Ocean University of China

Fisheries College Of OUC

Probabilistic Machine Learning

Author:
Han Fangcheng

Student ID:
18060013010

A note for Probabilistic Machine Learning

Only through hard work can we become good steel

July 12, 2021

Contents

Part I

Introduction

1 What is machine learning?

A popular definition of **machine learning** or **ML**, is as follows:

A computer program is said to learn from experience E with respect to some class of tasks T, and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

2 Supervised learning

The most common form of ML is **supervised learning**. In this problem, the task T is to learn a mapping f from inputs $x \in \mathcal{X}$ to output $y \in \mathcal{Y}$. The experience E is given in the form of a set of N input-output pairs $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N$, known as the **training set**.

2.1 Classification

In **classification** problems, the output space is a set of C unorder and matually exclusive labels known as **classes**, $\mathcal{Y} = \{1, 2, \cdots, C\}$. The problem of predicting the class label given an input is also called **pattern recognition**.

Empirical risk minimization

The goal of supervised learning is to automatically come up with classification models, so as to reliably predict the labels for any given input. A common way to measure performance on this task is in the terms of the **misclassification rate** on the training set:

$$\mathcal{L} \triangleq \frac{1}{N} \sum_{n=1}^{N} \mathbb{I}(y_n \neq f(x_n; \theta)) \tag{2.1}$$

		Estimate		
		Setosa	Versicolor	Virginica
Truth	Setosa	0	1	1
	Versicolor	1	0	1
	Virginica	10	10	0

Table 2.1: Hypothetical asymmetric loss matrix for Iris classification

This assumes all errors are equal. However it may be the case that some errors are more costly than others. For example, suppose we are foraging in the wilderness and we find some Iris flowers. Furthermore, suppose that setosa and versicolor are tasty, but virginica is poisonous. In this case, we might use the asymmetric **loss function** $\ell(y, \hat{y})$ shown in Table 2.1. We can then define **empirical risk** to be the average loss of the predictor on the training set:

$$\mathcal{L}(\boldsymbol{\theta}) \triangleq \frac{1}{N} \sum_{n=1}^{N} \ell\left(y_n, f\left(\mathbf{x}_n; \boldsymbol{\theta}\right)\right) \tag{2.2}$$

We see that the misclassification rate is equal to the empirical risk when we use **zero-one loss** for comparing the true label with the prediction:

$$\ell_{01} = \mathbb{I}(y \neq \hat{y}) \tag{2.3}$$

One way to define the problem of **modle fitting** or **training** is to find a setting of the parameters that minimizes the empirical risk on the training set:

$$\hat{\theta} = \arg\min_{\theta} \mathcal{L}(\theta) = \arg\min_{\theta} \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(x_n; \theta)) \tag{2.4}$$

Uncertainty

In many cases, we will not be able to perfectly predict the exact output given the input, due to lack of knowledge of the input-output mapping(this is called the **epistemic uncertainty** or **model uncertainty**), and/or due

to intrinsic (irreducible) stochasticity in the mapping (this is called **aleatoric uncertainty** or **data uncertainty**).

We can capture our uncertainty using the following **conditional probability distribution**:

$$p(y = c|x; \theta) = f_c(x; \theta) \tag{2.5}$$

where $f : \mathcal{X} \to [0, 1]^C$ maps inputs to a probability distribution over the C possible output labels. Since $f_c(x; \theta)$ returns the probability of class label c, we require $0 \leq f_c \leq 1$ for each c, and $\sum_{c=1}^{C} f_c = 1$. To avoid this restriction, it is common to instead require the model to return unnormalized log-probabilities. We can then convert these to probabilities using the **softmax function**, which is defined as follows:

$$\mathcal{S}(a) \triangleq [\frac{e^{a_1}}{\sum_{c'=1}^{C} e^{a_{c'}}}, \cdots, \frac{e^{a_C}}{\sum_{c'=1}^{C} e^{a_{c'}}}] \tag{2.6}$$

This maps $\mathbb{R}^C$ to $[0, 1]^C$, and satisfies the constraints that $0 \leq \mathcal{S}(a) \leq 1$ and $\sum_{c=1}^{C} \mathcal{S}(a)_c = 1$. The input to softmax, $a = f(x; \theta)$, are called **logits**. We thus define the overall model as follows:

$$p(y = c|x; \theta) = \mathcal{S}_c(f(x; \theta)) \tag{2.7}$$

A common special case of this arises when f is an **affine function** of the form

$$f(x; \theta) = b + w^T x = b + w_1 x_1 + w_2 x_2 + \cdots + w_D x_D \tag{2.8}$$

where $\theta = (b, w)$ are the parameters of the model. This model is called **logistic regression**.

To reduce notational clutter, it is common to absorb the bias term b into the weights w by defining $\widetilde{w} = [b, w_1, \cdots, w_D]$ and defining $\widetilde{x} = [1, x_1, \cdots, x_D]$, so that

$$\widetilde{w}^T \widetilde{x} = b + w^T x \tag{2.9}$$

This converts the affine function into a **linear function**. We will usually assume that this has been done, so we can just write the prediction function as follows:

$$f(x; w) = w^T x \tag{2.10}$$

Maximum likelihood estimation

When fitting probabilistic models, it is common to use the negative log probability as our loss function:

$$\ell(y, f(x;\theta)) = -\log p(y|f(x;\theta)) \tag{2.11}$$

The average negative log probability of the training set is given by

$$\text{NLL}(\theta) = -\frac{1}{N}\sum_{n=1}^{N}\log p(y_n|f(x_n;\theta)) \tag{2.12}$$

This is called the **negative log likelihood**. If we minimize this, we can compute the **maximum likelihood estimate** or **MLE**:

$$\hat{\theta}_{\text{mle}} = \arg\min_{\theta}\text{NLL}(\theta) \tag{2.13}$$

2.2 Regression

Now suppose that we want to predict a real-valued quantity $y \in \mathbb{R}$ instead of a class label $y \in \{1, \cdots, C\}$; this is known as **Regression**.

For regression, the most common used cost function is **quadratic loss**, or ℓ_2 **loss**:

$$\ell_2(y, \hat{y}) = (y - \hat{y})^2 \tag{2.14}$$

This penalizes large **residuals** $y - \hat{y}$ more than small ones. The empirical risk when using quadratic loss is equal to the **mean squared error** or **MSE**:

$$\text{MSE}(\theta) = \frac{1}{N}\sum_{n=1}^{N}(y_n - f(x_n;\theta))^2 \tag{2.15}$$

In regression problems, it is common to assume the output distribution is a **Gaussian** or **normal**, which is defined as

$$\mathcal{N}(y|\mu,\sigma^2) =\triangleq \frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{1}{2\sigma^2}(y-\mu)^2} \tag{2.16}$$

In the context of regression, we can make the mean depend on the inputs by defining $\mu = f(x_n; \theta)$. We therefore get the following conditional probability distribution:

$$p(y|x;\theta) = \mathcal{N}(y|f(x;\theta), \sigma^2) \tag{2.17}$$

If we assume that the variance σ^2 is fixed, the corresponding negative log likelihood becaues

$$\begin{aligned} \text{NLL} &= -\sum_{n=1}^{N} \log[(\frac{1}{2\pi\sigma^2})^{\frac{1}{2}} \exp(-\frac{1}{2\sigma^2}(y_n - f(x_n;\theta))^2)] \\ &= \frac{N}{2\sigma^2}\,\text{MSE}(\theta) + \text{const} \end{aligned} \tag{2.18}$$

We see that the NLL is proportional to the MSE. Hence computing the maximum likelihood estimate of the parameters will result in minimizing the squared, which seems like a sensible approach to model fitting.

Linear regression

For regression, we can fit the data using a **linear regression** model of the form

$$f(x;\theta) = b + w^T x \tag{2.19}$$

and $\theta = (w, b)$ are all the parameters of the model. By adjusting θ, we can minimize the sum of squared errors, until we find the **least squares solution**

$$\hat{\theta} = \arg\min_{\theta} \text{MSE}(\theta) \tag{2.20}$$

Polynomial regression

When linear model can not fit the data well, we can improve the fit by using **polynomial regression** model of degree D. This has the form $f(x; w) = w^T\phi(x)$, where $\phi(x)$ is a feature vector derived from the input, which has the following form:

$$\phi(x) = [1, x, x^2, \cdots, x^D] \tag{2.21}$$

This is a simple example of **feature preprocessing**, also called **feature engineering**.

Deep neural networks

We can create much more powerful models by learning to do nonlinear **feature extraction** automatically. If we let $\phi(x)$ have its own set of parameters, say V, then the overall model has the form

$$f(x; w, V) = w^T \phi(x; V) \tag{2.22}$$

We can recursively decompose the feature extractor $\phi(x; V)$ into a composition of simpler function. The result model then becomes a stack of L nested functions:

$$f(x; \theta) = f_L(f_{L-1}(\cdots(f_1(x))\cdots)) \tag{2.23}$$

where $f_\ell(x) = f(x; \theta_\ell)$ is the function at layer ℓ. The final layer is linear and has the form $f_L(x) = w^T f_{1:L-1}(x)$, where $f_{1:L-1}(x)$ is the learned feature extractor. This is the key idea behind **deep neural networks** or **DNN**s, which includes common variants such as **convolutional neural networks**(CNNs) for images, and **recurrent neural networks**(RNNs) for sequences.

2.3 Overfitting and generalization

We can rewrite the empirical risk in the following equivalent way:

$$\mathcal{L}(\theta; \mathcal{D}_{\text{train}}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \ell(y, f(x; \theta)) \tag{2.24}$$

where $|\mathcal{D}_{\text{train}}|$ is the size of the training set $\mathcal{D}_{\text{train}}$. This formulation is useful is useful because it makes explicit which dataset the loss is being evaluated on.

A model that perfectly fits the training data, but which is too complex, is said to suffer from **overfitting**.

To detect if a model is overfitting, let us assume that we have access the true distribution $p^*(x, y)$ used to generate the training set. Then, instead of computing the empirical risk we compute the theoretical expected loss or **population**

risk

$$\mathcal{L}(\theta; p^*) = \mathbb{E}_{p^*(x,y)}[\ell(y, f(x;\theta))] \tag{2.25}$$

The difference $\mathcal{L}(\theta; p^*) - \mathcal{L}(\theta; \mathcal{D}_{\text{train}})$ is called the **generalization gap**. If a model has a large generalization gap, it is a sign that it is overfitting.

In practice we don't know p^*. However, we can partition the data we do have into two subsets, known as the training set and the **test set**. Then we can approximate the population risk using the **test risk**:

$$\mathcal{L}(\theta; \mathcal{D}_{\text{test}}) = \frac{1}{|\mathcal{D}_{\text{test}}|} \sum_{(x,y)\in\mathcal{D}_{\text{test}}} \ell(y, f(x;\theta)) \tag{2.26}$$

In practice, we need to partition the data into three sets, namely the training set, the test set and a **validation set**; the latter is used for model selection, and we just use the test set to estimate future performance (the population risk), the test set is not used for model fitting or model selection.

2.4 No free lunch theorem

Given the large variety of models in the literature, it is natural to wonder which one is best. Unfortunately, there is no single best model that works optimally for all kinds of problems — this is sometimes called the **no free lunch theorem**. The reason is that a set of assumptions (also called **inductive bias**) that works well in one domain may work poorly in another.

3 Unsupervised learning

An arguably much more interesting task is try to "make sense of" data, as opposed to just learning a mapping. That is, we just get observed "inputs" $\mathcal{D} = \{x_n : n = 1 : N\}$ without any corresponding "outputs" y_n. This is called **unsupervised learning**. From a probability perspective, we can view the task of unsupervised learning as fitting an unconditional model of the form $p(x)$.

3.1 Clustering

A simple example of unsupervised learning is the problem of finding **clusters** in data. The goal is to partition the input into regions that contain "similar" points.

3.2 Discovering latent "factors of variation"

We often reduce the dimensionality by projecting it to a lower dimensional subspace which captures the "essence" of the data when dealing with high-dimensional data. One approach to this problem is to assume that each observed high-dimensional output $x_n \in \mathbb{R}^D$ was generated by a set of hidden or unobserved low-dimensional **latent factors** $z_n \in \mathbb{R}^K$. We can represent the model diagrammatically as follows: $z_n \rightarrow x_n$, where the arrow represents causation. Since we don't know the latent factor z_n, we often assume a simple prior probability model for $p(x_n)$ such as a Gaussian, which says that each factor is random K-dimensional vector.

The simplest example is when we use a linear model, $p(x_n|z_n;\theta) = \mathcal{N}(Wz_n + \mu, \Sigma)$. The resulting model is called **factor analysis**(FA). It is similar to linear regression, except we only observe the output x_n, and not the inputs z_n. In the special case that $\Sigma = \sigma^2 I$, this reduce to a model called probabilistic **principal components analysis(PCA)**.

Of course, assumping a linear mapping from z_n to x_n is very restrictive. However, we can create nonlinear extensiona by defining $p(x_n|z_n;\theta) = \mathcal{N}(x_n|f(z_n;\theta), \sigma^2 I)$, where $f(z;\theta)$ is a nonlinear model, such as a deep neural network.

3.3 Self-supervised learning

A recently popular approach to unsupervised learning is known as **self-supervised learning**. In this approach, we create proxy supervised tasks from unlabeled data. For example, we might try to learn to predict a color image from a grayscale image, or to mask out words in a sentence and then try to predict

them given the surrounding context. The hope is that the resulting predictor $\hat{x}_1 = f(x_2; \theta)$, where x_2 is the observed input and $\hat{x}_1$ is the predicted output, will learn useful features from the data, that can then be used in standard, downstream supervised tasks.

3.4 Evaluating unsupervised learning

A common method for evaluation unsupervised models is to measure the probability assigned by the model to unseen test examples. We can do this by computing the negative log likelihood of the data:

$$\mathcal{L}(\theta; \mathcal{D}) = -\frac{1}{\mathcal{D}} \sum_{x \in \mathcal{D}} \log p(x|\theta) \tag{3.1}$$

This treats the problem of unsupervised learning as one of **density estimation**.

Unfortunately, density estimation is difficult, especially in high dimensions.An alternative evaluation metric is to use the learned unsupervised representation as features or input to a downstream supervised learning method. If the unsupervised method has discovered useful patterns, then it should be possible to use these patterns to perform supervised learning using much less labeled data than when working with the original features. That is, we can increase the **sample efficiency** of learning by first learning a good representation.

4 Reinforcement learning

In addition to supervised and unsupervised learning, there is a third kind of ML known as **reinforcement learning(RL)**. In this class of problems, the system or **agent** has to learn how to interact with its environment. This can be encoded by means of a **policy** $a = \pi(x)$, which specifies which action to take in response to each possible input x.

The difference from supervised learning (SL) is that the system is not told which action is the best one to take. Instead, the system just receives an occasional reward (or punishment) signal in response to the actions that it takes.

This is like **learning with a critic**, who gives an occasional thumbs up or thumbs down, as opposed to **learning with a teacher**, who tells you what to do at each step.

5 data

5.1 Preprocessing discrete input data

Sometimes our input may have discrete input features, such as categorical variables like race and gender, or words from some vocabulary. In the sections below, we discuss some ways to preprocess such data to convert it to vector form.

One-hot encoding

The standard way to preprocess such categorical variables is to use a **one-hot encoding**, also called a **dummy encoding**. If a variable x has K values, we will denote its dummy encoding as follows: one-hot$(x) = [\mathbb{I}(x = 1), \cdots, \mathbb{I}(x = K)]$.

Feature crosses

A linear model using a dummy encoding for each categorical variable can capture the **main effects** of each variable, but cannot capture **interaction effects** between them. We can fix this by computing explicit **feature crosses**. For example, we can define a new composite feature with $m \times n$ possible values, to capture the interaction of variable one which has m categories and variable two which has n categories.

5.2 Preprocessing text data

Bag of words model

A simple approach to dealing with variable-length text documents is to interpret them as a **bag of words**, in which we ignore word order. To convert

this to a vector from a fixed input space, we first map each word to a **token** from some vocabulary.

Let x_{nt} be the token at location t in the nth document. If there are D unique tokens in the vocabulary, the we can represent the nth document as a D-dimensional vector $\widetilde{x}_n$, where $\widetilde{x}_{nv}$ is the number of times that word v occurs in document n:

$$\widetilde{x}_{nv} = \sum_{t=1}^{T} \mathbb{I}(x_{nt} = v) \tag{5.1}$$

where T is the length of document n. We can now interpret documents as vectors in $\mathbb{R}^D$. This is called the **vector space model** of text.

We traditionally store input data in an $N \times D$ design matrix denoted by X, where D is the number of features. In the context of vector space models, it is more common to represent the input data as a $D \times N$ term frequency matrix, where TF_{ij} is the frequency of term i in document j.

TF-IDF

To reduce the impact of words that occur manys times in general, we compute a quantity called the **inverse document frequence**, define as follows: $IDF_i \triangleq \log \frac{N}{1+DF_i}$, where DF_i is the number of documents with term i. We can combine these transformations to compute the **TF-IDF** matrix as follows:

$$TFIDF_{ij} = \log(TF_{ij} + 1) \times IDF_i \tag{5.2}$$

Word embeddings

Although the TF-IDF transformation improves the vector representation of words by placing more weight on "informative" words and less on "uninformative" words, it does not overcome the fundamental issue that semantically similar words, such as "man" and "woman", may be further apart (in vector space) than semantically dissimilar words, such as "man" and "banana".

The standard way to solve this problem is to use **word embeddings**, in which we map each spare one-hot vector, $x_{nt} \in \{0, 1\}^V$, to a lower-dimensional

dense vector, $e_{nt} \in \mathbb{R}^K$ using $e_{nt} = Ex_{nt}$, where E is learned such that semantically similar words are palced close by.

Once we have an embedding matrix, we can represent a variable-length vector by summing (or averaging) the embeddings:

$$\bar{e}_n = \sum_{t=1}^{T} e_{nt} = E\widetilde{x}_n \tag{5.3}$$

where $\widetilde{x}_n$ is the bag of words representation. We can then use this inside of a logistic regression classifier. The overall model has the form

$$p(y = c|x_n, \theta) = \mathcal{S}_c(WE\widetilde{x}_n) \tag{5.4}$$

We often use a **pre-trained word embedding** matrix E, in which case the model is linear in W, which simplifies parameter estimation.

Handling missing data

Sometimes we may have missing data, in which parts of the input x or output y may be unknown.

To model this, let M be an $N \times D$ matrix of binary variables, where $M_{nd} = 1$ if feature d in example n is missing, and $M_{nd} = 0$ otherwise. Let X_v be the visible parts of the input feature matrix, corresponding to $M_{nd} = 0$, and X_h be the missing parts, corresponding to $M_{nd} = 1$. Let Y be the output label matrix, which we assume is fully observed.

We divide missing values into three types:

a. If we assume $p(M|X_v, X_h, Y) = p(M)$, we say the data is missing completely at random or MCAR, since the missingness does not depend on the hidden or observed features.

b. If we assume $p(M|X_v, X_h, Y) = p(M|X_v, Y)$, we say the data is missing at random or MAR, since the missingness does not depend on the hidden features, but may depend on the visible features.

c. If neither of these assumptions hold, we say the data is not missing at random or NMAR.

Part II

Foundations

6 Probability: univariate models