilde{R}_{\text{val}}(M^l, D_{\text{val}}).$$

Example 1: Model Selection Let's take a simple example of having three hypothesis sets. One hypothesis set $H_{\text{perceptron}}$ contains all possible perceptrons, the next set H_{logreg} has all possible logistic regressions, and the last set H_{svm} has all possible support vector machines. We will find one perceptron, one logistic regression and one support vector machine from these hypothesis sets by using the learning rules we learned earlier. We select one of these two *models* based not on the empirical cost function but on the validation cost function.

Example 2: Early Stopping Can this be useful even if we have a single hypothesis set? Indeed. So far, two families of classifiers we have considered, which are perceptrons and logistic regression, learning happened iteratively. That is, we slowly evolve the parameters, or more specifically the weight vector. Let us denote the weight vector after the l-th update by $\mathbf{w}^l$, and assume that we have applied the learning rule L-many times. Suddenly I have L different classifiers, just like what we had with L different

[18] Though, as we discussed earlier in Eq. (1.6), it may be better to keep more than one M in certain cases.

[19] It is a usual practice to omit the model complexity term when computing the validation cost. We will get to why this is so shortly.

classifiers earlier when there were L hypothesis sets.[20] Instead of taking the very last weight vector $\mathbf{w}^L$, we will choose

$$\hat{M} = \underset{M^l | l=1,\ldots,L}{\arg\min}\ \tilde{R}_{\text{val}}(M^l, D_{\text{val}}),$$

where M^l is a classifier with its weight vector $\mathbf{w}^l$. This technique is often referred to as *early stopping*, and is a crucial recipe in iterative learning.

Cross-Validation Often we are not given a large enough set of examples to afford dividing it into two sets, and using only one of them to train competing models. Either the training set would be too small for the empirical (training) cost to well approximate the expected cost, or the validation set would be too small for the validation cost to be meaningful when selecting the final model (or the correct hypothesis set.) In this case, we use a technique called *K-fold cross validation*.

We first evenly split a given training set D_{tra} into K partitions while ensuring that the statistics of all the partitions are roughly similar, for instance, by ensuring that the label proportion (the number of positive examples to that of the negative examples) stays same. For each hypothesis set H, we train K classifiers, where D_{tra}
D^k_{tra} is used to train the k-th classifier and D^k_{tra} to compute its validation cost. We then get K validation costs of which the average is our estimate of the empirical cost of the current hypothesis set. We essentially reuse each example $K-1$ times for training and $K-1$ times for validation, thereby increasing both the efficiency of our use of training examples as well as the stability of our estimate. When K is set to the number of all training examples, we call it leave-one-out cross-validation (LOOCV).

Cross-validation is a good approach for model selection, but not usable for early-stopping.[21] Furthermore, when the training set is large, it may easily become infeasible to try cross-validation, as the computational complexity grows linearly with respect to K. It is however a recommended practice to use cross-validation whenever you have a manageable size of training examples.

Test Set As soon as we use the validation set to *select* among multiple hypothesis sets or models, the validation cost of the final model is not anymore a good estimate of its expected cost. This is largely because again of overfitting. Our choice of hypothesis set or model will agree well with the validation cost, but unavoidably the validation cost will have its own generalization error. Thus, we need yet another set of examples based on which we estimate the true expected cost. This set of examples is often called a *test set*.

Most importantly, *the test set must be withheld* throughout the whole process of learning *until the very end*. As soon as any intermediate decision about learning, such as the choice of hypothesis set, is made (even subconsciously) based on the test set, your estimate of the expected cost of the final model becomes biased. Thus, in practice,

[20] In some sense, we can view each of these classifiers as coming from L different (overlapping) hypothesis sets. Each hypothesis set can be characterized as *reachable* in l gradient updates from the initial weight vector.

[21] **Homework assignment**: Why is cross-validation not a feasible strategy for early-stopping?

what you must do is to split a training set into three portions; training, validation and test partitions, in advance of anything you do. Is there an ideal split? No.

Similarly to how we estimated the validation cost, it is often the case in which you do not have enough data and cannot afford to withhold a substantial portion of it as a test set. In that case, it is also a good practice to employ the strategy of K-fold cross-validation. In this case, it is worth noting that you need *nested* K-fold cross-validation. That is, for each k-th fold from the outer cross-validation loop, you use the inner cross-validation (K iterations of training and validation) for model selection. It is computationally expensive, as now it grows quadratically with respect to K, i.e., $O(K^2)$, but this is the best practice to accurately estimate the expected cost of your learning algorithm given only a small number of training examples.

1.5.3 Overfitting and Regularization

We now know how to measure the degree of overfitting by approximately computing the difference between the expected cost and the empirical cost. In this section, let us think of how we can use this in more detail. Let us start from the "Example 1: Model Selection" from above.

When we select a model, the first question that needs to be answered is what are our hypothesis sets. An obvious approach is to choose each hypothesis set to include all possible configurations of one model family, such as perceptron, logistic regression or support vector machine. Instead, we can also decide on a model family, and subdivide the gigantic hypothesis sets into several subsets. The latter is one we will discuss further here.

Let us consider the cost function of support vector machines from Eq. (1.22):

$$J_{\text{svm}}(M, D_{\text{tra}}) = \frac{C}{2} \underbrace{\|\mathbf{w}\|^2}_{\text{(a) Regularization}} + \frac{1}{|D_{\text{tra}}|} \sum_{(y,x) \in D_{\text{tra}}} D_{\text{hinge}}(y, \mathbf{x}; M).$$

The term (a) controls how much our separating hyperplane ($\mathbf{w}^\top \tilde{\mathbf{x}} = 0$) may deviate from a flat line ($\mathbf{w} = 0$). What does it mean? Let us now look at the gradient of the cost function:

$$\nabla_{\mathbf{w}} J_{\text{svm}} = \underbrace{C\mathbf{w}}_{(a)} + \frac{1}{|D_{\text{tra}}|} \sum_{(y,x) \in D_{\text{tra}}} \nabla_{\mathbf{w}} D_{\text{hinge}}(y, \mathbf{x}; M).$$

The first term corresponds to pulling the weight vector $\mathbf{w}$ closer to an all-zero vector, effectively disallowing the weight vector to move too far away from a vector with small values. The degree to which this *pull* toward a *simple* setting is determined by the *regularization coefficient* C. When $C = 0$, there is no force pulling the weight vector toward a small vector, while with a large C, this pulling force is greater.

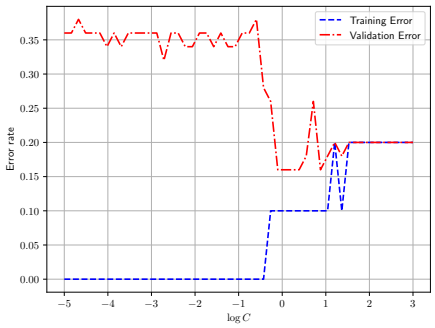


Figure 1.2: Training and test errors with respect to the weight decay coefficient. Notice that the test error grows back even when the training error is 0 as the weight decay coefficient decreases.

Based on this observation, we can now define a smaller hypothesis set $H_{C'}$, which is a subset of the larger hypothesis set of all support vector machines, as all the support vector machines reachable by minimizing the cost function of a support vector machine when the *regularization coefficient* C is set to C'. In other words, given a set of predefined coefficients $\{C_1, C_2, \ldots, C_M\}$, we can construct as many hypothesis sets $H_{C_1}, \ldots, H_{C_M}$.

We find a support vector machine that minimizes the cost function J_{svm} for each of these hypothesis sets. Then, as we discussed earlier in Sec. 1.5.2, we choose one of them based on the validation cost which in this case should omit the regularization term $\frac{C}{2}\|\mathbf{w}\|^2$.

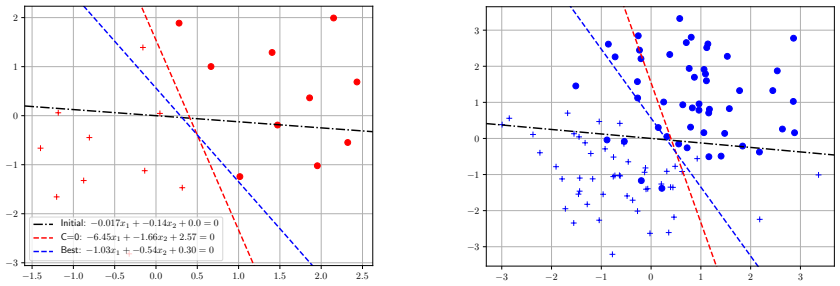
Two questions naturally arise here. First, why don't we estimate the regularization coefficient C just like we did with the weight vector $\mathbf{w}$? In other words, is it possible to simply find $\hat{C}$ such that

$$\hat{C} = \arg\min_{C} J_{\text{svm}}(M, D_{\text{tra}})?$$

It is because we are not supposed to use the training set to estimate such a regularization coefficient C. This would be equivalent to selecting a hypothesis set based on the training set, which we have learned *not* to do.

Second, why do we omit this regularization term when selecting among trained support vector machines each belonging to a different hypothesis set? Because at the end of the day, what we are really interested in is how well our classifier *classifies* unseen input vectors, which is precisely what the 0-1 loss in Eq. (1.18) measures. This is however not a universal practice, and you should choose how each hypothesis set, or the machine found within it, is scored based on your actual constraints. For instance, if an intrusion detection system can only run a low-end processor due to the power consumption constraint, you may have to *down-weight* the machines you've found from hypothesis sets with high computational requirement.

Example In this example there are 20 training pairs and 100 validation pairs. We search for the best regularization coefficient, or corresponding hypothesis sets, over the set of 50 equally-spaced $\log C$ from -5 to 3. We plot how the training and validation



(a) Training Set (b) Validation Set

Figure 1.3: 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.

errors (0-1 loss) change with respect to the regularization coefficient C (or equivalently $\log C$) in Fig. 1.2. It is clear from the plot that as the regularization strength weakens ($\log C \to -\infty$) the training error drops to zero. On the other hand, the validation error decreases until $\log C = 0$, but from there on, increases, which is a clear evidence of overfitting.

In Fig. 1.3, we see the difference between the support vector machines found using $C = 0$ (no regularization, red dashed line) and $\log C = 0$ (best regularization according to the validation error, blue dashed line). The red dashed decision boundary, which corresponds to the support vector machine without any regularization, classifies all the training input vectors perfectly, while the blue dashed decision boundary fails to classify one training input vector near $(-0.1, 1.3)$ correctly. However, when we consider the validation input vectors, the picture looks different. A better machine is now the regularized support vector machine.

1.6 Multi-Class Classification

So far, we have considered a *binary* classification only. In reality, there are often more than two categories to which an input vector belongs. It is indeed interesting to build a machine that can tell whether an object of a particular type, such as a dog, is in a given image, but we often want our machine to be able to classify an object in a given image into one of many object types. That is, we want our machine to answer "what is in an image?" rather than "is a dog in an image?" A slightly different formulation of the same question is "which of the following animals does this image describe, dog, cat, rabbit, fish, giraffe or tiger?" This kind of problem is a *multi-class classification*. Instead of two possible categories as in binary classification, now each input vector may belong to one of more than two categories.

Let us start from logistic regression in Sec. 1.3. We already learned that the logistic regression classifier outputs a Bernoulli distribution over two possible categories–

negative and positive. We extend it so that the logistic regression classifier is now returning a distribution over K-many possible categories

$$\mathscr{C} = \{0, 1, \ldots, K\}. \tag{1.25}$$

First, we must decide what kind of distribution, other than Bernoulli distribution, we should use. In this case of multi-class classification, we use a categorical distribution. The categorical distribution is defined over a set of K events (equivalent to K categories in our case) with a set of L probabilities $\{p_1, \ldots, p_K\}$. p_k is the probability of the k-th event happening, or the input vector belonging to the k-th category. As usual with any other probability, those probabilities are constrained to sum to 1, i.e.,

$$\sum_{k=1}^{K} p_k = 1.$$

Now given an input vector $\mathbf{x}$, we should let our new multi-class classifier output this categorical distribution. This is equivalent to building a machine that takes as input a vector $\mathbf{x}$ and outputs K probabilities that sum to 1. In order to do so, we turn the d-dimensional input vector into a K-dimensional vector by

$$\mathbf{a} = \mathbf{W}\tilde{\mathbf{x}},$$

where $\tilde{\mathbf{x}}$ is as before $[\mathbf{x}; 1]$. $\mathbf{W}$, to which we refer as a weight *matrix*, is a $K \times d$-dimensional matrix. Do you see how it has changed from a weight vector earlier to a weight matrix now?

We have K real numbers in $\mathbf{a}$. These numbers however are not constrained, meaning that their sum is not 1, and that some of them may even be negative. Let us now turn this K numbers into K probabilities of a categorical distribution. First, we make them positive by exponentiating each of them separately. That is,

$$\mathbf{a}^+ = \exp(\mathbf{a}) > 0.$$

Then, we force those K positive numbers to sum to 1 by

$$\mathbf{p} = \frac{1}{\sum_{k=1}^{K} a_k^+} \mathbf{a}^+. \tag{1.26}$$

That was easy, right? This transformation–exponentiation followed by normalization–is called *softmax* [**?**].

Hypothesis Set This new machine, often called *multinomial logistic regression*, is fully characterized by the weight matrix $\mathbf{W}$. In other words, our hypothesis set is a set of all K-by-d real-valued matrices.

Distance We define the distance function similarly to how we did with logistic regression. That is, it is the negative log-probability of the correct category returned by the reference machine M^*:

$$D(M^*(\mathbf{x}), M, \mathbf{x}) = -\log p(C = M^*(\mathbf{x})|\mathbf{x}) = -\log p_{M^*(\mathbf{x})}. \tag{1.27}$$

I used $p_{M^*(\mathbf{x})}$ to denote the $M^*(\mathbf{x})$-th probability value stored in $\mathbf{p}$, which is by our definition of the machine the probability of the correct category predicted by our machine. Let's expand it a bit further:

$$\begin{aligned} D(y^*, M, \mathbf{x}) &= -\log p_{M^*(\mathbf{x})} \\ &= -a_{y^*} + \log \sum_{k=1}^{K} \exp(a_k). \end{aligned}$$

Gradient of the Distance As we have done so with logistic regression and support vector machines earlier, we need to compute the gradient of the distance function in Eq. (1.27) with respect to the weight matrix.[22]

We will do this for each row of the weight matrix separately. First, we consider the y^*-th row vector, that corresponds to the correct class outputted by the reference machine:

$$\begin{aligned} \frac{\partial D(y^*, M, \mathbf{x})}{\partial \mathbf{w}_{y^*}} &= -\frac{\partial}{\partial \mathbf{w}_{y^*}} \left(\mathbf{w}_{y^*}^\top \tilde{\mathbf{x}} - \log \sum_{k=1}^{K} \exp(a_k) \right) \\ &= -(1 - p(C = y^* | \mathbf{x}))\tilde{\mathbf{x}}. \end{aligned}$$

Similarly, we can compute the gradient of the distance function with respect to the weight vector that corresponds to any other incorrect class $y \in \{1, \ldots, K\} \setminus \{y^*\}$:

$$\begin{aligned} \frac{\partial D(y^*, M, \mathbf{x})}{\partial \mathbf{w}_{y}} &= -\frac{\partial}{\partial \mathbf{w}_{y}} \left(\mathbf{w}_{y}^\top \tilde{\mathbf{x}} - \log \sum_{k=1}^{K} \exp(a_k) \right) \\ &= -(0 - p(C = y | \mathbf{x}))\tilde{\mathbf{x}}. \end{aligned}$$

We can combine them together into a single vector equation:

$$\nabla_{\mathbf{W}} D(y^*, M, \mathbf{x}) = -(\mathbf{y}^* - \mathbf{p})\,\tilde{\mathbf{x}}^\top,$$

where

$$\mathbf{y}^* = \begin{bmatrix} 0, \\ \vdots, \\ 1, \\ \vdots, \\ 0 \end{bmatrix} \leftarrow y^*\text{-th row} \tag{1.28}$$

is an *one-hot vector* corresponding to a desired output, and $\mathbf{p}$ is the actual output from the multinomial logistic regression from Eq. (1.26).

This equation above reminds us of the learning rule of logistic regression (and naturally that of perceptron.) See for instance Eq. (1.14) as a comparison. Both rules (logistic regression and multinomial logistic regression) have a multiplicative term in

[22] The full derivation is left for you as a **homework assignment**.

the front, and that multiplicative term is a difference between the predicted output (or the predicted conditional distribution over the categories given an input vector) and the desired output generated by the reference machine.

For the row vector of the weight matrix corresponding to the correct category y^*, this learning rule will add the input vector (augmented with an extra one) to the this vector so that they would align better. On the other hand, for any other category, the learning rule will subtract the input vector instead to make them less aligned. The degree to which the input vector is subtracted is decided based on how well the reference machine and our machine agree. Learning terminates, when the multinomial logistic regression puts all the probability mass (1) to the correct class.

1.7 What does the weight vector tell us?

Before we move on to more advanced topics, let us briefly discuss about what the weight vector or matrix tells us. In a standard setting of binary classification, each component x_j of an input vector $\mathbf{x}$ has a corresponding weight value w_j. When this associated weight value is close to 0, what does it mean? It means that this j-th component does not matter! This is easy to verify by looking at the score function we defined in Eq. (1.17) which can be rewritten as

$$s(y, \mathbf{x}; M) = y\mathbf{w}^\top \tilde{\mathbf{x}} = y\left(\sum_{j=1}^{d} w_j x_j + w_{d+1}\right).$$

Let's consider the k-th component of the input vector:

$$s(y, \mathbf{x}; M) = y\left(\sum_{j=1}^{k-1} w_j x_j + w_k x_k + \sum_{j'=k+1}^{d} w_{j'} x_{j'} + w_{d+1}\right),$$

which is equivalent to the equation below, if $w_k = 0$.

$$\begin{aligned} s(y, \mathbf{x}; M) =& y\left(\sum_{j=1}^{k-1} w_j x_j + \underbrace{w_k x_k}_{=0 \text{ if } w_k = 0} + \sum_{j'=k+1}^{d} w_{j'} x_{j'} + w_{d+1}\right) \\ =& y\left(\sum_{j=1}^{k-1} w_j x_j + \sum_{j'=k+1}^{d} w_{j'} x_{j'} + w_{d+1}\right). \end{aligned}$$

This is as if our input vector never had the k-th component to start with.

Along the same line of reasoning, we can see that the magnitude of each weight value $|w_k|$ roughly corresponds to how sensitive the output of a machine to the change in the value of the k-th component of the input vector x_k. Can we make it slightly more precise by defining the sensitivity more carefully? Indeed, we can. The sensitivity of the output of our machine, $\mathbf{w}^\top \tilde{\mathbf{x}}$ with respect to a single input component x_k is precisely the definition of the partial derivative of the output with respect to the input component.

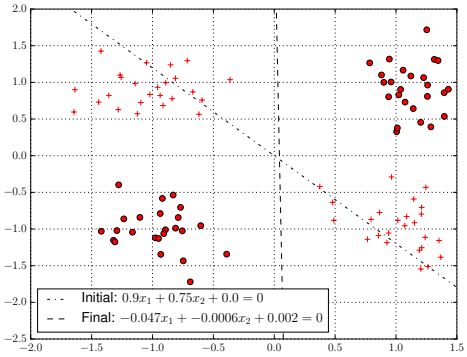


Figure 1.4: If the problem is not linearly separable as in the case shown, a linear classifier, such as perceptron, fails miserably. This is a famous example of a exclusive-or (XOR) problem (with noise.)

That is,

$$\begin{aligned}\frac{\partial \mathbf{w}^\top \tilde{\mathbf{w}}}{\partial x_k} &= \frac{\partial}{\partial x_k}\left(\sum_{j=1}^{k-1} w_j x_j + w_k x_k + \sum_{j'=k+1}^{d} w_{j'} x_{j'}\right) \\ &= \frac{\partial w_k x_k}{\partial x_k} \\ &= w_k.\end{aligned}$$

This definition of sensitivity via partial derivative will become handy in the later part of the course.

In other words, we can understand which components of the input vector are meaningful for or have high influence on the output of our machine by inspecting the weight vector. For an example of inspecting the weight vector, or matrix in the case of multi-class classification, see `https://github.com/nyu-dl/Intro_to_ML_Lecture_Note/blob/master/notebook/Weight%20Analyzer.ipynb`.

TODO: Add some explanation going through the example

1.8 Nonlinear Classification

So far in this course, we have looked at a linear classifier which defines a hyperplane ($\mathbf{w}^\top \tilde{\mathbf{x}} = 0$) that partitions the input space into two partitions. Clearly this type of classifier can only solve linearly separable problems. A famous example in which a linear classifier fails is exclusive-OR (XOR) problem shown in Fig. 1.4. In this section, we discuss how to build a classifier for problems which are not linearly separable.

1.8.1 Feature Extraction

We have so far assumed that an input vector $\mathbf{x}$ is somehow given together with data. Is this assumption reasonable? Let us think of what kind of data we run into in practice. For instance, in the example of intrusion detection system from earlier sections, the

input to a machine is not a vector but a picture taken by a camera installed at the front of the store. In the case of building a machine that categorizes a document, the input to a machine is again not a vector but a long list of words. If we are building a machine for detecting violent scenes from a movie, our machine takes as input a video not a single, flat vector. What all these examples suggest is that we need one more step in addition to what we have discussed as a full pipeline of machine learning. That is the step of feature extraction, or sometimes called feature engineering.

Let us introduce another symbol $\mathscr{X}$ to denote the original input which could be anything from a colour image, video clip to a social network of a person. Then, the feature extraction stage can be thought of as a function ϕ that maps from this arbitrary original input $\mathscr{X}$ to a corresponding input vector $\mathbf{x}$. That is,

$$\mathbf{x} = \phi(\mathscr{X}).$$

Why is this process called *feature extraction*? That is because the function ϕ can be thought of as extracting d-many characteristics of the original input $\mathscr{X}$. We extract features out of a given input $\mathscr{X}$ and build the corresponding input vector $\mathbf{x} \in \mathbb{R}^d$.

Example: Bag-of-Words Representation Let us consider an example of document categorization we learned about earlier. What is a property of a given document that largely determines the category or topic of the document? One thing that immediately comes to our mind is the existence of category-related words. For instance, if a word "hockey" is mentioned in the document, it is highly likely that it is a document about sports, and more specifically about hockey rather than baseball. Perhaps, it is also important how frequently such a word appeared in the document. If the word "baseball" appeared ten times more than the word "baseball", the topic of the document is likely "hockey" rather than "baseball", even though the word "baseball" appeared.

This observation leads us to use a so-called bag-of-words (BoW) feature representation of a document for document categorization. As the name suggests, this representation puts all the words in a given document into a bag and counts how often each word appeared in the document, ignoring any order among those words. This is equivalent to turning each word into a one-hot vector from Eq. (1.28) and sum them into a single vector. In order to do so, we will first build a vocabulary of all unique words in all the document from a training set (again, do not touch any test document!), which is similar to building a category set from Eq. (1.25). We then transform a document into a sequence of one-hot vectors w_i's, and sum all those one-hot vector to obtain a bag-of-words vector:

$$\mathbf{x} = \sum_{i=1}^{|\mathscr{X}|} w_i,$$

where $|\mathscr{X}|$ denotes the length, or the number of words, of the document $\mathscr{X}$. This BoW vector $\mathbf{x}$ can be used with any machine learning algorithm, such as any of the classifiers we have learned so far.

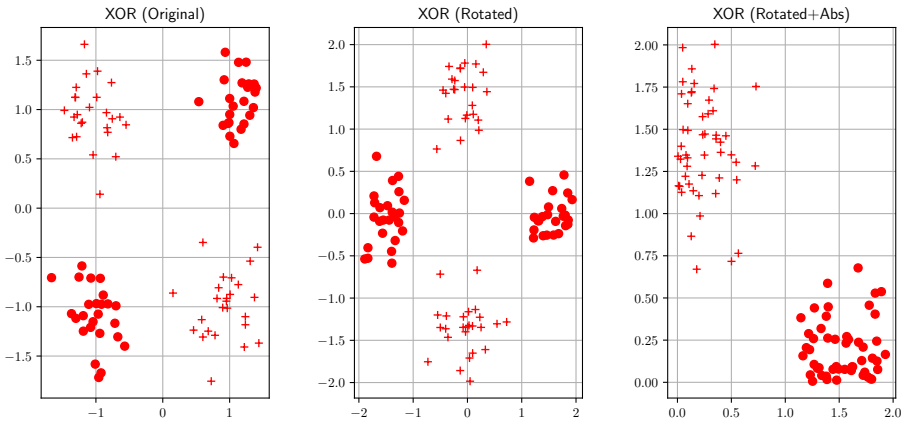


Figure 1.5: The XOR problem in the original coordinate system (left) is not linearly separable, but as shown in the right panel, it become linearly separable by transforming the space (center and right). See the main text for more details.

Linear Separable Feature Extraction Feature extraction serves two purposes. The first purpose is to build a fixed-dimensional vector $\mathbf{x}$ from an arbitrary input $\mathscr{X}$, of which an example was to build a bag-of-word vector from a document of arbitrary length. The second purpose is to make a given dataset *easier* for a classifier, or any other *linear* machines. More specifically for the case of classifiers we have learned so far, the goal of feature extraction is to make a dataset *linearly separable*, even when it is not so in the original input $\mathscr{X}$ space.

Let us go back to the example of XOR problem from earlier. In its original form, the XOR problem is not solvable by a linear classifier, because the positive and negative classes are not linear separable, meaning that there is no 1-D hyperplane (line) that separates the examples into the positive and negative classes. The question is then: can we somehow transform the original input–a vector in a 2-D space– so that the transformed data is linear separable?

First, let us rotate the whole space, or every single point in the space, clock-wise by 45 degrees. This can be done by first defining a rotation matrix as

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix},$$

where θ is in radian ($\text{rad}(45^\circ) \approx 0.785$). We then rotate each point in the space by

$$\mathbf{x}^r = R(\text{rad}(45^\circ))\mathbf{x}.$$

The XOR problem after this rotation is illustrated in the center panel of Fig. 1.5.

The problem is however not linearly separable yet. Let us then further apply one more transformation. That is, we will take the absolute value of each element of the

resulting, rotate vector:

$$\mathbf{x}^{ra} = \begin{bmatrix} |x_1^r| \\ |x_2^r| \end{bmatrix}.$$

Effectively, we fold the four quadrants of the rotated space into the first quadrant (the top-right one), and the resulting space is now linearly separable as shown in the right panel of Fig. 1.5. Within this new transformed space, the XOR problem, which was not linearly separable, is now linearly separable, and we can use any of the linear classification machines we have learned earlier.

This is a great news! Apparently, we can find a set of features–the elements in an input vector $\mathbf{x}$– that turn a problem, which is not linearly separable in its original form, into a linearly separable one. As long as we can find such a transformation, or a sequence of them, we are pretty much solve any classification problem.

Unfortunately, it is often impossible to find such a transformation manually, because the original inputs or the original feature vectors are almost always high-dimensional. In the remainder of this section, we will discuss how we can automate such a procedure.

1.8.2 k-Nearest Neighbours: Fixed Basis Networks

Let us consider another transformation for the XOR problem above. We start with selecting four points in the space that correspond to

$$\begin{aligned} \mathbf{r}^1 &= [-1,-1] \\ \mathbf{r}^2 &= [1,1] \\ \mathbf{r}^3 &= [-1,1] \\ \mathbf{r}^4 &= [1,-1] \end{aligned}$$

to which we refer as *basis vectors*. With these basis vectors, we will transform each two-dimensional input vector $\mathbf{x}$ into a four-dimensional vector $\phi(\mathbf{x})$ such that

$$\phi_i(\mathbf{x}) = \exp\left(-(\mathbf{x}-\mathbf{r}^i)^2\right). \tag{1.29}$$

This is equivalent to saying that the i-th element of the transformed vector $\phi(\mathbf{x})$ is inversely proportional to the distance between the input vector $\mathbf{x}$ and the i-th basis vector.

This function is often called a (Gaussian) *radial basis function*. This function's output is bounded between 0 and 1. The output is closer to 1, when the input vector $\mathbf{x}$ is close to the basis vector $\mathbf{r}$, but converges to 0 as the distance between them grows.

In this newly transformed space, the XOR problem is linearly separable. How do we confirm this? We can either train a linear classifier, such as the ones we have learned so far in the course, or manually find a weight vector $\mathbf{w} \in \mathbb{R}^5$ that solves the problem perfectly. Let us try the latter, based on the intuition we built from Sec. 1.7.

We first observe that any input vector that is close to one of the first two basis vectors $\mathbf{r}^1$ and $\mathbf{r}^2$ should be classified as a positive class, and an input vector closer to

either $\mathbf{r}^3$ or $\mathbf{r}^4$ should be classified as a negative class. In other words, any positive input vector would have either the first or second element of the transformed vector close to 1, while any negative input vector close to 0. For the third and fourth elements, they would have a value close to 1, if the original input vector is negative, and close to 0 otherwise.

Based on this observation, we can easily notice that the following weight vector will perfectly solve the problem:[23]

$$\mathbf{w} = [1, 1, -1, -1, 0]^\top .$$

The weight vector above can be written alternatively as

$$\mathbf{w} = \left[y^1, y^2, y^3, y^4, 0\right]^\top , \tag{1.30}$$

where y^i is the class label (-1 or 1) of the i-th basis vector.[24] In other words, the input vector $\mathbf{x}$ belongs to the class to which the *nearest* basis vector belongs.

Let us generalize this idea by assuming that we have K basis vectors and their corresponding labels:

$$\left\{(\mathbf{r}^1, y^1), \ldots, (\mathbf{r}^K, y^K)\right\}.$$

Each input vector $\mathbf{x}$ is transformed into

$$\phi(\mathbf{x}) = \begin{bmatrix} \exp\left(-(\mathbf{x} - \mathbf{r}^1)^2\right) \\ \vdots \\ \exp\left(-(\mathbf{x} - \mathbf{r}^K)^2\right) \end{bmatrix} \tag{1.31}$$

In the case of binary classification, the optimal weight vector is given in Eq. (1.30). In multi-class classification, the weight matrix $\mathbf{W}$ can be constructed as

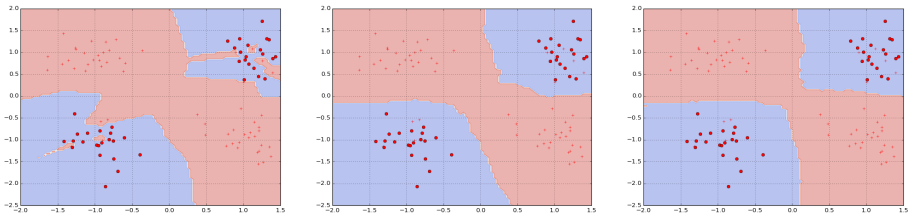
$$\mathbf{W} = \left[\mathbf{y}^1, \ldots, \mathbf{y}^K\right],$$

where $\mathbf{y}^i$ is the one-hot vector corresponding to the class to which the i-th basis vector belongs (see Eq. (1.28).) Again, it is left for you as a **homework assignment** to show that this construction of the weight matrix solves the problem of multi-class classification.

Suddenly the problem of finding a linearly separable transformation has become a problem of finding a set of good basis vectors and their own classes. Of course, now a big question is what these good basis vectors. An even bigger question is how we know to which class each of those basis vectors belongs. After all, the whole point of classification is to figure out this latter question.

[23] It is a **homework assignment** for you to show that this weight vector solves the XOR problem in the new space.

[24] It is your **homework assignment** to find such a weight vector when the class labels are given as 0 and 1.



(a) $k = 1$ (b) $k = 5$ (c) $k = 20$

Figure 1.6: The effect of varying k in k-nearest-neighbour classifier. We observe that the decision boundary becomes *smoother* as k increases, which suggests that a large k corresponds to having a stronger regularization term.

k-Nearest Neighbours We can push this idea to the extreme by declaring each and every input vector in a training set as basis vectors. Furthermore, instead of building a linear classifier in the transformed space (using all the training input vectors as basis vectors,) we can simply use the class label of the nearest basis vector, or more conventionally *nearest neighbour*.

This can be written down more formally by first defining as many basis vectors as there are training examples. That is,

$$\mathbf{r}^i = \mathbf{x}_i,$$

where $\mathbf{x}^i$ is the i-th input vector in a training set. Then, each input vector is transformed in two stages. First, we use the radial basis function to get $\phi(\mathbf{x})$ as done in Eq. (1.29). Second, we turn the resulting vector into an one-hot vector by setting the element with the largest value to 1 and all the others to 0:

$$\phi'(\mathbf{x}) = \arg\max(\phi(\mathbf{x})).$$

The weight matrix is constructed as before in Eq. (1.31). In practice, none of these formal steps is necessary. All that is needed is to find the nearest neighbour from a training set given an arbitrary (test) vector, and return the class of the nearest neighbour. We call this classifier a *nearest-neighbour classifier*.

The nearest-neighbour classifier can be written down in a single equation:

$$\underset{(\mathbf{x},y)\in D_{\text{tra}}}{\arg\min} \|\mathbf{x} - \mathbf{x}'\|^2,$$

where $\mathbf{x}'$ is a new input vector of which label must be found. Note that it is not necessary to use the Euclidean distance $\|a-b\|^2$, and any distance function may be used instead.

The nearest-neighbour classifier is rarely used in practice. Instead, it is more common to use its variant, called a *k-nearest-neighbour* (KNN) classifier. Unlike the nearest-neighbour classifier, the KNN classifier selects k nearest input vectors from a training set given a new input vector, and lets them vote on which category this new vector belongs to. The nearest-neighbour classifier is thus a special case of the KNN classifier, where k is fixed to 1.

Increasing k has the effect of regularization, similarly to the weight decay regularization term, or the max-margin regularization term from support vector machines (see Eq. (1.22) (a).) When $k = 1$, the empirical cost on a training set is perfect by definition, but this nearest-neighbour classifier is susceptible to outliers or noise. Imagine a case where one negative input vector was accidentally placed in the middle of a cluster[25] of positive input vectors. The nearest-neighbour classifier would assign any input vector in a small region around this negative input vector to a negative class, although it is quite clear that this training example is an outlier, or was labelled incorrectly. This behaviour, which is a classical example of overfitting, is easily mitigated by using $k > 2$, as this would ignore such an outlier training example. On the other hand, we can also think of a case where $k = |D_{\text{tra}}|$, in which case any new input vector would be assigned to a majority class, and such a KNN classifier wouldn't be able to correctly classify any training input vector belonging to a minority class. The latter case is considered *over-regularized* or *under-fitted*.

1.8.3 Radial Basis Function Networks

The most obvious weakness of the KNN classifier is that it requires (a) a large storage (since it must maintain the entire training set,) and (b) a sweep through the entire training set (unless some smart indexing with an appropriate approximation strategy is used.) This becomes more severe, as the size of a training set grows (which is precisely what is happening everyday,) and if there is a computational constraint in run-time (such as when run on a mobile phone.)

A radial basis function (RBF) network overcomes this weakness of the KNN classifier by selecting only a small number of basis vectors, according to memory and computational constraints. Of course, as we discussed earlier, how should we choose such basis vectors?

There are two widely-used approaches, when there is a constraint on the number of basis vectors. Let this constraint be K. The first approach is rather dumb in that it uniform-randomly selects K training input vectors without replacement:

1. Uniform-randomly select an index i from $\{1, \ldots, |D|\}$
2. $B \leftarrow B \cup \{\mathbf{x}_i\}$
3. $D \leftarrow D \setminus \{\mathbf{x}_i\}$
4. Go back to 1, if $|B| < K$.

where D is initialized with a training set D_{tra}, and B is a set of basis vectors initialized to be an empty set. This approach works surprisingly well, because (a) no basis vector is too far away from the training examples, and (b) uniform-randomly sampling ensures that the selected basis vectors are evenly distributed across the region occupied by the training examples. In this random-sampling approach, we know precisely what the correct label, or that predicted by a reference machine, of each basis vector is. This

[25] A cluster refers to a group of closely located input vectors.

allows us to set the weight vector, or weight matrix, exactly, as we have done with the nearest-neighbour classifier in Eq. (1.31).

The other approach is to find a set of clusters of training input vectors and pick the centroids[26] of these clusters as basis vectors. In the case of the XOR problem in Fig. 1.5, there are four clusters centered at (1,1), (-1,1), (-1,-1) and (1,-1), respectively. Hence we would take these four centroids $[1,1]$, $[-1,1]$, $[-1,-1]$ and $[1,-1]$ as basis vectors. This ensures that there is at least one basis vector for any group of training input vectors, and this guarantee is much stronger than we get with the random-sampling approach. Despite this nice property, we are now faced with two additional issues.

First, how do we get those clusters? In the case of low-dimensional input vectors (e.g., 2-D or 3-D), it may be possible for us to visually inspect the input vectors to manually select clusters. This however becomes implausible when the dimensionality d of the input vector grows beyond 3. We then resort to automatic clustering which is another class of algorithms in machine learning. We will learn about automatic clustering later in the course.

When we have found clusters and their centroids, we have the second question to answer. That is, how should we set the weight vector, or matrix, without having the correct labels for these basis vectors? Unlike the random-sampling approach, or the nearest-neighbour classifier (which is the special case of random-sampling approach with $K = |D_{\text{tra}}|$), the centroids are not necessarily included in a training set, and thereby, are without correct labels. Fortunately we already have a solution to this problem. In fact we have been learning how to answer this problem this entire semester.

Let us assume that we have K basis vectors $\left\{\mathbf{r}^1, \mathbf{r}^2, \ldots, \mathbf{r}^K\right\}$ corresponding to the centroids of K clusters. With these basis vectors, we now transform each and every input vector in a training set into a K-dimensional vector:

$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} \exp\left(-(\mathbf{x}-\mathbf{r}^1)^2\right) \\ \exp\left(-(\mathbf{x}-\mathbf{r}^2)^2\right) \\ \vdots \\ \exp\left(-(\mathbf{x}-\mathbf{r}^K)^2\right) \end{bmatrix} \in \mathbb{R}^K.$$

This transformation of every training input vector is equivalent to building a new training set consisting of pairs of a transformed input vector and its correct class. That is,

$$D_{\text{tra}} \leftarrow \left\{(\boldsymbol{\phi}(\mathbf{x}_1), y_1), \ldots, (\boldsymbol{\phi}(\mathbf{x}_N), y_N)\right\}.$$

Once this transformation is done, we can find the $K+1$-dimensional weight vector, or $(K+1) \times |\mathscr{C}|$-dimensional matrix, using any of the techniques we have learned earlier, including perceptron, logistic regression, support vector machines and multinomial logistic regression.

In other words, we have now learned how to use the linear classifiers we have learned so far for problems that are *not* linearly separable. We first find a set of basis vectors based on which each input vector is transformed using a radial basis function from Eq. (1.29), and fit a linear classifier on a new training set with these transformed

[26] A centroid is the average of all the input vectors in the corresponding cluster.

input vectors. A nonlinear classifier constructed in this way is often referred to as a *radial basis function network* (RBFN). Also, as both kNN and RBFN use a *fixed* set of basis vectors, we call this family of classifiers a *fixed basis network*.

1.8.4 Adaptive Basis Function Networks or Deep Learning

Adaptive Basis Function Networks A natural question that follows our discussion on the fixed basis network is whether it is necessary to *fix* basis vectors. Perhaps a more fundamental question would be how we know that those fixed basis vectors we have selected are good for the final classification accuracy. Is it possible that there is a better strategy for selecting basis vectors than the ones we have discussed already? Is it possible that this new strategy selects basis vectors not based on our intuition but based on the actual classification accuracy?

Let us go back to logistic regression and consider its distance function from Eq. (1.32):

$$\begin{aligned} D(M^*(\mathbf{x}), M, \mathbf{x}) &= -\log p(C = M^*(\mathbf{x})|\mathbf{x}) \\ &= -\left(M^*(\mathbf{x}) \log M(\mathbf{x}) + (1 - M^*(\mathbf{x})) \log(1 - M(\mathbf{x}))\right), \end{aligned} \tag{1.32}$$

where

$$M(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{w} + b)$$

and σ is a sigmoid function.[27] Given this distance function, we were able to derive a learning rule for logistic regression by computing its gradient with respect to the weight vector (and a bias scalar).

With K basis vectors, we can rewrite this distance function as

$$D(M^*(\phi(\mathbf{x})), M, \phi(\mathbf{x})) = -(M^*(\phi(\mathbf{x})) \log M(\phi(\mathbf{x})) + (1 - M^*(\phi(\mathbf{x}))) \log(1 - M(\phi(\mathbf{x})))),$$

where

$$\phi(\mathbf{x}) = \begin{bmatrix} \exp\left(-(\mathbf{x} - \mathbf{r}^1)^2\right) \\ \vdots \\ \exp\left(-(\mathbf{x} - \mathbf{r}^K)^2\right) \end{bmatrix}$$

from Eq. (1.31). In this rewritten form, we notice that we can compute the gradient of the distance with respect not only to the weight vector ($\mathbf{w}$ and b), but also to each and every basis vector. That is, we can compute

$$\nabla_{\mathbf{r}^k} D(M^*(\phi(\mathbf{x})), M, \phi(\mathbf{x})).$$

Let's compute this gradient:

$$\begin{aligned} \nabla_{\mathbf{r}^k} D(y^*, M, \phi(\mathbf{x})) &= \underbrace{-(y^* - \sigma(\mathbf{w}^\top \phi(\mathbf{x}) + b))}_{\frac{\partial D}{\partial a}} \underbrace{w_k}_{\frac{\partial a}{\partial \phi_k(\mathbf{x})}} \underbrace{(2\phi_k(\mathbf{x}))(\mathbf{x} - \mathbf{r}^k)}_{\nabla_{\mathbf{r}^k} \phi_k(\mathbf{x})} \\ &= -2(y^* - \sigma(\mathbf{w}^\top \phi(\mathbf{x}) + b)) w_k \phi_k(\mathbf{x})(\mathbf{x} - \mathbf{r}^k), \end{aligned}$$

[27] I have split the weight vector into two components; (a) the weight vector $\mathbf{w}$ and (b) the bias b, as this is a more standard notation in deep learning.

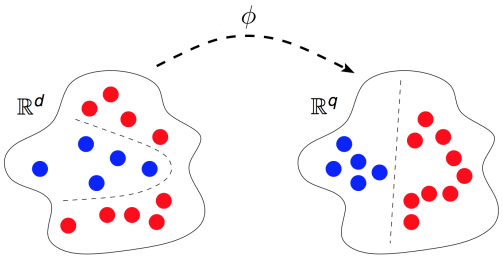


Figure 1.7: The goal of adaptive basis networks is to find a parametrized mapping from the original input space $\mathbf{x} \in \mathbb{R}^d$ to another space $\boldsymbol{\phi}(\mathbf{x}) \in \mathbb{R}^q$ that makes the problem linearly separable.

where $a = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}) + b$.[28] With this gradient, we now know how to adjust the k-th basis vector $\mathbf{r}^k$ to reduce the distance given a training example $(\mathbf{x}, y^*)$:

$$\mathbf{r}^k \leftarrow \mathbf{r}^k - \eta \nabla_{\mathbf{r}^k} D(y^*, M, \boldsymbol{\phi}(\mathbf{x})).$$

This is just like how we have learned to update the weight vector to minimize the distance earlier for the linear classifiers. Unlike those learning rules, however, this learning rule is not easily interpretable. For instance, when should the basis vector $\mathbf{r}^k$ become similar to the input vector $\mathbf{x}$? And, how much should it be adjusted toward or against $\mathbf{x}$? How does this value correlate with the classification accuracy? Despite the fact that we cannot or are not willing to answer these questions, one thing is clear; this learning rule will adjust the k-th basis vector to reduce the distance.

What does this imply? It implies that we can use the very same technique we learned earlier for training a linear classifier for automatically adapting the basis vectors as well. As long as the gradient of the distance function could be computed with respect to any of the basis vectors, we can simultaneously update the weight vector, or matrix, and all the basis vectors to minimize the distance, thereby maximizing the classification accuracy. Even better, we do not even need to compute the gradient ourselves, but can leave this tedious job to automatic differentiation.

In practice, we use one of the two selection strategies from the previous section to *initialize* basis vectors rather than fixing them. Then, we can use any off-the-shelf optimization algorithm to *jointly* tune both the weight vector, or matrix, and all the basis vectors to minimize the empirical cost function, using the gradient computed again automatically by automatic differentiation algorithm. When the basis vectors are not anymore fixed, we call this type of a classifier an *adaptive basis function network* (ABFN).

Deep Learning A further implication of gradient-based learning is that there is absolutely no constraint that the feature extraction function $\boldsymbol{\phi}$ be a radial basis function. In fact, $\boldsymbol{\phi}$ can be any parametrized, *differentiable* function that maps from an input vector $\mathbf{x}$ to its transformation. In the case of a radial basis function, for instance, $\boldsymbol{\phi}$ is a function from $\mathbb{R}^d$ to $\mathbb{R}^K$ parametrized by a set of K basis vectors. This feature extraction function is differentiable with respect to all the K basis vectors, and this allows us to use any gradient-based off-the-shelf optimization algorithm to train the whole classifier jointly.

[28] At this point, you should already know that the full derivation is left for you as a **homework assignment**.

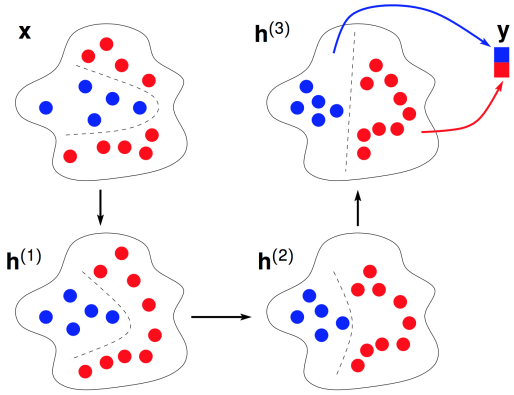


Figure 1.8: The goal of deep learning is to stack many feature extraction functions ϕ^l's on top of each other to gradually transform the problem to become linearly separable.

What is a good parametric, differentiable function ϕ? In order to answer this question, we should think of what we want this feature extraction function to do. We want this feature extraction function to make the problem *linear separable* so that the linear classifier acting on $\phi(\mathbf{x})$ can do its job well. See Fig. 1.7 for graphical illustration.

This can be done in two ways. First, we can make one large function ϕ that does the perfect job in one go. Or, we can compose a series of *simple feature extraction functions* such that each feature extraction function makes the problem slightly more linearly separable than it was beforehand. That is, we stack a series of feature extraction function $\phi^1, \dots \phi^L$ such that $\phi^L(\cdots \phi^1(\mathbf{x}))$ (or $\phi^1 \circ \cdots \circ \phi^L$) is linearly separable, as illustrated in Fig. 1.8.

What kind of simple feature extraction function should we use then, and how does the stack of such simple feature extraction functions make the problem more linearly separable? This question requires us finally to think of and understand the underlying structures or properties behind a target task (classification) and data. Based on the underlying structures, suitable feature extraction functions may be selected and composed into a *deep* feature extraction function which is often followed by a linear classifier. This composition of a linear classifier and a deep feature extraction function is jointly trained to minimize the empirical cost function using a gradient-based off-the-shelf optimization algorithm, and the field in which this type of machine learning models is studied is called *deep learning*.

There are many fascinating recent developments in deep learning, but they are out of the scope of this course. For comprehensive discussion on deep learning, I highly recommend you to read a newly published text book *Deep Learning* by Ian Goodfellow, Yoshua Bengio and Aaron Courville [**?**]. For a general overview of recent advances, see the review article [**?**]. If you are a student at New York University, you can also attend the course *Deep Learning* taught by Prof. Yann LeCun.

1.9 Kernel Support Vector Machines*

1.10 Decision Tree*

1.11 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.[1] A general setup of the problem remains largely identical to that from Sec. 1.1, meaning that it is probably a good idea to re-read that section at this point. In the context of regression, we will particularly focus on framing the whole problem as probabilistic modelling.

2.1 Linear Regression

2.1.1 Linear Regression

As we have done with classification, we will start with considering *linear* regression. In linear regression, our machine M is defined as

$$M(\mathbf{x}) = \mathbf{W}^\top \tilde{\mathbf{x}},$$

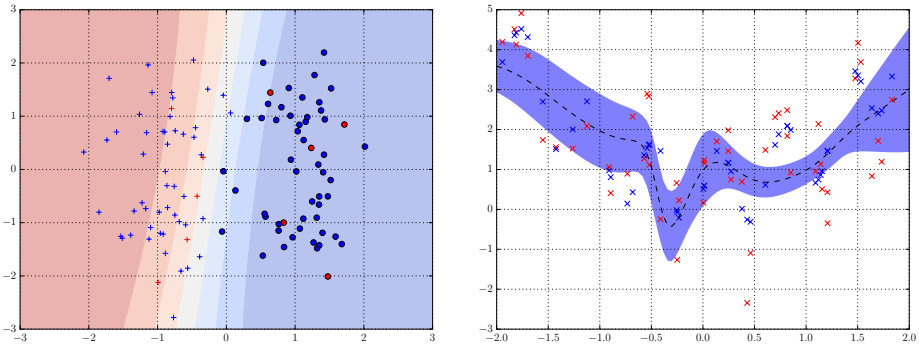
where we use $\tilde{\mathbf{x}}$ to denote the input vector with an extra 1 attached at the end. Similarly to the earlier classification problems, we are given a set of training examples:

$$D_{\text{tra}} = \{(\mathbf{x}_1, \mathbf{y}_1^*), \ldots, (\mathbf{x}_N, \mathbf{y}_N^*)\}.$$

Unlike classification, the output $\mathbf{y}_n^* \in \mathbb{R}^q$ is a q-dimensional real vector.

As should be obvious at this point, the goal of linear regression is to find a machine, or equivalently its weight vector, so as to minimize the distance between the reference machine's output and our machine's output. The reference machine's outputs are given as a part of the training set.

[1] This definition however is not universal, in that even when the output is from a finite set, the problem is sometimes called regression if there exists natural ordering of labels.



(a) (b)

Figure 2.1: (a) Bayesian logistic regression, and (b) Bayesian multilayer perceptron. Both of them were done with "ensemble samplers with affine invariance" [**?**] using Python emcee available at `http://dan.iel.fm/emcee/current/`.

Distance Functions Given two vectors, one natural way to define the distance between them is a Euclidean distance which is defined as

$$\|\mathbf{y}^* - \mathbf{y}\|_2 = \sqrt{\sum_{k=1}^{q} (y_k^* - y_k)^2}.$$

Of course, this is not the only way, and a straightforward generalization is to consider L_p norm of which Euclidean distance is a special case with $p = 2$:

$$\|\mathbf{y}^* - \mathbf{y}\|_p = \sqrt[p]{\sum_{k=1}^{q} (y_k^* - y_k)^p}.$$

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 *scores* an input vector according to how likely that vector is. If an input vector $\mathbf{x}$ is likely, the output of the machine $M(\mathbf{x})$ would be higher, and otherwise lower. The use of the term *score* should remind you of our earlier discussion on classification, and it is only natural because there is a strong connection between supervised learning and unsupervised learning. This connection is apparent when we consider $\mathbf{x}$ as a concatenation of an input vector $\mathbf{x}$ and its corresponding label y. Then, with a machine M trained by unsupervised learning, we can classify any given input vector by

$$\hat{y} = \arg\max_{y} M([\mathbf{x}; y]).$$

We can similarly perform regression in this framework.

This capability of scoring an input vector allows us to use it for *outlier detection* as well. Given an input vector $\mathbf{x}$, we will declare it as an outlier if its score is lower than a certain, predefined threshold:

$$\mathbf{x} \text{ is an outlier if } M(\mathbf{x}) < \tau.$$

The threshold τ may be found by any of the validation techniques we discussed in Sec. 1.5.2. Furthermore, we can even *generate* a novel input vector by finding an input vector that maximizes the score given by the model:

$$\hat{\mathbf{x}} = \arg\max_{\mathbf{x}} M(\mathbf{x}).$$

3.1.1 Naive Bayes Classifier

3.2 Dimensionality Reduction: Problem Setup

3.3 Principal Component Anslysis

3.3.1 Minimum Reconstruction Criterion

3.3.2 Maximum Variance Criterion

3.3.3 Probabilistic Principal Component Analysis

Expectation-Maximization Algorithm

3.4 Other Dimensionality Reduction Techniques*

Chapter 4

Clustering

Chapter 5

Sequential Decision Making

5.1 Sequential Decision Making as a Series of Classification Problems

5.2 Time-Series Modelling*