Brief Introduction to Machine Learning without Deep Learning

Kyunghyun Cho

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

January 30, 2018

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 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 [2]
- "Machine Learning: a probabilistic perspective" by Kevin Murphy [16]
- "A Course in Machine Learning" by Hal Daumé[1]

For practical exercises, Python scripts based on numpy and scipy are available online. They are under heavy development and subject to frequent changes over the course (and over the years I will be spending at NYU). 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

This course intentionally omits any recent advances in deep learning. This decision was made, because there is an excellent course "Deep Learning" taught at the NYU Center for Data Science by Yann LeCun himself. Also, every undergrad, who thinks they are interested in and passionate about machine learning, self-teaches themself deep learning, and I found it necessary for me to focus on anything that does not look like deep learning to give them a more balance view of machine learning. Of course, I have largely failed.

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

Contents

Exercises

The following exercise materials, based on python, numpy, scipy, autograd and scikit-learn, have been prepared by Varun D N.[2] Thanks, Varun! They are available at `https://github.com/nyu-dl/Intro_to_ML_Lecture_Note/tree/master/exercises`:

1. Perceptron: `Perceptron.ipynb`
2. Logistic Regression: `Logistic_Regression_1.ipynb`, `Logistic_Regression_2.ipynb`, `MNIST_Classification.ipynb`
3. Introduction to Autograd: `Autograd_Introduction.ipynb`
4. Support vector machines: `SVM.ipynb`, `SVM_vs_LogReg.ipynb`
5. Radial basis function networks: `RBFN.ipynb`
6. *k*-Nearest neighbour classifier: `KNN1.ipynb`
7. Principal component analysis: `PCA_MNIST.ipynb`, `PCA_Newsgroup20.ipynb`
8. Non-negative matrix factorization: `NMF.ipynb`, `NMF_Newsgroup20.ipynb`
9. Bayesian linear regression (requires PyMC3): `Bayesian_Linear_Regression_MCMC.ipynb`

These are well-written, but come also with some bugs. I will fix them next year, if I happen to teach the course once more.

[2] `https://www.linkedin.com/in/varundn/`

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: $\{\mathbf{x}^1,\ldots,\mathbf{x}^N\}$. 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.

1.2 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.

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

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]

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

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 (a) is -1, $\mathbf{w}^\top \tilde{\mathbf{x}}$ is positive, because $M(\mathbf{x})$ was 1, in which case the distance is again positive.

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

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

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

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

[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?

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,

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

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 [18], 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]

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.

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.

[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

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

[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**.

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

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:

- 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.

[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.

[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

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

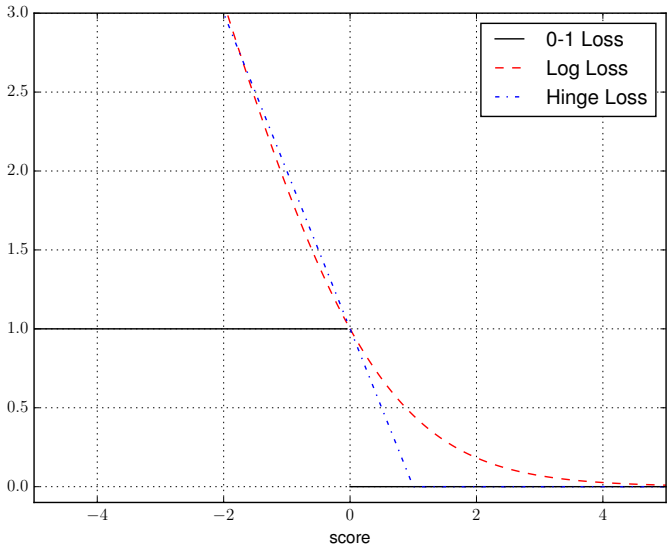


Figure 1.1: The figure plots three major loss functions–0-1 loss, log loss and hinge loss– with respect to the output of a score function s. Note that the log loss and hinge loss are upper-bound to the 0-1 loss.

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

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

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,

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

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.

[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.

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.

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

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

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.

[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**.

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

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:

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

where $C/2$ can be thought of as a regularization coefficient. This is a cost function for so-called support vector machines [7].

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:

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

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.

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

[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.

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 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 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.

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

[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.

[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.

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, 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 sub-divide the gigantic hypothesis sets into several subsets. The latter is one we will discuss further here.

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

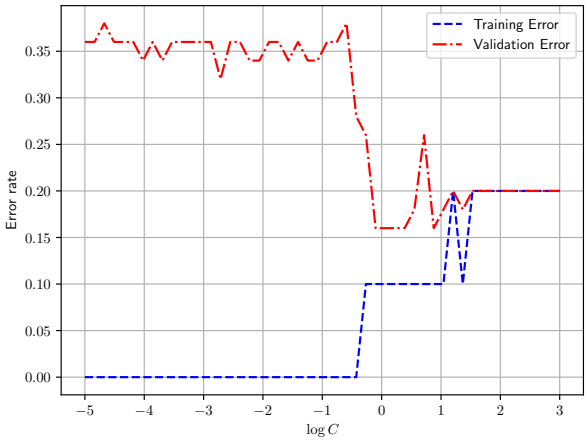


Figure 1.2: Training and test errors with respect to the weight decay coefficient. Notice that the test error grows back even when the training error is 0 as the weight decay coefficient decreases.

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.

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 $H_{C_1}, \ldots, H_{C_M}$.

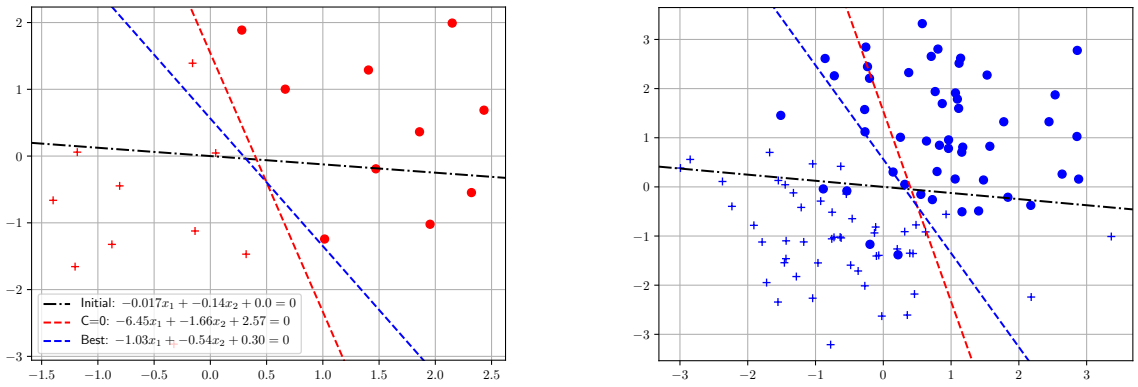
We find a support vector machine that minimizes the cost function J_{svm} for each of these hypothesis sets. Then, as we discussed earlier in Sec. 1.5.2, we choose one of them based on the validation cost which in this case should omit the regularization term $\frac{C}{2}\|\mathbf{w}\|^2$.

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.



(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.

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 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* [5].

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 $y \in \{1, \ldots, K\} \setminus \{y^*\}$:

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

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

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 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

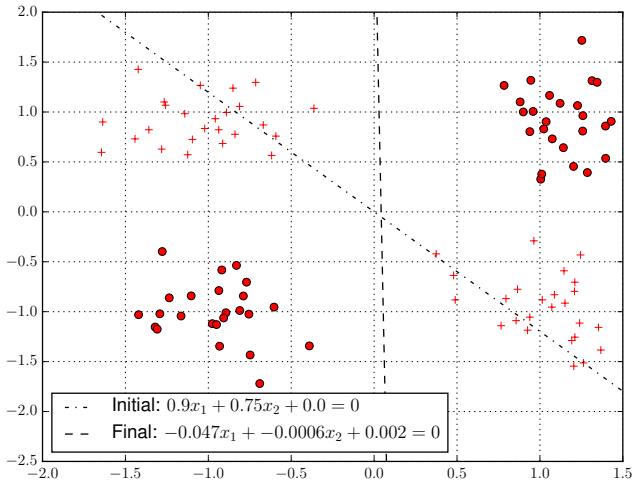


Figure 1.4: If the problem is not linearly separable as in the case shown, a linear classifier, such as perceptron, fails miserably. This is a famous example of a exclusive-or (XOR) problem (with noise.)

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. 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`.

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

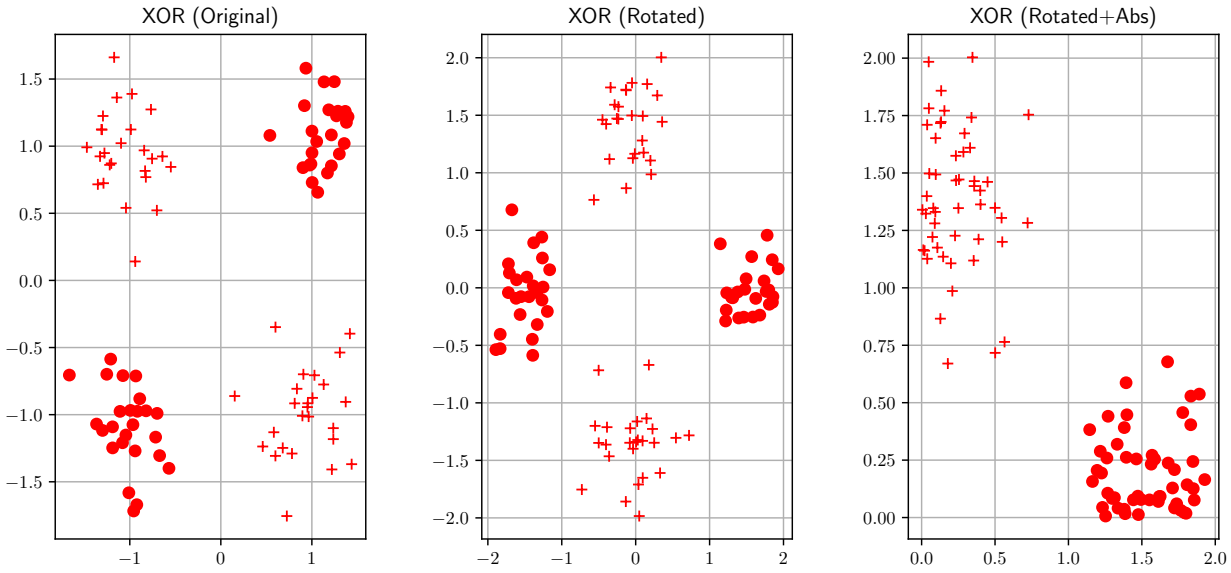


Figure 1.5: The XOR problem in the original coordinate system (left) is not linearly separable, but as shown in the right panel, it become linearly separable by transforming the space (center and right). See the main text for more details.

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. (4.1) 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.

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

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

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. 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. (4.1).) 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.

***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*.

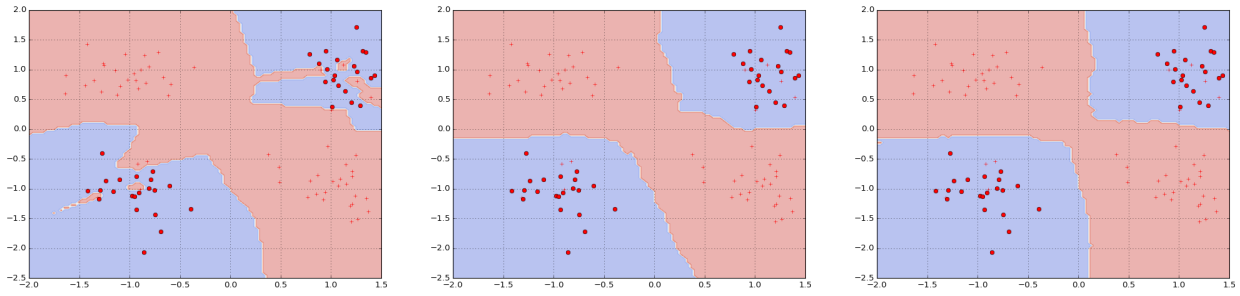
This can be written down more formally by first defining as many basis vectors as there are training examples. That is,

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

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:

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

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



(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.

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[24] 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 $\{1,\ldots,|D|\}$

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

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 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[25] 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 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

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

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.[26] 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

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

where

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

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

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

Let's compute this gradient:

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

where $a = \mathbf{w}^\top \boldsymbol{\phi}(\mathbf{x}) + b$.[27] 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).

[26] 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.

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

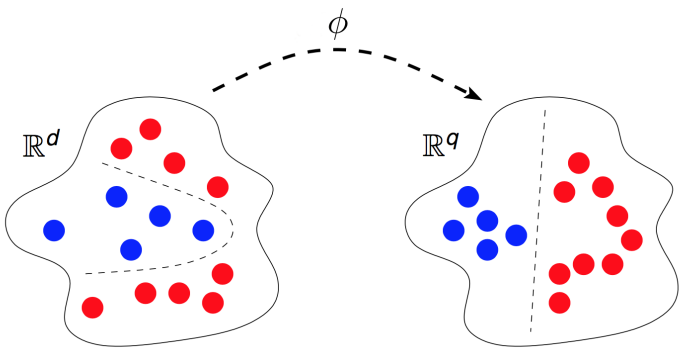


Figure 1.7: The goal of adaptive basis networks is to find a parametrized mapping from the original input space $\mathbf{x} \in \mathbb{R}^d$ to another space $\phi(\mathbf{x}) \in \mathbb{R}^q$ that makes the problem linearly separable.

Deep Learning A further implication of gradient-based learning is that there is absolutely no constraint that the feature extraction function ϕ be a radial basis function. In fact, ϕ 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, ϕ 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.

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 [8]. For a general overview of recent advances, see the review article [14]. If you are a student at New York University, you can also attend the course *Deep Learning* taught by Prof. Yann LeCun.

1.9 Further Topics*

The success of support vector machines does not necessarily come only from the use of maximum margin principle. Support vector machines are wildly popular when used together with a kernel technique, which extends a support vector machine to handle problems that are not linearly separable. Readers are recommended to read [23] by Schölkopf and Smola for more in-depth discussion of this *kernel* support vector machine.

One of the most successful classifiers in practice is a random forest classifier [4]. A random forest classifier consists of many randomly generated decision trees. Due to the lack of time, we cannot cover decision trees and how they are used to form a powerful random forest classifier. scikit-learn implements a random forest classifier, and I highly recommend you to try it out.

Related to the random forest classifier are ensemble methods. This family of ensemble methods is focused on how to combine multiple classifiers in order to achieve better generalization performance. For more discussion, I suggest you to read Sec. 14.2–3 of [2] and/or Sec. 16.2.5 and Sec. 16.4.3 of [16].

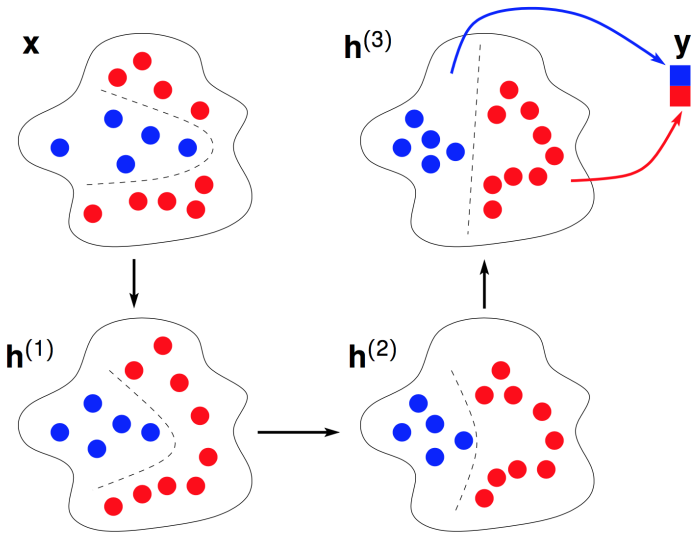


Figure 1.8: The goal of deep learning is to stack many feature extraction functions ϕ^{l}'s on top of each other to gradually transform the problem to become linearly separable.

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.

Distance Function: Log-likelihood Functional Given two vectors, one natural way to define the distance between them is a Euclidean distance which is defined as

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

Following our convention, we thus obtain the following distance function for *linear regression*:

$$D(M^*(\mathbf{x}), M, \mathbf{x}) = \frac{1}{2}\|M^*(\mathbf{x}) - M(\mathbf{x})\|_2^2 = \frac{1}{2}\sum_{k=1}^{q} (y_k^* - y_k)^2, \tag{2.1}$$

where the multiplicative factor $\frac{1}{2}$ was added to simplify the gradient.[2]

[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.

[2] Why does it simplify anything?

At this point of this course, you already know what you need to do. First, you would define an empirical cost using a training set as

$$\hat{R}(M, D_{\text{tra}}) = \frac{1}{N} \sum_{n=1}^{N} \|\mathbf{y}_n^* - M(\mathbf{x}_n)\|_2^2.$$

Then, you would compute the gradient of this empirical cost with respect to the weight matrix $\mathbf{W}$ (or more strictly its flattened vector):

$$\nabla_{\mathbf{W}} \hat{R}.$$

Then, you would use an off-the-shelf optimization algorithm to find a weight matrix that minimizes the empirical cost function. This would the final step, right? No! You should use validation to find a correct hypothesis set (or early-stop learning) to approximately minimize the expected cost function instead of empirical cost function.

Instead of going this whole pipeline once more in the setting of regression, we will consider a slightly difference framework in which these machine learning problems, including both classification and regression, could be embedded.

2.2 Recap: Probability and Distribution

Let us briefly go over a few concepts from probability here. They will become useful in the later discussion where a new framework is introduced.

Random Variables A random variable is not a usual variable in mathematics. A usual variable is often given a single value. When I say $x = 2$, the variable x is equivalent to the value 2, and it is equivalent to replacing x with 2 in any subsequent equations that include x. This is however not true in the case of a random variable, and sometimes we call a normal variable a *deterministic variable* as opposed to a random variable or a *stochastic variable*.

A *random variable* is assigned not a single value but a distribution over all possible values it could take. As an example, consider flipping a coin. We may declare a random variable X to denote the outcome of flipping a coin. X can then take one of two values $\Omega = \{0 \text{ (Head)}, 1 \text{ (Tail)}\}$. Because we do not know what the outcome would be, we do not assign X to any one of these values explicitly but assign to it a *distribution* over these two possible choices.

A distribution is characterized by a function p that maps from one of all possible values to its corresponding probability, and by a set of constraints on this function. In this example of coin flipping, $p : \Omega \to \mathbb{R}$. There are two constraints on this function p. First, the output of this function must be non-negative:

$$p(x) \geq 0.$$

Second, the probabilities of all possible values must sum to 1:

$$\sum_{x \in \Omega} p(x) = 1.$$

This function is called a *probability mass function*.

This idea of distribution can be extended to a continuous random variable. A continuous random variable is assigned correspondingly a *continuous distribution* over a continuous set Ω. Similarly to the definition of a distribution we defined above for a *discrete random variable*, we characterize a continuous distribution by a function F and a set of constraints. Unlike the earlier case, this function F does not return a probability of a given value, but computes a cumulative probability. That is,

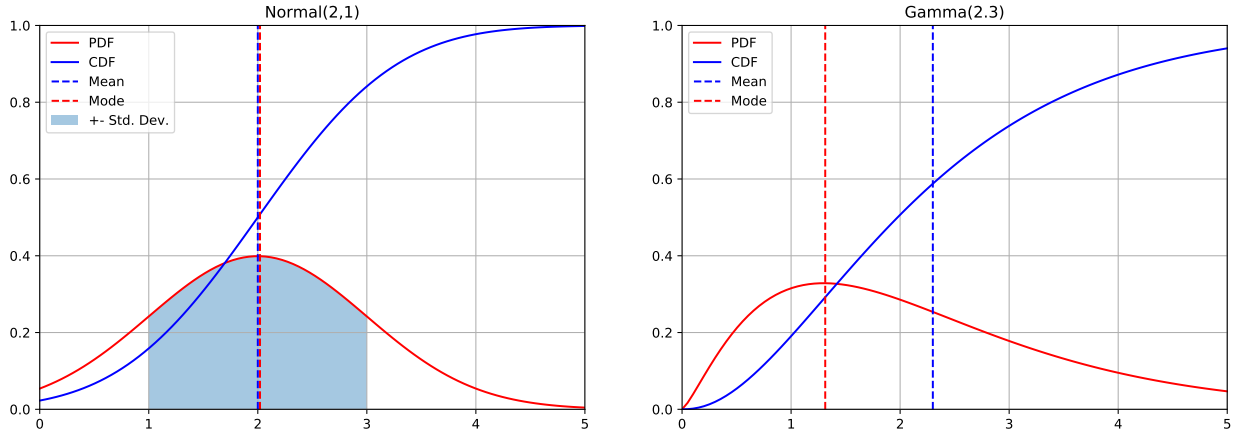
$$F(x) = P(X \leq x), \tag{2.2}$$

i.e., what is the probability of a random variable X being less than or equal to x?

There are two major constraints on this cumulative probability function F.[3] First, as was the case with the discrete random variable, the value of F must be bounded, and in this case, between 0 and 1:

$$0 \leq F(x) \leq 1, \forall x \in \Omega.$$

[3] In addition to two more constraints that are out of scope for this course.



(a) Normal Distribution (b) Gamma Distribution

Figure 2.1: In these two figures, we plot both a probability density function f, cumulative density function F, the mean and the mode. Additionally, in the case of (a) Normal distribution, we coloured the area corresponding to the deviation from the mean by one standard deviation.

Second, as the name suggests, this function must be monotonically non-decreasing:

$$F(x+\varepsilon) \geq F(x),$$

where $\varepsilon > 0$.

Of course, we want something similar to p with a continuous random variable as well in order to easily mix discrete and continuous random variables. The definition of the cumulative probability function F in Eq. (2.2) suggests such a function which acts on a single value x, rather than a set of values (i.e. $\{y \in \Omega | y \leq x\}$):

$$F(x) = \int_{-\infty}^{x} f(x)\mathrm{d}x.$$

This function f is called a *probability density function*.

It is a very bad habit, but a lot of people, including myself, often refer to both the probability mass function and probability density function simply as a *probability*. This is certainly not correct, but often helps us make our explanation as well as equations concise, without introducing much, if any, confusion. Especially, at the level of our discussion in this course, it is almost always okay to mix them up.[4]

Expectation, Variance and Other Statistics When a distribution is defined over many possible values, or sometimes infinitely many values as in the case of continuous random variables, it is useful to extract a small set of representative values of such a distribution. This is often what we do in everyday life as well. For instance, when we move to a new city, we often ask the average monthly rent of an apartment rather than a full distribution over all possible rent prices. Furthermore, we also want to know how much a usual monthly rent varies from this average monthly rent so as not to be surprised.

First, let us define what we mean by the average X by defining a quantity called *mean*. The mean of a random variable X, which has been assigned a distribution whose probability function is p, is defined as

$$\mathbb{E}[X] = \sum_{x \in \Omega} xp(x),$$

or in the case of a continuous variable X,

$$\mathbb{E}[X] = \int_{x \in \Omega} xp(x)\mathrm{d}x.$$

[4] But again, as your instructor, I must insist you know this distinction, and will likely ask you to clarify this distinction in the final exam.

When we say an "average" of a collection of N values, what we often mean is the following:

$$\frac{1}{N}\sum_{n=1}^{N} x_n.$$

Can we somehow connect this intuitive definition of average and the new definition above?

Indeed we can. This can be done by thinking of $p(x)$ as a frequency of x being selected out of all possible values in Ω. Let's say we have infinitely many realizations of the random variable X. $p(x)$ then corresponds to how many there are x in this set of infinitely many realizations of X, meaning how frequently x occurred when we observed X infinitely many times. In this case, multiplying x with the frequency $p(x)$ corresponds to adding all the occurrences of x. By doing this for all possible values of x, we end up with our everyday life definition of "average".

Next, let us define *variance*. We want to know how far each realization is from the mean on average. Based on what we have discussed, it should be clear that

$$\mathrm{Var}(X) = \sum_{x\in\Omega} (x - \mathbb{E}[X])^2 p(x).$$

When X is a continuous random variable, we replace the summation $\sum$ with the integral $\int$. The square-root of the variance is called a *standard deviation*, and is often easier to understand intuitively as it is in the same scale as the original input x.

Can we generalize this notion of variance further? How about this?

$$\mathrm{Moment}_p(X) = \sum_{x\in\Omega} (x - \mathbb{E}[X])^p p(x). \tag{2.3}$$

This is called a *p-th central moment*, and has turned out to have interesting properties. For instance, the 3-rd central moment, to which we refer as *skewness*, indicates whether the distribution is symmetric. For instance, the Normal distribution plotted in Fig. 2.1 (a) has a zero 3-rd central moment, while the Gamma distribution in Fig. 2.1 (b) has a non-zero 3-rd central moment. The 4-th central moment, called *kurtosis*, indicates the flatness of the distribution with respect to the Normal distribution. Many of these moments have fascinating use cases in machine learning, but they are out of scope for this course.

One interesting observation about the mean is that the mean may not correspond to an actual value. As a simple example, let us consider a distribution over a 5-star moving rating. Let us assume that *a priori* a 5-star moving rating obeys the following distribution:

$$p(1) = 0.3, p(2) = 0.2, p(3) = 0.05, p(4) = 0.15, p(5) = 0.3.$$

The mean is 2.95. This number is however not at all informative because of two reasons.

First, there is no rating of 2.95. In other words, this is some fantasy number that summarizes the whole distribution, however, without corresponding to any real rating. This could be problematic, if our goal is to use this mean to make a decision or act. For instance, consider placing a bet on predicting the rating of a newly released movie. I cannot place my bet on 2.95 but only on one of those five scores. Second, even if we decide to round the mean to select the nearest integer score, we notice that this is far from being representative of the distribution. The nearest score 3 has only 5% chance!

This latter issue encourages us to define yet another metric called a *mode* of a distribution. A mode of a distribution is a value of which probability is highest:

$$\mathrm{Mode}(X) = \arg\max_{x\in\Omega} p(x).$$

As you can easily guess, a mode is often not unique, and there may be multiple modes that have the same probability. In fact, a uniform distribution, which assigns the same probability of each and every possible value, has as many modes as the size of Ω (in the case of a continuous random variable, this would correspond to the infinity.) In practice, we often relax this definition, and consider all *local maxima* as modes of a distribution.

Examples of these statistics with real distributions are presented in Fig. 2.1.

Distributions There are a number of widely used distributions, both discrete and continuous. As this course is an introduction to machine learning, we will consider only a small subset of such distributions.

In the case of discrete variables, we have already learned two popular distributions. First, we used a *Bernoulli distribution* for logistic regression in Sec. 1.3. Bernoulli distribution is defined over a set of two possible values and fully specified by a single parameter μ:

$$p(x) = \mu^x (1-\mu)^(1-x),$$

where $x \in \{0,1\}$. The mean of the Bernoulli distribution is μ, and the variance is $\mu(1-\mu)$.

Later on in Sec. 1.6, we extended this Bernoulli distribution to a *categorical distribution* to build a multi-class logistic regression. Categorical distribution is defined over a set of C possible values, and is specified by C-many probabilities:

$$p(x) = p_x,$$

with a constraint that they are all non-negative and sum to 1:

$$\begin{aligned} p_x &\geq 0, \forall x \in C, \\ \sum_{x \in \Omega} p_x &= 1. \end{aligned}$$

In the case of continuous variables, we will solely use a normal distribution, or often Gaussian distribution, throughout this course. A normal distribution is defined over an unbounded real vector space $\mathbb{R}^d$, and is fully specified by two parameters–mean vector $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \in \mathbb{R}^{d \times d}$. Its probability density function is defined as

$$f(\mathbf{x}) = \frac{1}{Z} \exp\left(-\frac{1}{2} (\mathbf{x}-\mu)^\top \Sigma^{-1} (\mathbf{s}-\mu) \right), \tag{2.4}$$

where Z is the normalization constant that ensures the integral of the density function is 1. That is,

$$\begin{aligned} Z &= \int_{\mathbf{x} \in \mathbb{R}^d} \exp\left(-\frac{1}{2} (\mathbf{x}-\mu)^\top \Sigma^{-1} (\mathbf{s}-\mu) \right) \mathrm{d}\mathbf{x} \\ &= (2\pi)^{-d/2} |\Sigma|^{-1/2}, \end{aligned}$$

where $|\Sigma|$ is the determinant of the covariance matrix.

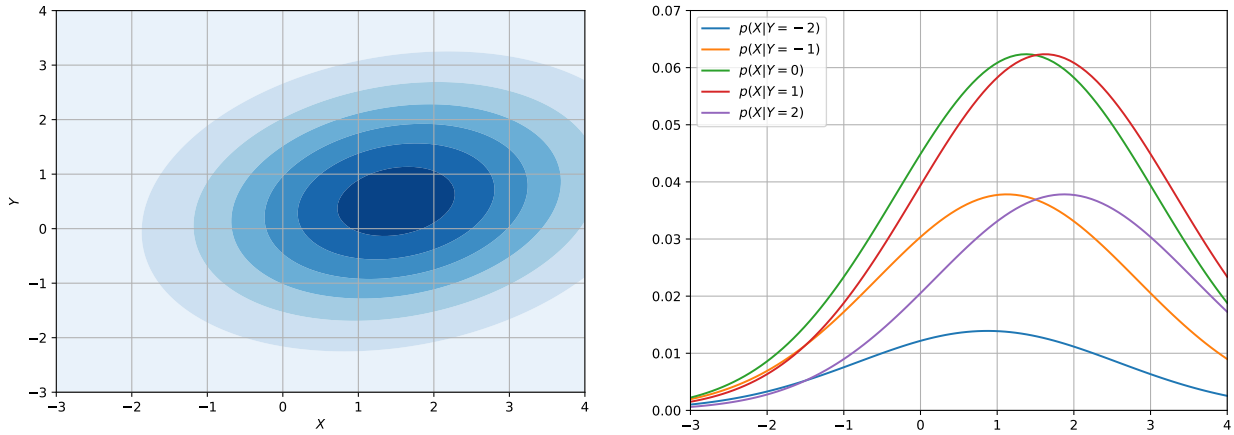
In practice, and for most of cases throughout this course, we will assume that the covariance matrix is a diagonal matrix, i.e.,

$$\Sigma = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & \sigma_d^2 \end{bmatrix}.$$

In other words, we assume that each dimension of the input $\mathbf{x}$ is decorrelated from each other. In this case, the probability density function simplifies to[5]

$$f(\mathbf{x}) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi}\sigma_i} \exp\left(-\frac{1}{2} \frac{1}{\sigma_i^2} (x_i - \mu_i)^2 \right). \tag{2.5}$$

[5] Of course, you may now have noticed that this simplification is left for you as a **homework assignment**.



(a) Joint Distribution

(b) (Un-normalized) Conditional Distributions

Figure 2.2: In the left figure, we plot the joint distribution over X and Y. In the right figure, we plot the conditional distributions over X given select values of Y.

Bayes' Rule Let us consider having two random variables X and Y. It is not too difficult to imagine a joint distribution over them, which could easily be done by assigning a *multivariate* distribution, such as the multivariate normal distribution from above, to a pair of X and Y:[6]

$$p(X = x, Y = y).$$

This distribution computes the probability of X and Y having the values x and y respectively. We call this distribution a *joint distribution* between random variables X and Y.

Based on this, we can further define a *conditional distribution*. A conditional distribution considers only a subset of all possible joint probabilities (via the joint distribution) by fixing the value of one random variable Y to a pre-specified value y:

$$p(X, Y = y).$$

In other words, given that the random variable Y is fixed to a value y, what is the distribution over the other *free* random variable X?

One thing we notice is that the above quantity $p(X, Y = y)$ is not a valid distribution as it will not sum to 1 in general. Instead, we need to normalize it by

$$p(X|Y = y) = \frac{p(X, Y = y)}{p(Y = y)}$$

to get a proper conditional probability $p(X|Y)$. We often omit $= y$ to denote that this equation holds regardless of which value Y was assigned to:

$$p(X|Y) = \frac{p(X, Y)}{p(Y)} \tag{2.6}$$

From this definition, we can establish one interesting notion of *independence*. If the random variables X and Y are independent from each other (or mutually independent), then the conditional distribution over X given Y should simply be the distribution over X, and vice versa. What does this notion of independence imply? If X is independent from Y, i.e.,

$$p(X|Y) = p(X),$$

[6] When it is not confusing, we will use $p(x, y)$ as a shorthand notation.

then

$$p(X|Y) = p(X) = \frac{p(X,Y)}{p(Y)} \iff p(X,Y) = p(X)p(Y). \tag{2.7}$$

In fact, the converse of this statement is also true. That is, if $p(X,Y) = p(X)p(Y)$, then X and Y are mutually independent.

These two definitions can easily be mixed in. Consider three random variables X, Y and Z. A joint distribution over all of them is $p(X,Y,Z)$. A conditional distribution over X and Y given Z can be written as $p(X,Y|Z)$. X and Y are *conditionally independent* from each other given Z *iff*

$$p(X,Y|Z) = p(X|Z)p(Y|Z). \tag{2.8}$$

From this definition of conditional distribution, we can compute the conditional distribution in the opposite direction:

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}. \tag{2.9}$$

This rule is called *Bayes' rule*, and it is left for you as a **homework assignment** to verify that this rule is indeed true. The importance of this rule will become self-evident in the following sections.

We have so far used $p(X)$ and $p(Y)$ together with a joint distribution $p(X,Y)$ without establishing their relationship. Of course, the definition of the conditional probability tells us that

$$p(X,Y) = p(X|Y)p(Y),$$

but this is simply a circular definition. Instead, can we define $p(X)$ (or $p(Y)$) on its own based on the joint distribution without resorting to the use of conditional distribution? Yes, and we do it by *marginalizing out* the other random variable:

$$p(X) = \sum_{y \in \Omega_Y} p(X, Y = y), \tag{2.10}$$

where Ω_Y is a set of all possible values for Y. With this marginalization, we can rewrite the Bayes' rule in Eq. (2.9) as

$$p(Y|X) = \frac{p(X|Y)p(Y)}{\sum_Y p(X,Y)}.$$

The usefulness of these probabilistic tools–joint distribution, conditional distribution, independence, marginalization and Bayes' rule– will become self-evident in the following sections, when we start to introduce a so-called Bayesian approach to machine learning.

2.3 Bayesian Linear Regression

2.3.1 Bayesian Linear Regression

Now that we have refreshed our memory on probability, let us frame what we have learned so far into this probabilistic terms. For this, we need to define two distributions; (1) likelihood or a conditional distribution, and (2) a prior distribution.

First, let us introduce a new symbol θ which we will use to denote any adjustable parameters of a machine. For instance, θ includes a weight vector in the case of linear classifiers and linear regression. In the case of adaptive basis function networks, or deep neural networks, θ includes a weight vector, or matrix, as well as all the basis vectors which may be adjusted to maximize the classification accuracy. All these seemingly diverse set of adjustable parameters can all be flattened and concatenated to form a single vector θ.

Second, let us slightly modify the distance function of linear regression in Eq. (2.1) so as to make it easier to be integrated into a probability framework. As our motivation speaks for itself, we want to turn the distance function to be a probability (density) function, similar to what we have done with logistic regression (see Eq. (1.32).) Unlike logistic

regression, linear regression outputs a continuous value, and therefore we use a normal distribution in Eq. (2.4) instead of categorical distribution. For simplicity, we assume that the covariance Σ is an identity matrix:

$$\Sigma = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & 0 & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} = \Sigma^{-1}.$$

In this case, the probability density function of normal distribution simplifies to

$$f(\mathbf{x}) = \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(x_i - \mu_i)^2\right). \tag{2.11}$$

By carefully inspecting this equation and linear regression distance function in Eq. (2.1), we notice that the latter appears as it is in the former. Furthermore, by replacing x_i with the i-th component of a ground-truth output y_i^* and μ with the output of our current machine M, we notice that minimizing the linear regression distance function is equivalent to maximizing the *log*-probability density of this Normal distribution. That is,

$$\arg\min_{M \in H} D(M^*(\mathbf{x}), M, \mathbf{x}) = \arg\min_{M \in H} -\log f(M^*(\mathbf{x}); \mu = M(\mathbf{x})) \tag{2.12}$$

$$= \arg\min_{M \in H} \sum_{i=1}^{q} \frac{1}{2}(y_i^* - \mu_i)^2 + C, \tag{2.13}$$

where C is a constant that does not depend on M, and the constant multiplicative factor $\frac{1}{2}$ does not change the optimization problem.

In other words, we can rewrite the distance function of linear regression as a negative log-probability of a ground-truth output vector y^* under a normal distribution whose mean is the output of the current machine M and covariance is an identity matrix. So, we have somehow turned linear regression into a probabilistic model. Let us continue.

Likelihood $p(D|\theta)$ If we consider this θ and an entire training set $D = D_{\text{tra}}$[7] as random variables, a notion of how likely the training set, or the set of training examples, is given some θ. This is precisely what conditional probability is (see Eq. (2.6),) and we call this specific conditional probability $p(D|\theta)$ a *likelihood*. How does this likelihood look like?

In this course, we have implicitly assumed that each training input vector x_n (and consequently the corresponding reference output y_n^*) was independently drawn from a single data distribution (see Eq. (1.3) from long time ago.) Then, based on the definition of conditional independence in Eq. (2.8), we can decompose the likelihood into

$$p(D|\theta) = \prod_{n=1}^{N} p((x_n, y^* n)|\theta).$$

The logarithm of this likelihood, to which we refer as *log-likelihood*, is then

$$\log p(D|\theta) = \sum_{n=1}^{N} \underbrace{\log p((x_n, y^* n)|\theta)}_{(a)}.$$

Where have we seen (a) before?

Yes, we have seen it twice already this course. We saw one when we defined the distance functions of logistic regression and multi-class classification, and we just saw this for linear regression right above in Eq. (2.12). We have just had a glimpse of an interesting relationship between the likelihood (or its logarithm version, log-likelihood) and the empirical cost function. That is, if the distance function of a model could be described in terms of a probability function, we can define an equivalent log-likelihood functional.[8]

[7] I will use D instead of D_{tra} throughout the remainder of this section for brevity.

[8] A *functional* is informally-speaking a mapping from a function to a scalar. Both the distance function and log-likelihood are functional in that the input to them is a function M. However, we can equivalently think of them as functions by considering the parameters θ of M as their input. This view is enough for the content of this course, but does not hold true in general.

Prior $p(\theta)$ Before we observe a training set D, or data, what kind of prior knowledge do we have about the parameters of a machine? Or, similarly what kind of prior information do we want to impose on the parameters? The answer to the former question may be problem-specific, while that to the latter may be algorithm-specific. A prior distribution is our way to impose or incorporate such prior knowledge.

Unlike the likelihood, the prior distribution $p(\theta)$ is defined solely on the parameters θ irrespective of actual data D. One of the most widely used prior distributions is again a normal distribution often with an all-zero mean vector and a diagonal covariance matrix, as in Eq. (2.5).[9] Intuitively, this choice of prior distribution states that each parameter is unlikely to deviate too much from 0, and how far it may deviate depends on the variance σ^2.[10] Mathematically, we express this prior distribution by

$$p(\theta) = \prod_{i=1}^{|\theta|} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{\theta_i^2}{\sigma^2}\right),$$

where $|\theta|$ is the dimensionality of the vector θ. The logarithmic form is

$$\log p(\theta) = \sum_{i=1}^{|\theta|} -\frac{\theta_i^2}{2\sigma^2} - \log\sqrt{2\pi}\sigma. \tag{2.14}$$

What does this remind you of?

Of course, this is not the only choice, and we can freely choose a prior distribution so as to impose certain structures. For instance, if we believe the first and second parameters are strongly correlated with each other, we may choose to abandon the diagonal covariance matrix and instead use a covariance matrix where the cross-correlation term between the first and second parameters is a large positive value. But, we will get to this later (hopefully!)

Posterior $p(\theta|D)$ At the end of the day, we have two goals in machine learning. First, we want to figure out what the parameters of a model look like once trained. We tackled this problem earlier by minimizing the empirical cost function (with some regularization, as in Sec. 1.5.3). In a probabilistic framework, however, we are rather interested in a *posterior* distribution over the parameters *given* a training set: $p(\theta|D)$.

Unlike the likelihood and prior, we will not designate the form of this posterior distribution ourselves. Because we already have the likelihood and prior distribution, we can instead compute the posterior distribution from them using Bayes' rule in Eq. (2.9):

$$p(\theta|D) = \underbrace{p(D|\theta)}_{\text{likelihood}} \underbrace{p(\theta)}_{\text{prior}} \Big/ \underbrace{p(D)}_{\text{evidence}}.$$

Clearly, we need one more distribution $p(D)$, to which we refer as *evidence*. Or, do we?[11]

Similarly to the likelihood, it is common to consider the *log*-posterior:

$$\log p(\theta|D) = \log p(D|\theta) + \log p(\theta) - \log p(D).$$

Other than the log-evidence $\log p(D)$, let's try to plug in what we know in the case of linear regression to the right-hand side of this log-posterior:

$$\log p(\mathbf{W}|D_{\text{tra}}) = \underbrace{\sum_{n=1}^{N}\sum_{i=1}^{q} \frac{1}{2}(y_{n,i}^* - [\mathbf{W}^\top \mathbf{x}_n]_i)^2 - \log\sqrt{2\pi}}_{=\log p(D|\theta)} + \underbrace{\sum_{i=1}^{d}\sum_{j=1}^{q} -\frac{w_{ij}^2}{2\sigma^2} - \log\sqrt{2\pi}\sigma}_{=\log p(\theta)} - \log p(D).$$

[9] Now you see why I introduced a normal distribution only when we were discussing about common distributions.

[10] Notice the lack of the subscript i. We assume that each and every parameter θ_i has the same variance σ.

[11] The following two questions are left for you as **homework assignments**.

1. Why do we call this distribution, or the probability of data D, evidence?
2. Do we need to define this evidence distribution ourselves? If not, what can we do about it?

Maximum-a-Posteriori (MAP) Estimation: Poor Man's Bayes Given this form of the log-posterior distribution, what should we do? The first thing that comes to my mind is to find the parameter vector θ, or correspondingly the weight matrix $\mathbf{W}$, that maximizes this posterior probability. That is, we want to find a *mode* of the posterior distribution:

$$\hat{\theta} = \arg\max_{\theta} \log p(\theta|D) = \arg\max_{\theta} \log p(D|\theta) + \log p(\theta) - \log p(D).$$

In the case of linear regression, this equation is equivalent to

$$\begin{aligned}
\arg\max_{\theta} \log p(\mathbf{W}|D_{\text{tra}}) &= \arg\max_{\theta} \sum_{n=1}^{N}\sum_{i=1}^{q} -\frac{1}{2}(y^*_{n,i} - [\mathbf{W}^\top \mathbf{x}_n]_i)^2 - \log\sqrt{2\pi} + \sum_{i=1}^{d}\sum_{j=1}^{q} -\frac{w_{ij}^2}{2\sigma^2} - \log\sqrt{2\pi}\sigma - \log p(D) \\
&= \arg\max_{\theta} \sum_{n=1}^{N}\sum_{i=1}^{q} -\frac{1}{2}(y^*_{n,i} - [\mathbf{W}^\top \mathbf{x}_n]_i)^2 - \sum_{i=1}^{d}\sum_{j=1}^{q} \frac{1}{2\sigma^2} w_{ij}^2 \\
&= \arg\min_{\theta} \underbrace{\sum_{n=1}^{N}\sum_{i=1}^{q} \frac{1}{2}(y^*_{n,i} - [\mathbf{W}^\top \mathbf{x}_n]_i)^2}_{\text{Empirical Cost}} + \underbrace{\frac{C}{2}\sum_{i=1}^{d}\sum_{j=1}^{q} w_{ij}^2}_{\text{Regularization}},
\end{aligned}$$

where $C = \frac{1}{\sigma^2}$. Notice that the log-partition functions $\log\sqrt{2\pi}$'s and the log-evidence $\log p(D)$ have been removed, as they are not dependent on the parameters θ.

By carefully inspecting the above equation, especially the final form, we notice that this is identical to the empirical cost function of linear regression augmented with a regularization term. More specifically, the regularization term is the maximum margin regularization term we learned earlier from support vector machines in Sec. 1.4.2. In other words, this probabilistic formulation of linear regression has resulted in a more traditional formulation of machine learning we have discussed so far, in the terms of empirical cost function and regularization.

Does this mean that all we have done during the past one and a half week has been to simply find a different way to end up with the exact same solution we have already learned? That would have been an awful waste of our time, wouldn't it?

Predictive Distribution $p((x^\star, y^\star)|D)$ Now we go one step further. In supervised machine learning, what is our ultimate goal? Was it to find the parameters that minimize the empirical cost function together with a regularization term? In some cases, yes. Our goal was however to build a machine that can predict the outcome of a future, unseen input vector as well as possible.

Can we push this even further? Perhaps what we want is only to know the outcome of a new, unseen input vector. If we can do so, do we really care about specifically which machine (equivalently, which set of parameters) has been used? Indeed, we may want to use more than one machines to make their own predictions and return us their (weighted) consensus. We discussed this possibility already at the very beginning of this course in Eq. (1.6).

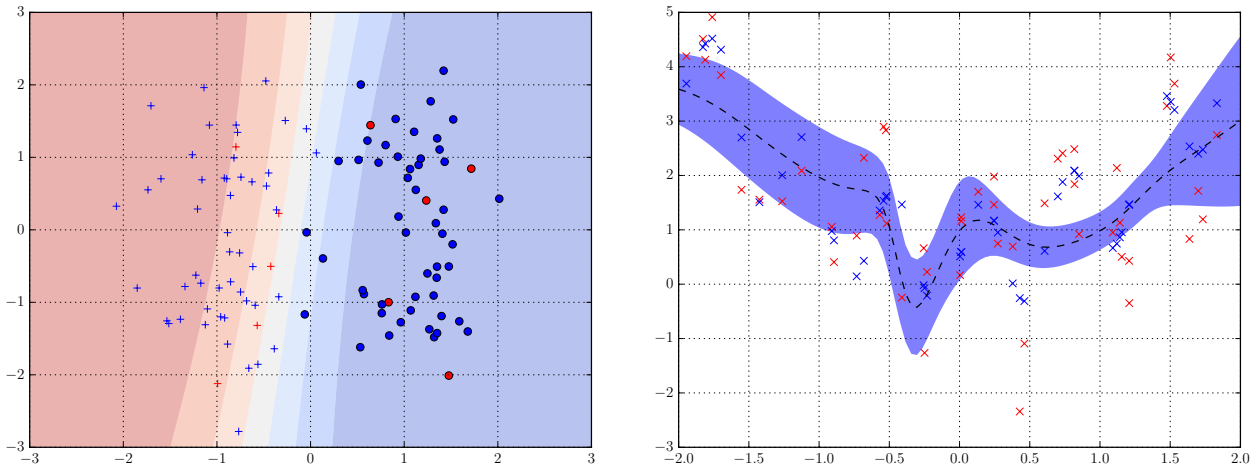
Eventually, what we want to know the conditional distribution over the new example $(x^\star, y^\star)$ given the training set.[12] We can do this by *marginalizing* out the parameters θ. As we have seen earlier in Eq. (2.10), we can do so by

$$\begin{aligned}
&p((x^\star, y^\star)|D) \\
&= \textstyle\sum_{\theta\in\Theta} p(x^\star, y^\star, \theta|D) && \text{marginalization} \\
&= \textstyle\sum_{\theta\in\Theta} p(x^\star, y^\star|\theta, D) p(\theta|D) && \text{conditional probability} \\
&= \textstyle\sum_{\theta\in\Theta} p(x^\star, y^\star|\theta) p(\theta|D), && \text{conditional independence}
\end{aligned}$$

where we assumed in the last line that the distribution over an example is independent from all the other examples *given* a set of parameters. Θ is a set of all possible sets of parameters. Examining the final form further, we notice that it is the expectation of the likelihood under the posterior distribution over the parameters:

$$p((x^\star, y^\star)|D) = \sum_{\theta\in\Theta} \underbrace{p(x^\star, y^\star|\theta)}_{\text{Likelihood}} \underbrace{p(\theta|D)}_{\text{Posterior}} = \mathbb{E}_{\theta|D}\left[p(x^\star, y^\star|\theta)\right].$$

[12] Note that $y^\star$ would not be given together with $x^\star$, but if we know their joint distribution $p(x^\star, y^\star|D)$, we can get the conditional distribution over $y^\star$ given $x^\star$ according to the definition in Eq. (2.6). Of course, in the case of supervised learning, we probably want to compute $p(y^\star|x^\star, D)$ directly.



(a) (b)

Figure 2.3: (a) Bayesian logistic regression, and (b) Bayesian multilayer perceptron. Both of them were done with "ensemble samplers with affine invariance" [9] using Python emcee available at `http://dan.iel.fm/emcee/current/`.

We call this resulting distribution a *predictive distribution*.

What does this equation imply intuitively? The answer is quite straightforward. We will let each machine, parametrized by θ, cast a vote by scoring each possible answer y given a new input vector $x^\star$. Their votes are however not equal, and the votes by those parameters, or equivalently models, which are more likely given the training data D (i.e., higher posterior $p(\theta|D)$) would be weighted more. In other words, those machines that do better in terms of empirical cost function are given higher weights than those that do worse. The term $\omega(M)$ in Eq. (1.6) thus corresponds to the posterior probability of M (equiv. to θ).

As an extra-credit **homework assignment**, you are asked to derive a closed-form probability density function of the predictive distribution in the case of linear regression. Linear regression with this derived predictive distribution is at the heart of *Bayesian linear regression*.

2.3.2 Bayesian Supervised Learning

Although we have used linear regression as an example[13] throughout the whole discussion on this probabilistic approach to machine learning, it should have been clear that this idea is generally applicable as long as we define the following distributions:

- Prior distribution over the parameters: $p(\theta)$
- Likelihood distribution: $p(D|\theta)$

From these two distributions and a training set D, we can derive the posterior distribution $p(\theta|D)$, and compute the predictive distribution.

Of course, this is not strictly true in that in almost all cases, we do not know how to compute the posterior distribution and/or predictive distribution exactly. Fortunately with linear regression, we were able to do so, but as soon as we encounter some non-trivial prior and/or likelihood distributions, it is impossible to derive an analytical form of either posterior or predictive distribution.

Thus, to be more precise, in addition to defining those two distributions above, we need to be able to compute, either exactly or approximately, the posterior distribution.[14] There are two major, general approaches to this prob-

[13] We call linear regression under this probabilistic approach a *Bayesian linear regression*.

[14] When I say *compute a distribution*, I mean that being able to (1) compute the (unnormalized) probability and (2) compute the expectation and the n-th central moments in Eq. (2.3).

lem of computing the posterior distribution. The first one is a sampling-based approach by which we can generate samples from a given distribution one at a time with an asymptotic guarantee that the collected samples are from the given distribution. The most widely used family of sampling-based algorithms is called Markov-Chain Monte-Carlo (MCMC). The second approach is variational inference in which a simpler approximate posterior distribution is fitted to the complex, but true posterior distribution. Once this simpler approximate distribution is found, we use it for any subsequent task including the computation of the predictive distribution. There are a number of other approaches, such as message passing (or belief propagation) that could be used. Unfortunately any of these approaches are far out of scope of this course.[15]

Instead, let me show you two examples in Fig. 2.3. On the left panel is the visualization of the predictive distribution of Bayesian logistic regression. The background gradient indicates the predictive probability of $y = 1$ given a training set (blue markers); red close to 1, and blue close to 0. We notice that the confidence, which can be thought of as how far away the predictive probability of $y = 1$ is from 0.5, is lower near the decision boundary. Furthermore, we notice that the low-confidence region grows as a new input vector is further away from the training examples.

On the right panel is the example of Bayesian multi-layer regression which is the regression version of the adaptive basis function network from Sec. 1.8.4. The plot shows both training examples, the mean of the predictive distributions as well as its standard deviation (shaded region surrounding the mean curve.) A noticeable feature is that the standard deviation shrinks when there are many training examples nearby, and grows in the other case.

Both examples were done using a specific MCMC algorithm to generate many samples from the posterior distribution.

2.3.3 Further Topic: Gaussian process regression*

In linear regression, when the prior distribution over the parameters is Gaussian and the likelihood is also Gaussian, the posterior distribution over the parameters as well as the predictive distribution are both Gaussian, due to the property of multivariate Gaussian distribution. It implies that we can fully characterize the prediction of a new input example given a training set by computing the mean and covariance of multivariate Gaussian distribution. The covariance matrix is often specified by a *covariance function*, and a suitable choice allows us to turn the linear regression into *nonlinear* regression. Unfortunately, Gaussian process regression is out of this course's scope, and I refer readers to a widely read textbook [25] by Williams and Rasmussen, and a great introduction tutorial[16] by David MacKay.

[15] I recommend [2] for further discussion.

[16] `http://videolectures.net/gpip06_mackay_gpb/`

Chapter 3

Dimensionality Reduction and Matrix Factorization

3.1 Dimensionality Reduction: Problem Setup

Let us assume that we are given a set of input vectors without their corresponding labels. What can we do about these input vectors

$$D = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}?$$

Perhaps, the first thing we should try is to look at all of these input vectors to see if we can find any interesting regularity behind them. If they are two-dimensional vectors, we can visualize them by plotting a two-dimensional scatter plot in which each vector is drawn as a single point. If they are three-dimensional vectors, we can still visualize them by plotting a three-dimensional scatter plot or by plotting a contour plot. This is however not as trivial as plotting two-dimensional points. If they are four-dimensional vectors, already we run out of any "general" way to visualize them.

When we are presented with high-dimensional vectors, we try to reduce their dimensionality in order to (1) visualize them more easily, and (2) more efficiently process them.[1] There is a family of machine learning techniques dedicated to this problem of reducing the dimensionality of input vectors. Applying one of these techniques is called *dimensionality reduction*.

In dimensionality reduction, given a set of input vectors

$$D = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$$

our goal is to find a set of corresponding lower-dimensional vectors

$$\tilde{D} = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_N\},$$

where $\mathbf{x}_i \in \mathbb{R}^d$, $\mathbf{z}_j \in \mathbb{R}^q$, and $d \gg q$.

There are two major families of techniques in dimensionality reduction. First, there are parametric dimensionality reduction techniques. Similarly to supervised learning we have learned earlier, we obtain a machine M that either maps from an input vector to its corresponding lower-dimensional vector

$$M : \mathbb{R}^d \to \mathbb{R}^q,$$

or maps from a lower-dimensional vector to its corresponding higher-dimensional vector[2]

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

[1] Why is it more efficient to process data points if they are lower-dimensional vectors? This question is left to you as a **homework assignment**.

[2] Note that finding either of these two mappings is equivalent when such a mapping is invertible. We will discuss this more in detail later when we introduce a deep autoencoder.

Again, as we learned with supervised learning, this type of machine may be categorized into either linear or nonlinear. In the case of linear, parametric dimensionality reduction, the problem can be formulated as *matrix factorization*. On the other hand, nonlinear, parametric dimensionality reduction is more general, and we will discuss one particular instantiation, called a *deep autoencoder* later in this chapter.

The other family of dimensionality reduction techniques consists of non-parametric techniques. A non-parametric dimensionality reduction technique does not provide a separate machine that could be used on a new example, but only returns a set of lower-dimensional vector $\tilde{D}$. These techniques are often strictly used for *visualization* of high-dimensional data.

In this course, we will mainly focus on the parametric family of dimensionality reduction techniques, starting from matrix factorization and ending with deep autoencoders. If time permits, we will study some non-parametric dimensionality reduction techniques that are widely used in the scientific community.

3.2 Matrix Factorization: Problem Setup

Let us build a large matrix that contains all the input vectors by lining them next to each other. That is,

$$\mathbf{X} = [\mathbf{x}_1; \mathbf{x}_2; \cdots \mathbf{x}_N] \in \mathbb{R}^{d \times N}.$$

Matrix factorization is then a problem of (approximately) representing this data matrix $\mathbf{X}$ as a product of two matrices:

$$\mathbf{X} \approx \mathbf{W}\mathbf{Z}, \tag{3.1}$$

where $\mathbf{Y}$ is a matrix that contains all the lower-dimensional vectors

$$\mathbf{Z} = [\mathbf{z}_1; \mathbf{z}_2; \cdots; \mathbf{z}_N],$$

and $\mathbf{W} \in \mathbb{R}^{d \times q}$ is a weight matrix. $\mathbf{Z}$ is often called a code matrix, and $\mathbf{W}$ a dictionary matrix. By setting q to be smaller than d, i.e., $q \ll d$, this matrix factorization allows us to find a set of lower-dimensional vectors. In other words, matrix factorization allows us to reduce the dimensionality of input vectors.

Matrix factorization is a widely-used, and perhaps oldest, dimensionality reduction technique. It is a *linear* dimensionality reduction technique, as evident from Eq. (3.1). Why is this so? Because, the equation in Eq. (3.1) could be understood as a set of N equations of which one is

$$\mathbf{x}_i = \mathbf{W}\mathbf{z}_i. \tag{3.2}$$

Matrix factorization is precisely a way to build a *linear* machine that maps from a lower-dimensional vector to its original space $\mathbb{R}^d$.

Eq. (3.2) reminds us of linear regression in Sec. 2.1. The only difference is that both $\mathbf{z}$ and $\mathbf{x}$ were known in linear regression, while only one of them, $\mathbf{x}$, is known in matrix factorization. This lack of input vectors, following the terminology from linear regression, implies that this problem is under-specified, meaning that there may be many solutions of $\mathbf{W}$ and $\mathbf{Z}$ that satisfy Eq. (3.1). This is quite obvious, if we consider the simplest case of $d = q = 1$, in which case the whole problem reduces to finding w and z that satisfies

$$x = wz,$$

where x is known. For any solution (w, z) that satisfies this equality, there are infinitely many other solutions (w', z') such that

$$w' = cw \text{ and } z' = \frac{z}{c},$$

where c is an arbitrary real number. A similar thing happens with $d, q > 1$ with any invertible matrix $\mathbf{C} \in \mathbb{R}^{q \times q}$, because

$$\mathbf{X} = (\mathbf{W}\mathbf{C})(\mathbf{C}^{-1}\mathbf{Z}) = \mathbf{W}(\underbrace{\mathbf{C}\mathbf{C}^{-1}}_{=\mathbf{I}})\mathbf{Z} = \mathbf{W}\mathbf{Z}.$$

What this implies is that there are many different ways to solve this matrix factorization. By imposing a set of clever constraints on the weight matrix $\mathbf{W}$ and/or $\mathbf{Z}$, we get a diverse set of linear dimensionality reduction algorithms. In the rest of this section, we investigate three such algorithms.

3.2.1 Principal Component Anslysis: Traditional Derivation

The first such algorithm is called principal component analysis (PCA). In PCA, there are two constraints. The first constraint is on $\mathbf{Z}$, or a set of code vectors $\mathbf{z}_1, \ldots, \mathbf{z}_N$. It states that each component of these code vectors must be *decorrelated* from all the other components. This is equivalent to

$$\sum_{n=1}^{N} z_{n,j} z_{n,i} = 0, \text{ for all } i \neq j,$$

assuming that $\mathbf{z}_n$'s are centered. We can write it more compactly by

$$\mathbf{Z}\mathbf{Z}^\top = \sum_{n=1}^{N} \mathbf{z}_n \mathbf{z}_n^\top = \text{diag}(\sigma_1^2, \ldots, \sigma_q^2),$$

where

$$\Sigma = \text{diag}(\sigma_1^2, \ldots, \sigma_q^2) = \begin{bmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma 2^2 & \cdots & 0 \\ \vdots & 0 & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & \sigma_q^2 \end{bmatrix}.$$

The second constraint is that these variances σ_j^2's are non-decreasing. In other words,

$$\sigma_i^2 \geq \sigma_j^2, \text{ for all } i > j.$$

With these constraints in our mind, let us consider the (scaled) covariance of the input vector matrix $\mathbf{X}$ which is defined as

$$\begin{aligned} \underbrace{\mathbf{X}\mathbf{X}^\top}_{\mathbf{C}=\text{Covariance of } \mathbf{X}} &= (\mathbf{W}\mathbf{Z})(\mathbf{W}\mathbf{Z})^\top \\ &= \mathbf{W} \underbrace{\mathbf{Z}\mathbf{Z}^\top}_{\Sigma=\text{Covariance of } \mathbf{Z}} \mathbf{W}^\top \\ &= \mathbf{W}\Sigma\mathbf{W}^\top \end{aligned}$$

Since any covariance matrix is by definition symmetric,[3] we immediately notice that this equation precisely describes a procedure called *eigendecomposition*:

$$\mathbf{C} = \mathbf{W}\Sigma\mathbf{W}^\top, \tag{3.3}$$

assuming $q = d$. $\mathbf{W}$ is then a matrix consisting of d eigenvectors of $\mathbf{C}$:[4]

$$\mathbf{W} = [\mathbf{v}_1; \mathbf{v}_2; \cdots; \mathbf{v}_d],$$

where

$$\mathbf{C}\mathbf{v}_j = \sigma_j^2 \mathbf{v}_j \tag{3.4}$$

which is the definition of the j-th eigenvector, and σ_j^2 is the corresponding j-th eigenvalue. These eigenvalues and the corresponding eigenvectors can be computed by first solving the characteristic polynomial of $\mathbf{C}$ to find all eigenvalues

[3] A matrix $\mathbf{C}$ is symmetric if and only if

$$\mathbf{C} = \mathbf{C}^\top.$$

[4] We assume that $N \gg d$, and that the subspace in which all the input vectors lie is at least q-dimensional.

and solving Eq. (3.4) for each of the eigenvalues.[5] We see that Eq. (3.3) is nothing but a stack of Eq. (3.4) in the decreasing order of σ_j^2's. In other words, we can compute the weight matrix $\mathbf{W}$ by eigendecomposition of the (scaled) covariance matrix $\mathbf{C}$ of the input matrix $\mathbf{X}$.

This is a good start as this selection of the weight matrix $\mathbf{W}$ ensures that the code vectors are decorrelated, satisfying the first constraint. But, how do we compute the code matrix $\mathbf{Z}$? One important property of the eigenvectors is that they are orthogonal to each other.[6] Since the inverse of any orthogonal matrix is its transpose, we can compute the code matrix by

$$\mathbf{Z} = \mathbf{W}^\top \mathbf{X}. \tag{3.5}$$

Along the derivation, we have made one assumption that $q = d$. This effectively means that we have not done any dimensionality *reduction*. How do we then use this whole procedure to reduce the dimensionality, that is $q \ll d$? We can do it easily by simply taking only the first q rows of the weight matrix, or taking only the first q eigenvectors. That is,

$$\mathbf{W} = [\mathbf{v}_1; \mathbf{v}_2; \cdots; \mathbf{v}_q].$$

Because the eigenvectors were sorted in the decreasing order of the eigenvalues which correspond to the variance of each component of the code vectors, this satisfies the second constraint. Of course, this breaks the equality in Eq. (3.3), but is still a good approximation, as we will see shortly.

In summary, PCA is done in the following steps:

1. Centering: $\mathbf{x}_n \leftarrow \mathbf{x}_n - \frac{1}{N}\sum_{n=1}^N \mathbf{x}_n$
2. Covariance: $\mathbf{C} = \mathbf{X}\mathbf{X}^\top$
3. Eigendecomposition: $\mathbf{C} = \mathbf{W}\Sigma\mathbf{W}^\top$
4. Top-q extraction: $\mathbf{W} \leftarrow \mathbf{W}_{1:q}$
5. Code vectors: $\mathbf{Z} = \mathbf{W}^\top \mathbf{X}$

Despite the simplicity in description, this whole process is computational expensive, especially due to the computation of the covariance matrix $\mathbf{C}$. It is therefore desirable to skip computing the covariance matrix, and fortunately there is a way to do so by resorting to singular value decomposition.

Singular value decomposition (SVD) decomposes a given matrix $\mathbf{X}$ into a product of three matrices:

$$\mathbf{X} = \mathbf{W}\mathbf{S}\mathbf{V}, \tag{3.6}$$

where $\mathbf{W} \in \mathbb{R}^{d\times d}$ and $\mathbf{V} \in \mathbb{R}^{d\times N}$ are orthogonal matrices, and $\mathbf{S} \in \mathbb{R}^{d\times d}$ is a diagonal matrix with non-decreasing diagonal entries. Let us use this decomposition to compute the covariance matrix $\mathbf{C}$:

$$\begin{aligned}
\mathbf{C} =& \mathbf{X}\mathbf{X}^\top \\
=& \mathbf{W}\mathbf{S}\mathbf{V}(\mathbf{W}\mathbf{S}\mathbf{V})^\top \\
=& \mathbf{W}\mathbf{S}\underbrace{\mathbf{V}\mathbf{V}^\top}_{=\mathbf{I}}\underbrace{\mathbf{S}^\top}_{=\mathbf{S}}\mathbf{W}^\top \\
=& \mathbf{W}\mathbf{S}\mathbf{S}\mathbf{W}^\top \\
=& \mathbf{W}\Sigma\mathbf{W}^\top.
\end{aligned}$$

Surprised? Yes, the matrix $\mathbf{W}$ could be precisely recovered by SVD on the input matrix $\mathbf{X}$ without computing the covariance matrix. Because of this, it is common to directly use SVD on the input matrix in practice.

[5] It is out of the scope of this course to teach how to find eigenvalues and eigenvectors. I refer students to take, for instance, MATH-UA 140 LINEAR ALGEBRA listed at the Department of Mathematics.

[6] This is true only when the associated eigenvalues are positive.

Proportion of Variance Explained: PV The variance of the sum of decorrelated random variables is simply the sum of the variances of all those random variables, i.e.,

$$\mathrm{Var}(\sum_{i=1}^{q} Z_i) = \sum_{i=1}^{q} \mathrm{Var}(Z_i).$$

By considering each component of the code vectors as a random variable, this implies that the (empirical) variance of the sum of these code components is simply the sum of the first q eigenvalues: $\sum_{i=1}^{q} \sigma_i^2$. We can then compute how much variance has been *explained* by the code vectors by inspecting the ratio between the variance of the code components and the variance of the whole input:

$$\mathrm{PV}(q) = \frac{\sum_{i=1}^{q} \sigma_i^2}{\sum_{j=1}^{d} \sigma_d^2}.$$

What does this proportion of variance explained (PV) correspond to? We will see it in the upcoming demonstration session.

What are principal components? The principal components are the column vectors of the weight matrix $\mathbf{W}$. They are orthogonal to each other, and form a q-dimensional subspace in the original d-dimensional real space $\mathbb{R}^d$. The code vector $\mathbf{z}$ of an input vector $\mathbf{x}$ is then an orthogonal projection of $\mathbf{x}$ onto this subspace. This projection $\mathbf{z}$ obtained by Eq. (3.5) then corresponds to the *reconstruction* in the original space by

$$\hat{\mathbf{x}} = z_1\mathbf{w}_1 + z_2\mathbf{w}_2 + \cdots + z_q\mathbf{w}_q = \mathbf{W}\mathbf{z}. \tag{3.7}$$

Note that

$$z_i = \mathbf{w}_i^\top \mathbf{x}.$$

With this reconstruction, we can define a *reconstruction error* by

$$\|\mathbf{x} - \hat{\mathbf{x}}\|_2^2.$$

Now the question is what kind of subspace the procedure above finds. From the first constraint, that the code vectors are decorrelated with each other, we know that the principal components are eigenvectors of the input covariance matrix $\mathbf{C}$.[7] What does the second constraint tell us about our selection of the first q eigenvectors?

First, we notice that we can represent any input vector as a weighted sum of the eigenvectors, similarly to Eq. (3.7):

$$\mathbf{x} = z'_1\mathbf{w}_1 + z'_2\mathbf{w}_2 + \cdots + z'_d\mathbf{w}_d,$$

where

$$z'_i = \mathbf{w}_i^\top \mathbf{x}.$$

[7] Orthonormal vectors are orthogonal to each other and have a unit norm.

The reconstruction error can then be written as[8]

$$\begin{aligned}
\|\mathbf{x}-\hat{\mathbf{x}}\|_2^2 &= \left\| \sum_{i=1}^{q} (z'_i - z_i)\mathbf{w}_i + \sum_{j=q+1}^{d} z'_j \mathbf{w}_j \right\|_2^2 \\
&= \left\| \sum_{j=q+1}^{d} z'_j \mathbf{w}_j \right\|_2^2 \\
&= \sum_{j=q+1}^{d} z'^2_j \underbrace{\|\mathbf{w}_j\|^2}_{=1} \\
&= \sum_{j=q+1}^{d} z'^2_j \\
&= \sum_{j=q+1}^{d} \mathbf{w}_i^\top \underbrace{\mathbf{x}\mathbf{x}^\top}_{\mathbf{C}=\text{covariance}} \mathbf{w}_i \\
&= \sum_{j=q+1}^{d} \mathbf{w}_i^\top \mathbf{C}\mathbf{w}_i.
\end{aligned}$$

Now this looks awfully similar to the eigendecomposition from Eq. (3.3), where we learned that[9]

$$\begin{aligned}
&\Sigma = \mathbf{W}^\top \mathbf{C}\mathbf{W} \\
\iff &\sigma_j^2 = \mathbf{w}_j^\top \mathbf{C}\mathbf{w}_j, \text{ for all } j = 1,\ldots,d.
\end{aligned}$$

In other words, the reconstruction error equals to the sum of the eigenvalues of the discarded eigenvectors:

$$\|\mathbf{x}-\hat{\mathbf{x}}\|_2^2 = \sum_{j=q+1}^{d} \sigma_j^2.$$

Thus, we minimize the reconstruction error by selecting the top-q eigenvectors according to their corresponding eigenvalues as the principal components. If we want to add one more, we add another eigenvector whose eigenvalue is greater than equal to that of any other remaining eigenvector.

This view of PCA as finding a subspace that minimizes the reconstruction error becomes handy when we extend it to nonlinear PCA later. A deep autoencoder, which is one realization of nonlinear PCA, directly and explicitly minimizes the reconstruction error.

3.2.2 PCA: Minimum Reconstruction Error with Orthogonality Constraint

We have derived PCA from the two constraints; (1) the elements of a code vector $\mathbf{z}$ are decorrelated, and (2) the variances of the elements of a code vector are sorted in a decreasing order. We have seen that the first constraint led to the orthogonality of the weight matrix, or the dictionary matrix, $\mathbf{W}$. The second constraint, on the other hand, led to the minimum reconstruction error criterion. This latter revelation tells us that the second constraint of PCA is perhaps not a criterion on its own, but rather a consequence of optimization which has been at the core of machine learning so far in this course (except for the Bayesian linear regression.)

Here, we will re-formulate PCA by starting with an optimization cost, which is similar to the empirical cost function we have used throughout our discussions on supervised learning. The optimization cost function is defined as the reconstruction error over a given training set D_{tra}:

$$\begin{aligned}
\hat{R}(\mathbf{W},\mathbf{Z};D_{\text{tra}}) &= \sum_{n=1}^{N} \|\mathbf{x}_n - \hat{\mathbf{x}}_n\|_2^2 \\
&= \|\mathbf{X} - \mathbf{W}\mathbf{Z}\|_F^2.
\end{aligned}$$

[8] Note that I am looking at a single input vector $\mathbf{x}$, but this equation trivially extends to, and is in fact necessary to have, multiple input vectors $\mathbf{x}_1,\ldots,\mathbf{x}_N$. For brevity and simplicity, my derivation here considers a single input vector, but it is left for you as a **homework assignment** to extend this derivation to use multiple input vectors.

[9] It is your **homework assignment** to show that the equations below are true.

Looking closely at the equation at the bottom, we in fact see that this corresponds to finding the weight matrix $\mathbf{W}$ and the code matrix $\mathbf{Z}$ such that their product $\mathbf{WZ}$ is close to the input matrix $\mathbf{X}$, which was the main goal of matrix factorization from the very beginning as in Eq. (3.1).

This minimization alone does not however result in PCA, as naive minimization would not find a solution that satisfies the first constraint. We therefore impose that the first constraint be satisfies during optimization. That is,

$$\min_{\mathbf{W},\mathbf{Z}} \hat{R}(\mathbf{W},\mathbf{Z})$$

subject to the constraint that

$$\mathbf{W} \text{ is a orthogonal matrix.}$$

Assuming that this constraint is always satisfied during optimization, we can now safely replace $\mathbf{Z}$ with $\mathbf{W}^\top\mathbf{X}$, because $\mathbf{W}^\top\mathbf{W} = \mathbf{I}$. This results in

$$\hat{R}(\mathbf{W},\mathbf{Z};D_{\text{tra}}) = \|\mathbf{X} - \mathbf{W}\mathbf{W}^\top\mathbf{Z}\|_F^2$$

subject to subject to the constraint that

$$\mathbf{W} \text{ is a orthogonal matrix.}$$

This alternative derivation of PCA gives us a more general framework on top of which various matrix factorization algorithms could be implemented. In this general framework, the goal is to minimize the reconstruction error $\|\mathbf{X} - \mathbf{WZ}\|_F^p$. This is natural, as our goal is to find the weight and code matrices whose product is approximately the input matrix. We then need another mechanism g , or a function, that allows us to map from a given input vector $\mathbf{x}$ to its corresponding code vector $\mathbf{z}$, i.e. $\mathbf{z} = g(\mathbf{x},\mathbf{W})$. That is, we should be able to infer what the code matrix $\mathbf{Z}$ is given the input matrix $\mathbf{X}$ and the weight matrix $\mathbf{W}$.[10] Then, we need a certain constraint on either the weight matrix $\mathbf{W}$ and/or the code matrix $\mathbf{Z}$. In summary, a matrix factorization algorithm, or framework, is defined based on the following items:

1. Cost function (almost always reconstruction error)
2. Constraints on $\mathbf{W}$ and/or $\mathbf{Z}$
3. Inference mechanism g: infer $\mathbf{z}$ from $\mathbf{x}$ given $\mathbf{W}$

Furthermore, the appropriate choice of constraints relaxes the basic assumption we had earlier about the dimensionality of the code vector q that it is often much smaller than d.

To reiterate, in the case of PCA, the cost function is a reconstruction error defined in terms of Euclidean distance (or Frobenius norm). The inference mechanism is simply $\mathbf{z} = g(\mathbf{x}) = \mathbf{W}^\top\mathbf{x}$, and there is a single constraint that $\mathbf{W}$ is orthogonal.

Many different matrix factorization algorithms could be formulated under this framework. For instance, sparse coding [17] is defined by

1. Cost function: L2 reconstruction error
2. Constraint: $\|\mathbf{z}\|_0 = k$, for some $k > 0$
3. Inference mechanism: minimization with respect to $\mathbf{Z}$

It is often the case with sparse coding that $q \gg d$. Independent component analysis (see, e.g., [11])is on the other hand defined by

1. Cost function: L2 reconstruction error
2. Constraint: $z_i \perp z_j$, for all $i \neq j$

[10] Note that g may not be a function, but a process in which the reconstruction cost function is minimized with respect to the code matrix $\mathbf{Z}$ (instead of $\mathbf{W}$). This is a usual practice in sparse coding.

3. Inference mechanism: minimization with respect to $\mathbf{Z}$ or $g(\mathbf{x}) = \mathbf{U}\mathbf{z}$

$\mathbf{U}$ is a matrix separate from *mW* that recovers the code vector from an input vector. $\perp$ is used to denote the independence between two random variables (see Eq. (2.7).) Because of different choices of inference mechanism and constraints, these algorithms often learn different weight matrices and code matrices even when provided with the same input matrix $\mathbf{X}$.

In the next subsection, we will consider another matrix factorization algorithm, called non-negative matrix factorization, that is capable of learning a so-called *part-based representation*.

3.2.3 Non-negative Matrix Factorization: NMF

Let us consider another matrix factorization scheme called non-negative matrix factorization (NMF, [15]). Let us go step-by-step here. What is our main objective? Exactly the objective function we have used for principal component analysis (PCA):

$$\hat{R}(\mathbf{W},\mathbf{Z}) = \|\mathbf{X} - \mathbf{W}\mathbf{Z}\|_F^2.$$

What kind of constraints do we want? The name itself suggests them. That is, both the weight matrix $\mathbf{W}$ and the code matrix $\mathbf{Z}$ are *non-negative*. Of course, this naturally assumes that the input matrix $\mathbf{X}$ is also non-negative, because the sum, product or their combination, of non-negative numbers would never result in negative. What this implies is that we must ensure that the input matrix is non-negative by, for instance, subtracting the minimum value of $\mathbf{X}$ from $\mathbf{X}$. This is a preprocessing step that is similar to the centering step of PCA.

In summary, the optimization problem of NMT is defined as

$$\arg\min_{\mathbf{W},\mathbf{Z}} \|\mathbf{X} - \mathbf{W}\mathbf{Z}\|_F^2 \tag{3.8}$$

subject to

$$w_{ij} \geq 0, \text{ for all } i = 1,\ldots,d \text{ and } j = 1,\ldots,q$$
$$z_{ij} \geq 0, \text{ for all } i = 1,\ldots,q \text{ and } j = 1,\ldots,N.$$

Before learning how to solve this constrained optimization problem, let us discuss why NMF is an interesting case of matrix factorization.

As we discussed earlier, the weight matrix $\mathbf{W}$ is often called a dictionary matrix. This dictionary matrix contains a set of q atoms:

$$\mathbf{W} = [\mathbf{w}_1; \mathbf{w}_2; \cdots; \mathbf{w}_q].$$

These atoms are then combined according to their coefficients, given by a code vector, to form one of the input vectors. That is,

$$\hat{\mathbf{x}} = z_1\mathbf{w}_1 + z_2\mathbf{w}_2 + \cdots + z_q\mathbf{w}_q = \mathbf{W}\mathbf{z}.$$

Now let's do some quick thought experiment. Input vectors are human faces. What would be those atoms, when we are constrained to only *add* them? In other words, those positive-valued atoms and their positive-valued coefficients prevent us from cancelling out the contributions of the atoms. For instance, we cannot have an atom that has three positive noses and another atom that has two negative noses so that summing them would leave us only one nose. In order to build a full face with NMF, we would need to have an atom for a nose, an atom for a pair of eyes, an atom for a mouth, an atom for a pair of ears and so on. In other words, each positive-valued atom needs to contain a set of parts of a face that do not overlap with each other, and each positive-valued coefficient indicates how much contribution the corresponding atom makes to the full face.

This is precisely the motivation behind the NMT. Lee and Seung [15] showed that if you solve NMF on a set $\mathbf{X}$ of (normalized) faces, the dictionary matrix $\mathbf{W}$ contains face parts (such as mouth, eyes, nose, and so on), and the code matrix $\mathbf{Z}$ captures how some of those parts are selected and combined to form full faces. This process for one face is shown in Fig. 3.1.

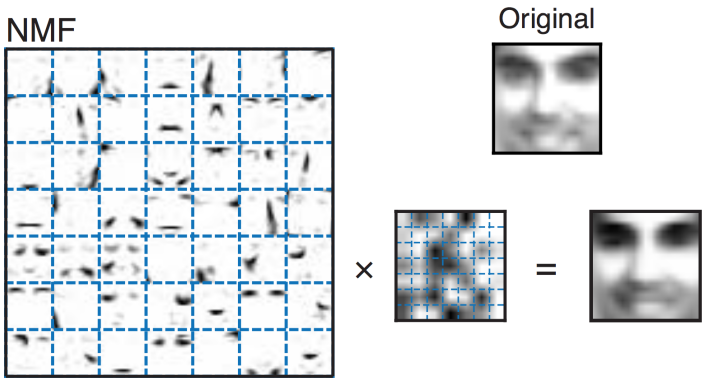


Figure 3.1: A graphical illustration of how non-negative matrix factorization models a face as a weighted sum of parts in a dictionary matrix $\mathbf{W}$. The figure was taken from [15].

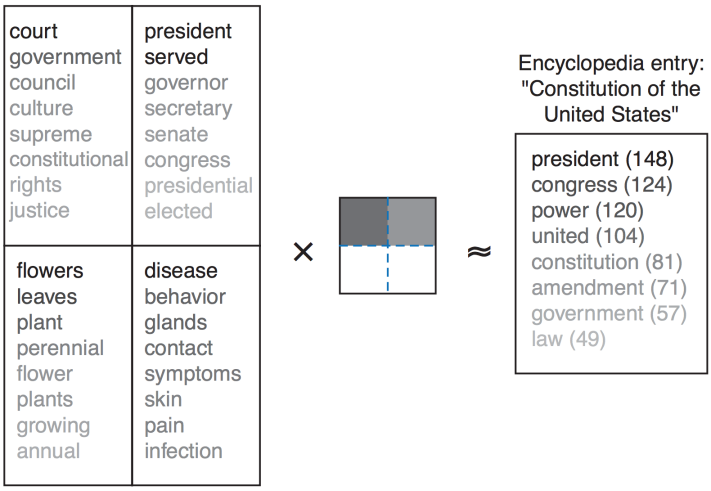


Figure 3.2: A graphical illustration of how non-negative matrix factorization models a document as a weighted sum of topics from a dictionary matrix $\mathbf{W}$. The figure was taken from [15].

Let us do another thought experiment. How about modelling a document? As we did earlier in Sec. 1.8.1, we assume each document is represented as a bag of words. What would be a good dictionary of atoms in this case? Perhaps each atom should represent a topic. If a document is about ice hockey, this document would be a result of adding multiple topics such as sports, hockey and ice. Again, each of these topics would have to be exclusive, since the non-negativity constraint prevents us from two atoms to *cancel out* each other. Again, this was one of the examples given in [15], as shown in Fig. 3.2.

The constrained optimization problem for NMF in Eq. (3.8) is not a trivial problem, and many optimization algorithms have been proposed to solve NMF. Since most of them are way out of the scope of this course, I will briefly describe a straightforward extension of gradient-descent algorithm, called projected gradient-descent algorithm. Unlike the usual gradient descent algorithm, the projected gradient-descent algorithm consists of two steps. The first step is to update the parameters (in our case, $\mathbf{W}$ and $\mathbf{Z}$) following the negative gradient direction:

$$\tilde{\mathbf{W}} = \mathbf{W} - \eta \nabla_{\mathbf{W}} \hat{R} = \mathbf{W} - \eta(\mathbf{X} - \mathbf{W}\mathbf{Z})\mathbf{Z}^\top, \tag{3.9}$$

$$\tilde{\mathbf{Z}} = \mathbf{Z} - \eta \nabla_{\mathbf{Z}} \hat{R} = \mathbf{Z} - \eta \mathbf{W}^\top(\mathbf{X} - \mathbf{W}\mathbf{Z}), \tag{3.10}$$

which is precisely the gradient-descent algorithm.

The second step (note that we have not updated the actual matrices yet) involves *projecting* these candidate matrices back to a feasible region which consists of all points that satisfy the constraints. That is,

$$\mathbf{W} = \arg\min_{\mathbf{W} \geq 0} \|\mathbf{W} - \tilde{\mathbf{W}}\|_F^2,$$

$$\mathbf{Z} = \arg\min_{\mathbf{Z} \geq 0} \|\mathbf{Z} - \tilde{\mathbf{Z}}\|_F^2.$$

In other words, we want to find, for instance, a weight matrix $\mathbf{W}$ with all non-negative values that is close to the updated matrix $\tilde{\mathbf{W}}$, and the same for the code matrix $\mathbf{Z}$. Finding such a projection is generally difficult, but in the case of NMF, the projection is simply

$$w_{ij} = \max(0, \tilde{w}_{ij}),$$
$$z_{ij} = \max(0, \tilde{z}_{ij}).$$

That is, we simply cut off any negative values from both the weight matrix and code matrix. This projected gradient algorithm is widely used for non-negative matrix factorization in addition to the original multiplicative update rules from [15].

We will see NMF in action during the next demonstration session.

3.3 Deep Autoencoders: Nonlinear Matrix Factorization

A major limitation of matrix factorization $\mathbf{X} \approx \mathbf{WZ}$ is that the relationship between a code vector $\mathbf{z}$ and its corresponding input vector $\mathbf{x}$ is *linear*. Given a fixed dimensionality q of a code vector, what happens if there is no good linear map from it to the corresponding input vector? Perhaps what we need is to relax this linearity assumption. That is,

$$\mathbf{x} = f_\theta(\mathbf{z}),$$

where f_θ is a nonlinear function parametrized by a set θ of parameters. We can use, for instance, a *deep* feature extraction function (or deep neural network) from Sec. 1.8.4.

As usual with the other matrix factorization methods above, we first define a cost function as a reconstruction error. A usual one is Euclidean distance as below:

$$\hat{R}(\theta, \mathbf{Z}) = \frac{1}{N} \sum_{n=1}^{N} \|\mathbf{x} - f_\theta(\mathbf{z})\|_2^2.$$

This is however not the only option. For instance, if $\mathbf{x}$ is a binary vector, it would be a good idea to use the negative log-probability under Bernoulli distribution, as we did with logistic regression earlier (remember?)

Now we need an inference mechanism for computing $\mathbf{z}$ given $\mathbf{x}$. In deep autoencoders, this is done by having another deep feature extraction function g_ϕ that maps from an input vector $\mathbf{x}$ to its corresponding code vector $\mathbf{z}$. That is,

$$\mathbf{z} = g_\phi(\mathbf{x}).$$

The question is where this new set ϕ of parameters comes from. The answer turns out to be more straightforward. We can simply find both θ and ϕ to minimize the reconstruction error. In other words,

$$\hat{R}(\theta, \phi) = \frac{1}{N} \sum_{n=1}^{N} \|\mathbf{x} - \underbrace{f_\theta}_{\text{decode}}(\underbrace{g_\phi}_{\text{encode}}(\mathbf{x}))\|_2^2.$$

In this problem, we see that g_ϕ encodes an input vector $\mathbf{x}$, and f_θ decodes the obtained code vector back into the input vector $\hat{\mathbf{x}}$. We thus call this approach a deep *autoencoder* [10].

It is now time to decide on the constraint. Unfortunately there is not a well-agreed-upon set of constraints for deep autoencoders. Some impose sparsity on the code vector $\mathbf{z}$ [6], some constraint that the dimensionality of the code vector is smaller than that of the input vector [10], or encourages the code vectors to be close to a binary vector by adding noise [21]. It is also possible to let the encoder f and decoder g share parameters in order to avoid a trivial solution in which f is a random mapping. It is out of this course's scope to discuss all these constraints, and I recommend you to read [8] and take the course *Deep Learning* taught by Prof. Yann LeCun.

3.4 Dimensionality Reduction beyond Matrix Factorization

3.4.1 Metric Multi-Dimensional Scaling (MDS)

So far we have considered matrix factorization for dimensionality reduction. A major property of such approaches was that we assumed a linear, or nonlinear in the case of deep autoencoders, mapping from a code vector to the corresponding input vector. That is, $\mathbf{x} \approx \mathbf{W}\mathbf{z}$ or $\mathbf{x} \approx f(\mathbf{z})$. In this section, we ask whether this is necessary.

A major disadvantage from this matrix factorization based approach is that all the input data points must be presented in their *vector* forms. What if this is not easily doable nor natural? For instance, consider embedding a graph which consists of many nodes, or vertices, and their connections, or edges. Unfortunately each node is anonymous in the sense that there is no description attached to it. Without any description, such as its absolute coordinate, it is not easy to label each node with its vector representation. Instead, we have a set of edges that define the connectivity among those nodes, and perhaps each edge comes with a weight that defines the similarity or distance between a pair of nodes. What would be represented as such a graph in a real life?

Let us think of a graph whose nodes correspond to all the cities in the world with their own airports. A pair of two cities i and j is connected with an edge, if there is more than one direct flight connection between them. The weight of the edge is defined inversely-proportional to the number of direct connections between them. In this graph, we do not have any obvious vector representation of each city, but we are only given the *distances* among them, defined as the edge weights d_{ij}. For any non-existing edge, we will assign $d_{\max} > 1$. Now we want to find N code vectors $\{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$ in a low-dimensional space $\mathbb{R}^q$, assuming that there are N cities.

The first step is to define a distance between each pair of code vectors. Let us use $f(\mathbf{z}_i, \mathbf{z}_j)$ for this distance. One example would be to use a Euclidean distance such that

$$f(\mathbf{z}_i, \mathbf{z}_j) = \|\mathbf{z}_i - \mathbf{z}_j\|_2^2.$$

Given this distance function, we now define a cost function of metric multidimensional scaling (MDS) as

$$\hat{R}(\mathbf{Z}; \mathbf{D}) = \left(\sum_{i<j} \frac{(f(\mathbf{z}_i, \mathbf{z}_j) - d_{ij})^2}{Z} \right)^{1/2}, \tag{3.11}$$

where Z is a normalization constant that determines a specific type of MDS. We can of course set it to 1.

What does this cost function of MDS do? The answer is quite straightforward. The cost function decreases as the distance between each pair of the code vectors is close to the distance between the corresponding input data points. The latter distance may be given arbitrarily without having to have a distance function defined over the input vector space.

Back to our example case earlier, I am plotting the cities according to how well connected they are to each other in terms of direct flight connections in Fig. 3.3. This plot was generated by the metric MDS using "scikit-learn" using the data made available from OpenFlights.[11] The plot is the result of applying MDS to the top-100 cities according to the number of direct flight connections from any other cities included in the database.

We notice some clusters of cities according to their continents, which is understandable as cities in the same continent are likely to have more connections among them. This is however not necessarily true across the continent boundaries, as hub cities are closely connected to each other. These hub cities are often placed on the boundaries between two or three continents. For instance, New York and Seoul are places between US and Asia. Similarly, Beijing is placed between Europe and Asia.

A major strength of MDS is that it does not require us to have a precise vector representation of each input data point. All that is necessary is a *dissimilarity* matrix which is a symmetric matrix of which each (i, j)-th element indicates how distant the i-th and j-th input points are. In this sense, one can think of MDS as a method of inferring the underlying vectors given the dissimilarity measures. This strength is at the same time its major weakness. That is, it is not trivial to find a code vector of a new data point which was not available at the beginning.

MDS is closely related to principal component analysis (PCA) we studied earlier. In PCA, an input vector $\mathbf{x}$ is assumed to be given by $\mathbf{W}\mathbf{z}$, where $\mathbf{W}$ is an orthonormal matrix. In this case, the cost function of MDS in Eq. (3.11) is zero automatically, if d_{ij} is the Euclidean distance between the input vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. It is your **homework assignment** to show that this is indeed a case.

[11] `http://openflights.org/data.html`

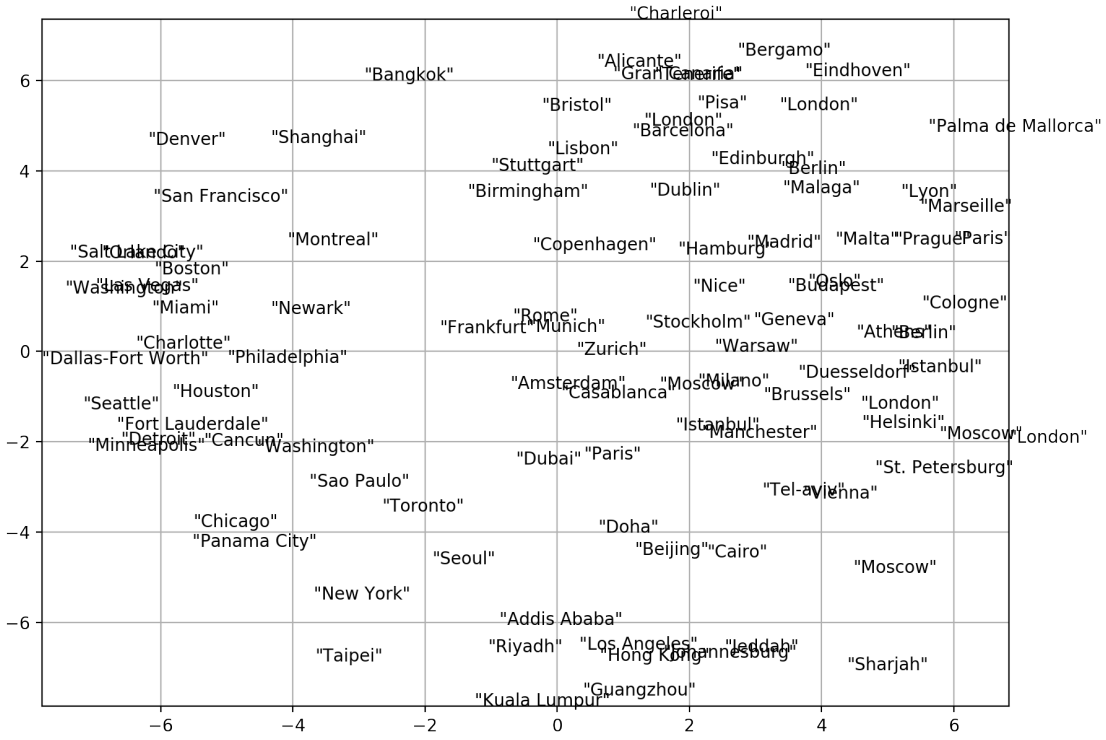


Figure 3.3: The scatter plot of cities according to how close they are in terms of the number of direct flight connections among them. The more direct flights there are the closer the cities are. Notice two major clusters of the cities. On the left, we see a cluster of the cities in the United States, while the cities in Europe are clustered on the right. However, because the distances are defined in terms of the airline connections, we see that their arrangements do not necessarily correspond to their geographical locations. For instance, "New York", "Toronto" and "Los Angeles" are located close to the cities in Asia, likely due to many intercontinental connections to those cities. See the jupyter notebook "MDS.ipynb" for details.

3.5 Further Topics*

Yet another way to derive principal component analysis, or factor analysis in general, is to build a probabilistic latent variable model. This approach is called probabilistic principal component analysis [20, 24], and is particularly interesting, because it naturally allows us to perform principal component analysis even when there are missing values in the input matrix [12, 22]. Similarly to Bayesian linear regression, we can consider both the weight and code matrices as random variables and marginalize both of them. This results in Gaussian process latent variable model and allows us to perform nonlinear matrix factorization [13].

Student-t stochastic neighbourhood embeddings (tSNE) !!!

Chapter 4

Clustering

4.1 k-Means Clustering

Clustering vs. Dimensionality Reduction From the 10,000 feet above the ground, the problem of dimensionality reduction was simply to find a set of code vectors $\{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$ given a set of input vectors (or a dissimilarity matrix, in the case of multidimensional scaling,) while preserving the similarities of input vectors as well as possible in the code vector space. As we went down closer to the ground, we notice that there are many different types of dimensionality reduction algorithms, including principal component analysis (PCA, Sec. 3.2.1) and non-negative matrix factorization (NMF, Sec. 3.2.3). These algorithms were generally distinguished from each other by different types of constraints that have been put either directly on the code vector or on the encoding or decoding functions.

Let us now consider an extreme constraint on a code vector that the code vector can only be an one-hot vector. An one-hot vector, as we learned earlier in Eq. (4.1), is a vector whose elements but one are all zeros. That is,

$$\mathbf{z} = \begin{bmatrix} 0, \\ \vdots, \\ 0, \\ 1, \\ 0, \\ \vdots, \\ 0 \end{bmatrix} \in \{0,1\}^q. \tag{4.1}$$

As we discussed earlier, an one-hot vector is equivalent to an integer index between 1 and q, and this allows us to use this code vector as an indicator of which of the q possible bins the corresponding input vector $\mathbf{x}$ belongs to. In other words, by constraining the code vector to be one-hot, we effectively transformed the problem of dimensionality reduction into clustering, where a code vector denotes which of q clusters the corresponding input vector belongs to.

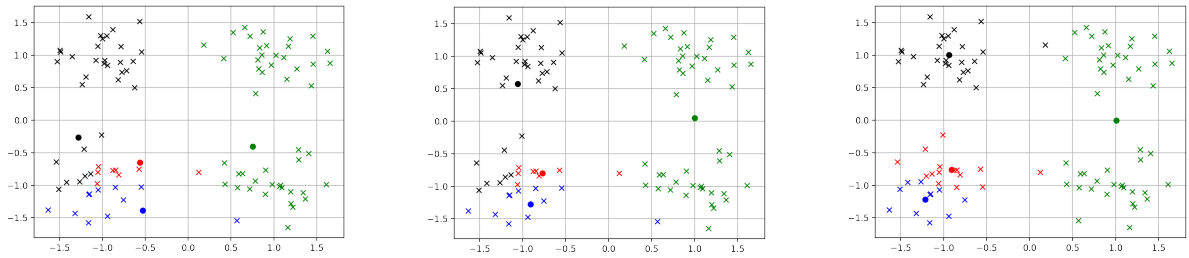
k-Means Clustering Let us start using k instead of q to denote the dimensionality of a code vector in order to be more consistent of a traditional description of clustering. That is, our goal is to cluster a given set of data points $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ into k groups, which is equivalent to obtaining a set of code vectors $\{\mathbf{z}_1, \ldots, \mathbf{z}_N\}$ with the constraint that each and every code vector is an one-hot vector.

As usual with the matrix factorization algorithms we have learned earlier, our goal is to minimize the reconstruction error:

$$\hat{R}(\mathbf{Z}, \mathbf{W}; \mathbf{X}) = \|\mathbf{X} - \mathbf{W}\mathbf{Z}\|_F^2 .$$

Because we constrain each code vector to be an one-hot vector, the reconstruction error can be rewritten as

$$\hat{R}(\mathbf{Z}, \mathbf{W}; \mathbf{X}) = \sum_{k'=1}^{k} \underbrace{\sum_{j:z_{j,k'}=1} \|\mathbf{w}_{k'} - \mathbf{x}_j\|_2^2}_{\text{within-cluster error}},$$



(a) Cluster Assignment (b) Centroid Estimation (c) Convergence

Figure 4.1: (a) The cluster assignment given cluster centroids. (b) The estimation of the cluster centroids given the cluster assignment. (c) The final result after iterating between the centroid estimation and the cluster assignment until convergence.

where $z_{j,k'}$ is the k'-th element of the j-th code vector $\mathbf{z}_j$. In other words, the reconstruction error can be computed separately for each k'-th cluster, given a fixed set of code vectors $\mathbf{Z}$.

Let us assume that we indeed have a fixed set of code vectors $\mathbf{Z}^{(0)}$, i.e., $\left\{\mathbf{z}_1^{(0)},\ldots,\mathbf{z}_N^{(0)}\right\}$. This implies that we have already divided the input vector set into k clusters, and we can minimize the reconstruction error by minimizing the within-cluster reconstruction error. For each k'-th cluster, we need to minimize

$$\hat{R}_{k'}(\mathbf{Z}^{(0)},\mathbf{W};\mathbf{X}) = \sum_{j:z_{j,k'}^{(0)}=1} \|\mathbf{w}_{k'}-\mathbf{x}_j\|_2^2$$

with respect to the associated weight vector $\mathbf{w}_{k'}$. This within-cluster reconstruction error has a unique solution which is

$$\mathbf{w}_{k'} \leftarrow \frac{1}{\left|\{j : z_{j,k'}^{(0)}=1\}\right|} \sum_{j:z_{j,k'}^{(0)}=1} \mathbf{x}_j. \tag{4.2}$$

In other words, the optimal associated weight vector $\mathbf{w}_{k'}$ is the arithmetic mean of all the input vectors in, or a *centroid* of, the corresponding k'-th cluster. Of course, these weight vectors, or centroids, are only optimal with respect to the current set of code vectors $\mathbf{Z}^{(0)}$. Let us use the superscript (0) to denote that these weight vectors $\mathbf{W}^{(0)}$ are optimal with respect to $\mathbf{Z}^{(0)}$.

Given these centroid $\mathbf{W}^{(0)}$, we now adjust the code vectors in order to minimize the reconstruction error. Unlike before when we looked at each cluster separately, we look at each input vector $\mathbf{x}$ separately this time:

$$\hat{R}_n(\mathbf{Z},\mathbf{W}^{(0)};\mathbf{X}) = \|\mathbf{x}_n-\mathbf{W}^{(0)}\mathbf{z}_n\|_2^2.$$

Minimizing this per-input reconstruction error is equivalent to first computing the distance between the input vector and each of the cluster centroids, and then selecting the cluster with the minimal distance. That is,

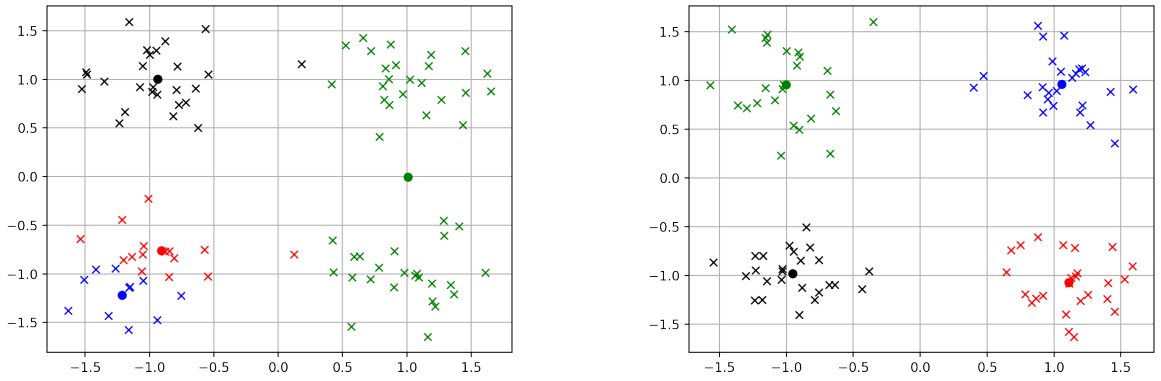
$$\mathbf{z}_n \leftarrow \arg\min_{k'} \hat{R}_n = \arg\min_{k'} \left\{\|\mathbf{x}_n-\mathbf{w}_{k'}\|_2^2\right\}. \tag{4.3}$$

In other words, the optimal code vector $\mathbf{z}_n$ for the input vector $\mathbf{x}_n$ corresponds to the assignment of the input vector to the cluster whose centroid is nearest. This is true for every input vector, and the optimal code vector set, or the *cluster assignment*, $\mathbf{Z}^{(1)}$ is the set of cluster assignment of each input vector, given the cluster centroids $\mathbf{W}^{(0)}$.

Of course, now that we have found new cluster assignment $\mathbf{Z}^{(1)}$, the current cluster centroids $\mathbf{W}^{(0)}$ are not anymore optimal. We hence go back to the earlier step and recompute the optimal cluster centroids $\mathbf{W}^{(1)}$. Then, the current cluster assignment $\mathbf{Z}^{(1)}$ becomes sub-optimal, and we must recompute them by assigning each input vector to the nearest cluster $\mathbf{Z}^{(2)}$. This iterative process continues until both the cluster centroids and the cluster assignment stop changing. Fortunately, this algorithm is guaranteed to terminate, although there is no guarantee that the converged solution is globally optimal.[1]

This algorithm is known as *k-means clustering*, and is widely used for clustering high-dimensional input vectors.

[1] A solution is globally optimal, when there exists no pair $\mathbf{Z}$ and $\mathbf{W}$ that results in the reconstruction error smaller.



(a) Solution 1

(b) Solution 2

Figure 4.2: Depending on the initialization of the cluster centroids, k-means clustering may end up with a different solution.

4.2 Further Topics*

Mixture of Gaussians* !!!

Spectral Clustering* !!!

Hierarchical agglomerative clustering* !!!

Chapter 5

Sequential Decision Making

5.1 Sequential Decision Making as a Series of Classification

The material covered so far in this course has largely assumed a static world, meaning that our machines were expected to work with a single input vector $\mathbf{x}$ regardless of what past nor future input vectors were. This setup covers a large portion of use cases of machine learning in practice. Consider, for instance, image caption generation, which is being rolled out as we speak in many commercial social net services. This caption generator would take as input a single image, and perhaps a bit of associated metadata, and generate an appropriate caption, regardless of which series of images it has worked on earlier and which series of images it will work on later.

Instead, let us now think of a more general setting in which a machine needs to make multiple decisions in order to arrive at a conclusion. There are many such cases. Think of, for example, driving. Every moment we make a decision on how to turn a steering wheel, whether to hit a brake or whether to hit a gas pedal. The effect, or success, of a series of all these decisions will only be apparent at the end of a trip, based on how quickly, safely and successfully the cat has arrived at an intended destination. Furthermore, your decision affects the surrounding environment. For instance, how you have decided to turn your steering wheel early on will affect where you end up in the future, and whether you have hit the brake immediately influences the speed at which the car drives in the next moment.

This setting could be understood as running a series of classification and/or regression. Our machine maps an input vector $\mathbf{x}$, that is an observation of the world, to its action $\mathbf{y}$. Unlike the previous static-world machines, the machine in this *sequential decision making* setting may optionally expose an additional auxiliary data about itself. This auxiliary data, represented as a vector $\mathbf{h}$ will be used as a part of the input in the next time, along the actual action taken by the machine in the previous step. In a more concise form,

$$[\mathbf{y}_t; \mathbf{h}_t] = M([\mathbf{x}_t; \mathbf{y}_{t-1}; \mathbf{h}_{t-1}]),$$

where t is the time index. If we assume for now that $\mathbf{h}_t = 0$ for all t, meaning that the machine does not maintain its own internal state, the equation simplifies to

$$\mathbf{y}_t = M([\mathbf{x}_t; \mathbf{y}_{t-1}]).$$

This equation reminds us of all the supervised learning methods, including classifiers and regression models, we have learned throughout this course; the machine tries to output a correct answer given an input vector.

Supervised Learning Considering this, what should be our first approach? Obviously, it is supervised learning. We assume that there exists a reference machine M^* that solves this kind of sequential decision making problem (nearly) perfectly. We let the reference machine run over and over, while collecting observations and the corresponding actions taken by the reference machine. That is, we build a training set:

$$D_{\text{tra}} = \left\{ (\mathbf{x}_1^1, \mathbf{y}_1^1), \ldots, (\mathbf{x}_T^1, \mathbf{y}_T^1), \ldots (\mathbf{x}_1^N, \mathbf{y}_1^N), \ldots, (\mathbf{x}_T^N, \mathbf{y}_T^N) \right\}.$$

For each pair $(\mathbf{x}, \mathbf{y}) \in D_{\text{tra}}$, we can define a distance function:

$$D(\mathbf{y}, M, \mathbf{x}),$$

as we have done for all the supervised learning methods so far. With this distance function, we can define and minimize an empirical cost function to fit our machine to solve a sequential decision making problem.

This approach based on supervised learning has been found to be surprisingly effective. For instance, Bojarski et al. [3] showed that you can train such a supervised classifier to drive an actual car on a road.

Of course, this does not mean that there is no issue with this supervised learning approach. A major issue is that error made by the classifier accumulates quadratically with respect to the length of an episode, where an episode is defined as a single, full run of the classifier. In order to avoid this, a number of *imitation learning*, or *learning to search*, algorithms have been proposed (see, e.g., [19]). Unfortunately those algorithms are out of this course's scope.

Reinforcement Learning: Policy Gradient In an extreme case, there may not be an expert from which we collect training example pairs. Instead we may only receive weak supervision by a supervisor, or the environment in which sequential decision making takes place, in the form of a scalar reward at each time step. Such a reward will be arbitrary, but eventually at the end of each episode, the sum of such rewards will be a good indicator of whether the sequence of decisions made during the episode was good.

We start with a random classifier M^0 which as before takes as input an observation of the surrounding environment $\mathbf{x}$ and outputs a decision $\hat{\mathbf{y}}$. Unlike supervised learning, we restrict ourselves to a probabilistic classifier, such as logistic regression, that outputs a conditional distribution $p(\mathbf{y}|\mathbf{x})$ over all possible decisions, or actions, $\mathbf{y}$ given an observation $\mathbf{x}$.

We now collect training examples using this random classifier M^0 by letting it run multiple times, i.e., multiple episodes. At each time step t of each m-th episode, M^0 receives an observation $\mathbf{x}_t^m$ and computes $p(\mathbf{y}|\mathbf{x}_t^m)$ from which one action $\hat{\mathbf{y}}_t^m$ is sampled. The classifier performs the sampled action, and the environment, or a (weak) supervisor, provides a reward r_t^m. From each episode, we then collect

$$((\mathbf{x}_1^m, \hat{\mathbf{y}}_1^m, r_1^m), \ldots, (\mathbf{x}_T^m, \hat{\mathbf{y}}_T^m, r_T^m))$$

which we turn into

$$((\mathbf{x}_1^m, \hat{\mathbf{y}}_1^m, Q_1^m), \ldots, (\mathbf{x}_T^m, \hat{\mathbf{y}}_T^m, Q_T^m)).$$

Q_t^m is the accumulated future reward defined as

$$Q_t^m = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}^m,$$

and is the indicator of whether the decision $\hat{\mathbf{y}}_t^m$ given an observation $\mathbf{x}_t^m$ was good or not. $\gamma \in (0, 1]$ is a discount factor, and in our case of a finite-length episode, can trivially set to 1.

Why is the accumulated future reward Q more important than the immediate reward r? Because a decision is not a good one even with a high immediate reward, if it eventually leads to a situation in the future in which only low rewards could be collected. A decision is only good, if it leads to a situation in the future in which an episode accumulates as much reward as possible.

We run the random classifier M^0 multiple times to collect as many such tuples of input vector, sampled decision and the corresponding reward. What we get is then a training set:

$$D_{\text{tra}}^0 = \{(\mathbf{x}_1, \hat{\mathbf{y}}_1, Q_1), \ldots (\mathbf{x}_N, \hat{\mathbf{y}}_N, Q_N)\}.$$

Using this training set, we now train our next classifier M^1. How should we do this?

One thing clear from the above dataset is that not every pair of observation and the corresponding decision was born equal. We want to make sure that the next classifier puts a higher probability on an associated decision $\mathbf{y}$ given an observation $\mathbf{x}$ *if and only if* the pair led to a high accumulated future reward, or Q. If the associated Q was low, we want to make sure that the next classifier M^1 puts a low probability. We can achieve this behaviour by modifying the empirical cost function into

$$\hat{R}(M^1; D_{\text{tra}}^0) = \frac{1}{N} \sum_{n=1}^{N} \underbrace{(Q_n - V_n)}_{\text{advantage}} D(\mathbf{y}_n, M^1, \mathbf{x}_n), \tag{5.1}$$

where D is a usual distance function defined per classifier. In the case of (multinomial) logistic regression, this distance function would be a cross-entropy loss from Eq. (1.27), and it would be a simple Euclidean distance in the case of linear regression.

What is V_n? V_n is a value of the observation $\mathbf{x}_n$, defined as

$$V_n = V(\mathbf{x}_n) = \mathbb{E}_{\mathbf{y} \sim M^0(\mathbf{x}_n)} \left[Q(\mathbf{y}, \mathbf{x}_n) = \sum_{\mathbf{y}} p(\mathbf{y}|\mathbf{x}_n, M^0) Q(\mathbf{y}, \mathbf{x}_n) \right],$$

where $Q(\mathbf{y}, \mathbf{x}_n)$ is the accumulated future reward should $\mathbf{y}$ was selected given an observation $\mathbf{x}_n$. In words, V_n computes what we believe would be the future accumulated reward given an observation $\mathbf{x}_n$ should we have let M^0 make a decision, on average. This is contrast to Q_n which computes how much reward in the future has been accumulated by selecting $\mathbf{y}_n$ given $\mathbf{x}_n$.

With this in our mind, the advantage term in Eq. (5.1) measure whether and how much better or worse the specific choice of $\mathbf{y}_n$ given $\mathbf{x}_n$ was, measured as Q_n, with respect to the expected performance V_n. If the choice led to something better than our expected performance, we encourage our next classifier to put more probability mass on the choice, and otherwise, we encourage it to put less probability mass.

Where does the V, often called a value function, come from? Clearly, it is not feasible to compute the value function exactly unless with a set of strict assumptions. It is thus a usual practice, at least recently, to train a yet another regression model to estimate the value of an observation $\mathbf{x}$ by minimizing

$$\frac{1}{N} \sum_{n=1}^{N} (V(\mathbf{x}_n) - Q_n)^2.$$

Once we train the next classifier M^1, we can go back to the trajectory collection stage. We run M^1 for multiple episodes to collect training example tuples of which each consists of an observation, sampled action and associated reward. We then minimize the empirical cost function in Eq. (5.1) in order to obtain another classifier M^2. We continue iterating these steps until we obtain our final classifier M^∞.

Training a machine for sequential decision making with only weak supervision is called reinforcement learning, and the specific approach discussed here is called (advantaged-based) *policy gradient*. Reinforcement learning is perhaps the most general form of machine learning. Unfortunately, the course is now ending, and I will have to refer anyone who is interested in learning more about reinforcement learning to the following materials:

1. Neuro-dynamic programming by Bertsekas and Tsitsiklis [1]
2. Introduction to Reinforcement Learning by Pineau: `http://videolectures.net/deeplearning2016_pineau_reinforcement_learning/`

5.2 Further Topics*

Learning to search !!!

Time-series modelling !!!

Bibliography

[1] D. P. Bertsekas and J. N. Tsitsiklis. Neuro-dynamic programming. In *Decision and Control, 1995., Proceedings of the 34th IEEE Conference on*. Athena Scientific, Belmont, MA, 1996.

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

[3] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[4] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[5] J. S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 211–217. Morgan-Kaufmann, 1990.

[6] K. Cho. Simple sparsification improves sparse denoising autoencoders in denoising highly corrupted images. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 432–440, 2013.

[7] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT Press, 2016.

[9] J. Goodman and J. Weare. Ensemble samplers with affine invariance. *Communications in applied mathematics and computational science*, 5(1):65–80, 2010.

[10] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[11] A. Hyvärinen, J. Karhunen, and E. Oja. *Independent component analysis*, volume 46. John Wiley & Sons, 2004.

[12] A. Ilin and T. Raiko. Practical approaches to principal component analysis in the presence of missing values. *Journal of Machine Learning Research*, 11(Jul):1957–2000, 2010.

[13] N. D. Lawrence. Gaussian process latent variable models for visualisation of high dimensional data. In *Advances in neural information processing systems*, pages 329–336, 2004.

[14] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[15] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.

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

[17] B. A. Olshausen and D. J. Field. Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision research*, 37(23):3311–3325, 1997.

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

[19] S. Ross, G. J. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, volume 1, page 6, 2011.

[20] S. Roweis. Em algorithms for pca and spca. *Advances in neural information processing systems*, pages 626–632, 1998.

[21] R. Salakhutdinov and G. Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009.

[22] R. Salakhutdinov and A. Mnih. Probabilistic matrix factorization. In *Neural Information Processing Systems*, volume 21, 2007.

[23] B. Schölkopf and A. J. Smola. *Learning with kernels: Support vector machines, regularization, optimization, and beyond*. the MIT Press, 2002.

[24] M. E. Tipping and C. M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999.

[25] C. K. Williams and C. E. Rasmussen. Gaussian processes for machine learning. *the MIT Press*, 2(3):4, 2006.