Tong Xiao

Jingbo Zhu

Natural Language Processing

Neural Networks and Large Language Models

NATURAL LANGUAGE PROCESSING LAB
NORTHEASTERN UNIVERSITY

&

NIUTRANS RESEARCH

https://github.com/NiuTrans/NLPBook
https://niutrans.github.io/NLPBook

NATURAL LANGUAGE PROCESSING LAB, NORTHEASTERN UNIVERSITY
&
NIUTRANS RESEARCH

https://github.com/NiuTrans/NLPBook
https://niutrans.github.io/NLPBook

June 12, 2025

Tong Xiao and Jingbo Zhu
June, 2025

Chapter 1

Foundations of Machine Learning

> *The goal of machine learning is to develop methods that can automatically detect patterns in data, and then to use the uncovered patterns to predict future data or other outcomes of interest.*
>
> – Murphy [2012]

> *Machine learning can be broadly defined as computational methods using experience to improve performance or to make accurate predictions.*
>
> – Mohri et al. [2018]

Data-driven NLP fits the above definitions[1]. It teaches computers to learn language *experience* from corpora, and to understand and utilize language based on that *experience*. Connecting machine learning (ML) with natural language processing is much more than a means that makes computers mimic human language intelligence from data. It is leading a revolution in both areas: natural language processing evolves by using a powerful tool of deriving meaning from corpora, and machine learning evolves by addressing the NLP challenges and testing on real-world data.

In this chapter, we present several basic concepts and models in machine learning. There are no tough bits but some preliminaries for the subsequent chapters. Here we focus on how to apply machine learning to NLP problems, in particular how to define an NLP problem as a statistical learning problem. To do this, we start with classification — one of the most widely-used examples in most introductory books. We then present several fundamental issues of machine learning. They are followed by a discussion on NLP problems from the machine learning perspective.

[1]We drop the term *data-driven* from now on and assume that all NLP models are data-driven in the remainder of this document.

1.1 Math Basics

In the remainder of this chapter and the following chapters, we will talk about machine learning problems using the tool of applied mathematics. Here are the math basics. If you find the details trivial, you can skip to Section 1.2 directly.

1.1.1 Linear Algebra

1. Vectors and Matrices

Scalar may or may not be the simplest concept in linear algebra, but is surely the most common concept that one learns in high school or in university. A scalar is a number. It is a quantity that has a magnitude but has no direction. For example, height, weight, distance, temperature are all examples of scalars. Here we use an italic number to denote a scalar, for example, a, b, x, A, and so on.

Vector and **matrix** are defined on top of scalar. A vector is an array of scalars, or simply a number list. A matrix is a rectangular array of scalars. In this book, we follow the convention of using bold letters to denote vectors and matrices. For example, an n-dimensional vector can be written as

$$\mathbf{a} = \begin{bmatrix} a_1 & a_2 & ... & a_n \end{bmatrix} \tag{1.1}$$

where $\{a_1, a_2, ..., a_n\}$ are the elements (or entries) of the vector. Each indicates a dimension. For convenience of notation, we write a_i as $a(i)$ sometimes. A vector is a real-valued vector only if all the elements are real numbers (i.e., $a_i \in \mathbb{R}$ for each i), denoted as $\mathbf{a} \in \mathbb{R}^n$. Likewise, we can write an $m \times n$ matrix as

$$\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & ... & A_{1n} \\ A_{21} & A_{22} & ... & A_{2n} \\ ... & ... & ... & ... \\ A_{m1} & A_{m2} & ... & A_{mn} \end{bmatrix} \tag{1.2}$$

where m is the number of rows and n is the number of columns. A_{ij} is the entry (i, j) of the matrix. A real-valued matrix is denoted as $\mathbf{A} \in \mathbb{R}^{m \times n}$. Occasionally, we use $\mathbf{A}_{m \times n}$ to emphasize that the shape of the matrix is $m \times n$.

There are a few special matrices. For example, a matrix whose elements are all zeros is a **zero matrix**, denoted as $\mathbf{0}$. Another example is **identity matrix**, denoted as $\mathbf{I}$. It is a square matrix whose diagonal elements are all 1, and other elements are 0. Vectors can be treated as a special sort of matrices, too. For example, the vector in Eq. (1.1) is a matrix with only one row.

2. Matrix Transpose

The **transpose** of a matrix $A_{m\times n}$ is a matrix $B_{n\times m}$ subject to $A_{ij} = B_{ji}$ for each pair of i and j. Often, $\mathbf{A}$'s transpose is denoted as $\mathbf{A}^{\mathrm{T}}$. For example, for a matrix

$$\mathbf{A} = \begin{bmatrix} 8 & 0 & 0 \\ 2 & 9 & 7 \end{bmatrix} \tag{1.3}$$

the transpose is

$$\mathbf{A}^{\mathrm{T}} = \begin{bmatrix} 8 & 2 \\ 0 & 9 \\ 0 & 7 \end{bmatrix} \tag{1.4}$$

One can transpose a vector as well. For a vector

$$\mathbf{a} = \begin{bmatrix} 1 & 9 & 7 & 3 \end{bmatrix} \tag{1.5}$$

the transpose is

$$\mathbf{a}^{\mathrm{T}} = \begin{bmatrix} 1 \\ 9 \\ 7 \\ 3 \end{bmatrix} \tag{1.6}$$

In general, a vector with only one row is called a **row vector** (as in Eq. (1.5)), and a vector with only one column is called a **column vector** (as in Eq. (1.6)). In this book, all vectors are row vectors by default.

3. Element-wise Operations on Matrices

Suppose $\mathbf{A}$ and $\mathbf{B}$ are two matrices of the same shape, say $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m\times n}$. The **matrix addition** of $\mathbf{A}$ and $\mathbf{B}$ is written as $\mathbf{A}+\mathbf{B}$. $\mathbf{A}+\mathbf{B}$ is a matrix in $\mathbb{R}^{m\times n}$ such that each element is the sum of the corresponding elements of $\mathbf{A}$ and $\mathbf{B}$. Here is an example.

$$\begin{aligned} \mathbf{A}+\mathbf{B} &= \begin{bmatrix} 8 & 0 & 0 \\ 2 & 9 & 7 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 4 \end{bmatrix} \\ &= \begin{bmatrix} 9 & 1 & 1 \\ 3 & 9 & 11 \end{bmatrix} \end{aligned} \tag{1.7}$$

In a similar way, we can define element-wise minus ($\mathbf{A}-\mathbf{B}$), product ($\mathbf{A}\odot\mathbf{B}$), division ($\mathbf{A}\oslash\mathbf{B}$) and other operations. A special case of element-wise product is that we multiply a matrix $\mathbf{A}$ with another matrix whose elements are all the same (say k). It is equal to scaling $\mathbf{A}$ with a scalar k, denoted as $k\times\mathbf{A}$ or $k\mathbf{A}$. This is also called **scalar product**. See below for an

example for $k = 2$ and $\mathbf{A} = \begin{bmatrix} 8 & 0 & 0 \\ 2 & 9 & 7 \end{bmatrix}$.

$$\begin{aligned} k\mathbf{A} &= 2\begin{bmatrix} 8 & 0 & 0 \\ 2 & 9 & 7 \end{bmatrix} \\ &= \begin{bmatrix} 16 & 0 & 0 \\ 4 & 18 & 14 \end{bmatrix} \end{aligned} \tag{1.8}$$

Let $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ be matrices in $\mathbb{R}^{m \times n}$, and k and l be scalars in $\mathbb{R}$. Some properties of the matrix operations are:

- Property of the zero matrix:

$$\begin{aligned} \mathbf{A} &= \mathbf{A} + \mathbf{0} \\ &= \mathbf{0} + \mathbf{A} \end{aligned} \tag{1.9}$$

- **Commutativity**:

$$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A} \tag{1.10}$$

$$kl\mathbf{A} = lk\mathbf{A} \tag{1.11}$$

- **Associativity**:

$$(\mathbf{A} + \mathbf{B}) + \mathbf{C} = \mathbf{A} + (\mathbf{B} + \mathbf{C}) \tag{1.12}$$

$$(kl)\mathbf{A} = l(k\mathbf{A}) \tag{1.13}$$

- **Distributivity**:

$$k(\mathbf{A} + \mathbf{B}) = k\mathbf{A} + k\mathbf{B} \tag{1.14}$$

$$(k + l)\mathbf{A} = k\mathbf{A} + l\mathbf{A} \tag{1.15}$$

4. Dot Product

The **dot product** of two same-sized vectors $\mathbf{a} = \begin{bmatrix} a_1 & a_2 & ... & a_n \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} b_1 & b_2 & ... & b_n \end{bmatrix}$ is defined to be:

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_1 b_1 & a_2 b_2 & ... & a_n b_n \end{bmatrix} \tag{1.16}$$

In geometry, a real-valued vector $\mathbf{a}$ can be seen as a geometric object having both magnitude (denoted as $|\mathbf{a}|$) and direction. The dot product of $\mathbf{a}$ and $\mathbf{b}$ can also be defined as

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| \times |\mathbf{b}| \times \cos(\theta) \tag{1.17}$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

5. Matrix Product

Matrix product (or **matrix-matrix product**) operates on two matrices. Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times p}$ and a matrix $\mathbf{B} \in \mathbb{R}^{p \times n}$, the matrix product of $\mathbf{A}$ and $\mathbf{B}$ produces a matrix $\mathbf{C} \in \mathbb{R}^{m \times n}$ whose elements are defined as:

$$\begin{aligned} C_{ij} &= \sum_{k=1}^{p} A_{ik} \times B_{kj} \\ &= A_i 1 \times B_{1j} + A_{i2} \times B_{2j} + ... + A_{ip} \times B_{pj} \end{aligned} \tag{1.18}$$

Matrix product requires that the number of columns in $\mathbf{A}$ is exactly the same as the number of rows in $\mathbf{B}$. In this book we use $\mathbf{AB}$ to denote the matrix product of $\mathbf{A}$ and $\mathbf{B}$. Here are a few properties of matrix product.

- Distributivity. For $\mathbf{A} \in \mathbb{R}^{m \times p}$ and $\mathbf{B}, \mathbf{C} \in \mathbb{R}^{p \times n}$, the left distributivity is defined as

$$\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB}+\mathbf{AC} \tag{1.19}$$

  For $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m \times p}$ and $\mathbf{C} \in \mathbb{R}^{p \times n}$, the right distributivity is defined as

$$(\mathbf{A}+\mathbf{B})\mathbf{C} = \mathbf{AC}+\mathbf{BC} \tag{1.20}$$

- Associativity. For $\mathbf{A} \in \mathbb{R}^{m \times p}$, $\mathbf{B} \in \mathbb{R}^{p \times q}$ and $\mathbf{C} \in \mathbb{R}^{q \times n}$, the associativity defines

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC}) \tag{1.21}$$

- Transpose. For $\mathbf{A} \in \mathbb{R}^{m \times p}$ and $\mathbf{B} \in \mathbb{R}^{p \times n}$, we have

$$(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}} \tag{1.22}$$

Matrix product is not commutative, i.e., we do not have $\mathbf{AB} = \mathbf{BA}$ for all $\mathbf{A}$ and $\mathbf{B}$ even if $\mathbf{A}$ and $\mathbf{B}$ are square matrices with the same shape. Based on matrix-matrix product, we can define vector-matrix product and matrix-vector product accordingly. This is trivial because all we need is to see a vector as a matrix in multiplication.

6. Norm

In a vector space, **norm** is a measure of vector "length". Given a vector $\mathbf{a} = \begin{bmatrix} a_1 & a_2 & ... & a_n \end{bmatrix} \in \mathbb{R}^n$, the norm on $\mathbf{a}$ is a function that maps from $\mathbb{R}^n$ to $\mathbb{R}$. It is written as $\|\cdot\|_p$, or l_p for short. $\|\mathbf{a}\|_p$ is defined as

$$\|\mathbf{a}\|_p = \left(\sum_{i=1}^{n} |a_i|^p \right)^{1/p} \tag{1.23}$$

It is called p**-norm** or l_p **norm**. The popular versions of p-norm are those when $p = 1, 2$ and

∞:

$$\|\mathbf{a}\|_1 = \sum_{i=1}^{n} |a_i| \tag{1.24}$$

$$\|\mathbf{a}\|_2 = \sqrt{\sum_{i=1}^{n} |a_i|^2} \tag{1.25}$$

$$\|\mathbf{a}\|_\infty = \max\{|a_1|, |a_2|, ..., |a_n|\} \tag{1.26}$$

2-norm and ∞-norm are also called **Euclidean norm** and **maximum norm**. p-norm can also be used in measuring the distance of two points in an n-dimensional space. Let b be another vector in $\mathbb{R}^n$. The **p-norm distance** between $\mathbf{a}$ and $\mathbf{b}$ is given by the equation:

$$\|\mathbf{a} - \mathbf{b}\|_p = \left(\sum_{i=1}^{n} |a_i - b_i|^p\right)^{1/p} \tag{1.27}$$

1.1.2 Probability and Statistics

1. What is Probability

Probability is a matter of uncertainty. It is a quantity that describes how likely an event is to happen. For example, if the event is certain to happen, we will say that the probability is 1; if the event will never happen, we will say that the probability is 0.

Then, what is an event? In short, an event is an experimental outcome. It could be simply the result of everything. For example, an event could be the action that you raised your arms, the scene that you were seeing the sunset, the result that you figured out for a math quiz, and so on. A set of related events is described by a **random variable** or **variable** for short. For example, we can define the outcome of tossing a coin as a variable x. As there are two outcomes (heads or tails), we have two choices for the value of x. We can define that $x = 1$ when the coin lands heads, and $x = 0$ otherwise. Hence, x is a **binary variable** or more precisely a 0-1 variable. A variable choosing a value means that an event happens. For example, $x = 1$ means the event of the coin landing heads.

In mathematics, probability is a measure on the probability space comprising events (call it a **probability measure**). As a measure, probability should satisfy certain properties, such as countable additivity [Ash and Doléans-Dade, 1999]. This means that not all functions defined on the interval $[0, 1]$ could be a probability measure. Here we do not discuss the precise definition of probability measure. We just simply treat it as a function that outputs a real number in $[0, 1]$.

Usually, a probability measure is denoted as a function $\Pr(\cdot)$, called a **probability function**. When the input of $\Pr(\cdot)$ is defined on a discrete set of events, the output of the function is the probability that an event happens. For example, $\Pr(x = 1)$ means the probability of x equalling 1. Note that $\Pr(x)$ is a function that varies its output by choosing different values of x. Suppose x_1 is a value that x can take. When we write $\Pr(x_1)$, it means $\Pr(x = x_1)$. Because the probability over all events should be 1, any probability function should be subject

to:

$$\sum_{x_i \in X} \Pr(x_i) \quad = \quad 1 \tag{1.28}$$

where X is the set of all events. A probability function can be defined on two variables or more. Here are a few widely-used cases.

- **Joint Probability**. It is the probability that two events x_1 and y_1 happen at the same time, denoted as $\Pr(x_1, y_1)$. As a special case, the joint probability will be decomposed into the product of the probabilities of x_1 and y_1, if x_1 and y_1 are independent of each other.

$$\Pr(x_1, y_1) = \Pr(x_1) \Pr(y_1) \tag{1.29}$$

- **Conditional Probability**. It is the probability that x_1 happens in the presence of y_1 happening, denoted as $\Pr(x_1|y_1)$. $\Pr(x_1|y_1)$ can be defined as:

$$\Pr(x_1|y_1) \quad = \quad \frac{\Pr(x_1, y_1)}{\Pr(y_1)} \tag{1.30}$$

- **Marginal Probability**. It is another way to define the probability of a single variable. Given the joint probability on two variables, the marginal probability defines that

$$\Pr(x_1) \quad = \quad \sum_{y_j \in Y} \Pr(x_1, y_j) \tag{1.31}$$

  where Y is the event space of y_j. Eq (1.31) says that $\Pr(x_1)$ is unconditioned on Y.

Another note on joint probability. In some cases, one would like to use conditional probabilities to represent a joint probability. To this end, one can rewrite the joint probability by the **product rule** or the **chain rule**, like this

$$\Pr(x_1, y_1) = \Pr(x_1|y_1) \Pr(y_1) \tag{1.32}$$

So far, we have defined several kinds of probability on discrete variables. For continuous variables, we do not have a "probability" at a certain point. Instead, we have a density for that point. More formally, given a continuous variable x, a **probability density** of x is written as $\Pr(x)$. Suppose $x \in \mathbb{R}$. The probability of x lying in the interval $[a, b]$ is defined via an integral:

$$\Pr(x \in [a, b]) \quad = \quad \int_a^b \Pr(x) dx \tag{1.33}$$

Obviously, we have

$$\int_{-\infty}^{+\infty} \Pr(x)dx = 1 \tag{1.34}$$

For other properties, such as joint probability and conditional probability, the forms for continuous variables are almost the same as those for discrete variables. We just need to replace the sums in the formulas with the integrals.

2. Distribution and Expectation

A **probability distribution** (or **distribution** for short) is the probabilities of different values for a variable. It is defined by probability functions (for discrete variables) or probability density functions (for continuous variables). For example, a uniform distribution on a discrete variable x that chooses values from $\{x_1, x_2\}$ can be described as $\Pr(x_i) = 1/2$ because $\Pr(x_1) = \Pr(x_2)$ and $\Pr(x_1) + \Pr(x_2) = 1$; A uniform distribution on a continuous variable in $[-2, 2]$ can be described as $\Pr(x) = 1/4$ because $\Pr(x)$ is a constant for any $x \in [-2, 2]$ and $\int_{-2}^{2} \Pr(x)dx = 1$. Statisticians have developed many distributions for describing the world we are living in, such as binomial distribution, Bernoulli distribution and Gaussian/normal distribution. One can find details of these distributions in most textbooks on statistics [McClave and Sincich, 2006; Freedman et al., 2007].

For describing properties of a variable, a popular means is to compute the **expected value** or **expectation** of the variable. Let x be a discrete variable that takes values from $\{x_1, ..., x_n\}$, and $\Pr(x)$ be a distribution on x. The expected value of x is defined to be

$$\mathbb{E}_{x\sim\Pr(x)}(x) = \sum_{i=1}^{n} x_i \cdot \Pr(x_i) \tag{1.35}$$

where the subscript $x \sim \Pr(x)$ indicates that x follows the distribution $\Pr(x)$. In many cases, we can drop the subscript and rewrite it as $\mathbb{E}(x)$. $\mathbb{E}(x)$ is essentially the weighted average value of x under the distribution $\Pr(x)$. It is a measure of central tendency, and is sometimes called the **mean** of a variable (denoted as μ).

Then, we can define the **variance** of a variable as the squared variation of the variable from the mean value, like this

$$\mathrm{Var}(x) = \mathbb{E}[(x - \mathbb{E}(x))^2] \tag{1.36}$$

Informally, it describes how far the values are from the mean. $\mathrm{Var}(x)$ is usually written as σ^2, where σ is called **standard deviation**.

For a continuous variable $x \in \mathbb{R}$, the expected value is defined as:

$$\mathbb{E}(x) = \int_{-\infty}^{+\infty} x \cdot \Pr(x)dx \tag{1.37}$$

where $\Pr(x)$ is a probability density function. For computing the variance of x, we just reuse

Eq. (1.36).

3. Entropy

Entropy is one of the most important tools of describing random variables and processes [Shannon, 1948]. It is a measure of expected surprise. The more deterministic the events occur, the less surprise and less information there will be. For simplicity, we restrict the discussion on discrete variables here[2]. Let x be a variable and $\Pr(x)$ be a distribution on x. The entropy is written as:

$$H(x) = -\sum_{i=1}^{n} \Pr(x_i) \cdot \log_b \Pr(x_i) \tag{1.38}$$

where b is the base of the logarithm function. The value of b is typically set to 2, 10 and e.

In addition to obtaining the entropy of a single distribution, we can determine the similarity of two distributions from the entropy point of view. Suppose $p(x)$ and $q(x)$ are distributions on x. The **relative entropy** of p with respect to q is defined to be:

$$D_b(p||q) = \sum_{i=1}^{n} p(x_i) \cdot \log_b \frac{p(x_i)}{q(x_i)} \tag{1.39}$$

We can treat $p(x_i)$ as a weight to the log likelihood ratio $\log_b \frac{p(x_i)}{q(x_i)}$. Hence, $D_b(p||q)$ is a weighted sum of the likelihood ratios over all possible values. A smaller value $D_b(p||q)$ indicates that distributions p and q are closer. For example, p and q will be identical if $D_b(p||q) = 0$. The relative entropy is also called the **Kullback-Leibler (KL) divergence**. Note that the relative entropy is asymmetric, i.e., we cannot guarantee $D_b(p||q) = D_b(q||p)$.

Another concept that is popular in machine learning is **cross-entropy**. It is a measure of the information (in terms of the total number of bits) that we need to transit the events from a source in one distribution with another distribution. More formally, we write the cross-entropy of the distribution p with the distribution q as $H_{\text{cross}}(p, q)$. It is defined to be:

$$H_{\text{cross}}(p, q) = \sum_{i=1}^{n} p(x_i) \cdot \log_2 q(x_i) \tag{1.40}$$

Like relative entropy, cross-entropy is asymmetric too. Both relative entropy and cross-entropy are widely used in designing the objective of learning NLP systems although they are different quantities. The difference lies in that relative entropy calculates the average number of bits when replacing p with q, while cross-entropy calculates the total number of bits in the same process.

[2]For continuous variables, we have similar calculations.

1.2 Designing a Text Classifier

Classification is one of the most common problems in machine learning. It aims at automatically categorizing something into a set of classes. These classes are called **labels**, or **tags**, or **categories**. In general, a program of classification is called a **classifier** or **classification system**. There are a vast number of practical applications of classifiers. A simple example is spam filtering in that one needs to label an email as "spam" or "not-spam". More challenging examples include classifying computed tomography images of organs into "normal" or "not-normal", determining whether a piece of Chinese text is written by native speakers or not, labeling a patent application with a set of IPC codes it belongs to, and so on.

Many machine learning theories and algorithms are modeled and tested on classification tasks. Following this convention, we consider text classification as an example to get started. Assume that we have a corpus like this.

Text	Label
The game was wonderful.	Not-food
I've tried my best to recreate it in my kitchen. It tastes heavenly.	Food
For centuries seaweed was considered a food for normal people.	Food
Have you finished your coding work today?	Not-Food
I was wondering how you could miss the bus.	Not-Food
I like fruit because it is good for health.	Food
Natural language processing research is amazing.	Not-Food
...	...

Each line of the corpus is a tuple of a piece of text (we simply call it a document) and a label that indicates whether the text is about food or not. We call such tuples **samples**, or more precisely **labeled samples**. Labeled samples are essentially question-answer pairs although they do not strictly follow the general forms of questions and answers. For example, in the samples presented above, one can take a document as a question and take its label as the answer. In the next few chapters, we will show that such a form of describing machine learning problems is general and fits most problems in NLP.

Next, let us assume that we have a classifier that learns from those samples the way of labeling documents. The classifier is then used to label every new document as "Food" or "Not-Food". For example, for the text

Fruit is not my favorite but I can enjoy it.

the classifier would categorize it as "Food".

However, text classification, though seems simple on the surface, is much more than classifying or sorting unlabeled samples into classes. It presents a wide variety of issues, especially when considering the ambiguities and richness of language. Modern classifiers are not a system comprising a set of hand-crafted rules. They instead model the classification problem in a probabilistic manner, making it possible to learn the ability of classification from large-scale labeled data.

1.2.1 Problem Statement

Let x be a document and c be a label. Here we assume a probabilistic classifier which would estimate how likely we choose c as the label of x, denoted as $\Pr(c|x)$. $\Pr(c|x)$ is in general a **classification model**. It describes a distribution over the set of all possible labels, satisfying

$$\sum_{c \in C} \Pr(c|x) \quad = \quad 1 \tag{1.41}$$

where C is the label set. For any document, we choose the most probable label as output via the classification model, like this

$$\hat{c} \quad = \quad \arg\max_{c \in C} \Pr(c|x) \tag{1.42}$$

where $\hat{c}$ is the "best" label predicted by the model. $\arg\max$ is the abbreviation of the arguments of the maxima. It returns the value of the argument that maximizes some function.

Eq. (1.42) is the fundamental equation of classification. It implies three problems

- **The modeling problem**. $\Pr(c|x)$ is a computational challenge because it is not obvious how to obtain the value of $\Pr(c|x)$ for each pair of x and c. To make an adequate model, one may need to represent x and c in some way that is easy to use, and may need to develop some mathematical form connecting x and c together with the algorithms necessary to compute the form.
- **The learning problem**. From a statistical learning point of view, the general form of $\Pr(c|x)$ represents a range of models configured with different variables or parameters. These models are essentially of the same form but would behave differently if we choose different values of those parameters. Thus, we need to choose a "good" model among them. This is typically addressed by optimizing the parameters on labeled data by some criterion.
- **The prediction problem**. We are addressing a **binary classification** problem here. Predicting document class is thus trivial as we just need to determine which class is more probable than the other. However, one can hardly imagine how difficult the prediction problem is in the real world, especially when predicting tree or graph-like structures and other non-linear structures[3]. For many NLP problems, prediction needs effective and efficient search algorithms.

These problems are general and cover many machine learning and natural language processing tasks. Binary classification, though is one of the simplest cases, can fully complete the goal of getting familiar with machine learning. On the other hand, classification has several variants. Here are two examples.

- **Multi-class classification**. It is an updated version of binary classification. In multi-class

[3] Predicting trees or other structures is not recognized as a standard sub-problem of classification. It is typically referred to as **structure prediction**. We will show in the later sections that both classification and structure prediction can share a similar machine learning paradigm.

classification, one needs to classify samples into one of three or more classes.

- **Multi-label classification**. This might be confusing because multi-class classification and multi-label classification seem to be the same thing. By conventional use of the terms, multi-label classification is referred to as assigning multiple labels to a sample. By contrast, the problem presented in Eq. (1.42) is a **single-label classification** problem.

Classification would be more interesting and challenging if we extend it to the case of dealing with hierarchical data. For example, for biological and patent data, some classes can be grouped into a super-class. This makes a hierarchy of the classes and requires a hierarchical classification schema.

1.2.2 Documents as Feature Vectors

The first problem we confront in designing text classification models is how to represent a document. Treating x as a string is simply not a good solution. One may want a representation by which a human being can understand the text. For example, we can parse each sentence in a document into a syntax tree and use trees as a text representation. This, however, requires efforts for developing additional NLP tools (such as syntactic parsers).

Representation, of course, is a fundamental issue in NLP. We skip here those diverse, state-of-the-art models, but present a simple and effective model — **the bag-of-words (BOW) model**. The bag-of-words model is a feature-based model of representing documents. In machine learning, a **feature** is a property of a sample. One can define a feature not only as some concrete attribute, such as a name and a gender, but also as a quantity that is countable for machine learning systems, such as a real number.

In the bag-of-words model, a feature corresponds to the occurrence times of a word. Let V be a vocabulary. A document can be represented as a $|V|$-dimensional feature vector. Each dimension describes a word count feature. It counts the occurrence of the i-th word of V in the document. More formally, let $\mathbf{x}$ be a feature vector. The i-th entry of $\mathbf{x}$ is defined as:

$$x_i \quad = \quad \text{count}(V_i) \tag{1.43}$$

where $\text{count}(\cdot)$ is a counting function. Consider, for example, the following lines of text[4].

As I went to Bonner

I met a pig

Without a wig,

Upon my word and honor.

[4]The text is from *Mother Goose rhymes*.

We then have a vocabulary extracted from the corpus[5], like this

$$\begin{aligned} V \quad = \quad & \{\text{“a”},\ \text{“and”},\ \text{“as”},\ \text{“Bonner”},\ \text{“honor”}, \\ & \text{“I”},\ \text{“met”},\ \text{“word”},\ \text{“my”},\ \text{“pig”}, \\ & \text{“to”},\ \text{“upon”},\ \text{“went”},\ \text{“wig”},\ \text{“without”}, \\ & \text{“,”},\ \text{“.”}\} \end{aligned}$$

Each line of the text can be seen as a document and represented as a feature vector. See below for the feature vectors generated by using the bag-of-words model.

	a	and	as	Bonner	honor	I	met	word	my	pig	to	upon	went	wig	without	,	.
As I went to Bonner	0	0	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0
I met a pig	1	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0
Without a wig,	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
Upon my word and honor.	0	1	0	0	1	0	0	1	1	0	0	1	0	0	0	0	1

The bag-of-words model defines a vector space[6]. In this space, the similarity of two vectors is measured in some way like dot-product. It helps when one wants to establish the relationship between documents — two documents with more overlapping words are more similar. This intuitive picture has guided many people when building classification systems.

The beauty of the bag-of-words model comes from its simple form in that any word is independent of other words. The independent assumption makes it possible to encode a document with almost infinite word relations as a countable, feasibly sized vector. On the other hand, representing a document as a feature vector of word counts is not the only option. As an improvement, one might take more context into account, and/or design more powerful features. This leads to an active line of research on text representation methods, ranging from heuristics-based methods to representation learning methods. We will see a few of them in the subsequent chapters.

1.2.3 Linear Classifiers

Linear classifier is one of the simplest classification models. Suppose we take a feature vector $\mathbf{x} = \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}$ as input, and take a weight vector $\mathbf{w} = \begin{bmatrix} w_1 & \dots & w_n \end{bmatrix}$ and a scalar b as parameters. A linear function has a form like this

$$\begin{aligned} s(\mathbf{x}, \mathbf{w}, b) \quad &= \quad \mathbf{w} \cdot \mathbf{x} + b \\ &= \quad w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b \end{aligned} \tag{1.44}$$

where b is a bias term. The dot product $\mathbf{x} \cdot \mathbf{w}$ is a linear combination of $\begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}$ where each x_i is weighted by w_i. For a more condensed formulation, we can define a new input vector

[5] We removed the case of the word at the beginning of each line.

[6] A vector space should be closed under vector addition and scalar multiplication.

$\mathbf{x}' = \begin{bmatrix} x_1 & \dots & x_n & 1 \end{bmatrix}$ and a new weight vector $\mathbf{w}' = \begin{bmatrix} w_1 & \dots & w_n & b \end{bmatrix}$. We then rewrite Eq. (1.44) as:

$$\begin{aligned} s(\mathbf{x}', \mathbf{w}') &= s(\mathbf{x}, \mathbf{w}, b) \\ &= \mathbf{w}' \cdot \mathbf{x}' \end{aligned} \tag{1.45}$$

In the following, we drop the bias term b for simplicity and use $s(\mathbf{x}, \mathbf{w})$ to denote a linear function. For classification, a linear function is used to describe class membership. Each class is assigned a score by the function equipped with a unique weight vector. Consider again the binary classification as an example. Let c_a and c_b be two classes. We can define two weight vectors $\mathbf{w}_a$ and $\mathbf{w}_b$ so that the function can discriminate between c_a and c_b.

For prediction, we can infer a class based on $s(\mathbf{x}, \mathbf{w})$. To achieve this, activation functions $\psi(\cdot)$ are in general used for mapping the value of $s(\mathbf{x}, \mathbf{w})$ to a class. For example, for binary classification, we can define an activation function like this,

$$\psi(x) = \begin{cases} c_a & x > 0 \\ c_b & \text{otherwise} \end{cases} \tag{1.46}$$

Then, we make a prediction by

$$\psi(s(\mathbf{x}, \mathbf{w}_a) - s(\mathbf{x}, \mathbf{w}_b)) = \begin{cases} c_a & s(\mathbf{x}, \mathbf{w}_a) - s(\mathbf{x}, \mathbf{w}_b) > 0 \\ c_b & \text{otherwise} \end{cases} \tag{1.47}$$

As $s(\mathbf{x}, \mathbf{w}_a) - s(\mathbf{x}, \mathbf{w}_b) = s(\mathbf{x}, \mathbf{w}_a - \mathbf{w}_b)$, the final prediction function is $\psi(s(\mathbf{x}, \mathbf{w}_a - \mathbf{w}_b))$. We call it a **discriminant function**. Note that $s(\mathbf{x}, \mathbf{w}_a - \mathbf{w}_b)$ is linear. So this is a **linear discriminant function**.

A discriminant function assigns an input vector $\mathbf{x}$ directly to a class but it does not describe how likely a class would appear given $\mathbf{x}$. There are other activation functions for generating a desirable output. For example, we may want a probability-like output (see Eq. (1.42)), and thus define $\psi(\cdot)$ as a normalized function[7]. Then, the classification probabilities are given by the equation

$$\begin{aligned} \begin{bmatrix} \Pr(c_a|\mathbf{x}) & \Pr(c_b|\mathbf{x}) \end{bmatrix} &= \psi\Big(\begin{bmatrix} s(\mathbf{x}, \mathbf{w}_a) & s(\mathbf{x}, \mathbf{w}_b) \end{bmatrix} \Big) \\ &= \begin{bmatrix} \frac{s(\mathbf{x}, \mathbf{w}_a)}{s(\mathbf{x}, \mathbf{w}_a) + s(\mathbf{x}, \mathbf{w}_b)} & \frac{s(\mathbf{x}, \mathbf{w}_b)}{s(\mathbf{x}, \mathbf{w}_a) + s(\mathbf{x}, \mathbf{w}_b)} \end{bmatrix} \end{aligned} \tag{1.48}$$

where $\psi(\cdot)$ is a **vector function**[8]. It normalizes the entries of the input vector by the sum of these entries. The decision rule is simple: we predict c_a if $\Pr(c_a|\mathbf{x}) > \Pr(c_b|\mathbf{x})$, and c_b otherwise. Since $\frac{s(\mathbf{x}, \mathbf{w}_a)}{s(\mathbf{x}, \mathbf{w}_a) + s(\mathbf{x}, \mathbf{w}_b)}$ and $\frac{s(\mathbf{x}, \mathbf{w}_b)}{s(\mathbf{x}, \mathbf{w}_a) + s(\mathbf{x}, \mathbf{w}_b)}$ share the same denominator, the prediction can also be made by comparing the numerators, i.e., we are doing the same thing as that in Eq. (1.47). In subsequent chapters, we will show that the trick of transforming

[7] A normalized function is a function whose integral over its domain is equal to 1.

[8] A vector function reads a vector and returns a new vector.

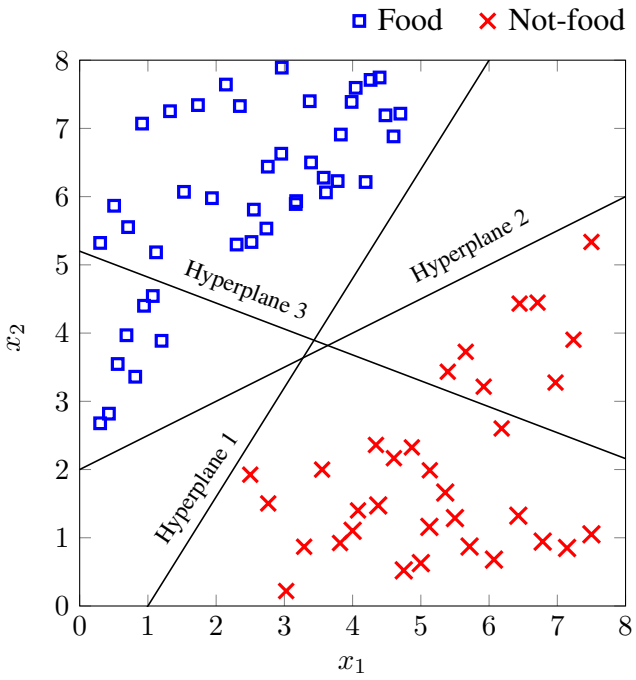


Figure 1.1: Data points and separating hyperplanes in two dimensions. There are two classes of data points (food and non-food). Both hyperplanes 1 and 2 separate the space into two sub-spaces where the two classes of data points are isolated. In this sense, the problem here is linearly separable. On the other hand, hyperplane 3 fails to separate the two classes, that is, the data points in the same class are classified into two different classes.

comparing probabilities to comparing real-valued scores is frequently used for addressing NLP problems.

For ease of understanding, one can see a linear classification model as a hyperplane (or **decision surface**, or **decision boundary**) that separates data points into different groups. Figure 1.1 shows example hyperplanes in a 2D space where each $\mathbf{x}$ is a data point. Hyperplanes 1 and 2 successfully separate the data points into the correct classes, while hyperplane 3 fails to do so. In this sense, the task of classification is to find hyperplanes that make the correct separation of data points.

«««< HEAD It is worthy of note that linearity is the basis of many classifiers although most of them do not in the same form as Eqs. (1.44 - 1.45). A linear model can be nearly a perfect solution if the problem is linearly separable[9]. Even for non-linearly separable problems, linear models can be married to other models with a non-linear separation ability. A good example is that one can achieve non-linear classification by marrying a linear model with a non-linear activation function. ======= It is worthy of note that linearity is the basis of many classifiers although most of them do not exist in the same form as Eqs. (1.44 - 1.45). A linear model

[9]Linear separability checks if there is a way that we put a hyperplane to reside a group of data points from the remaining data points. For example, the problem shown in Figure 1.1 is linearly separable.

can be nearly a perfect solution if the problem is linearly separable[10]. Even for non-linearly separable problems, linear models can be married to other models with a non-linear separation ability. A good example is that one can achieve non-linear classification by marrying a linear model with a non-linear activation function. »»»> origin/master

1.2.4 Generative vs Discriminative

There are two ways, though not restricted to linear models, to make use of linearity in classification — **generative models** and **discriminative models**. While most statistical classifiers are of the modeling variety, they choose the backbone design from either or both of these two types of models.

1. Generative Models

A goal of classification is to learn $\Pr(c|\mathbf{x})$. Generative models do not explicitly model this conditional probability. Instead, they model the joint probability $\Pr(\mathbf{x}, c)$, and use the Bayes' rule to compute $\Pr(c|\mathbf{x})$. This is given by the following equation.

$$\begin{aligned} \Pr(c|\mathbf{x}) &= \frac{\Pr(\mathbf{x}, c)}{\Pr(\mathbf{x})} \\ &= \frac{\Pr(c)\Pr(\mathbf{x}|c)}{\Pr(\mathbf{x})} \end{aligned} \tag{1.49}$$

where $\Pr(\mathbf{x}, c)$ is rewritten as $\Pr(\mathbf{x}|c)\Pr(c)$. For an optimal class $\hat{c}$, we choose a class c by maximizing $\Pr(c|\mathbf{x})$ (see Eq.(1.42)). Since the denominator $\mathbf{x}$ is a constant for any c, we just need to maximize the numerator. Then, we rewrite Eq.(1.42) in the form

$$\begin{aligned} \hat{c} &= \arg\max_{c \in C} \Pr(c|\mathbf{x}) \\ &= \arg\max_{c \in C} \frac{\Pr(c)\Pr(\mathbf{x}|c)}{\Pr(\mathbf{x})} \\ &= \arg\max_{c \in C} \Pr(c)\Pr(\mathbf{x}|c) \end{aligned} \tag{1.50}$$

where $\Pr(c)$ is the prior of c, and $\Pr(\mathbf{x}|c)$ is the conditional probability of the input document vector $\mathbf{x}$ given c. Computing $\Pr(c)$ is easy. For example, the **maximum likelihood estimation** (**MLE**) defines $\Pr(c)$ as a relative frequency:

$$\Pr(c) = \frac{\text{count}(c)}{\sum_{c' \in C} \text{count}(c')} \tag{1.51}$$

where $\text{count}(c)$ counts the occurrences of c in a corpus.

But computing $\Pr(\mathbf{x}|c)$ is non-trivial as data sparseness prevents us from accurately estimating the probability of a high-dimensional document vector. Recall that the bag-of-words model defines x_i as the word frequency of V_i in the document. We assume here that the feature

[10] Linear separability checks if there is a way that we put a hyperplane to separate a group of data points from the remaining data points. For example, the problem shown in Figure 1.1 is linearly separable.

vector $\mathbf{x} = \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}$ is generated by a multinomial $(p_1(c), p_2(c), ..., p_n(c))$, where $p_i(c)$ is the probability of V_i occurring given c. Based on MLE, we can estimate $p_i(c)$ by the relative frequency estimation:

$$p_i(c) = \frac{\text{count}(V_i, c)}{\sum_{1 \le i' \le n} \text{count}(V_{i'}, c)} \tag{1.52}$$

where $\text{count}(V_i, c)$ is the number of occurrences of V_i in all documents labeled as c. Then, $\Pr(\mathbf{x}|c)$ is given by

$$\Pr(\mathbf{x}|c) = \frac{(\sum_{i=1}^n x_i)!}{x_1! \cdot x_2! \cdot \dots \cdot x_n!} \cdot \prod_{i=1}^n p_i(c)^{x_i} \tag{1.53}$$

Substituting Eq. (1.53) into Eq. (1.50), we have

$$\hat{c} = \arg\max_{c \in C} \Pr(c) \cdot \frac{(\sum_{i=1}^n x_i)!}{x_1! \cdot x_2! \cdot \dots \cdot x_n!} \cdot \prod_{i=1}^n p_i(c)^{x_i} \tag{1.54}$$

Note that $\frac{(\sum_{i=1}^n x_i)!}{x_1! \cdot x_2! \cdot \dots \cdot x_n!}$ is independent of any c. We drop it in $\arg\max$, and rewrite the right-hand side of the equation in log scale:

$$\hat{c} = \arg\max_{c \in C} \log(\Pr(c)) + \sum_{i=1}^n x_i \cdot \log(p_i(c)) \tag{1.55}$$

Obviously, this is a linear model. It defines the feature vector and weight vector as below

$$\mathbf{x} = \begin{bmatrix} 1 & x_1 & \dots & x_n \end{bmatrix} \tag{1.56}$$

$$\mathbf{w} = \begin{bmatrix} \log(\Pr(c)) & \log(p_1(c)) & \dots & \log(p_n(c)) \end{bmatrix} \tag{1.57}$$

Such a form is sometimes called a **log-linear** model, as the linearity comes from transforming the original problem via a logistic function.

In general, Eq (1.55) is called the **multinomial naive Bayes** approach. There are, of course, the naive Bayes variants for other types of feature vectors. For example, for binary value feature vectors, one can assume a Bernoulli distribution on each entry of a vector and design a **Bernoulli naive Bayes** classifier; for vectors with continuous features, one can assume a Gaussian distribution over continuous data and design a **Gaussian naive Bayes** classifier.

2. Discriminative Models

The model defined by $\Pr(c|\mathbf{x}) = \frac{\Pr(\mathbf{x},c)}{\Pr(\mathbf{x})} = \frac{\Pr(c)\Pr(\mathbf{x}|c)}{\Pr(\mathbf{x})}$ (see Eq. (1.49)) is called generative because it assumes some way of generating data $\mathbf{x}$ given label c. The idea is to use $\Pr(\mathbf{x}, c)$ as a pivot to compute $\Pr(c|\mathbf{x})$. As an alternative possibility for modeling $\Pr(c|\mathbf{x})$, **discriminative models** do not try to model the distribution of $\mathbf{x}$ but estimate $\Pr(c|\mathbf{x})$ directly. An example is **logistic regression**. For binary text classification, a naive Bayes classifier predicts label c_a for

a given document $\mathbf{x}$ only if the following function is positive:

$$f_a(\mathbf{x}) = \log \frac{\Pr(c_a|\mathbf{x})}{\Pr(c_b|\mathbf{x})} \tag{1.58}$$

One can assume that this quantity follows a linear model:

$$\log \frac{\Pr(c_a|\mathbf{x})}{\Pr(c_b|\mathbf{x})} = \mathbf{w} \cdot \mathbf{x} \tag{1.59}$$

Since $\Pr(c_b|\mathbf{x}) = 1 - \Pr(c_a|\mathbf{x})$, we have

$$\Pr(c_a|\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w} \cdot \mathbf{x})} \tag{1.60}$$

This is a logistic function, or more precisely a Sigmoid function. With such a model, we predict c_a only if $\mathbf{w} \cdot \mathbf{x}$ is positive. Eq. (1.60) is also called the logistic regression classifier. It is simply a discriminative analog of the naive Bayes classifier.

An advantage of discriminative models is that they offer flexibility in viewing classification (or other machine learning problems) from different angles. While generative models try to estimate the data distribution of $\mathbf{x}$, discriminative models try to find a good boundary between classes. Discriminative models, therefore, care more about which class is prioritized over another given $\mathbf{x}$, or in possibility language which class is more likely to appear, instead of making assumptions on individual data points. This makes it possible to learn a classifier by minimizing the number of some errors, not necessarily guaranteeing the maximum likelihood on those data points. This approach is generally called **error-driven learning**.

There are many ways to define errors. Like generative models, one can learn a discriminative model by fitting parameters $\mathbf{w}$ to maximize the likelihood on the training data. Let $\{(\mathbf{x}^{(1)}, c^{(1)}), ..., (\mathbf{x}^{(K)}, c^{(K)})\}$ be a set of labeled documents, where $\mathbf{x}^{(k)}$ is a document and $c^{(k)}$ is the corresponding class. The best parameter vector $\hat{\mathbf{w}}$ is given by the equation

$$\hat{\mathbf{w}} = \arg\max_{\mathbf{w}} \sum_{k=1}^{K} \log \Pr(c^{(k)}|\mathbf{x}^{(k)}) \tag{1.61}$$

Taking Eq. (1.60), the process can be seen as maximizing the likelihood

$$\begin{aligned}
\hat{\mathbf{w}} &= \arg\max_{\mathbf{w}} \sum_{k=1}^{K} \log \delta(c_a, c^{(k)}) \Pr(c_a|\mathbf{x}) + \log \delta(c_b, c^{(k)}) \Pr(c_b|\mathbf{x}) \\
&= \arg\max_{\mathbf{w}} \sum_{k=1}^{K} \log \delta(c_a, c^{(k)}) \frac{1}{1+\exp(-\mathbf{w} \cdot \mathbf{x}^{(k)})} + \\
&\qquad \log \delta(c_b, c^{(k)}) \Big(1 - \frac{1}{1+\exp(-\mathbf{w} \cdot \mathbf{x}^{(k)})}\Big)
\end{aligned} \tag{1.62}$$

where $\delta(\cdot, \cdot)$ is an **indicator function** that returns 1 if the two arguments are equal, and 0

otherwise.

Alternatively, we can train the model by minimizing 0-1 errors, that is, we count an error when the output is not the correct label. This process can be formulated as

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \sum_{k=1}^{K} \delta(\hat{c}^{(k)}, c^{(k)}) \tag{1.63}$$

where $\hat{c}^{(i)}$ is the prediction made by the classifier. The 0-1 error can be further extended by taking the posterior into account:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} \sum_{k=1}^{K} \Pr(\hat{c}^{(k)}|\mathbf{x}^{(k)}) \cdot \delta(\hat{c}^{(k)}, c^{(k)}) \tag{1.64}$$

The training objective plays an important role in discriminative models. This topic, however, is so broad and beyond the scope of this section. We will present some in Section 1.3.4. As another bonus, discriminative models do not restrict features to forming a probabilistic generative story. In a broader sense, $\mathbf{x}$ could be any feature vector that is designed by researchers and engineers. One does not even need to guarantee the probabilistic meaning for these features. For example, let $g(\mathbf{x})$ be the output of another system, say some scores. Following Eq. (1.60), a new binary classification model can be designed in a logistic regression manner:

$$\Pr(c_a|\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w} \cdot g(\mathbf{x}))} \tag{1.65}$$

For learning $g(\mathbf{x})$, one can either pre-train it on some additional data, or train it jointly with $\mathbf{w}$ (i.e., the parameters of the upper-level model $\Pr(c_a|\mathbf{x})$).

1.2.5 OOV Words and Smoothing

The **out-of-vocabulary** (**OOV**) problem occurs when some of the words of a document are not found in the vocabulary. OOV words are common in NLP because new words are always there no matter how much text we have seen. Figure 1.2 gives two curves to illustrate this problem. As shown in Figure 1.2 (a), new words continuously appear when more data is available. When we fix the data that is used for testing the coverage of a vocabulary, OOV words remain even if we have an extremely large vocabulary (Figure 1.2 (b)).

For practical systems, OOV words are common because the vocabulary is often restricted to a "small" number of entries. A standard method is to keep the top-n most frequent words and discard the rest. In this case, OOV words are treated as *unknown* words. For example, a new symbol <unk> is introduced into the vocabulary so that all OOV words are denoted as <unk>. A more aggressive idea is to build an **open-vocabulary** system that accepts every possible word, but it would require more sophisticated algorithms and probably a task-specific design of data structures. The <unk> trick is still the de facto standard for the development of current NLP systems.

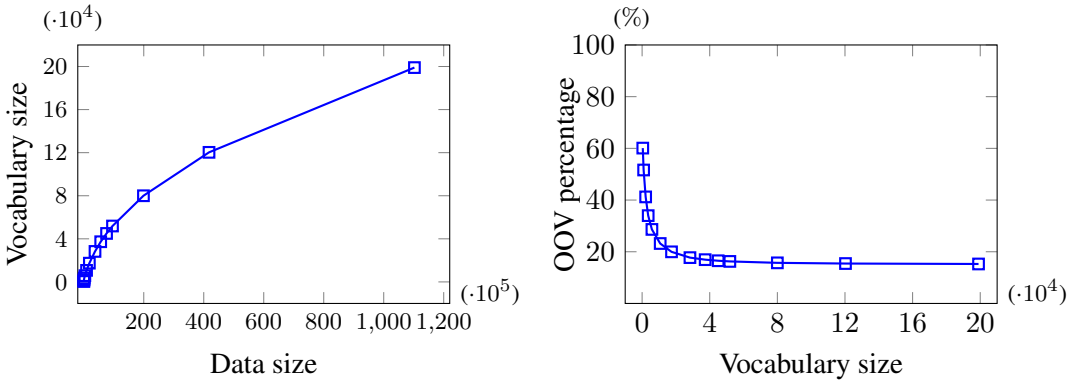


Figure 1.2: Data size (in number of words), vocabulary size and percentage of OOV. The statistics are collected on the English data provided in the WMT21 Zh-En translation task. The more data we use, the larger vocabulary we have. The increase in vocabulary size continues even if we build the vocabulary on data of more than 100 million words. However, the slope of the curve tends to be smaller as more data is involved. Interestingly, the OOV percentage converges to a certain level as the vocabulary size increases, indicating that new words always occur no matter how many words we have observed.

For words that are already in the vocabulary, there are also **unseen words** that are absent in the parameter estimation phase but appear in a new document. A naive implementation of the model described in the previous section would be tough when dealing with unseen words. For example, Eq. (1.52) will simply give a zero probability if the word V_i does not occur in any training example labeled with c. In consequence, for a new example containing V_i, Eq. (1.53) will assign it a zero probability. This result is obviously unreasonable. However, we should not simply attribute it to the model design itself. Instead, a primary reason for this is the insufficient data used for parameter estimation. This is also explained by **Zipf's Law**: a small number of words occur quite often, while a large number of words occur rarely.

However, we cannot suppose that we always have access to some data where every word occurs sufficiently. Alternatively, one could adopt smoothing techniques to redistribute the probability over the vocabulary. Consider Eq. (1.52) as an example. We can add a small number to each V_i, and rewrite the equation as:

$$p_i(c) \quad = \quad \frac{\text{count}(V_i, c) + \alpha}{\sum_{1 \leq i' \leq n} \text{count}(V_{i'}, c) + n \cdot \alpha} \tag{1.66}$$

where α is the default count that we assign to each word. In this way, $p_i(c)$ gives a non-zero probability even if $\text{count}(V_i, c) = 0$. Eq. (1.66) is doing something like subtracting word counts from high-frequency words and reassigning the subtracted counts to low-frequency words. This method is called **additive smoothing** or **add-α smoothing**. It is one of the simplest smoothing methods. For other methods, we refer the reader to language modeling papers where smoothing is in heavy use [Chen and Goodman, 1999].

1.3 General Problems

Building a simple text classifier is a good start but not enough for solving complicated, diverse real-world problems. For a more general picture of how modern machine learning systems work, we now discuss some problems that are important when designing such systems.

1.3.1 Supervised and Unsupervised Models

Supervised learning deals with labeled samples, by which we mean that an input x is associated with an output y. Given a set of input-output pairs $\{(x^{(1)}, y^{(1)}), ..., (x^{(K)}, y^{(K)})\}$, the task here is to learn a function $f(\cdot)$ that maps each $x^{(k)}$ to $y^{(k)}$:

$$y^{(k)} = f(x^{(k)}) \tag{1.67}$$

This process is called supervised because learning $f(\cdot)$ is guided by the manually annotated answer $y^{(k)}$ for $x^{(k)}$. In general, $y^{(k)}$ is called the ground-truth or gold-standard label of $x^{(k)}$. After that, when a new input x_{new} comes, we use the learned function $f(\cdot)$ to predict the output. We will say that the supervised learning succeeds if the prediction $f(x_{\text{new}})$ is the same as the ground-truth y_{new}.

The vast majority of NLP can be framed as supervised learning problems. Assigning a class to a document is no doubt one of the simplest cases. Other NLP tasks include but are not limited to producing a sequence of labels, a piece of text, a syntax tree, and a graph.

In contrast to supervised learning, **unsupervised learning** deals with unlabeled samples, in other words, for each sample, we have an input x in the absence of the correct output y. In this case, we need an algorithm that learns from the unlabeled data $\{x^{(k)}\}$ the mapping function from $x^{(k)}$ to some output. Since there is no human intervention on the output of the function, algorithms of this kind need to discover patterns in $\{x^{(k)}\}$ and optimize the way we represent $\{x^{(k)}\}$ and function outputs by some criteria. These criteria are typically inspired by human prior knowledge so that the resulting function could output something that we expect.

A common example in unsupervised learning is **word clustering**. It groups a set of words by assigning similar words into the same cluster[11]. Based on such a criterion, we can bias $f(\cdot)$ towards outputting the same cluster for similar words. A more difficult case is **unsupervised bilingual dictionary induction**. It learns a word-level mapping between two languages without the need of parallel data. The problem is usually addressed, in part, by making use of the isomorphism of word representation spaces of different languages.

A halfway between supervised learning and unsupervised learning is **semi-supervised learning**. It deals with the case in which some of the input data is labeled and the rest is unlabeled. Thus, it can receive benefits from both supervised and unsupervised learning. For example, in machine translation, we may have a certain amount of parallel data (i.e., labeled data) and orders of magnitude larger monolingual data (i.e., unlabeled data). Often, a base system is learned on the parallel data in a supervised learning fashion. On top of it, improvements can be made from training components of the base system on large-scale

[11] It might be difficult and ambiguous to determine if two words are similar or not. We leave this issue to Chapter 3.

monolingual data. This is implemented by either combining a translation model learned on the bilingual data and a language model learned on the target-language monolingual data, or pre-training parts of the translation model on monolingual data and fine-tuning the entire model on the bilingual data.

Broadly speaking, all learning algorithms need supervision. This sounds weird because unsupervised learning seems to not be signaled by any ground-truth data. However, from a general learning perspective, we should not restrict ourselves to labeled data for receiving supervision signals. Even for an unsupervised learning problem, we still need to supervise the learning process by prior knowledge and hidden patterns in the input data $\{x^{(k)}\}$. In this sense, unsupervised learning is not "learning without supervision".

Taking "supervision" as a concept in a broader sense, more paradigms can be seen as instances of machine learning, though not necessarily belonging to either supervised learning or unsupervised learning. An example is **reinforcement learning**. It models how a system makes a sequence of decisions. This is achieved by operating an agent in an environment. The agent receives a feedback (or a reward) from the environment when making a decision (or taking an action). The goal of reinforcement learning is to learn a decision model that maximizes the reward along the steps the agent takes. The real reward here is available only when the agent reaches some state, such as the end of a game. As such, reinforcement learning can describe problems where the reward is over a longer period (call it **distant reward**). This differentiates reinforcement learning sharply from standard supervised learning, traditionally concerned with instant supervision signals that are encoded in labeled data in advance. This characteristic fits many NLP problems. For example, a text generation system generally generates a sequence of words from left to right, but it is hard to determine if a word is properly predicted until the whole text has been generated.

Another example is **self-supervised learning**. It addresses unsupervised learning problems in a supervised learning manner. A general idea is to frame the unsupervised learning task as a pretext task that can be used in solving the original problem. In the pretext task, ground truth can be generated from input data. The learning therefore receives supervision from self-made signals instead of manual labels.

Self-supervised learning has indeed been quite successful in NLP. Perhaps **pre-training** is one of those which have made the most incredible progress. In pre-training, one can train some model (such as a language model) via self-supervised learning, and then apply parts of the model to some downstream system (such as a sentiment analysis system). It offers two advantages. First, the self-supervised, pre-trained models can be scaled to a huge amount of data because they require no labeled data. Second, pre-training is general itself and can be applied to a wide range of downstream tasks. In Chapter 7, we will see a few examples of applying self-supervised learning to NLP models.

1.3.2 Inductive Bias

We informally describe (supervised) machine learning problems as an **inductive reasoning** (or **inductive inference**) process: we use specific observations (such as labeled documents) to make a generalized model (such as a classifier). For example, we may observe that word

cooking frequently occurs in some documents talking about food. We would say that, based on inductive reasoning, *cooking* is an important indicator for all food documents. This "specific-to-general" method is also called **induction** sometimes. Once we have an induced model, we can apply it to describe new observations, as a **deduction** process.

Induction is the most widely-used principle in designing learners of modern machine learning systems[12]. Imagine that there is a hypothesis space (or model space, or learnable function space) consisting of all possible models that we could make. Learning a model is thus the same as selecting a model in the hypothesis space by inducing from the given training samples. However, we cannot simply assume an oracle model that works well on all unseen samples. A reason for this is that searching for the "best" model in a huge hypothesis space is computationally infeasible. There would be an infinite number of dimensions along which we can design models if the hypothesis space is unconstrained. This problem is essentially some sort of **the curse of dimensionality**[13]. Another reason is that many models in the hypothesis space can fit training samples well, but only some of them can fit unseen samples. There is a risk that we select a "weak" model for test data although it is "strong" for training data. This is relevant to **overfitting**, a concept that we will discuss later.

A natural solution is to define priors on the hypothesis space in a way that allows some models to be more preferable than others. A simple example is that we restrict classifiers to linear models (see Section 1.2.3). It is doing something like we impose a prior that excludes all non-linear models from the hypothesis space.

Such a prior is generally called an **inductive bias**. In a nutshell, an inductive bias is a set of assumptions on the problem[14]. For example, one can design models in certain mathematical forms (i.e., model bias); one can choose specific algorithms for learning a model (i.e, algorithm bias); one can assume the way of generating samples (i.e, sample bias), and so on[15].

Inductive biases try to tell in what way we should describe a problem. Better results are generally favorable when inductive biases meet what really happens. This explains why solutions to some problems prefer certain model architectures (or model biases). Of course, more and stronger inductive biases could make it easier to solve a problem. However, inductive biases are not always helpful, especially when they are not close to the reality.

Let us consider a dice rolling game. Suppose you have a 6-sided dice. Before rolling the dice, you guess a side (say a number from 1 to 6). You will win if the dice lands on the same side you guess. You are a gambler and try to win as many times as possible. In your experience, a random guess is the best choice in this game because all sides should have an equal chance of appearing (i.e. a chance of $1/6$). This is true when you play fair dice. However, one day,

[12] Not all machine learning methods should follow an induction process for learning a model. There are other options for different types of problems, including deductive reasoning, abductive reasoning, analogical reasoning (or transduction), and so on [Hurley, 2011].

[13] The curse of dimensionality refers to the problems that generally appear as the dimensionality of the hypothesis space increases. For example, data sparseness is a common problem that arises when processing high-dimensional data, and is thus a kind of the curse of dimensionality.

[14] A more formal definition can be found in machine learning textbooks [Mitchell, 1997]

[15] As an aside it is worth noting that the term *bias* is used in many different ways, and there are other meanings for *bias* in certain contexts. We will make it clear when a different meaning is used.

we played weighted dice, and it was not easy to win as before. You found that the appearance of different sides did not follow a uniform distribution. Then, you assumed a multinomial distribution (because you had the experience of developing naive Bayes classifiers). Before the game started, you rolled the dice 100 times. You chose the most frequent side for new games, and you won more. In this example, you made an initial assumption that all six of the sides are equally likely to occur. This is a very strong inductive bias because your model has 0 degrees of freedom. It seems to be obvious but does not work for weighted dice. The second inductive bias, though seems more complicated, is actually a weaker assumption, because a multinomial distribution defines a larger family of models and gives room to finding appropriate models.

In general, all machine learning models need some sort of inductive bias. Many of them are implicit assumptions. Sometimes, we are even not aware that we are making these assumptions because they are so "obvious" and "logical". On the other hand, if the assumption is wrong then it is harmful to problem-solving. So we still need some experience to avoid easily neglected mistakes.

1.3.3 Non-linearity

Non-linearity is the nature of most real-world problems, whereas it is not easy to use a linear model to solve a non-linear problem. See Figure 1.3 for examples of varying degrees of classification difficulty. In Figure 1.3 (a), the two classes can be separated by a hyperplane. In this case, the problem is **linearly separable** because the decision boundary can be represented as a linear function. In contrast, in Figure 1.3 (b), we cannot draw hyperplanes to perfectly separate the two classes. Instead, we need some non-linearity for better separation, such as hyperspheres. A more difficult case is shown in Figure 1.3 (c) where the decision boundary is highly complex.

Although the theory of non-linear systems still has not been fully studied, there are several methods that help us introduce non-linearity into machine learning systems.

- **Feature mapping** and **kernel methods**. Recall that a linear classifier can be formulated as a function $f(\mathbf{w}\cdot\mathbf{x})$, where $\mathbf{w}$ is the weight vector, $\mathbf{x}$ is the feature vector, and $f(\cdot)$ is the function that returns one class (say c_a) if its argument >0 and the other class (say c_b) otherwise. The idea of feature mapping is that we map the feature vector $\mathbf{x}$ into a higher-dimensional space so that the problem is linearly separable in the new space. For example, let $\phi(\cdot):\mathbb{R}^2\rightarrow\mathbb{R}^4$ be a mapping function and $\mathbf{x}=\begin{bmatrix}x_1 & x_2\end{bmatrix}$ be a 2-dimensional vector. We assume that

$$\phi(\begin{bmatrix}x_1 & x_2\end{bmatrix}) = \begin{bmatrix}x_1^2+x_2^2 & x_1 & x_2 & 1\end{bmatrix} \tag{1.68}$$

  By choosing $\mathbf{w}=\begin{bmatrix}1 & -8 & -8 & 28\end{bmatrix}$, we get a new classifier:

$$\begin{aligned} f(\mathbf{w}\cdot\phi((x))) &= f(\begin{bmatrix}1 & -8 & -8 & 28\end{bmatrix}\cdot\begin{bmatrix}x_1^2+x_2^2 & x_1 & x_2 & 1\end{bmatrix}) \\ &= f\big((x_1-4)^2+(x_2-4)^2-4\big) \end{aligned} \tag{1.69}$$

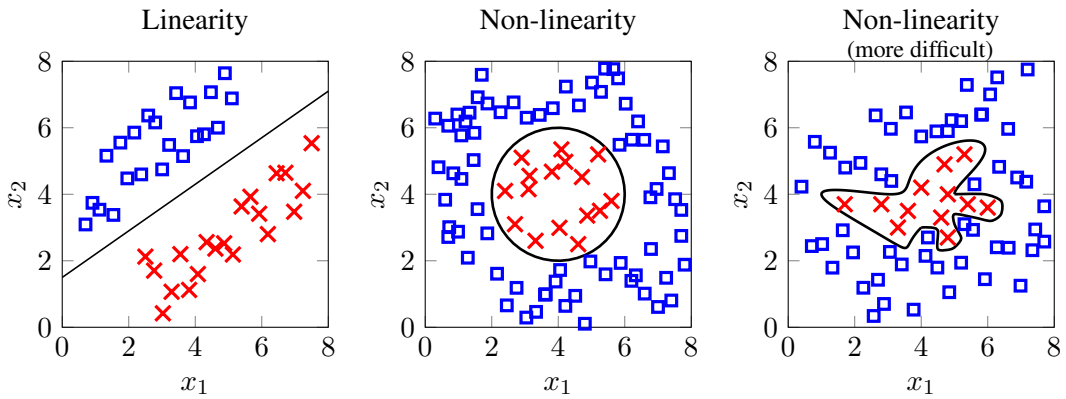


Figure 1.3: Linearity and non-linearity in binary classification. The first problem (left) is linearly separable because there exists (at least) a hyperplane that perfectly separates the data points in the two classes. The property of linear separability does not hold in the second problem (middle). Rather, we need a circle-like decision boundary. The decision boundary would be more complex if there are areas where the two classes of the data points are mixed and more or less indistinguishable (right).

It defines a decision boundary (i.e., a hypersphere $(x_1 - 4)^2 + (x_2 - 4)^2 = 4$) that perfectly classifies the samples in Figure 1.3 (b). In other words, we use a linear model on the mapped feature space to create a non-linear model. However, computing the mapping function might be inefficient. This is typically addressed by using kernel methods. In kernel methods, the calculation of vector dot-product in the new space is performed efficiently by using a kernel function in the old space. This method is called **the kernel trick**. It has been successfully adopted in classification and other machine learning models, such as **support vector machines** [Cortes and Vapnik, 1995].

- **Non-linear activation functions**. Another way to add non-linearity is to use activation functions. A common method is to stack a non-linear activation function on top of a linear model. For example, the function $f(\cdot)$ used in the above example is itself a non-linear function. There are many kinds of non-linear activation functions. We can choose from them, depending on what form of the output we want. For more sophisticated models, more activation functions can be inserted into the intermediate computing steps to develop a more powerful and expressive model. For example, a **deep neural network** is a stack of sub-models (call them layers) where each sub-model may involve one or more activation functions.
- **Non-parametric methods**. Non-parametric is a term that is originated from statistics. In non-parametric statistics, statistical inferences are made without any assumption on underlying distributions of data. In machine learning, non-parametric methods follow the same idea. They do not assume any mapping function from input to output as Eq. (1.69). This differentiates them from **parametric methods** that explicitly learn a mathematical form of variables (or parameters) to describe the problem. An example of

non-parametric methods is k**-nearest neighbors**. It makes a prediction for a new sample based on the k nearest neighboring samples in training data. Note that non-parametric does not mean parameter-free. Rather, it means that parameters can change. On another hand, non-parametric methods do not ensure a fixed model. They grow in model size as more training samples are available. As a reward, they can handle highly non-linear problems when training samples are sufficient.

Still, non-linear methods do not work alone. Linearity is surely an important component for most practical machine learning systems. This has two flavors. First, more non-linearity is not always better. We do not need to complicate the modeling if a linear model is enough for solving the problem. An example is that most state-of-the-art machine learning models are a combination of linear and non-linear sub-models. This is also an instance of **Occam's Razor** — *the simplest solution is almost always the best*. The second flavor is the linear approximation of non-linear behaviors. Linear models are a good alternative if the non-linearity of the problem is not obvious. In such cases, using linear functions to approximate precise solutions is probably more efficient for practical purposes.

1.3.4 Training and Loss Functions

Almost all machine learning algorithms involve a training step. Typically, it refers to the process of estimating the mapping function and the associated parameters from data. Here we follow a conventional definition of the training problem: given a model or mapping function, we improve some objective by evaluating the model through some training experience [Mitchell, 1997]. For example, training a naive Bayes text classifier requires maximizing a likelihood function (i.e., the objective) on a number of labeled documents (i.e., the training experience).

Often, the training problem can be framed as an **optimization** problem. As such, we optimize some **objective function** via some training algorithm. Although an ideal objective function is a performance measure on test samples, we cannot take it in optimization since the test samples and corresponding labels are assumed to be inaccessible in the training phase. Practical objective functions are instead defined as a surrogate for the measure on test data. On the other hand, these objective functions are not necessarily some sort of performance measure, but some metrics that are assumed to correlate with the performance on test data.

Let us consider a general case. Suppose $\mathbf{y}_\theta = f_\theta(\mathbf{x})$ is a model that reads a feature vector $\mathbf{x}$ and produces an n-dimensional vector $\mathbf{y}_\theta$. For example, in text classification, $\mathbf{x}$ is the bag-of-words representation of a document, and $\mathbf{y}_\theta$ is a distribution over a set of classes. θ is the parameters of the model. The subscript emphasizes that the model is determined by θ. We further suppose that $\mathbf{y}_{\text{gold}}$ is the gold-standard vector. Then, we define the objective function as a function that counts errors in $\mathbf{y}_\theta$ with respect to $\mathbf{y}_{\text{gold}}$, denoted as $L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}})$. It measures how bad it would be if we predict $\mathbf{y}_\theta$ instead of $\mathbf{y}_{\text{gold}}$. Given a model, the training problem can be described as finding the "best" parameters $\hat{\theta}$ so that $L(\mathbf{y}_{\hat{\theta}}, \mathbf{y}_{\text{gold}})$ is minimized:

$$\hat{\theta} = \arg\min_\theta L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) \tag{1.70}$$

This formulation can be easily extended to the case of K training samples:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{K} \sum_{k=1}^{K} L(\mathbf{y}_{\theta}^{(k)}, \mathbf{y}_{\text{gold}}^{(k)}) \tag{1.71}$$

Once we obtain $\hat{\theta}$, we can use $f_{\hat{\theta}}(\mathbf{x})$ as a fixed model for prediction.

$L(\mathbf{y}_{\theta}, \mathbf{y}_{\text{gold}})$ and $\frac{1}{K}\sum_{k=1}^{K} L(\mathbf{y}_{\theta}^{(k)}, \mathbf{y}_{\text{gold}}^{(k)})$ are usually called **loss functions** (or **cost functions**). A loss function can be defined in many ways, depending on what type of problem we address and what prior we want to impose upon training. Here, we first consider the case in which $\mathbf{y}_{\theta}$ is a probability distribution. It is quite common in NLP, e.g., $\mathbf{y}_{\theta}$ could be a distribution over a vocabulary, a distribution over a list of documents, a distribution over a set of syntactic labels. For such a type of model output, the most commonly-used loss functions are measures of divergence:

- **Divergence-based Loss**. Divergence-based loss functions compute the degree of difference between the two distributions $\mathbf{y}_{\theta}$ and $\mathbf{y}_{\text{gold}}$. For example, cross-entropy (see Section 1.1.2) is one of the most popular loss functions used in NLP. One can, of course, choose other divergence-based measures, such as the KL divergence and the Jensen-Shannon (JS) divergence, which can be found in most statistics textbooks. Note that MLE is also a special instance of the divergence-based objective. It is the same as the cross-entropy loss if $\mathbf{y}_{\text{gold}}$ is a one-hot vector where the entry of the correct label is 1 and other entries are all 0.

However, machine learning systems are not always restricted to distribution-like output. Rather, $\mathbf{y}_{\theta}$ could be a vector in $\mathbb{R}^n$. An example is the discriminant functions used in classification (see Section 1.2.3). They assign a score to each class, indicating how strong the model believes it is the answer. One way to define the loss functions on real-valued vectors is to transform them into distribution-like forms[16], and resort to the divergence-based loss. However, normalization is not always necessary, especially when we need a score out of the range of $[0, 1]$. It is more common to compute losses on the raw output of these models. Here are some examples.

- **Distance-based Loss**. It is natural to take loss as some sort of distance in geometry. A general example is the p-norm distance (see Section 1.1.1):

$$L(\mathbf{y}_{\theta}, \mathbf{y}_{\text{gold}}) = \Big(\sum_{i=1}^{n} |y_{\theta}(i) - y_{\text{gold}}(i)|^p \Big)^{1/p} \tag{1.72}$$

  For example, we would have a Euclidean distance-based loss function if $p = 2$. The distance-based loss intrinsically describes a **curve fitting** problem: we learn a curve $\mathbf{y}_{\theta} = f_{\theta}(\mathbf{x})$ to fit the points $\{(\mathbf{x}^{(k)}, \mathbf{y}_{\text{gold}}^{(k)})\}$. It is also called **regression** [17]. A simple

[16] For example, we can normalize the entries of a vector by the sum of these entries.

[17] When the model output is a vector with two or more dimensions, the problem is called **multivariate regression**.

example is quality estimation of machine translation[18]. It learns to predict translation quality (i.e., y_θ) for any pair of source and target sentences (i.e., $\mathbf{x}$). We would say that the prediction is accurate if the predicted score is close to that made by humans. By using Eq. (1.72), one can design many loss functions for regression models. For example, **mean square error** (**MSE**) is a popular regression loss function. It is the sum of squared Euclidean distances between the prediction and the gold standard:

$$L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) = \sum_{i=1}^{n} |y_\theta(i) - y_{\text{gold}}(i)|^2 \tag{1.73}$$

Another example is **mean absolute error** (**MAE**). It is precisely the form of Eq. (1.72) when $p = 1$.

- **0-1 Loss**. The 0-1 loss is widely used in classification problems. It chooses a value of either 1 (penalty) or 0 (no penalty), and penalizes the case in which the predicted label and the gold-standard label are not the same. Let $c_\theta = \arg\max_c y_\theta(c)$ be the label that is predicted by selecting the entry in $\mathbf{y}_\theta$ with the highest value. Likewise, let $c_{\text{gold}} = \arg\max_c y_{\text{gold}}(c)$ be the gold-standard label. The 0-1 loss is defined to be:

$$\begin{aligned} L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) &= L_{\text{0-1}}(c_\theta, c_{\text{gold}}) \\ &= \begin{cases} 1 & c_\theta \neq c_{\text{gold}} \\ 0 & c_\theta = c_{\text{gold}} \end{cases} \end{aligned} \tag{1.74}$$

- **Margin-based Loss**. A **margin** is the difference between the predicted scores of the correct label c_{gold} and an incorrect label c:

$$\text{magin}(c, c_{\text{gold}}) = y_\theta(c_{\text{gold}}) - y_\theta(c) \tag{1.75}$$

It indicates a distinction between c_{gold} and c. So, a natural idea is to ensure that the margin is sufficiently large, or at least exceeds a minimum. This is called **large-margin training**. Let $\triangle(c, c_{\text{gold}})$ be a predefined cost of replacing label c_{gold} with label c, satisfying $\triangle(c, c_{\text{gold}}) \geq 0$, and $\triangle(c, c_{\text{gold}}) = 0$ only if $c = c_{\text{gold}}$. Our goal is to enlarge $\text{margin}(c, c_{\text{gold}}) - \triangle(c, c_{\text{gold}})$, in other words, the larger this value is, the smaller the loss is. Then, the margin-based loss is given by:

$$\begin{aligned} L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) &= \max\Big(0, \max_c -\big(\text{magin}(c, c_{\text{gold}}) - \triangle(c, c_{\text{gold}})\big)\Big) \\ &= \max\Big(0, \max_c \big(y_\theta(c) - y_\theta(c_{\text{gold}}) + \triangle(c, c_{\text{gold}})\big)\Big) \\ &= \max_c \Big(0, y_\theta(c) - y_\theta(c_{\text{gold}}) + \triangle(c, c_{\text{gold}})\Big) \end{aligned} \tag{1.76}$$

[18] In machine translation, quality estimation comprises several different tasks (see https://www.statmt.org/wmt21/quality-estimation-task.html). Here we use the term to refer to the task that predicts an evaluation score directly.

Designing $\triangle(c, c_{\text{gold}})$ depends on the problem. A simple choice is $\triangle(c, c_{\text{gold}}) = 1$ for $c \neq c_{\text{gold}}$. This makes Eq. (1.76) a type of the **hinge loss**. Another variant of Eq. (1.76) is using a sum instead of a max:

$$L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) = \sum_c \max\Big(0, y_\theta(c) - y_\theta(c_{\text{gold}}) + \triangle(c, c_{\text{gold}})\Big) \tag{1.77}$$

- **Ranking-based Loss**. Ranking-based loss (or ranking loss) is used in several different areas, such as information retrieval, classification and metric learning. It deals with the problem where we want to order a set of scored items. Suppose the model output $\mathbf{y}_\theta$ corresponds to a set of n items $\{c_i\}$, each for an entry of $\mathbf{y}_\theta$. We define $\{\psi_\theta(c_i)\}$ as the order of $\{c_i\}$ by $\{y_\theta(i)\}$. For example, given $\{y_\theta(i)\} = \{0.3, -2, 1\}$, we have $\psi_\theta(c_1) = 2$, $\psi_\theta(c_2) = 3$ and $\psi_\theta(c_3) = 1$. Likewise, we can define $\{\psi_{\text{gold}}(c_i)\}$ as the gold-standard ranks. Note that $\{\psi_{\text{gold}}(c_i)\}$ can be induced in some way without the need of $\mathbf{y}_{\text{gold}}$ if the problem only requires orders, rather than scores. An idea of ranking-based loss is to model the ranking mistakes in $\{\psi_\theta(c_i)\}$ with respect to $\{\psi_{\text{gold}}(c_i)\}$. There are many ways to "count" the mistakes. A simple method is to penalize the case in which a pair of items are ordered incorrectly. As such, the ranking-based loss somewhat shares the same spirit of that used in binary classification — we categorize a pair of items as correct or incorrect. Let Ω be a set of ordered item pairs:

$$\Omega = \{(i, j) | \phi_{\text{gold}}(i) < \phi_{\text{gold}}(j)\} \tag{1.78}$$

The loss function is given by the equation:

$$L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) = \sum_{(i,j)\in\Omega} L_{\text{pair}}(y_\theta(i), y_\theta(j)) \tag{1.79}$$

where $L_{\text{pair}}(y_\theta(i), y_\theta(j))$ is a classification loss, such as the hinge loss used in [Collobert and Weston, 2008]:

$$L_{\text{pair}}(y_\theta(i), y_\theta(j)) = \max\big(0, y_\theta(i) - y_\theta(j) + 1\big) \tag{1.80}$$

This method is called the **pairwise method**. Also, one can define the ranking-based loss in a pointwise or listwise manner. These loss functions are extensively used in developing systems to rank objects.

- **Contrastive Loss**. Contrastive loss is typically used in **contrastive learning**. It assumes that, given a sample, there is a similar sample that is labeled as "positive", and there are a number of dissimilar samples that are labeled as "negative". A natural idea is to minimize the distance between similar samples and simultaneously maximize the distance between dissimilar samples. Return to the formulation here. For a model output $\mathbf{y}_\theta$, let $\mathbf{y}^+$ be the positive output and $\mathbf{Y}^- = \{\mathbf{y}^-\}$ be the set of negative outputs. Also, we use $\mathbf{y}_{\text{gold}}$ to denote the tuple of $\mathbf{y}^+$ and $\mathbf{Y}^-$ instead of a single gold-standard vector.

A form of the contrastive loss function is given by the equation:

$$\begin{aligned} L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) &= L(\mathbf{y}_\theta, \mathbf{y}^+, \{\mathbf{y}^-\}) \\ &= \log D(\mathbf{y}_\theta, \mathbf{y}^+) - \log \sum_{\mathbf{y}^- \in \mathbf{Y}^-} D(\mathbf{y}_\theta, \mathbf{y}^-) \\ &= \log \frac{D(\mathbf{y}_\theta, \mathbf{y}^+)}{\sum_{\mathbf{y}^- \in \mathbf{Y}^-} D(\mathbf{y}_\theta, \mathbf{y}^-)} \end{aligned} \tag{1.81}$$

where $D(\alpha, \beta)$ is a measure of the distance between α and β. For example, we can define $D(\alpha, \beta)$ as the Euclidean distance (see Eq. (1.72)). A problem here is how to generate positive and negative model outputs. In the supervised learning setup, one can simply treat the gold-standard vector as the positive output. For negative outputs, the model $f(\mathbf{x})$ can output a number of $\mathbf{y}$ through accepting different $\mathbf{x}$. In the unsupervised learning setup, $\mathbf{y}^+$ and $\mathbf{Y}^-$ are often defined based on some "natural" annotation. For example, $f(\cdot)$ can be a function that maps $\mathbf{x}$ to something and back to $\mathbf{x}$ (call it **auto-encoding**). Then, $\mathbf{y}^+$ is $\mathbf{x}$ itself or some neighbors of $\mathbf{x}$, and $\mathbf{Y}^-$ is a set of randomly generated vectors.

- **Error-based Loss**. Evaluation metrics, as generally used in counting errors in system output, can also be taken to be part of a loss function. For example, in machine translation, a popular evaluation metric is BLEU[19]. Thus, we can take minimizing $1 - \text{BLEU}$ as the objective. Let $\text{Error}(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}})$ be the "number" of errors in comparing $\mathbf{y}_\theta$ with $\mathbf{y}_{\text{gold}}$. The error-based loss is just the same as this number:

$$L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) = \text{Error}(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) \tag{1.82}$$

So far we have presented several loss functions for a wide variety of problems, such as classification, regression, and ranking. As we will see in this book, different loss functions have different effects on model behavior. However, testing all possible loss functions is simply impractical because there are so many of them. Users instead need to choose or design the most suitable loss functions for their own problems. This may take time but is necessary.

On another hand, there are general methods to improve the design of loss functions. For example, we can assume that the model output $\mathbf{y}_\theta$ is not a single vector but a variable with some probability. The loss $L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}})$ is thus treated as a variable too. Then, we redefine the loss function as the expectation of $L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}})$ under the distribution of $\mathbf{y}_\theta$:

$$\mathbb{L}(\{\mathbf{y}_\theta\}, \mathbf{y}_{\text{gold}}) = \mathbb{E}_{\mathbf{y}_\theta \sim \Pr(\mathbf{y}_\theta|\mathbf{x})}\left[L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) \cdot \Pr(\mathbf{y}_\theta|\mathbf{x})\right] \tag{1.83}$$

where the use of $\{\mathbf{y}_\theta\}$ means that $\mathbf{y}_\theta$ is not fixed. By accessing the space of possible $\mathbf{y}_\theta$, it offers a better estimation of the loss. This is essentially an instance of **the Bayesian approach**. $\mathbb{L}(\{\mathbf{y}_\theta\}, \mathbf{y}_{\text{gold}})$ is called **the Bayesian risk** or **risk** for short, sometimes.

Another way to improve training is introducing priors into the objective. A typical method

[19] BLEU is a precision-like score between 0 and 1. The higher the better.

is to add a regularization term R to the objective, like this:

$$\hat{\theta} = \arg\min_{\theta} L(\mathbf{y}_\theta, \mathbf{y}_{\text{gold}}) + \alpha \cdot R \tag{1.84}$$

where R could be another function that describes some aspect of the problem, such as the number of parameters. α is a hyperparameter controlling how much we respect R in training. The design of R is itself an important problem for many practical machine learning systems. Although we do not discuss them here, we will look at a few later in this book.

Once the objective is determined, we need some training algorithm to perform optimization. This is a very broad topic in machine learning, such that we do not even try to describe any of them in detail in this chapter. Anyway, one should not expect a universal algorithm that can solve all training problems, and there are indeed some algorithms that are suitable for certain types of problems. For example, we can use gradient descent to train a neural language model with the cross entropy-based loss [Bengio et al., 2003], can use quadratic programming to train an SVM model with the hinge loss [Cortes and Vapnik, 1995], and can use **minimum error-rate training** (**MERT**) to train a statistical machine translation model with the $1 - \text{BLEU}$ loss [Och and Ney, 2002].

1.3.5 Overfitting and Underfitting

The standard process of (supervised) machine learning comprises a training step and a test step. While one may try to minimize the loss on training samples, the learned model is used to deal with new samples that are never seen before. It is like what we experienced in our lives, for example, a student studies hard and wishes to get good grades in final exams. Yes, *studying hard = good grades* should always be true, but it does not mean that memorizing all the questions and answers in textbooks is a good way to perform well in exams. It always happens that the test questions are something different from what we learned. We therefore need some ability of **generalization** .

In machine learning, generalization is used to describe how well a model learned through experience predicts on new data. A system is thought to be of excellent generation performance if it learns little from training data but forms its prediction ability based on some "god" inductive biases on the problem. However, good generalization does not mean less training. Instead, practitioners would like to train a machine learning model on more training data to prevent it from memorizing all the things. Generalization is a very complex issue determined by several factors, including problem complexity, model architecture, amount of training data, training algorithm and so on. While there are no standard rules to ensure good generalization, researchers always try to address it somehow.

To describe how well a model generalizes to new data, there are two important terms, **underfitting** and overfitting. Underfitting refers to the phenomenon that a model does not learn sufficiently from the training data and thus has poor performance on new data. For example, we interrupt training accidentally and deploy the immature model for prediction. The model cannot perform well on either the training data or the test data. If a model underfits the training data, then one could improve it in some simple ways. For example, one could train the model

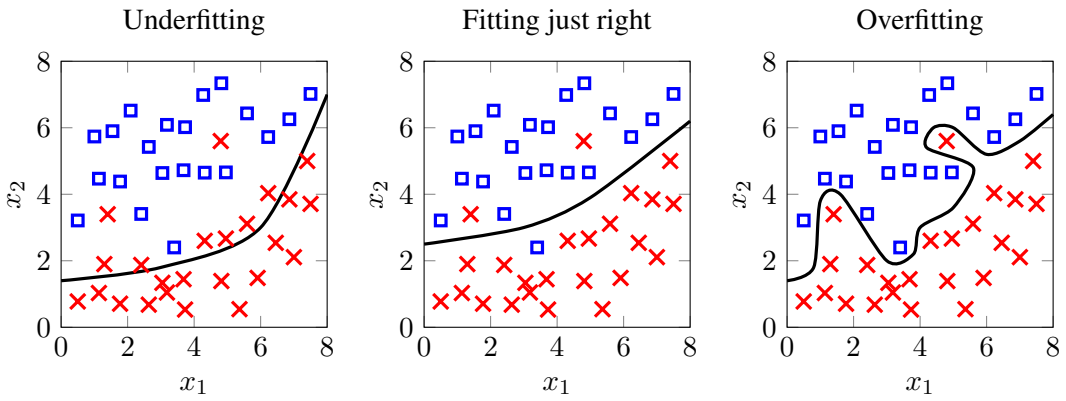


Figure 1.4: Decision boundaries of a binary classification problem. left = underfitting, right = overfitting, and middle = fitting just right. In the underfitting case, there are several obvious mistakes that are made in separating the two classes of data points. By shifting the decision boundary up a bit (middle), we obtain a satisfactory separation result, where most of the data points belonging to the same class are placed on the same side of the decision boundary. By contrast, a perfect separation requires a highly complex decision boundary instead (right).

for a longer time; one could remove unimportant portions from the training data; one could use a model with a simpler architecture instead.

In contrast to underfitting, overfitting refers to the phenomenon that a model fits the training data well but generalizes poorly on the test data (see Figure 1.4). A simple example of overfitting here is the OOV problem (see Section 1.2.5). It would be a disaster if a text classification model just fits those words that have been seen but gets stuck when new words appear.

The causes of overfitting are diverse. An example is learning a complex model on a small training dataset. The model complexity often matters when we design a machine learning model. If the model is complex and has many parameters, then it would be much easier to overfit a small number of samples (see Figure 1.4). The problem would be more difficult if there is noisy data, because of the errors of "garbage in, garbage out" in training. In addition, excessive training is another cause of overfitting. For example, we can heavily tune a system to enforce it to model the data with no errors. The system would be fragile for new samples, even when there are small fluctuations in input.

Overfitting can be alleviated in many ways. Here are some commonly-used techniques.

- **Using more (high-quality) training data**. Large-scale training helps the model capture the true patterns in data. However, adding noisy data would do this in a negative way.
- **Using validation data**. Validation data is some test data but used in training. For example, a dataset can be divided into **held-out data** and training data. One can simply **early stop** the training process when the performance drops on the held-out data.
- **Using simpler model architectures**. As noted previously, Occam's Razor is a principle

we can follow in model design. Models with more complex architectures, though powerful, would be more likely to fit the noisy data points if the problem is not so difficult itself. Using a simpler model architecture instead could make it easier to model the dominant patterns in the data.

- **Regularization**. Regularization is another way to control the model complexity. Typically, it regularizes model parameters by priors. An example is smoothing (see Section 1.2.5). It re-estimates the distribution of words after training. A more general method is regularized training (see Eq. (1.84)). For example, we can define the regularization factor as the l_1 norm of the parameters, and bias the model to those whose parameters are not in large absolute values.
- **Combining multiple models**. A better prediction can also be made by ensembling multiple models. These models (call them **component models**) are in general of different parameters or architectures, and/or are trained with different portions of the data. The variance in models can reduce the risk that all these models overfit the data in exactly the same manner. These models are, therefore, less likely to make similar mistakes in prediction.

1.3.6 Prediction

Although we restricted our discussion to classification in previous sections, (supervised) machine learning is not just a task of predicting a label for an input object. There are many types of machine learning problems, depending on what form of the prediction is defined.

- **Classification**. Classification is perhaps one of the most common machine learning problems. A classification system is required to assign one or more classes to an input object.
- **Regression**. In statistics, regression studies the relationship between a **dependent variable** (or an outcome) and an **independent variable**. While regression has many applications, it is often framed as score prediction in NLP. For example, taking a movie review as input (i.e., an independent variable), the regression model learns to predict a recommendation score (i.e., a dependent variable).
- **Ranking**. A ranking model is to predict the order of a set of input objects. For example, a model ranks a number of translations in terms of translation quality.
- **Structure prediction**. Many machine learning models are required to output not only a real value or a class but a tree or a sequence. The task of predicting structured outputs is called structure prediction. For example, a syntactic parser is a structure prediction system, as its output is a tree structure.

In addition to these, **mining** is a term that is frequently used in the community, although it is somehow not a standard machine learning problem. The problem of mining refers to discovering unknown patterns in the data. An example we would like to categorize into this is word clustering. Given a number of words, the clustering system "predicts" the cluster for each word. The output of such systems is not pre-defined. Patterns in data are themselves hard

to describe. Thus, the term "mining" could cover a range of problems. To avoid confusion, we will use more specific terms (such as word clustering) to refer to mining-related problems.

Despite a fundamental aspect of machine learning, prediction is conventionally assumed to be trivial, given that many models and methods are tested on standard classification and regression tasks. On the other hand, prediction is non-trivial in structure prediction, such as parsing and machine translation, which are very common in NLP. Essentially, predicting a tree or a sequence is a **search problem**. For example, there exist a theoretically infinite number of translations given a source-language sentence. Even if we have a model to evaluate every translation, finding the optimal translation in the search space is obviously a computational challenge. In such cases, we need some way to make it feasible to perform search. This is implemented by either resorting to the general search algorithms in artificial intelligence or developing new algorithms for specific problems. As an aside, the study on the search problem offers a new view on the mistakes made by a machine learning model: some of the errors are due to inaccurate modeling (call them **model errors**), and the rest are due to inaccurate search (call them **search errors**). For prediction, eliminating search errors is a goal but often at the cost of a considerably large amount of search effort. We sometimes must trade off between efficiency and accuracy if a machine learning model is deployed for practical purposes. We will see a few examples in Chapter 5.

1.4 Model Selection and Evaluation

For most machine learning problems, the goal is to find a model that would perform the best on new data. Two problems can be separated out from this goal [Hastie et al., 2009]:

- **Model selection**. Selecting the best model on training data by some criteria.
- **Model evaluation**. Estimating the performance of a given model on new data.

As noted in Section 1.3.4, loss functions (or error functions) are common ways of measuring errors in a prediction $\mathbf{y}_\theta = f_\theta(\mathbf{x})$ with respect to a gold-standard $\mathbf{y}_{\text{gold}}$. Given K labeled training samples $\{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}_{\text{gold}}), ..., (\mathbf{x}^{(K)}, \mathbf{y}^{(K)}_{\text{gold}})\}$, the **training error** is given by

$$\text{Err}_{\text{train}} = L(\{\mathbf{y}^{(k)}_\theta\}, \{\mathbf{y}^{(k)}_{\text{gold}}\}) \tag{1.85}$$

where $\{\mathbf{y}^{(k)}_\theta\}$ are the predictions over the training dataset, and $\{\mathbf{y}^{(k)}_{\text{gold}}\}$ are the corresponding gold-standards. $L(\{\mathbf{y}^{(k)}_\theta\}, \{\mathbf{y}^{(k)}_{\text{gold}}\})$ is in general defined as the averaged loss over all training samples:

$$L(\{\mathbf{y}^{(k)}_\theta\}, \{\mathbf{y}^{(k)}_{\text{gold}}\}) = \frac{1}{K}\sum_{k=1}^{K} L(\mathbf{y}^{(k)}_\theta, \mathbf{y}^{(k)}_{\text{gold}}) \tag{1.86}$$

or defined as a single measure on the entire set of training samples. Likewise, we can define the **test error** on the test dataset, denoted as Err_{test}. Err_{test} is also called **generalization error**. It indicates how well a model generalizes to new data.

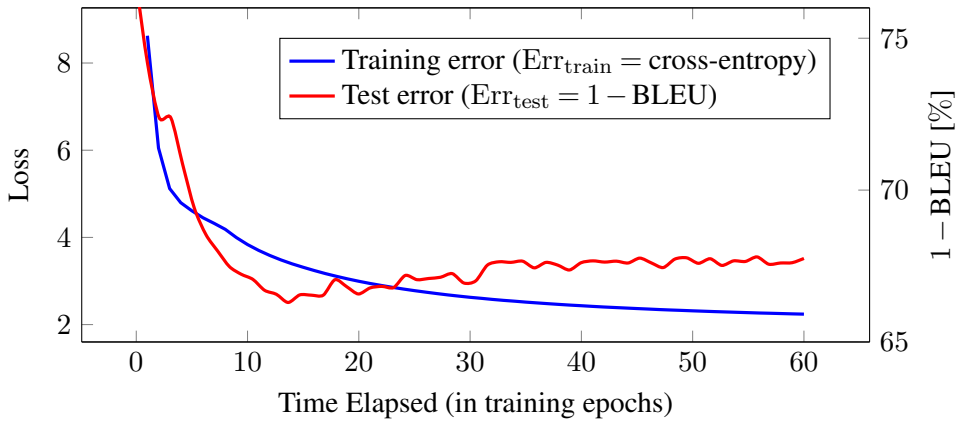


Figure 1.5: Curves of training error and test error for a machine learning system. The training error is measured in terms of the cross-entropy loss, and the test error is measured in terms of $1 - \mathrm{BLEU}$. All statistics are collected by running a neural machine translation system on the IWSLT De-En benchmark. The training error continues to drop as more training epochs are involved. The test error, on the other hand, follows a trend of first going down and then going up. When the test error starts to increase, the model is likely to overfit the training data.

In the preceding sections we assumed that minimizing $\mathrm{Err}_{\mathrm{train}}$ is the objective of training, i.e., $\hat{\theta} = \arg\min_\theta \mathrm{Err}_{\mathrm{train}}$. However, we cannot assume that $f_{\hat{\theta}}(\cdot)$ can obtain the minimum $\mathrm{Err}_{\mathrm{test}}$ in the same way. See Figure 1.5 for learning curves of a machine translation system. Clearly, $\mathrm{Err}_{\mathrm{test}}$ does not correlate with $\mathrm{Err}_{\mathrm{train}}$ well. The training error keeps reducing as training proceeds. However, the test error goes up after following the same trend as the training error for a period of time, indicating overfitting of the model. This makes the problem a bit more complicated, as we cannot always trust $\mathrm{Err}_{\mathrm{train}}$ although it is and should be the measure of the goodness of training. Surely, we need some way to select a better model, in addition to looking at $\mathrm{Err}_{\mathrm{train}}$ only.

1.4.1 Strategies for Model Selection

Choosing the optimal model on the training data is challenging because the motivation here is "greedy" itself — we hope that a machine learning model can generalize from a finite, even a "small" number of samples. From the statistical learning point of view, the challenge is due to the way we define the learning problem. An implicit assumption in machine learning is that all data is generated by some distribution. Thus, the learning problem is determined by generating the training data via a data-generation distribution and the test data via another distribution.

For example, if both the training and test datasets are sufficiently large and obtained via the same data-generation distribution, then the learned model can perform on the test data as well as on the training data. In this case, it is easy to generalize the model from the training data to the test data. By contrast, if all training and test data is generated in an arbitrary manner (say a uniform distribution over the entire space of data points), then the model will fail to generalize, as everything learned on the training data does nothing with the test data.

It will be more interesting if we consider all possible problems. **The no free lunch theorem** states that all learning algorithms will perform equally well if we average the test error over all problems[20]. In other words, all learning will make no sense if there is no preference for certain problems. However, developing a universally good machine learning model on all problems is idealistic. In real-world applications, the training and test data is always assumed to at least in part follow some distribution. Therefore, there are indeed some ways to capture this distribution and improve the generalization ability of a model. Two scenarios are generally considered in improving machine learning systems:

- Given the model design and the training algorithm, how to develop or select training data to reduce the test error.
- Given the training data, how to develop or select a model to reduce the test error.

The first scenario is complicated and relates to many practical issues, e.g., annotation, data cleaning, data quality estimation and so on. Since these issues are not the focus for model selection, we do not discuss them but leave some to subsequent sections. Here, we focus on the model selection problem in the second scenario.

1. Model Complexity

The simplest method of model selection might be testing the models on validation data. Typically, this data does not overlap with either training or test data, but is assumed to be generated in the same way as the test data. However, such data is not always available. In some cases, we do not even know anything about the test data. So many model selection methods are validation-free.

A common way is to use **model complexity** (or **model capacity**) as an indicator of the selection. In machine learning, model complexity can be interpreted in several different ways. For example, a non-linear model is intuitively more complex than a linear model. Also, a model with more parameters is more complex than a model with fewer parameters under the same model architecture. More formal definitions could be found in the theoretical part of machine learning, such as **the Vapnik-Chervonenkis dimension** or **the VC dimension** [Vapnik and Chervonenkis, 1971]. Here we simply treat model complexity as a measure of the expressive power of a model, i.e., a higher model complexity indicates more hypotheses that the model can express.

While complex models are usually assumed to be more powerful, higher model complexities are not always helpful. In fact, complex models are more likely to overfit the data, especially when a small dataset is used for training. By contrast, too simple models are often prone to underfitting. We therefore need to seek an "optimal" level of model complexity. Figure 1.6 plots training and test errors against model complexity. An "optimal" complexity can be chosen when the training error tends to convergence. While Figure 1.6 shows an intuitive example, it is still hard to say at what point we can choose the model. The common practice, though not formally described in most cases, is to choose among those "good" models by using

[20]The no free lunch theorem was originally presented in a classification scenario [Wolpert, 1996], and was further extended to search and optimization problems [Wolpert and Macready, 1997].

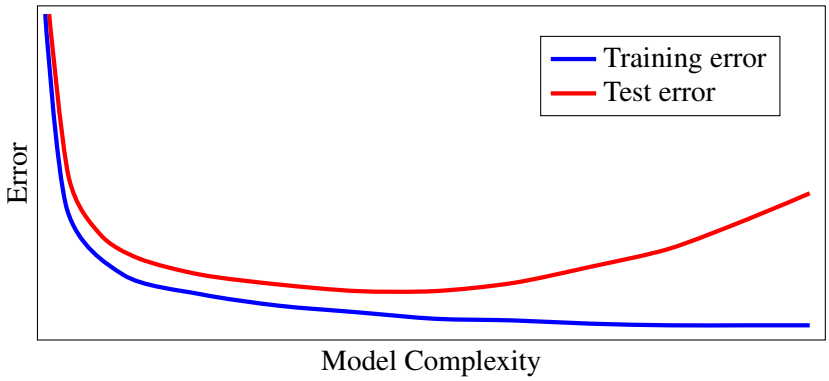


Figure 1.6: Curves of training error and test error under different model complexities. Complex models help in reducing the training error as they can compute complex functions in fitting data points. However, a too large model complexity is more likely to lead to overfitting and is harmful to the generalization ability of the models. For example, the test error increases as more complexity is added.

Occam's Razor. Suppose we have a set of models that perform comparably well on the training data but are of different complexities. According to Occam's Razor, the simplest model is the "best" choice. Many criteria are available to measure the model complexity. For example,

- **Number of parameters**. Though very simple, counting the number of parameters is the most intuitive yet effective method. It can be extended to counting the **effective number of parameters** which is defined to be the trace of the matrix used to transform $\mathbf{y}_{\text{gold}}$ to $\mathbf{y}_\theta$.
- **p-norm of parameters**. The p-norm of a parameter matrix is also an indicator of how complex a model is (see Section 1.1.1). For example, according to the l_1 norm, a model with larger absolute values for parameters is more complex.
- **Description length**. Description length is a term used in data compression. For example, it could be the number of bits used to store a model. Thus, the **minimum description length** (or **MDL**) indicates the most compressed model.
- **The VC dimension**. It is originally from computational learning theory. In short, the VC dimension can be defined as the maximum number of data points that can be shattered by the classifier.

In addition, there are other choices for defining the criterion, including **the Akaike information criterion** (**AIC**), **the Bayesian information criterion** (**BIC**), **the minimum message length** (**MML**) and so on. They can be found in most textbooks on statistics and/or statistical learning [Burnham and Anderson, 2002; Konishi and Kitagawa, 2007; Hastie et al., 2009].

2. Bias-Variance Tradeoff

Controlling model complexity to avoid overfitting and underfitting is also linked to the tradeoff between bias and variance. Bias (or prediction bias) is the amount that the model prediction differs from the true value. In statistics, bias is a **systematic error** that cannot cancel out even if we run a large number of repeated experiments. In general, bias error results from the wrong assumptions about the problem, such as approximating a non-linear problem via a linear model. This is very interesting! We can establish the connection of the bias error here with the inductive bias used in mode design (see Section 1.3.2). For example, given training data, a large bias model is usually due to the fact that there are more assumptions and the model is not complex enough. To make it simple, we would say that more (or stronger) inductive biases can result in a lower model complexity and more bias error in prediction. Occasionally, the term *bias* is used as a short for both bias in prediction (from a statistics perspective) and inductive bias (from a model design perspective), although they are considered to have different meanings[21].

Variance, on the other hand, describes how spread the prediction is when there are variations in training data. The variance error also correlates with model complexity. For example, a complex model tends to exhibit higher variance.

Both bias and variance are sources of errors of a system. A common example is the bias-variance decomposition of mean squared error. Here we use some notation that differs slightly from that used in previous sections. Let D be a set of K training samples and $f_{\hat{\theta}(D)}(\cdot)$ be a model leaned on D. Further, given a new sample $\mathbf{x}$, let $\mathbf{y}_{\hat{\theta}(D)} = f_{\hat{\theta}(D)}(\mathbf{x})$ be the model prediction and $\mathbf{y}_{\text{gold}}$ be the "true" prediction. The bias and variance are defined as:

$$\text{bias} = \mathbb{E}_D[\mathbf{y}_{\hat{\theta}(D)}] - \mathbf{y}_{\text{gold}} \tag{1.87}$$

$$\text{variance} = \mathbb{E}_D\big[\big(\mathbb{E}_D[\mathbf{y}_{\hat{\theta}(D)}] - \mathbf{y}_{\hat{\theta}(D)}\big)^2\big] \tag{1.88}$$

where $\mathbb{E}_D[\mathbf{y}_{\hat{\theta}(D)}]$ is the mean of $\mathbf{y}_{\hat{\theta}(D)}$ over all possible K sample training datasets. Thus, the bias is some sort of difference between the mean and the true value, and the variance is some sort of difference between the mean and the predicted value. Taking the mean squared error as the error measure, we can write the expected error as:

$$\begin{aligned} \text{error} &= \mathbb{E}_D\big[\big(\mathbf{y}_{\hat{\theta}(D)} - \mathbf{y}_{\text{gold}}\big)^2\big] \\ &= \text{bias}^2 + \text{variance} \end{aligned} \tag{1.89}$$

For lower mean squared error, reducing both bias and variance simultaneously is obviously an ideal goal. However, it is difficult to make a model that exhibits both low bias and variance. When one of the two decreases, the other increases (see Figure 1.7). Researchers must choose the optimal level of model complexity while preventing training from overfitting and underfitting. This also depends on the problem we intend to solve. For example, a simple

[21] Bias is more often used in statistics to describe some aspect of an estimator.

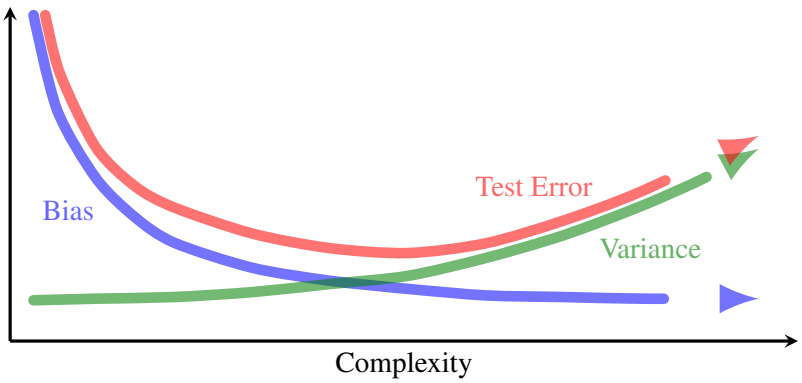


Figure 1.7: Bias and variance against model complexity [Goodfellow et al., 2016]. The curves show a conflict in reducing the bias error and the variance error simultaneously. By varying the model complexity, one can obtain either a low-bias, high-variance model or a high-bias, low-variance model. Both of the two cases exhibit high test error. For example, a high-variance model is often of a larger model complexity. While such a model is able to deal with complex problems, it is more likely to overfit the data. On the other hand, a high-bias model often means a simpler model but tends to underfit the data. To improve the generalization on test data, one can seek a tradeoff between bias and variance. For example, there is low test error when a "middle sized" model is chosen.

model generally has low variance but high bias. However, if we use the simple model (say a linear model) to describe a complex problem (say a non-linear problem), then underfitting would probably occur because the problem is too "hard" for the model.

Returning to the model selection problem, the bias-variance tradeoff is not a rule for model selection, but a principle we must keep in mind. Often, one needs to make compromises to create a model that makes reasonably good predictions. It is also worth noting that, in many applications, complex models are usually accompanied with the inefficiency problem. An appropriate method might be to start with a simple model and only add complexity when it is needed.

3. Model Combination

Selecting from a set of models is not the only way to reduce generalization error. Alternatively, one can do this in the opposite way, and combine these models for a "stronger" model. Such a method is called **ensemble learning** [Seni et al., 2010; Zhou, 2012]. A key idea of ensemble learning is to create a set of component models (or ensemble models), such that they can vote for a better prediction. The simplest of these is a **mixture model** that averages the predicted scores of multiple component models (call it **model averaging**), whereas a more sophisticated method can combine the sub-structures of these models.

Component models are in general generated in some way that they can exhibit some diversity. For example, they can be learned on different portions of the training data, or by using different initializations for model parameters. Interestingly, it is found that such methods

can guarantee the reduction of generalization error somehow. For example, bagging helps to lower variance [Breiman, 1996], and boosting helps to lower bias [Schapire, 1990]. These are linked back to what we presented in Section 1.4.1: the generalization error can be reduced by either reducing the bias error or reducing the variance error.

But discussing how to combine models is beyond the scope of this chapter. While it is even not appropriate to categorize model combination as a topic related to model selection, it can be seen as a means of improving the generalization ability. In this sense, both model combination and model selection address problems on a similar theme. In fact, model combination is remarkably effective for many NLP tasks. For example, most state-of-the-art systems in NLP are based on the combination of multiple models.

1.4.2 Training, Validation and Test Data

We turn now to the data problem. As discussed in the previous sections, in the training stage, a training dataset is used to fit the parameters of the model. In the test stage, a test dataset is used to evaluate the learned model. Closely related to test data is validation data, which has come up a few times in this chapter. A validation dataset is a test dataset as well but can be used in the training stage. It is commonly used for model selection and tuning hyperparameters.

In many cases, one may imagine that there is some data for training and some additional data for validation and test. This assumption, however, is not realistic in many real-world applications. For example, developers cannot always access the data of system use after deploying a system. From a scientific point of view, there is no "real" new data for test — when you see new data, it is not new anymore. Therefore, what we address is essentially an analogue of the problem.

A simple method, as in many research papers, is to verify machine learning models on benchmark tasks. In these tasks, all data is prepared in advance, and all you need is to run your models on the data. Such a method makes it easy to compare different systems directly, as all these systems are trained and tested on the same datasets. Occasionally, we are just given a number of samples but not told which are for training and which are for test. In such cases, the data can be divided into parts each of which is used for some purposes. For example, a split could be 60% for training, 20% for validation, and 20% for test.

While data splitting provides a way to assess the performance of a model, the assessment result is not always stable due to sampling bias. Sometimes, the performance varies greatly across different runs of data splitting. The problem is more obvious when the dataset is too small to perform sufficient training or test.

A common way to weaken the effect of this bias is **cross-validation**. Cross-validation is a resampling method. Each round of cross-validation is a new split of data and the result is the combination of the assessment over the rounds. A simple method is random subsampling that repeats random partition of the data and averages the performance over runs. Another method is k-fold cross-validation. It divides the data into k parts. In each round of cross-validation, some parts are used as training data, and other parts are used as validation and test data. For example, in 10-fold cross-validation, a model can be trained and validated/tested for 10 times, each choosing one of the ten parts as the test dataset.

Another note on the scale of data. For practitioners, one of the most frequent questions is how many samples are enough for learning a good model. This may be the most difficult question on which different people can have consensus answers. There are many theoretical results that can tell the bound of errors given a certain amount of data, whereas in most cases we just simply follow the "the more the better" idea. In another line of thought, a system could be **sample efficient**. In general, a sample efficient system can reach a good level of performance by using fewer samples or seeing the same sample for fewer times. For example, tuning a pre-trained model is sample efficient because the samples are not used for learning from scratch but a modest update of the model. Another example is **few-shot learning**. It aims to generalize from observing very few samples for a task.

1.4.3 Performance Measure

As an essential part of every machine learning problem, a performance measure describes how well a system performs given some data. Usually it is used in either designing the training objectives or evaluating the result of the final system. For example, all those loss functions described in Section 1.3.4 are some kinds of performance measures.

As for evaluating the performance on test data, a measure is often designed in a way that we can count the real errors. Thus, re-using the loss functions in training might not be a good choice for reporting the final score. For example, the widely-used measures for classification problems are precision, recall and F_1 score. They are proposed to quantify the ability of a classification system in certain aspects: given a class c, precision computes the fraction of correct predictions in predicting c, and recall computes the fraction of correct predictions on all samples labeled as c. The F_1 score is a measure that combines precision and recall.

Notice that performance measures are not necessarily designed for optimization. In this sense, they may not guarantee some mathematical properties, such as differentiable and continuous functions. An example is the BLEU metric used in machine translation. BLEU is a function combining precision scores and a penalty score [Papineni et al., 2002]. This in turn makes the metric non-differentiable and discontinuous. In NLP, there are many such evaluation measures that are ad-hoc for certain tasks. These raise an interesting problem that the loss function used in training may differ from what we actually use in evaluating the final model. Thus, one sometimes needs to take into account the discrepancy between the objectives of training and test.

Another problem with performance measures in NLP is that there might be two or more "answers" for the same "question". For example, there are generally multiple good translations for a source-language sentence. One solution is to take multiple gold-standards into account when designing a performance measure. BLEU is such a case. It counts the maximum number of the correct translation segments over all reference translations. The second solution involves human evaluation. Such a way of evaluation is more accurate but of course is more expensive. When developing practical systems, practitioners usually train and tune the systems using automatic measures, and call for human evaluations for the final test.

1.4.4 Significance Tests

Now, assuming you are improving a system in some way, you might be wondering if the improvement is significant enough or not. All you have is a performance measure. So you can tell the performance difference between any two points in developing the system, but you cannot tell if the difference is real or happens by chance.

In this example, you implicitly try to reject or accept a claim that a system is better than another system (or not). In statistics, **significance tests** are a method to model this problem. Suppose we have two systems A and B. And there are a number of datasets on each of which we evaluate the two systems via the same performance measure. Then, we make two hypotheses

$$H_0\text{: System } A \text{ performs worse than or equally well as system } B.$$

$$H_1\text{: System } A \text{ performs better than system } B.$$

where H_0 is the **null hypothesis**, and H_1 is the **alternative hypothesis** that is contradictory to the null hypothesis. By testing these hypotheses, we can claim that system A is significantly better than system B (i.e., reject H_0 and accept H_1) or not (i.e., accept H_0 and reject H_1). We probably make errors in the test, for example, incorrectly rejecting a true null hypothesis (type I error), or incorrectly accepting a false null hypothesis (type II error). The two types of errors are at odds with each other. A decrease of one may lead to an increase of the other. Alternatively, we can decrease one while guaranteeing that the other is upper bounded. For example, we can reduce the type II error as much as possible, and keep the type I error below a constant α. α is typically called the **significance level** of a test. It is standard practice to choose the significance level in the interval $[1\%, 5\%]$. When conducting statistical testing, we can obtain the probability of the type I error (call it a p**-value**). A p-value that is lower than the significance level can make a rejection of the null hypothesis. For example, in the above example, with a significance level of 5%, a p-value $= 3\%$ means that the improvement is statistically significant. For more information about the p-value, we refer the reader to other books on statistics [McClave and Sincich, 2006; Freedman et al., 2007; Freedman, 2009].

Note that the conclusion of significance tests depends on several factors, such as the number of experiments and the variance in the results of experiments. A problem with applying significance tests to NLP tasks is that there are often very few datasets for running the experiments [Dror et al., 2020]. Ideally, we know the true data distribution and can consider it in the test. This method is called the **parametric test**. If we cannot find the true data distribution, then, as a **non-parametric** test method, we can generate a number of experiments by sampling over a dataset or adding randomness into the test.

Significance tests are important for drawing convincing conclusions in developing machine learning systems, although they are often ignored unintentionally. Figure 1.8 shows evaluation results of three models. Each of them is run for several times with different initial parameters. While system A is superior to system B in terms of the averaged performance, there are large variances in their results. The significance test indicates that the difference is not significant. By contrast, the difference between system A and system C is significant because their performance differs greatly enough in most cases. On the other hand, researchers have found

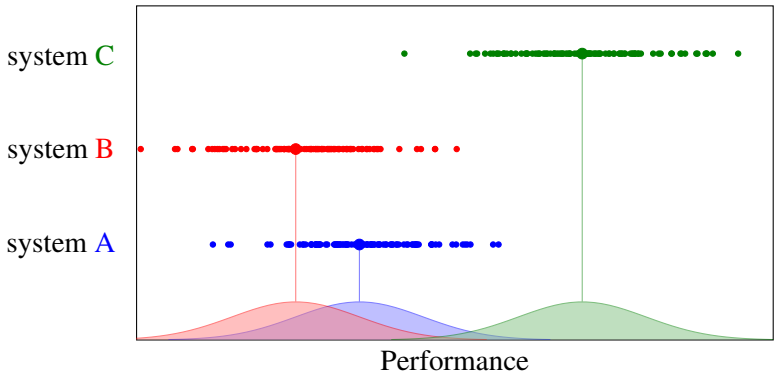


Figure 1.8: Performance of three machine learning systems. For each system, there are many different results because we introduce some randomness into training (e.g., data shuffling, random starting points, etc). Although it seems that System A outperforms System B, there is no real distinction between them, because they overlap a lot in the distributions of the performance (see the bottom of the figure). When comparing System C with System A or B, the difference in performance is significant because we could accept the H_1 hypothesis (i.e., System C outperforms System A or B) given a large number of experiments.

that there are indeed some thresholds of performance gain to indicate significance under certain circumstances. For example, we would say that the significance can be roughly indicated by a certain metric gain if we compare similar systems [Berg-Kirkpatrick et al., 2012].

1.5 NLP Tasks as ML Tasks

While there are a wide variety of NLP tasks, many of them can be formulated as the same machine learning problem. This enables a universal solution to a group of NLP problems by using a general machine learning approach. Typically, an NLP task can be described as learning to map language units to some output. Following the notation used in this chapter, we use $\mathbf{x}$ to denote the input feature vector (or matrix) of an NLP task, and use $f(\mathbf{x})$ to denote the function that is learned to process $\mathbf{x}$. Here are some of the common tasks in NLP.

1.5.1 Classification

Suppose there are a set of classes or labels C. Each class is represented by a distinct integer in $\{1, ..., |C|\}$. A classification model is a function that maps the input $\mathbf{x}$ to a $|C|$-dimensional vector $\mathbf{y}$, i.e., $\mathbf{y} = f(\mathbf{x})$. Each entry of $\mathbf{y}$ is a score corresponding to class i, denoted by $y(i)$. The task here is to assign $\mathbf{x}$ to one or more classes having the highest scores. Consider single-label classification as an example. The prediction is given by the equation

$$\hat{c} = \underset{1 \leq i \leq |C|}{\arg\max}\, y(i) \tag{1.90}$$

where $\hat{c}$ is the "best" class assigned to $\mathbf{x}$. Sometimes, one needs a probability-like output (see

Section 1.2.1). Let $\psi(\cdot)$ be a function that normalizes a vector into a distribution[22]. We then obtain a probabilistic classifier:

$$\mathbf{y} = \psi(f(\mathbf{x})) \tag{1.93}$$

Classification may be the most common problem in NLP. There are many applications in addition to categorizing documents into predefined classes. Among them are choosing a sense for a word [Yarowsky, 1994], determining the polarity of a sentence [Pang et al., 2002], checking whether two entities should be linked [Krebs et al., 2018], classifying the way of associating a semantic argument with a verb [Gildea and Jurafsky, 2002], and so on. When adapting a classification model to these tasks, all you need is to design the form of $\mathbf{x}$ and the set of classes.

1.5.2 Sequence Labeling

An extension to standard classification is to classify a set of samples simultaneously. **Sequence labeling** is an example of such a problem. In sequence labeling, the input is a sequence of n tokens, such as a sequence of n words. A sequence labeling system is required to assign each input token $\mathbf{x}(i)$ a label $l(i)$. Here the boldface in $\mathbf{x}(i)$ is used to emphasize that the token is represented as a feature vector. For convenience, we write $\mathbf{x}(i)$ as $\mathbf{x}_i$ and $l(i)$ as l_i. The function $f(\cdot)$ maps the sequence $\mathbf{x}_1 ... \mathbf{x}_n$ into another sequence $\mathbf{y}_1 ... \mathbf{y}_n$, where $\mathbf{y}_i$ is the output vector corresponding to $\mathbf{x}_i$. This can be formulated as:

$$\begin{bmatrix}\mathbf{y}_1 & ... & \mathbf{y}_n\end{bmatrix} = f(\begin{bmatrix}\mathbf{x}_1 & ... & \mathbf{x}_n\end{bmatrix}) \tag{1.94}$$

For vector $\mathbf{y}_i$, each entry $y_i(c)$ corresponds to the prediction score of a class $c \in C$. Note that $f(\cdot)$ allows for the use of a larger context. For example, one can condition the prediction $\mathbf{y}_i$ on the entire input sequence [Lafferty et al., 2001]. The final output of the system can be defined as the "optimal" label sequence induced from $\mathbf{y}_1 ... \mathbf{y}_n$. A simple method is to choose the label sequence that maximizes the sum of the scores over all positions, like this

$$\begin{bmatrix}\hat{l}_1 & ... & \hat{l}_n\end{bmatrix} = \underset{l_1,...,l_n \in C}{\arg\max} \sum_{i=1}^{n} y_i(l_i) \tag{1.95}$$

A straightforward application of sequence labeling to NLP is to tag each token of the input sequence, such as **part-of-speech tagging** (or **POS tagging**). Furthermore, sequence labeling

[22] A similar idea can be found in Eq. (1.48). Given a vector $\mathbf{a} = \begin{bmatrix}a(1) & ... & a(n)\end{bmatrix}$, the normalization function has the form:

$$\psi(\mathbf{a}) = \begin{bmatrix}\frac{a(1)}{\sum_{i=1}^{n} a(i)} & ... & \frac{a(n)}{\sum_{i=1}^{n} a(i)}\end{bmatrix} \tag{1.91}$$

Another way is using the Softmax function:

$$\psi(\mathbf{a}) = \begin{bmatrix}\frac{\exp(a(1))}{\sum_{i=1}^{n} \exp(a(i))} & ... & \frac{\exp(a(n))}{\sum_{i=1}^{n} \exp(a(i))}\end{bmatrix} \tag{1.92}$$

Tokens :	Most	are	expected	to	fall	below	previous-month	levels	.
POS tags :	JJS	VBP	VBN	TO	VB	IN	JJ	NNS	·
Chunk tags :	B-NP	B-VP	I-VP	I-VP	I-VP	B-PP	B-NP	I-NP	O
	NP	VP				PP	NP		

Figure 1.9: An example of sequence labeling for POS tagging and chunking. The example is from the training data of the CoNLL 2000 shared task. Each token is labeled with a POS tag and a chunk tag. A chunk tag has an initial character chosen from $\{\mathrm{B}, \mathrm{I}, \mathrm{O}\}$, where B = beginning of a chunk, I = inside a chunk, and O = outside a chunk. So, a chunk always starts with a "B" tag, optionally followed by "I" tags. For example, the VP (verb phrase) chunk in the example spans over the chunk tag sequence "B-VP I-VP I-VP I-VP".

is able to deal with more complex problems by using labels in a clever way. A well-known example is the use of the "IOB" label format in identifying chunks spanning multiple tokens (call it **chunking**). In this method, "I", "O" and "B" stand for a token inside a chunk, a token outside a chunk, and the leftmost token of a chunk [Ramshaw and Marcus, 1995]. As such, a chunk always starts with a "B" and ends just before the next "B" or a new "O". See Figure 1.9 for POS tagging and chunking results on an example sentence. As sequence labeling allows the labeling of both tokens and spans, it has been applied with strong results to many tasks, including POS tagging [Bahl and Mercer, 1976], chunking [Tjong Kim Sang and Buchholz, 2000], **named entity recognition** (**NER**) [Tjong Kim Sang, 2002], and so on.

1.5.3 Language Modeling/Word Prediction

Statistical language modeling (or **language modeling** for short) is a task of assigning a probability $\Pr(w_1, ..., w_n)$ to a sequence of words $w_1 ... w_n$. This joint probability is generally decomposed into a product of conditional probabilities, by using the chain rule:

$$\begin{aligned} \Pr(w_1, ..., w_n) &= \Pr(w_1) \cdot \Pr(w_2|w_1) \cdots \Pr(w_n|w_1, ..., w_{n-1}) \\ &= \prod_{i=1}^{n} \Pr(w_i|w_1, ..., w_{i-1}) \end{aligned} \tag{1.96}$$

Eq. (1.96) describes a procedure that generates a word sequence from left to right (call it **auto-regressive** generation). Estimating $\Pr(w_i|w_1, ..., w_{i-1})$ is essentially a missing word prediction problem: we mask out the last word of a sequence and guide the language model to predict the correct word at that position. See below for a word sequence where the last word is missing.

Pride and prejudice is one of the best known ___

We can reuse the idea in classification to model the probability distribution $\Pr(\ \_\ |$

Pride, *and*, ..., *known*). Let $\mathbf{x}_i$ be the vector representation of w_i. We can define a function that reads $\mathbf{x}_1 ... \mathbf{x}_{i-1}$ and produces a vector $\mathbf{h}_i$:

$$\mathbf{h}_i = f(\mathbf{x}_1, ..., \mathbf{x}_{i-1}) \tag{1.97}$$

where $\mathbf{h}_i$ is the intermediate states of the word distribution at position i. For a sounding distribution, we normalize $\mathbf{h}_i$ by some normalization function $\psi(\cdot)$. Thus, the distribution at position i would be

$$\begin{aligned} \mathbf{y}_i &= \psi(\mathbf{h}_i) \\ &= \psi(f(\mathbf{x}_1, ..., \mathbf{x}_{i-1})) \end{aligned} \tag{1.98}$$

Obviously, $y_i(w_i)$ is the probability of w_i given previous words, i.e., $y_i(w_i) = \Pr(w_i | w_1, ..., w_{i-1})$. Note that Eq. (1.96) only considers the left context when predicting a word. A natural extension to this is to condition the prediction on all available context. Consider, for example, a sentence with a masked word in the middle.

Pride and ___ is one of the best-known novels

In this example, we can predict the masked word by using both the left context (*Pride and*) and the right context (*is one of the best known novels*):

$$\mathbf{y}_i = \psi(f(\mathbf{x}_1, ..., \mathbf{x}_{i-1}, \mathbf{x}_{i+1}, ..., \mathbf{x}_n)) \tag{1.99}$$

This is a bidirectional model, and is commonly used in **auto-encoding** methods for learning sequence representation models [Devlin et al., 2019].

1.5.4 Sequence Generation

Sequence generation covers a range of NLP problems, including machine translation, summarization, question answering, dialogue systems, and so on. Usually, it refers to mapping some data to a sequence. Here we focus on the **sequence-to-sequence** problem, in that a source-side sequence is transformed to a target-side sequence, although sequence generation is not specialized to work with chain structures on the source-side.

For notation convenience, we use boldface variables to denote sequences from now on. For example, $\mathbf{a}$ is a sequence of size n. It can be written as either $\begin{bmatrix} a_1 & ... & a_n \end{bmatrix}$ or $a_1 ... a_n$. Let $\mathbf{s} = s_1 ... s_m$ and $\mathbf{t} = t_1 ... t_n$ be the sequences to transform from and to. The sequence-to-sequence problem can be described as finding a target-side sequence that maximizes $\Pr(\mathbf{t}|\mathbf{s})$:

$$\hat{\mathbf{t}} = \arg\max_{\mathbf{t}} \Pr(\mathbf{t}|\mathbf{s}) \tag{1.100}$$

Like language modeling, $\Pr(\mathbf{t}|\mathbf{s})$ can be formalized in an auto-regressive fashion:

$$\begin{aligned}\Pr(\mathbf{t}|\mathbf{s}) &= \Pr(t_1,...,t_n|\mathbf{s}) \\ &= \Pr(t_1|\mathbf{s}) \cdot \Pr(t_2|\mathbf{s},t_1) \cdots \Pr(t_n|\mathbf{s},t_1,...,t_{n-1}) \\ &= \prod_{i=1}^{n} \Pr(t_i|\mathbf{s},t_1,...,t_{i-1}) \end{aligned} \tag{1.101}$$

Eq. (1.101) differs from Eq. (1.96) only in the additional condition (i.e., $\mathbf{s}$) introduced to these probabilities. In this sense, we can use Eqs. (1.97-1.98) to solve $\Pr(t_i|\mathbf{s},t_1,...,t_{i-1})$. On the other hand, involving $\mathbf{s}$ makes the problem more difficult, as we need to model the cross-sequence relationship between $\mathbf{s}$ and t_i. A recent trend in sequence generation is to formulate $\Pr(t_i|\mathbf{s},t_1,...,t_{i-1})$ in the **encoder-decoder** paradigm. There are two steps: an encoder is first used to represent $\mathbf{s}$ as some intermediate form (e.g., a vector), and a decoder is then used to model both the target-side words and the correlation between the encoder output and the target-side words. Putting these together, the output of the encoder-decoder model can be defined to be

$$\mathbf{y}_i = \text{Dec}(\text{Enc}(\mathbf{s}),t_1,...,t_{i-1}) \tag{1.102}$$

where $\text{Enc}(\cdot)$ is the encoder, and $\text{Dec}(\cdot)$ is the decoder. $\mathbf{y}_i$ is a distribution of the target-side word at position i, i.e., $y_i(t_i) = \Pr(t_i|\mathbf{s},t_1,...,t_{i-1})$. Chapter 5 will provide a detailed description of the encoder-decoder model.

1.5.5 Tree Generation

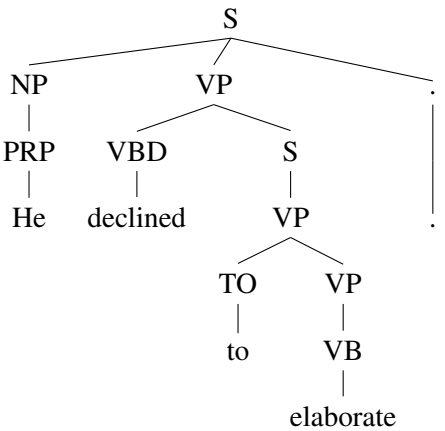
In NLP, trees are usually used to represent the structures or meanings of sequential data. For example, a **syntactic parser** analyzes a sentence to form a **syntax tree** or **parse tree**. More formally, given a sequence of words $\mathbf{s} = s_1...s_m$, the parsing problem can be defined as:

$$\hat{d} = \arg\max_{d \in D} \Pr(d|\mathbf{s}) \tag{1.103}$$

where d is a parse tree, and D is the set of all parse trees yielding $s_1...s_m$. Computing $\Pr(d|\mathbf{s})$ is challenging, as the modeling complexity increases exponentially when moving from sequences to trees. In **statistical parsing**, a solution is to model d as a derivation of syntactic rules. In this way, $\Pr(d|\mathbf{s})$ can be formulated as a product of rule probabilities. Figure 1.10 presents an example of parsing with context-free grammar (CFG) rules. Alternatively, $\Pr(d|\mathbf{s})$ can be modeled in an end-to-end manner. For example, some recent approaches perform parsing by defining a neural network over the parse tree. The probability of a sub-tree rooting at a node is computed by considering the interaction between this node and child nodes.

Another idea is to frame parsing as sequence generation. For example, one can linearize a parse tree and represent it as a sequence of words and syntactic labels, or transform the tree generation process as a sequence of actions. This allows the use of sequence-to-sequence techniques in addressing a sequence-to-tree problem.

Parse Tree:



CFG Rules:

r_1 : PRP → He
r_2 : VBD → declined
r_3 : TO → to
r_4 : VB → elaborate
r_5 : . → .
r_6 : NP → PRP
r_7 : VP → VB
r_8 : VP → TO VP
r_9 : S → VP
r_{10} : VP → VBD S
r_{11} : S → NP VP .

$$\begin{aligned}\mathrm{P}(d|\mathbf{s}) = &\mathrm{P}(\mathrm{PRP} \rightarrow \mathrm{He}) \cdot \mathrm{P}(\mathrm{VBD} \rightarrow \mathrm{declined}) \cdot \mathrm{P}(\mathrm{TO} \rightarrow \mathrm{to}) \cdot \mathrm{P}(\mathrm{VB} \rightarrow \mathrm{elaborate}) \cdot \\ &\mathrm{P}(. \rightarrow .) \cdot \mathrm{P}(\mathrm{NP} \rightarrow \mathrm{PRP}) \cdot \mathrm{P}(\mathrm{VP} \rightarrow \mathrm{VB}) \cdot \mathrm{P}(\mathrm{VP} \rightarrow \mathrm{TO\ VP}) \cdot \mathrm{P}(\mathrm{S} \rightarrow \mathrm{VP}) \cdot \\ &\mathrm{P}(\mathrm{VP} \rightarrow \mathrm{VBDS}) \cdot \mathrm{P}(\mathrm{S} \rightarrow \mathrm{NP\ VP\ .}) \\ = &\prod_{i=0}^{11} \mathrm{P}(r_i)\end{aligned}$$

Figure 1.10: An example parse tree and CFG rules. The sentence is from the training data of the CoNLL 2000 shared task. The parse tree is represented as a derivation of CFG rules. The probability of the parse tree is defined as the product of rule probabilities.

In linguistics and NLP, tree structures are in heavy use for syntactic analysis. In addition to parsing sentences, they are also attributed to words, phrases, and discourses. On the other hand, trees are not the only way of visualizing complex non-linear structures. A more general concept is a graph. While trees can be thought of as special graphs, there are cases that trees cannot handle [Fellbaum, 2005; Singhal, 2005; Banarescu et al., 2013]. For example, in the semantic representation of a sentence, we often need a graph to connect verbs and arguments. While learning general graphs is harder than parsing a sentence into a tree, we can reuse many of the methods developed in sequence and tree generation.

1.5.6 Relevance Modeling

Generally speaking, relevance is referred to as how well a thing relates to another. The concept of relevance is used in many different sub-fields of NLP and information science. For example, in information retrieval, relevance is used to describe to what extent a retrieved document meets the query. Additional uses of this concept can be found in question answering, dialogue systems, semantic analysis, and all other tasks that require a matching or retrieval process.

Let us consider a more general description. Assume that we have a query $query$ and a key key that represents something we intend to match with $query$. Then, we define the feature

vectors of $query$ and key as

$$\mathbf{q} = Q(query) \tag{1.104}$$
$$\mathbf{k} = K(key) \tag{1.105}$$

$Q(\cdot)$ and $K(\cdot)$ are feature extractors. The relevance between $query$ and key is given by the function:

$$r = f(\mathbf{q}, \mathbf{k}) \tag{1.106}$$

$f(\cdot)$ could be on one hand simply a distance measure if $\mathbf{q}$ and $\mathbf{k}$ are in the same vector space, and on the other hand a more complex model that performs some non-linear transformations. In fact, the way of defining relevance can be adopted in several different scenarios. Sometimes, relevance is also termed as similarity or correlation. A general example is how similar two objects are. Let x and y be two samples (say two words). The similarity of x and y is given by

$$r = f(g(x), g(y)) \tag{1.107}$$

where $g(\cdot)$ is a feature extractor, and $f(\cdot)$ is a **similarity function**. Learning both $g(\cdot)$ and $f(\cdot)$ is called **similarity learning**. In one setup of similarity learning, we fix $f(\cdot)$ and learn $g(\cdot)$ in a way that similar samples exhibit similar outputs of $g(\cdot)$. The learning of the feature extractor is not even required to work with the similarity function. For example, for obtaining the similarity between words, we can learn $g(\cdot)$ in a language model and use it together with various similarity functions. This puts the problem in a larger topic of machine learning: the learning of a sub-model is independent of the problem where we use it. Such an idea is widely adopted in pre-training, advancing the recent state-of-the-art on many NLP tasks.

In another setup of similarity learning, we can learn $f(\cdot)$ directly. This can be performed by either jointly learning $f(\cdot)$ and $g(\cdot)$, or learning $f(\cdot)$ on top of fixed $g(\cdot)$. The problem is also related to **metric learning**. Typically, metric learning is framed as a supervised problem [Kulis, 2013]. A desired similarity function could be learned with the supervision regarding some gold-standard similarity. However, in practice there is usually no such supervised information in NLP. In this case, one could take relative distance as some supervision. For example, the similarity function can be learned by optimizing a contrastive loss (see Section 1.3.4).

Measuring the similarity between objects plays an important role in many machine learning methods, such as clustering and nearest neighbor classification. On the side of NLP, it is useful for exploring the relationship between words, phrases, sentences, and documents, e.g., similarity is a way to examine how word vectors correspond to our understanding of word meanings [Mikolov et al., 2013; Pennington et al., 2014].

1.5.7 Linguistic Alignment

Linguistic alignment is a set of problems where we establish some correspondence between two sets of linguistic units. In NLP, the sequence-to-sequence and sequence-to-tree problems are typically linguistic alignment problems, as they both connect two linguistic units. However,

by convention, the term *alignment* is referred to as aligning multiple objects simultaneously[23].

As an example, consider the well-known **word alignment** task: we align the words of a sentence to the words of another sentence. We reuse the notation in Section 1.5.4 as both the sequence-to-sequence and word alignment tasks perform on a pair of sequences. Given a source-side word sequence $\mathbf{s} = s_1 ... s_m$ and a target-side word sequence $\mathbf{t} = t_1 ... t_n$, the word alignment between the two sequences is denoted as an $m \times n$ matrix $\mathbf{A}$. $A(i,j) = 1$ if there is an **alignment link** between s_i and t_j, and $A(i,j) = 0$ otherwise. The optimal alignment can be defined as:

$$\hat{\mathbf{A}} = \arg\max_{\mathbf{A}} \Pr(\mathbf{A}|\mathbf{s},\mathbf{t}) \tag{1.108}$$

where $\Pr(\mathbf{A}|\mathbf{s},\mathbf{t})$ is the word alignment probability. Like in other machine learning problems, we can model $\Pr(\mathbf{A}|\mathbf{s},\mathbf{t})$ in either a generative or discriminative manner (see Section 1.2.4). For example, in Brown et al. [1993]'s work, the word alignment model is factored into several generative steps, each accounting for some assumptions about the problem[24].

A 0-1 alignment matrix indicates a hard way of word alignment. A problem here is that the hard model may not describe well the highly ambiguous word alignments. We therefore can represent $\mathbf{A}$ as a real-valued matrix (call it a **soft word alignment matrix** or a **word alignment weight matrix**). Assume that the source-side words are represented as a sequence of feature vectors $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 & ... & \mathbf{x}_m \end{bmatrix}$. Likewise, the target-side words are represented as $\mathbf{y} = \begin{bmatrix} \mathbf{y}_1 & ... & \mathbf{y}_n \end{bmatrix}$. A soft word alignment model is given by:

$$\mathbf{A} = a(\mathbf{s},\mathbf{t}) \tag{1.109}$$

where $a(\cdot)$ is an word alignment function that computes the alignment weight $A(i,j)$ for each pair of $\mathbf{x}_i$ and $\mathbf{y}_j$. In fact, all the methods discussed in Section 1.5.6 are applicable to the design of $a(\cdot)$. This somehow links the modeling of word alignment with the modeling of similarity, and makes it possible to address different NLP problems by using the same machine learning approach.

Eq. (1.109) offers a very general way to discover the underlying connection over pairs of variables. In addition to aligning words in sequences, it is useful for aligning unordered objects. For example, in bilingual dictionary induction, we can learn such a weight matrix to estimate how strong a word in one language corresponds to a word in another language.

Here is another note on linguistic alignment models. While linguistic alignment could be thought of as an independent NLP task, it is commonly used in designing sub-models of some downstream systems. Many systems that model word-level relationships involve implicit representation of linguistic alignment. As a consequence, linguistic alignment is treated as some latent states, and is a by-product of these systems. For example, in the early

[23] The concept of alignment is wide-ranging. We use the term *linguistic alignment* here to differentiate it from the *alignment* of large language models discussed in subsequent chapters.

[24] More precisely, Brown et al. [1993] model $\Pr(\mathbf{A},\mathbf{s}|\mathbf{t})$ which is a surrogate of $\Pr(\mathbf{A}|\mathbf{s},\mathbf{t})$, as $\Pr(\mathbf{A}|\mathbf{s},\mathbf{t}) = \frac{\Pr(\mathbf{A},\mathbf{s}|\mathbf{t})}{\Pr(\mathbf{s}|\mathbf{t})} = \frac{\Pr(\mathbf{A},\mathbf{s}|\mathbf{t})}{\sum_{\mathbf{A}'}\Pr(\mathbf{A}',\mathbf{s}|\mathbf{t})}$

age of statistical machine translation, word alignment is a hidden variable used in modeling the mapping between sequences. The word alignment result can be easily induced from a machine translation model. More recently, neural sequence-to-sequence models — most notably attentional models [Bahdanau et al., 2014] — have attempted to do something similar to word alignment by computing attention weights among words.

1.5.8 Extraction

In NLP, *extraction* is not a kind of task but a kind of behavior that a system exhibits. Informally, it denotes a process of gathering, distilling structured information from some information sources. So, the term extraction generally appears together with other terms to form a specific task, such as **keyword extraction**, **event extraction**, and **relation extraction**. Many of these tasks can be categorized into an area — **information extraction**. Information extraction is perhaps the broadest topic in NLP. There is even no exhaustive list of information extraction tasks. According to Jurafsky and Martin [2008]'s book, it includes but is not limited to named entity recognition, reference resolution, relation extraction, event extraction, **template filling**, and so on.

However, since information extraction is a "miscellany" of many different problems, it cannot be formulated as a single machine learning problem. Fortunately, most of these problems can be framed as standard machine learning problems, such as classification and sequence labeling, and can be solved by using the off-the-shelf tools. In some cases, it may require a slight update of existing models for adaptation to a new task. For example, extracting a specific segment from text may require the system to produce a span that indicates the beginning and ending positions of the extracted segment (call it **span prediction**).

On the practical side, machine learning is not always necessary in extracting information from text. Many problems can be solved by using hand-crafted rules. An example is using regular expressions to identify locations and dates in text. In practice real-world systems are usually combinations of heuristic methods and automatic machine learning methods.

1.5.9 Others

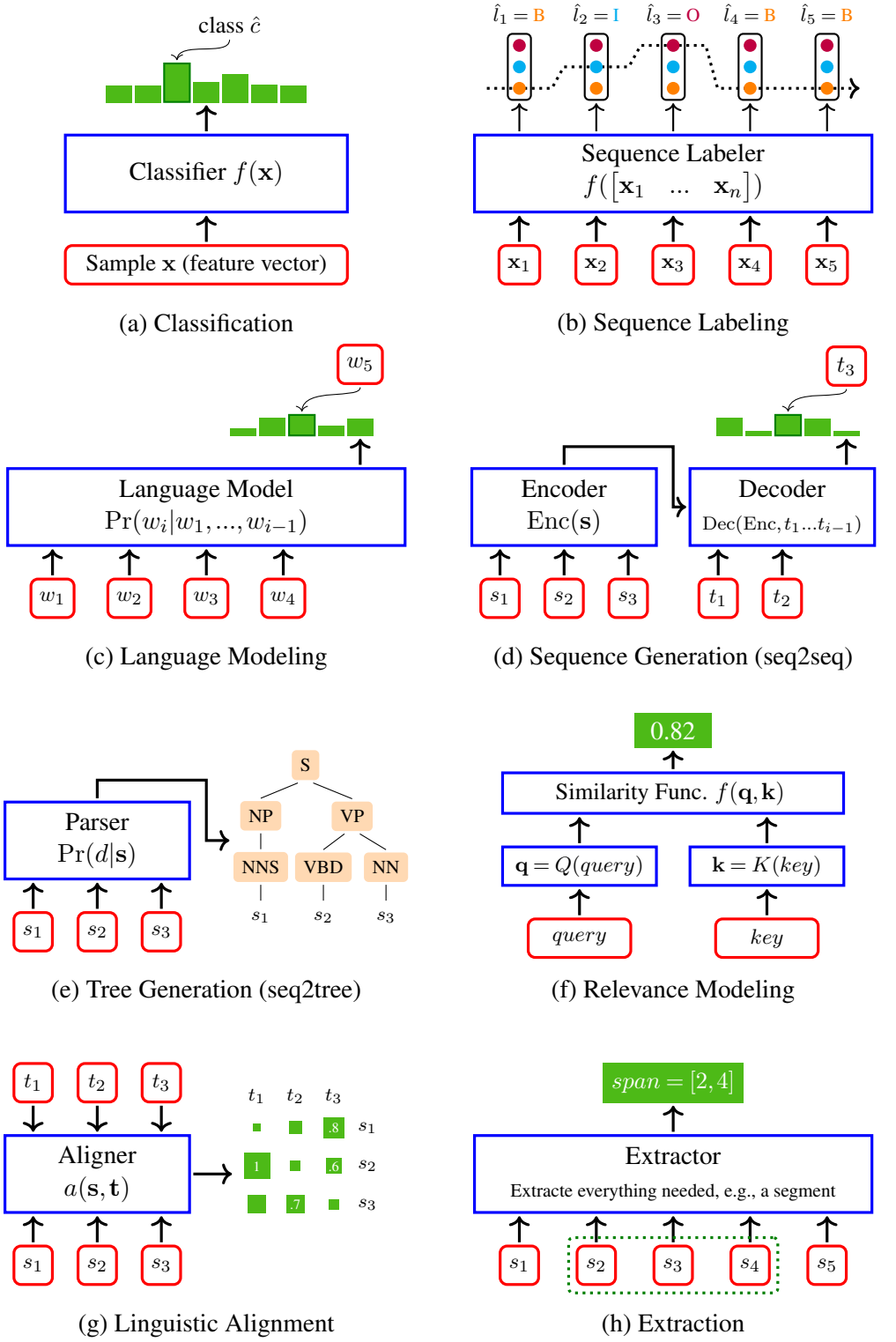
Figure 1.11 shows illustrative examples of the above NLP tasks. Note that many of the discussions here are still preliminary and incomplete. For example, we only talked about NLP problems in the supervised learning paradigm. Many unsupervised tasks are important for NLP research as well. For example, it is common to cluster unlabeled words or documents to ease the processing in downstream systems. Several methods are directly applicable to this task [Murphy, 2012]. A recent trend in NLP is that it is not necessary to set a strict boundary between the use of supervised learning and the use of unsupervised learning. In many cases, unsupervised methods help supervised tasks, and vice versa. A notable example is that we learn a pre-trained feature extractor on unlabeled data and build a supervised classifier on top of it. This leads to another trend running towards improving representation models (i.e., feature extractors) without the need of accessing downstream supervised tasks.

1.6 Summary

This chapter has given the basic ideas of machine learning and its applications to NLP problems. In particular, we have presented a simple text classification problem to get started with machine learning. Also, we have discussed several general problems on machine learning, e.g., types of machine learning methods, inductive biases, loss functions, overfitting and so on. They are followed by a discussion on model selection and assessment. In addition, we have described how model NLP problems are framed as machine learning problems.

However, machine learning is a huge research field. There are several interesting topics we left out. One topic that we said little about is reinforcement learning. In general, reinforcement learning is very powerful. It should be and has been considered as an approach to addressing NLP problems, e.g., training a sequence-to-sequence by using a risk-based loss function. A reinforcement learning textbook will offer the general ideas of reinforcement learning [Sutton and Barto, 2018]. Another topic we missed here is Bayesian learning [Gelman et al., 2020; McElreath, 2020; Downey, 2021]. It opens up a notable strand of research in statistical learning, and has been successfully used in NLP tasks. Moreover, there are many other topics that are specialized in certain aspects of machine learning and are of interest to NLP researchers and engineers. Some of them are efficient machine learning [Tay et al., 2020], multi-task learning [Zhang and Yang, 2021], few-shot/zero-shot learning [Wang et al., 2019; 2020], and so on.

A final point to wrap up this chapter. We skip the detailed discussion on certain machine learning models and algorithms, such as classification and regression models, because the reader interested in them can find several excellent, comprehensive introductions [Bishop, 2006; Hastie et al., 2009; Murphy, 2012; Mohri et al., 2018]. In the next chapter we will discuss a bit more about artificial neural networks which are the basis of deep learning and recent state-of-the-art NLP models.



(a) Classification

(b) Sequence Labeling

(c) Language Modeling

(d) Sequence Generation (seq2seq)

(e) Tree Generation (seq2tree)

(f) Relevance Modeling

(g) Linguistic Alignment

(h) Extraction

Figure 1.11: Natural language processing tasks from a machine learning perspective.

Bibliography

[Ash and Doléans-Dade, 1999] Robert B. Ash and Catherine A. Doléans-Dade. *Probability & Measure Theory*. Academic Press, 1999.

[Bahdanau et al., 2014] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[Bahl and Mercer, 1976] L. R. Bahl and R. L. Mercer. Part of speech assignment by a statistical decision algorithm. In *Proceedings of IEEE International Symposium on Information Theory*, pages 88–89, 1976.

[Banarescu et al., 2013] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, 2013.

[Bengio et al., 2003] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[Berg-Kirkpatrick et al., 2012] Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. An empirical investigation of statistical significance in NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 995–1005, 2012.

[Bishop, 2006] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Breiman, 1996] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[Brown et al., 1993] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993.

[Burnham and Anderson, 2002] Kenneth P. Burnham and David R. Anderson. *Model selection and multimodel inference: a practical information-theoretic approach*. Spinger, 2002.

[Chen and Goodman, 1999] Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13:359–394, 1999.

[Collobert and Weston, 2008] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning (ICML08)*, pages 160–167, 2008.

[Cortes and Vapnik, 1995] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mind*, Machine Learning:273–297, 1995.

[Devlin et al., 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the*

2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 4171–4186, 2019.

[Downey, 2021] Allen B. Downey. *Think Bayes: Bayesian Statistics in Python (2nd ed.)*. O'Reilly Media, 2021.

[Dror et al., 2020] Rotem Dror, Lotem Peled-Cohen, and Segev Shlomov. *Neural Network Methods for Natural Language Processing*. Morgan & Claypool Publishers, 2020.

[Fellbaum, 2005] Christiane Fellbaum. Wordnet and wordnets. In Keith Brown, editor, *Encyclopedia of Language and Linguistics (2nd ed.)*. Elsevier, 2005.

[Freedman et al., 2007] David Freedman, Robert Pisani, and Roger Purves. *Statistics (4th ed.)*. W. W. Norton & Company, 2007.

[Freedman, 2009] David A. Freedman. *Statistical Models: Theory and Practice (2nd ed.)*. Cambridge University Press, 2009.

[Gelman et al., 2020] Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian Data Analysis (2nd ed.)*. Chapman and Hall/CRC, 2020.

[Gildea and Jurafsky, 2002] Daniel Gildea and Daniel Jurafsky. Automatic labeling of semantic roles. *Computational Linguistics*, 28(3):245–288, 2002.

[Goodfellow et al., 2016] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[Hastie et al., 2009] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Spinger, 2009.

[Hurley, 2011] Patrick Hurley. *A Concise Introduction to Logic (11th ed.)*. Wadsworth Publishing, 2011.

[Jurafsky and Martin, 2008] Dan Jurafsky and James H. Martin. *Speech and Language Processing (2nd ed.)*. Prentice Hall, 2008.

[Konishi and Kitagawa, 2007] Sadanori Konishi and Genshiro Kitagawa. *Information Criteria and Statistical Modeling*. Spinger, 2007.

[Krebs et al., 2018] Alicia Krebs, Alessandro Lenci, and Denis Paperno. SemEval-2018 task 10: Capturing discriminative attributes. In *Proceedings of The 12th International Workshop on Semantic Evaluation*, pages 732–740, 2018.

[Kulis, 2013] Brian Kulis. Metric learning: A survey. *Foundations and Trends® in Machine Learning*, 5(4):287–364, 2013.

[Lafferty et al., 2001] John Lafferty, Andrew McCallum, and Fernando Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning 2001*, pages 282–289, 2001.

[McClave and Sincich, 2006] James T. McClave and Terry Sincich. *Statistics (10th ed.)*. Prentice Hall, 2006.

[McElreath, 2020] Richard McElreath. *Statistical Rethinking: A Bayesian Course with Examples in R and STAN (2nd ed.)*. Chapman and Hall/CRC, 2020.

[Mikolov et al., 2013] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of*

the 26th International Conference on Neural Information Processing Systems - Volume 2, pages 3111–3119, 2013.

[Mitchell, 1997] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Education, 1997.

[Mohri et al., 2018] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning (2nd ed.)*. MIT Press, 2018.

[Murphy, 2012] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[Och and Ney, 2002] Franz Josef Och and Hermann Ney. Discriminative training and maximum entropy models for statistical machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 295–302, 2002.

[Pang et al., 2002] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. Thumbs up? sentiment classification using machine learning techniques. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing (EMNLP 2002)*, pages 79–86, 2002.

[Papineni et al., 2002] Kishore Papineni, Salim Roukos, Todd Ward, and Wei jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[Pennington et al., 2014] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[Ramshaw and Marcus, 1995] Lance Ramshaw and Mitch Marcus. Text chunking using transformation-based learning. In *Third Workshop on Very Large Corpora*, 1995.

[Schapire, 1990] Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2): 197–227, 1990.

[Seni et al., 2010] Giovanni Seni, John Elder, and Robert Grossman. *Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions*. Morgan and Claypool Publishers, 2010.

[Shannon, 1948] Claude E. Shannon. A mathematical theory of communication. Report, Bell Labs, 1948.

[Singhal, 2005] Amit Singhal. Introducing the knowledge graph: things, not strings, 2005.

[Sutton and Barto, 2018] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (2nd ed.)*. The MIT Press, 2018.

[Tay et al., 2020] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *CoRR*, abs/2009.06732, 2020.

[Tjong Kim Sang, 2002] Erik F. Tjong Kim Sang. Introduction to the CoNLL-2002 shared task: Language-independent named entity recognition. In *Proceedings of COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002)*, 2002.

[Tjong Kim Sang and Buchholz, 2000] Erik F. Tjong Kim Sang and Sabine Buchholz. Introduction to the CoNLL-2000 shared task chunking. In *Proceedings of Fourth Conference on Computational Natural Language Learning and the Second Learning Language in Logic Workshop*, 2000.

[Vapnik and Chervonenkis, 1971] Vladimir Vapnik and Alexey Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–279, 1971.

[Wang et al., 2019] Wei Wang, Vincent Wenchen Zheng, Han Yu, and Chunyan Miao. A survey of zero-shot learning. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10:1–37, 2019.

[Wang et al., 2020] Yaqing Wang, Quanming Yao, James T. Kwok, and Lionel M. Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys*, 53(3):1–34, 2020.

[Wolpert, 1996] David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computatoin*, 8(7):1341–1390, 1996.

[Wolpert and Macready, 1997] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[Yarowsky, 1994] David Yarowsky. Decision lists for lexical ambiguity resolution: Application to accent restoration in Spanish and French. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 88–95, 1994.

[Zhang and Yang, 2021] Yu Zhang and Qiang Yang. A survey on multi-task learning. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2021.

[Zhou, 2012] Zhi-Hua Zhou. *Ensemble Methods: Foundations and Algorithms*. Chapman and Hall/CRC, 2012.