22

SUPERVISED LEARNING PRINCIPLES

22.1 The supervised learning problem

- $\mathcal{X}$: input space, the set of all possible examples or instances. An instance or an example is usually represented by a feature vector $x = (x_1, ..., x_n) \in \mathcal{X} = \mathbb{R}^n$
- $\mathcal{Y}$: the set of all possible labels or target values. For example $\mathcal{Y} = \{-1, 1\}$ in binary classification.
- $\mathcal{D}$: the distribution of examples defined on $\mathcal{X}$ that are used to draw samples.
- D: a training data set $D = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(n)}, y^{(n)})\}$ or $D = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$ drawn from $\mathcal{D}$.
- $\mathcal{H}$: hypothesis space or hypothesis set

22.1.1 Concepts

Consider a problem of predicting house prices given a set of house characteristics like the year built, type, parking, neighborhood, and utilities. In supervised learning, we are presented with a number of examples of house prices and their characteristics, and our goal, from mathematical modeling perspective, is to seek an optimal function or mapping $y = f(x)$ that captures the relationship between house characteristics, i.e., the x, and their prices y.

In another example, in a loan lending problem, we need to predict a customer's default behavior, in terms of default probability, based on the customer's characteristics such as credit score, age, occupation, past repayment behavior, etc. Before the prediction, we are given past data examples showing customers' characteristics and their historical default behaviors.

These two examples outline the key aspects of supervised learning problems: we aim to make predictions given a set of features or characteristics; we are given observed examples to derive the relationship between the outcomes and the features.

We denote the **feature** by x taking value in a **feature space** $\mathcal{X}$ and the output or label by y taking value in an **output space** $\mathcal{Y}$. The observed examples are called the **training data set** D consists of independent identically distributed samples given by

$$D = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}.$$

The two most common supervised learning tasks are **classification** and **regression**:

- **Classification**: In classification tasks, our goal is to learn a mapping $y = f(x)$ from input x to output $y \in \{1, 2, ..., C\}$, which encodes categories. In some variant of classification, we are learning a probability distribution over categories given x, in this case, $f(x) = P(y|x)$.

When $C = 2$, the classification is called **binary classification**; when $C > 2$, the classification is called **multi-label classification**.

- **Regression**: In regression tasks, the goal is to predict a numerical value given some inputs. To solve this task, the learning algorithm is asked to output a function $f : \mathbb{R}^n \to \mathbb{R}$. This type of task is similar to classification, except that the format of output is different. An example regression task is the prediction of the expected claim amount that an insured person will make (used to set insurance premiums), or the prediction of future prices of securities.

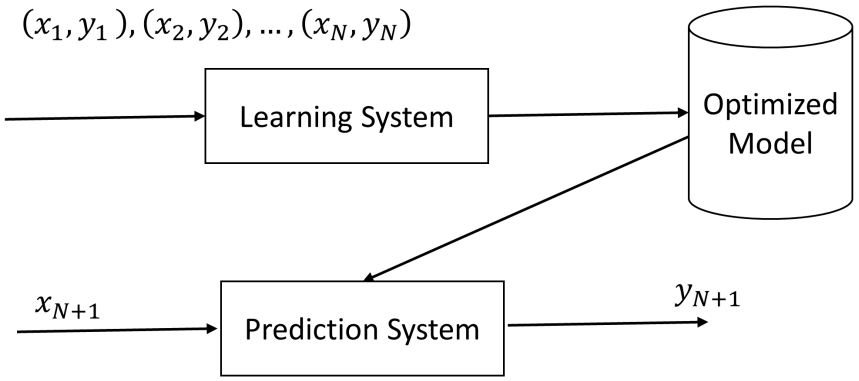

Figure 22.1.1: Scheme of a supervised learning task. Training samples are fed into a learning system to obtain an optimized model, which will be further used in a prediction system for regression and classification tasks.

The input feature x can take a wide variety of forms, including nominal, ordinal, binary, count, time, or interval, categorical, integer, continuous number, image, video, time series, audio, text, etc. For example

	binary	categorical	integer	continuous	binary target
ID	Home owner	Marital Status	Age	Annual Income	Defaulted Borrower
1	Yes	Single	20	75K	Yes
2	No	Married	25	100K	No
3	No	Divorced	35	170K	No
4	Yes	Single	24	90K	Yes
…	…	…	…	…	…
10	No	Married	28	120K	No

Figure 22.1.2: Input feature data type examples.

22.1.2 Framework

Supervised learning is primarily about seeking function mapping. To set up a formal framework, we first introduce the concept of **hypothesis**, which is equivalent to the concept of function mapping, and **hypothesis space**.

Definition 22.1.1 (hypothesis, hypothesis space). *Let $\mathcal{X}$ and $\mathcal{Y}$ be the feature space and output space, respectively.*

- *A **hypothesis** f is a mapping from $\mathcal{X}$ to $\mathcal{Y}$.*
- *The set of all hypothesis in a learning task form the **hypothesis space** $\mathcal{H}$. $\mathcal{H}$ is usually represented by a family of functions parameterized by some parameters θ.*

Example 22.1.1 (hypothesis space of linear functions). Consider a hypothesis space $\mathcal{H}$ consisting of all linear functions with respect to the input $x \in \mathbb{R}^n$. $\mathcal{H}$ can be parameterized by

$$\mathcal{H} = \{f(x;\theta) = w^T x + b | w \in \mathbb{R}^n, b \in \mathbb{R}\}.$$

We further adopt a probabilistic description on the training data set. We assume feature X and output Y follow joint distribution $P(X,Y)$ or $\mathcal{D}$. The training data set D can be viewed as IID samples from $\mathcal{D}$.

We measure the prediction performance via a **loss function** $L(y, f(x))$, which maps to larger values as the prediction $f(x)$ deviates from the ground truth y. Particularly, in our probabilistic framework, we measure prediction performance via expected loss function defined as

Definition 22.1.2 (expected loss/error). *Given a hypothesis f, the expected loss of f with respect to the sample distribution $\mathcal{D}$ is given by*

$$R_{\text{exp}}(f) = E_{\mathcal{D}}[L(Y, f(X))] = \int_{\mathcal{X}\times\mathcal{Y}} L(y, f(x))P(x,y)dxdy.$$

Now we are in a position to state the goal of supervised learning:

The goal of supervised learning is to seek a hypothesis that minimizes prediction error, which is given by

$$\min_{f\in\mathcal{H}} R_{\text{exp}}(f).$$

Unfortunately, the joint distribution $P(X, Y)$ is often unknown. We instead seek the optimal hypothesis by optimizing over the **empirical loss** on the training data set, which is given by

Definition 22.1.3 (empirical loss/error). *Given a hypothesis f, the **empirical loss** of f with respect to a training data set D is given by*

$$R_{\text{emp}}(f) = \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, f\left(x_i\right)\right).$$

Although as $N \to \infty$, $R_{\text{emp}} \to R_{\text{exp}}$, in practice, we only have finite N, and consequently, the optimal hypothesis based on the finite-sized training set is not optimal with respect to the full population. Such non-optimality, or generalization error, can be measured on a test data set that is not used in the optimization progress. We thus define the **empirical test error** as

Definition 22.1.4 (empirical test loss). *Given a hypothesis f, the **empirical test loss** of f with respect to an unseen test data set D^T is given by*

$$e_{\text{test}} = \frac{1}{N^T} \sum_{i=1}^{N^T} L\left(y_i, \hat{f}\left(x_i\right)\right).$$

22.2 Variance bias trade-off

22.2.1 Underfitting and Overfitting

The most fundamental compromise we make in the presence of **finite training data** is to introduce restriction on the hypothesis space, that is, to directly limit the complexity of hypothesis or model we are seeking.

The necessity of reducing model complexity can be understood by the following regression problem [Figure 22.2.1]. We are given a set of training data that is generated from a ground truth model given by

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X_3 + \epsilon.$$

Consider models of different complexity, from linear model, to polynomial models of different degree freedoms, to be optimized to fit the observed data. clearly, a linear model is too simple to yield an adequate fit to the training data set, which we generally refer to as **underfitting**. On the other hand, higher order polynomials ($d = 7$) fit the training data better than polynomial of degree 3. but it will have larger expected error with respect to the true sample distribution $\mathcal{D}$ or with respect ot unseen testing data; we refer to the phenomenon that a model fits training data well but not on unseen data as **overfitting**.

Thanks to the availability of a wealth of powerful statistical learning methods (e.g., decision tree, ensemble learning, neural networks), in practice, models usually suffer more from overfitting than underfitting. We have following summary on overfitting.

Definition 22.2.1 (overfitting). *We say a hypothesis $f \in \mathcal{H}$ **over-fits training data (overfitting)** if there is an alternative hypothesis $f' \in \mathcal{H}$ such that h performs better with respect to training data, that is,*

$$R_{\text{emp}}(h) < R_{\text{emp}}(h')$$

but h performs worse with respect to unseen testing data, that is

$$e_{\text{test}}(h) > e_{\text{test}}(h');$$

or more generally in terms of generalization performance,

$$R_{\text{exp}}(h) > R_{\text{exp}}(h').$$

Empirically, with a given training data set, the relationship among training error, testing error and model complexity can be summarized in Figure 22.2.2. At the limit of

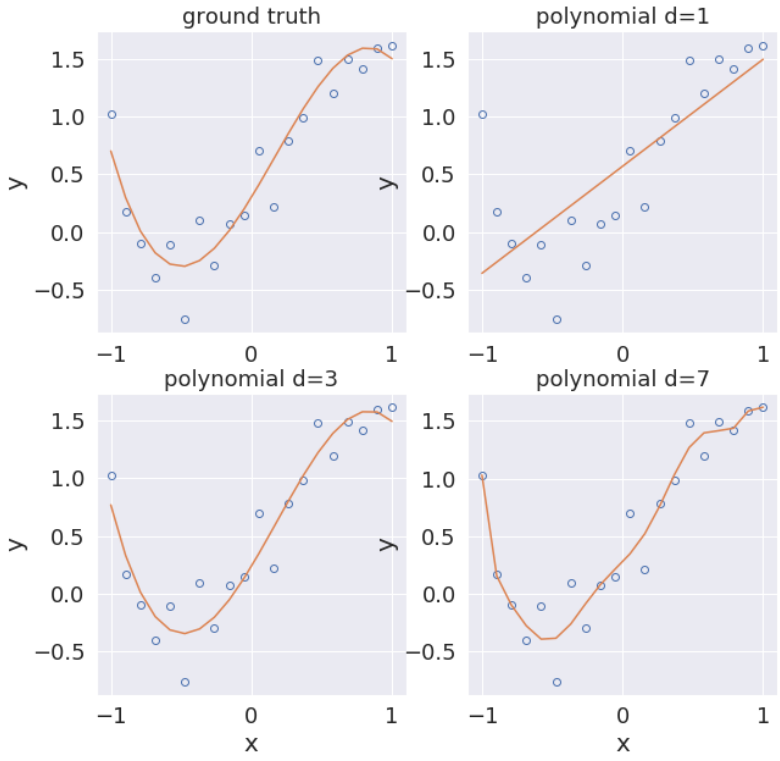

Figure 22.2.1: A simple regression problem illustrating underfitting and overfitting. Solid lines are models, and points are samples.

simple models, underfitting occurs such that both training and testing error are large; as we increase the model complexity, training error continues to drop but testing error will first drop and then increase. The turning point of the testing error is where overfitting occurs.

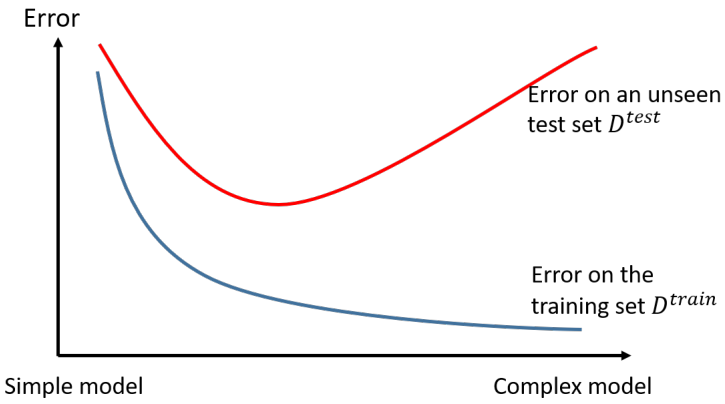

Figure 22.2.2: The commonly observed phenomenon of overfitting and underfitting in machine learning.

22.2.2 Variance and bias trade-off

The cause of underfitting and overfitting and the characterizations of training and testing error can be analyzed in-depth via variance and bias. Such characterization offer insights on the roots of underfitting and overfitting, and further lay the foundation for seeking better hypotheses. We first introduce the variance-bias decomposition [Lemma 22.2.1], which states that the testing error can be attributed to the variance component and the error component.

Lemma 22.2.1 (variance-bias decomposition for prediction error). *Suppose examples* (x, y) *are generated via ground truth model*

$$y = h(x) + \epsilon$$

where ϵ *is* ***an independent random variable with zero mean****. Let* $\hat{f}$ *be our proposed model with model parameters needs to be estimated from training examples* $(x, y) \sim \mathcal{D}$*. Let us measure our learning performance via* $E_D[(y - \hat{f}(x|D))^2]$*, where* $\hat{f}(x|D)$ *denotes that our model is estimated from finite-sized sample* D*, which is randomly drawn from sample distribution* $\mathcal{D}$*.*

We have

$$E_D[(y - \hat{f}(x|D))^2] = Var_D[\hat{f}(x|D)] + [Bias(\hat{f}(x|D))]^2 + Var[\epsilon]$$

where $Bias(\hat{f}(x|\mathcal{D})) = E_D[E_D[\hat{f}(x|D)] - h(x)]^2$.

Proof. (1) Use the following decomposition:

$$\begin{aligned}
&E_D[(y - \hat{f}(x|D))^2] \\
=&E_D[(y - h(x) + h(x) - \hat{f}(x|D))^2] \\
=&E_D[(y - h(x))^2] + 2E_D[(h(x) - \hat{f}(x|D))(y - h(x))] + E_D[(h(x) - \hat{f}(x|D))^2] \\
=&E_D[(y - h(x))^2] + 2E_D[(h(x) - \hat{f}(x|D))\epsilon] + E_D[(h(x) - \hat{f}(x|D))^2] \\
=&\underbrace{E_D[(y - h(x))^2]}_{\text{noise term}} + 0 + \underbrace{E_D[(h(x) - \hat{f}(x|D))^2]}_{\text{model estimation error}} \\
=&E_D[(y - h(x))^2] + E_D[(h(x) - E_D[\hat{f}(x|D)] + E_D[\hat{f}(x|D)] - \hat{f}(x|D))^2] \\
=&E_D[(y - h(x))^2] + E_D[(h(x) - E_D[\hat{f}(x|D)])^2] + E_D[(E_D[\hat{f}(x|D)] - \hat{f}(x|D))^2] \\
=&\underbrace{E_D[(y - h(x))^2]}_{\text{noise term}} + \underbrace{E_D[(h(x) - E_D[\hat{f}(x|D)])]^2}_{\text{bias}^2} + \underbrace{Var_D[\hat{f}(x|D)]}_{\text{variance}}
\end{aligned}$$

where the noise term is without our control and the model estimation error is what we want to minimize. Further, we decompose model estimation error into variance and bias terms using similar derivation techniques as we derive the variance-bias decomposition for a statistical estimator [Theorem 13.1.1]. □

Remark 22.2.1 (interpretation of variance term).

- $Var[\hat{f}(x|D)]$ represents the amount by which $\hat{f}$ would change if we estimate the model parameter using a different set of training examples of fixed size n.
- If this term is large, then we are likely to have large test error.
- When increasing the number training examples, this term will be reduced.
- Given a fixed number of training examples, a large capacity model (model with more parameters, such as neural networks) will tend to have large variance since model parameter could be estimated poorly.

Remark 22.2.2 (interpretation of bias term).

- $E_D[\hat{f}]$ is the resulting trained model when given infinitely number of training examples. Therefore, $(E_D[\hat{f}] - h)^2$ reflect the fundamental error associated with model proposal/assumption. For example, if h is a nonlinear model but $\hat{f}$ is a linear model, then no matter how hard we train, we cannot eliminate this error.

- A less flexible model(such as linear model) will tend to have large bias. When using "universal approximation" models, like neural network and decision tree, the bias term could be reduced, however, at the risk of increasing Variance term.

Remark 22.2.3 (interpretation of noise term).

- Even we have perfect model, that is $Bias[\hat{f}] = 0$, and perfect estimation of the model $Var[\hat{f}] = 0$, we will cannot reduce $E[(y_0 - \hat{f}(x_0))^2]$ to zero because the y_0 always contains a noise that cannot be predicted.

The training and testing error are random variables as they are obtained from a finite-sized sample drawn from a population. A numerical method to estimate error is given as follows.

Methodology 22.2.1 (estimating generalization error). *Fix the test example (x_0, y_0), we can do the following experiment:*

- *Draw a training examples D_1 of size N, $D_1 = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$.*
- *Train linear regression $\hat{f}(x|D_1)$ using D_1.*
- *Calculate the squared error*

$$err(D_1) = (\hat{f}(x_0|D_i) - y_0)^2.$$

- *Repeat the above procedures by drawing different training examples $D_i, i = 2, ..., M$, and get the mean squared error via*

$$err = \frac{1}{M}\sum_{i=1}^{M} err(D_i).$$

Finally, we provide a graphic summary on the connection of variance and bias to testing and training errors. Consider models with different variance and bias characteristics. As we increase the training data size, the testing error and training error usually behave as Figure 22.2.3 depicts. Altogether, we can summarize the error convergence as follows

Remark 22.2.4 (error convergence properties). As showed in Figure 22.2.3, we have

- In general, for a given model, increasing training data will increase training error as more training data will exceed the model's capacity; on the other hand, increasing training data will decrease testing error because the trained model is closer to the ground truth model. As training data size goes to infinitely large, training error and testing error will converge to the same error.
- For a biased model, the limiting error will be larger than that of an unbiased model.
- For a high variance model, its convergence will be slower than a low variance model.

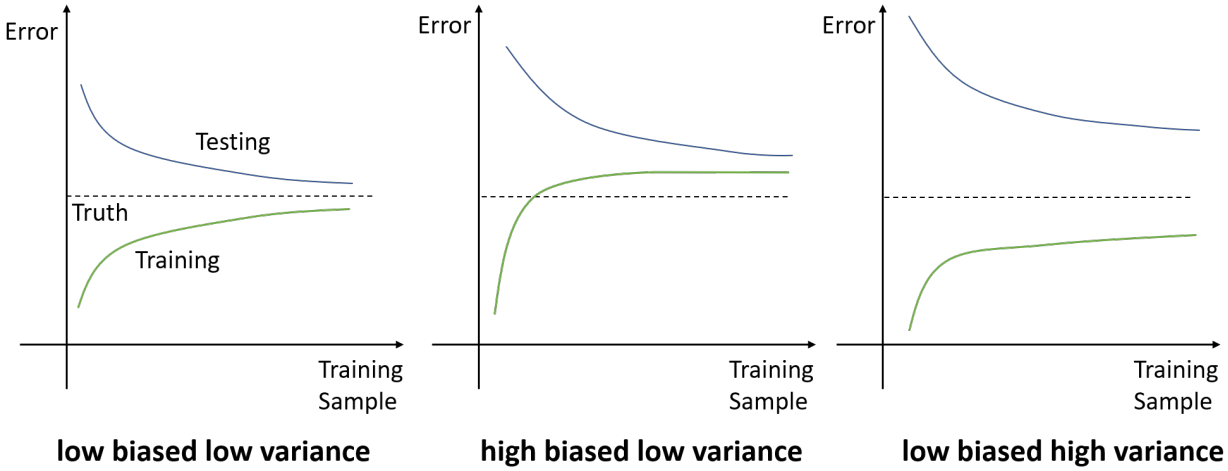

Figure 22.2.3: The performance of different types of models as training proceeds.

22.2.3 Examples

22.2.3.1 *Linear regression*

We use following Lemma regarding linear regression model to illustrate the variance and bias. Note that by increasing sample size, variance can be reduced.

Lemma 22.2.2 (linear regression variance-bias analysis). *Assume the true model is* $h(x) = x^T\beta$ *and the prediction model is*

$$\hat{f}(x|D) = x^T\hat{\beta},$$

where $\hat{\beta} = (X^TX)^{-1}X^TY$. *It follows that*

- *Unbiasedness:*
$$E_D[\hat{f}(x)|D] = h(x) = x^T\beta.$$
- *Variance:*
$$Var \triangleq E_D[(\hat{f}(x|D) - E_D[\hat{f}(x)|D])^2] = \sigma^2 x^T(X^TX)^{-1}x.$$
- *If inputs* X *are standardized and* x *is a random vector with distribution* $MN(0, I_{p\times p})$, *then*
$$Var \approx \sigma^2\frac{p}{n}.$$

Proof. (1)(2) see Theorem 15.1.1. (3)

$$\begin{aligned} E_x[Var] &= E_x[\sigma^2 x^T (X^T X)^{-1} x] \\ &= \sigma^2 E_x[Tr(x^T (X^T X)^{-1} x)] \\ &= \sigma^2 E_x[Tr((X^T X)^{-1} x x^T)] \\ &= \sigma^2 E_x[Tr(x x^T (X^T X)^{-1})] \\ &= \sigma^2 Tr(E_x[x x^T](X^T X)^{-1}) \\ &= \sigma^2 Tr(I_{p\times p}) \end{aligned}$$

□

22.2.4 No free lunch theorem

The goal of supervised learning is to seek optimal hypothesis given finite-sized observation via algorithms. Does there exist one algorithm always perform the best - in terms of seeking optimal hypothesis - for *all* problems? or even simpler, can one algorithm always be better than the other algorithm for *all* problems?

The answer is provided by **No free lunch theorem**[1]. Let's go through a relatively simple proof [2, p. 9], and then we will discuss its implications.

Let $\mathcal{X}$ be a finite-size discrete feature space, let the output space be $\{0, 1\}$, and let $\mathcal{H}$ be all the possible functions mapping from $\mathcal{X}$ to $\{0, 1\}$. Clearly, there are totally $2^{\{|\mathcal{X}|\}}$ different mapping in $\mathcal{H}$. Denote L_a as the algorithm we consider.

Let $P(h|X, \mathfrak{L}_a)$ be the probability of generating h based on algorithm $\mathfrak{L}_a$ and training data D.

The average out-of-sample error given algorithm $\mathfrak{L}_a$ and training data D and the true hypothesis f is given by

$$E_{ote}\left(\mathfrak{L}_a|X, f\right) = \sum_h \sum_{x\in\mathcal{X}-X} P(x) I(h(\boldsymbol{x}) \neq f(x)) P\left(h|X, \mathfrak{L}_a\right)$$

Because we are considering the performance for *all* possible problems, we should let f be any mapping that is uniformly drawn from $\mathcal{H}$. Then the average error of algorithm $\mathfrak{L}_a$ given training data D is then

$$\begin{aligned}
\sum_f E_{ote}\left(\mathfrak{L}_a|X,f\right) &= \sum_f \sum_h \sum_{x\in\mathcal{X}-X} P(x)\mathbb{I}(h(x)\neq f(x))P\left(h|X,\Sigma_a\right) \\
&= \sum_{x\in\mathcal{X}-X} P(x)\sum_h P\left(h|X,\Sigma_a\right)\sum_f \mathbb{I}(h(x)\neq f(x)) \\
&= \sum_{x\in\mathcal{X}-X} P(x)\sum_h P\left(h|X,\mathcal{L}_a\right)\frac{1}{2}2^{|\mathcal{X}|} \\
&= \frac{1}{2}2^{|\mathcal{X}|}\sum_{x\in\mathcal{X}-X} P(x)\sum_h P\left(h|X,\Sigma_a\right) \\
&= 2^{|\mathcal{X}|-1}\sum_{x\in\mathcal{X}-X} P(x)\cdot 1
\end{aligned}$$

where we informally use $\mathcal{X}-D$ to represent out-of-samples.

Surprisingly, **the average error of an algorithm given finite-sized training samples across all possible true hypothesis is independent of the algorithm.** This result has multiple important implications.

Remark 22.2.5 (interpretation).

- The most critical assumption is the ground truth f can be anything uniformly drawn from $\mathcal{H}$. In reality, when we face a class of problems (such as computer vision or natural language processing), f is not uniformly distributed on $\mathcal{H}$. Consider the housing price prediction problem, a larger house tends to be more expensive (instead of being at any price possible).
- The most important implication of No free lunch theorem is there exists no algorithms that excel for all possible problems. The performance and quality of an algorithm should only be assessed with respect to a certain type of problems.

22.3 Model loss and evaluation

22.3.1 Common loss functions

Supervised learning relies heavily on mathematical optimization method to find model parameters by optimizing over loss functions. Here we briefly review different loss functions used in regression and classification problems.

Definition 22.3.1 (regression loss function). *Denote $\hat{y}$ as the estimated target output, y the corresponding (correct) target output, $\hat{y}_i$ the predicted value of the i-th sample, and y_i the corresponding true value*

- *(mean squared error) The **mean squared error (MSE)** estimate over N samples is defined as*

$$L_{MSE}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2.$$

- *(mean absolute error) The **mean absolute error (MAE)** estimate over N samples is defined as*

$$L_{MAE}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}\left|y_i - \hat{y}_i\right|.$$

- *(median absolute error) The **median absolute error (MedSE)** estimate over N samples is defined as*

$$L_{MedAE}(y, \hat{y}) = median(\left|y_1 - \hat{y}_1\right|, ..., \left|y_N - \hat{y}_N\right|,)^2.$$

- *(Log-Cosh loss) The **Log-Cosh loss** estimate over N samples is defined as*

$$L_{LC}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}\log(\cosh(y_i - \hat{y}_i)).$$

- *(Huber loss) The **Huber loss**, with parameter δ, estimate over N samples is defined as*

$$L_{Huber}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}H_\delta(y_i - \hat{y}_i),$$

where

$$L_\delta = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & if\ |(y - \hat{y})| < \delta \\ \delta\left((y - \hat{y}) - \frac{1}{2}\delta\right) & otherwise \end{cases}$$

Some loss functions are showed in Figure 22.3.1.

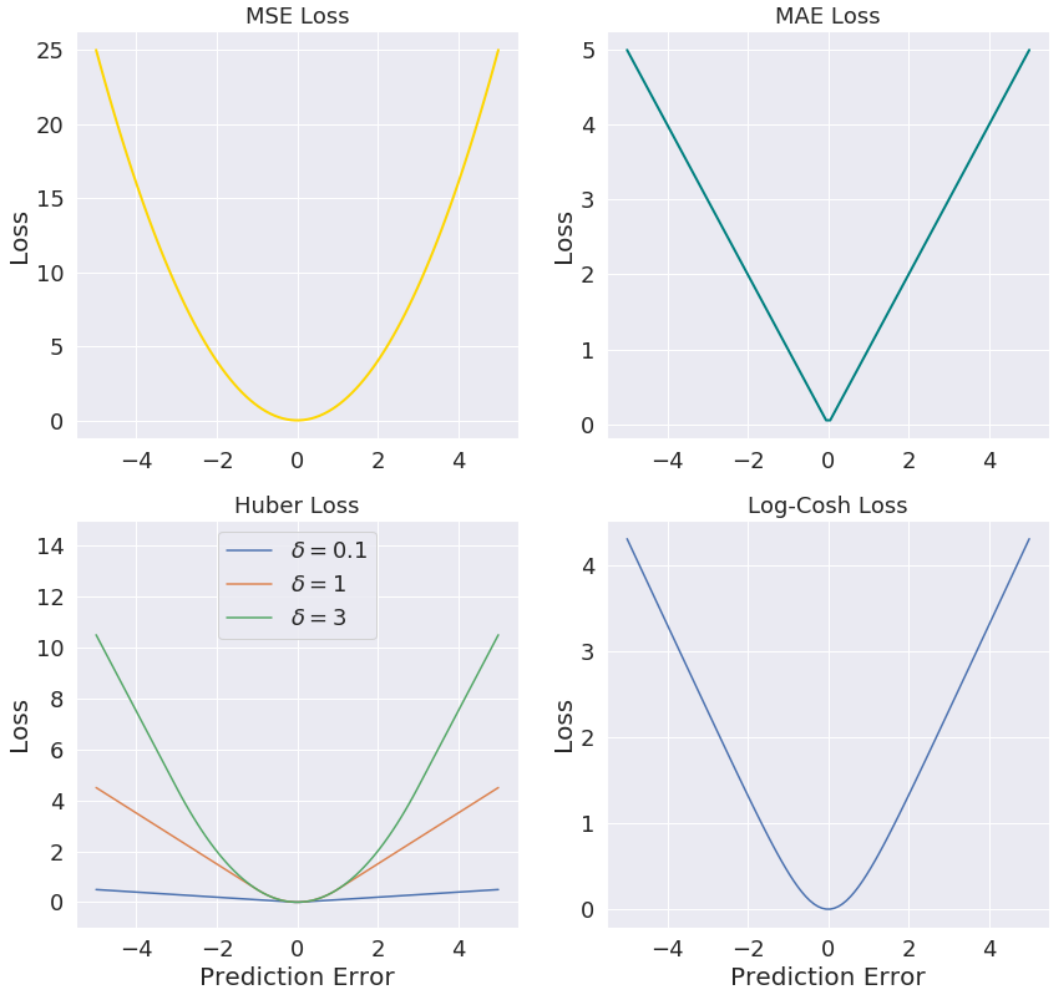

Figure 22.3.1: Regression loss functions: MSE Loss, MAE Loss, Huber loss ($\delta = 0.1, 1, 3$), and Log-Cosh Loss.

Remark 22.3.1.

- Compared to MSE, MAE is more robust to outliers.
- One big problem in using MAE loss is its constant gradient, which can hinder learning when model parameters are near optimal values (where small steps are preferred).

- Huber loss can be viewed as a hybrid version (controlled by δ)of MAE loss and MSE loss. It is less sensitive to outliers in data than the squared error loss and it is also differentiable at 0.
- $\log(\cosh(x))$ is approximately equal to $\frac{x^2}{2}$ for small x and to $|x| - \log(2)$ for large x. In other words, $\log(\cosh(x))$ works mostly like the mean squared error, but will also robust to outliers. It has all the advantages of Huber loss, and it is also twice differentiable everywhere, unlike Huber loss. Note that for machine learning frameworks like XGBoost (covered in following chapters), twice differentiable functions are more favorable.

Definition 22.3.2 (classification loss function). *Denote $\hat{y}$ as the estimated target output, y the corresponding (correct) target output, $\hat{y}_i$ the predicted value of the i-th sample, and y_i the corresponding true value.*

- *(zero-one loss)*
$$L_{0-1} = -\sum_{i=1}^{N} \mathbf{1}(y_i! = \hat{y}_i).$$
- *(binary cross entropy, log loss) Let y take value in $\{0, 1\}$Let $\hat{p}_i$ be the model predicted probability for class label 1. The **cross-entropy loss** for binary classification is given by:*
$$L_{BCE} = -\sum_{i=1}^{N} (y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)).$$
- *(multi-class cross entropy) Let $\hat{p}_i \in \mathbb{R}^M$ be the model predicted probability mass over M class labels. Let y_i be one-hot encoding of class labels. The cross-entropy loss over N samples is defined as*
$$L_{MCE} = \sum_{i=1}^{N} \sum_{c=1}^{M} y_{i,c} \log(\hat{p}_{i,c}).$$
- *(Perceptron loss) In binary classification, let $\hat{y}_i, y_i$ take values in $\{-1, 1\}$, then the **Perceptron loss** estimated over N samples is defined as*
$$L_{per}(y, \hat{y}) = -\sum_{i=1}^{N} \min(\hat{y}_i y_i, 0) = \sum_{i=1}^{N} \max(-\hat{y}_i y_i, 0).$$
- *(Hinge loss) In binary classification, let $\hat{y}_i, y_i$ take values in $\{-1, 1\}$, then the **Hinge loss** estimated over N samples is defined as*
$$L_{hinge}(y, \hat{y}) = -\sum_{i=1}^{N} \max(0, 1 - \hat{y}_i y_i).$$

- *(Squared Hinge loss) If $\hat{y}_i$ is the decision function output for the i-th sample, and y_i is the corresponding true value taking value $\{0,1\}$, then the **squared Hinge loss** estimated over N samples is defined as*

$$L_{per}(y,\hat{y}) = -\sum_{i=1}^{N} \max(0, 1-\hat{y}_i y_i)^2.$$

- *(Modified Huber loss) If $\hat{y}_i$ is the decision function output for the i-th sample, and y_i is the corresponding true value taking value $\{0,1\}$, then the **modified Huber loss** estimated over N samples is defined as.*

$$L_{Huber}(y,\hat{y}) = \begin{cases} \max(0, 1-y\hat{y})^2 & \text{for } y\hat{y} \geq -1 \\ -4y\hat{y} & \text{otherwise} \end{cases}$$

Some loss functions are showed in Figure 22.3.2.

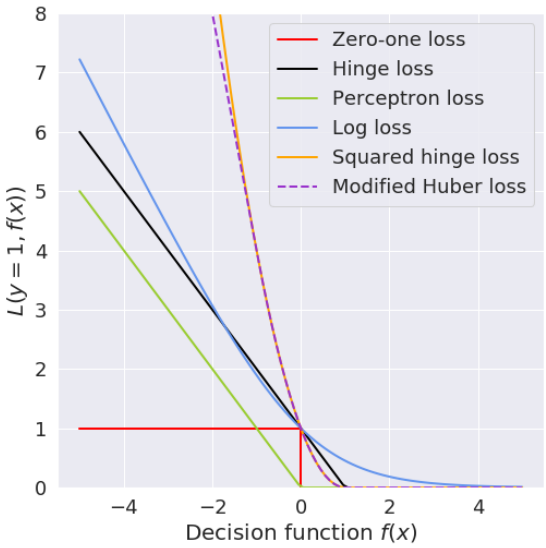

Figure 22.3.2: Common classification loss functions.

Remark 22.3.2.

- Zero-one loss is mostly used for monitoring rather than optimization since it is not smooth.

- Hinge loss not only penalizes incorrect predictions, but also penalize correct but less confident cases (i.e., decision function/score is not large enough). As a comparison, Perceptron loss only penalize incorrect classification.
- Log loss optimizes well-defined probabilistic output; Hinge loss encourages maximization of classification margins.
- Squared hinge loss and modified Huber loss are similar and they aim to overcome the non-differentibility issue in Hinge loss.

Example 22.3.1. Consider a 3-class classification problem. Assume one sample has label $y = (1, 0, 0)$, and the prediction gives $f(x; \theta) = (0.5, 0.3, 0.2)$. Then the cross entropy contribution from this example is given by

$$-(1 \times \log(0.5)) - (0 \times \log(0.3)) - (0 \times \log(0.2)).$$

22.3.2 Model evaluation metrics

22.3.2.1 *Regression metrics*

Although loss functions provide assessments on model performance and quality, we also need additional metrics to provides interpretation and evaluation on models specific to applications; this is particularly true for classifications. Note that most evaluation metrics are not differentiable and not suitable gradient descent type of optimization; therefore, they cannot be used ad loss functions.

Common evaluation metrics are summarized below.

Definition 22.3.3 (regression performance metrics). *Denote $\hat{y}$ as the estimated target output, y the corresponding (correct) target output, $\hat{y}_i$ the predicted value of the i-th sample, and y_i the corresponding true value*

- *(explained variance) The **explained variance** is estimated as follow:*

$$explained - variance(y, \hat{y}) = 1 - \frac{Var[y - \hat{y}]}{Var[y]}.$$

- *(mean absolute error) The **mean absolute error (MAE)** estimated over N samples is defined as*

$$MAE(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} \left| y_i - \hat{y}_i \right|.$$

- *(mean squared error) If $\hat{y}_i$ is the predicted value of the i-th sample, and y_i is the corresponding true value, then the **mean squared error (MSE)** estimated over N samples is defined as*

$$MSE(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2.$$

- *(median absolute error) If $\hat{y}_i$ is the predicted value of the i-th sample, and y_i is the corresponding true value, then the **mean squared error (MedSE)** estimated over N samples is defined as*

$$MadAE(y, \hat{y}) = median(\left|y_1 - \hat{y}_1\right|, ..., \left|y_N - \hat{y}_N\right|,)^2.$$

22.3.2.2 *Classification metrics*

Definition 22.3.4 (classification accuracy metrics). *Denote $\hat{y}$ as the estimated target output, y the corresponding (correct) target output, $\hat{y}_i$ the predicted value of the i-th sample, and y_i the corresponding true value.*

- ***Accuracy (acc)***

$$acc = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\left(y - i = \hat{y}_i\right)$$

- ***Error (err)***

$$err = 1 - acc = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}\left(y_i \neq \hat{y}_i\right)$$

For a specific class label c, there are four classification results.

- ***true positive, TP**, where the sample's true label is c and the predicted label is c. More formally, we define*

$$TP_c = \sum_{i=1}^{N} I\left(y_i = \hat{y}_i = c\right).$$

- ***false positive, FP**, where the sample's true label is not c but the predicted label is c. More formally, we define*

$$FP_c = \sum_{i=1}^{N} I\left(y_i \neq c \wedge \hat{y}_i = c\right)$$

- ***true negative, TN**, where the sample's true label is not c and the predicted label is not c.*

$$TN_c = \sum_{i=1}^{N} I\left(y_i \neq c \wedge \hat{y}_i \neq c\right)$$

- ***false negative, FN***, *where the sample's true label is c and the predicted label is not c. More formally, we define*

$$FN_i = \sum_{i=1}^{N} I\left(y_i = c \wedge \hat{y}_i \neq c\right).$$

Definition 22.3.5 (classification precision, recall, and F measure). *For a specific class label c, we can further define metrics of* ***precision***, ***recall*** *and* ***F measure****.*

- ***Precision*** *is the ratio of the number of correct prediction over the number of total prediction on label c:*

$$P_c = \frac{TP_c}{TP_c + FP_c}$$

- ***Recall*** *is is the ratio of the number of correct prediction over the number of total label c:*

$$R_c = \frac{TP_c}{TP_c + FN_c}$$

- *F* ***measure***

$$F_c = \frac{\left(1 + \beta^2\right) \times P_c \times R_c}{\beta^2 \times P_c + R_c}$$

where β is the parameter to adjust the relative importance of precision and recall Usually, $\beta = 1$, and in this case, F measure to denoted by F_1.

Remark 22.3.3. Depending on the application, precision and recall are playing different roles. Note that **precision is about exactness, and recall is about completeness.**

- For rare disease detection, recall is more important than precision, since we want all true disease to be detected. Precision is of less concern since it is ok to make false positive mistake. In other words, in the medical community, a false negative is usually more disastrous than a false positive for preliminary diagnoses. For one extreme example, health care plans will generally offer the flu shot to everyone, disregarding precision entirely.
- For phone advertisement where we need to identify customers who want to buy the product, if the cost of making a phone call is low, then making false-positive mistake (call a customer who has no interest) is acceptable, so precision can be low. But recall needs to be high since we need to find out all customers who are interested. On the other hand, if the cost of making a phone call and an advertisement is very expensive, we would want precision to be high since we hope every call or advertisement is targeted to the interested customer.

We can also average over **precision**, **recall** and **F measure** over all labels.

Definition 22.3.6 (average of precision and recall).

- *Macro average are averages over all class*

$$P_{\text{marcro}} = \frac{1}{K}\sum_{c=1}^{K} P_c$$
$$R_{\text{marcro}} = \frac{1}{K}\sum_{c=1}^{K} R_c$$
$$F_{1,\text{marcro}} = \frac{2 \times P_{\text{marcro}} \times R_{\text{marcro}}}{P_{\text{marcro}} + R_{\text{marcro}}}$$

- *Micro average is defined by*

$$P_{\text{miro}} = \frac{\overline{TP}}{\overline{TP} + \overline{FP}} = \frac{\sum_{c=1}^{K} TP_c}{\sum_{c=1}^{K} TP_c + \sum_{c=1}^{K} FP_c}$$
$$R_{\text{micro}} = \frac{\overline{TP}}{\overline{TP} + \overline{FN}} = \frac{\sum_{c=1}^{K} TP_c}{\sum_{c=1}^{K} TP_c + \sum_{c=1}^{K} FN_c}$$
$$F_{\text{micro}} = \frac{2 \times P_{\text{micro}} \times R_{\text{micro}}}{P_{\text{micro}} + R_{\text{micro}}}$$

Remark 22.3.4 (macro-average vs. micro-average).

- Macro-averaging assigns the same weight to each category, whereas micro-averging gives every sample the same weight.
- We should use micro-averaging when we evaluate classification performance on majority categories, but use macro-averaging when we evaluate classification performance on minority categories,

22.3.2.3 *ROC and PRC metrics*

In binary classification, the prediction based on feature x is often characterized by some probability function $f(x)$. If $f(x) \geq T$, where T is the threshold, we predict the positive label, and otherwise. After choosing the threshold, the classifier is determined and its performance will be characterized by recall, precision, etc, as we covered in the previous section.

Besides calculating the metrics at chosen threshold, we can also calculate metrics at different at threshold points and put them in the same plot. One popular plot is the Receiver Operating Characteristic (ROC) curve.

The curve is a plot of false positive rate (FPR) versus the true positive rate (TPR) for a number of threshold values. A user may plot the ROC curve and choose a threshold that gives a desirable balance between the false positives and false negatives.

Specifically, the false positive rate on the x axis is given by

$$\text{False Positive Rate } = \frac{FP}{FP + TN}$$

And the the true positive rate on the y axis is given by

$$\text{True Positive Rate } = \frac{TP}{TP + FN}$$

The best possible classifier could be characterized by a point in the upper left corner or coordinate (0,1) of the ROC space, indicating 100% accuracy for all classes. A random guess is characterized by a point along a diagonal line (the concrete value depends on the number of samples in different classes).

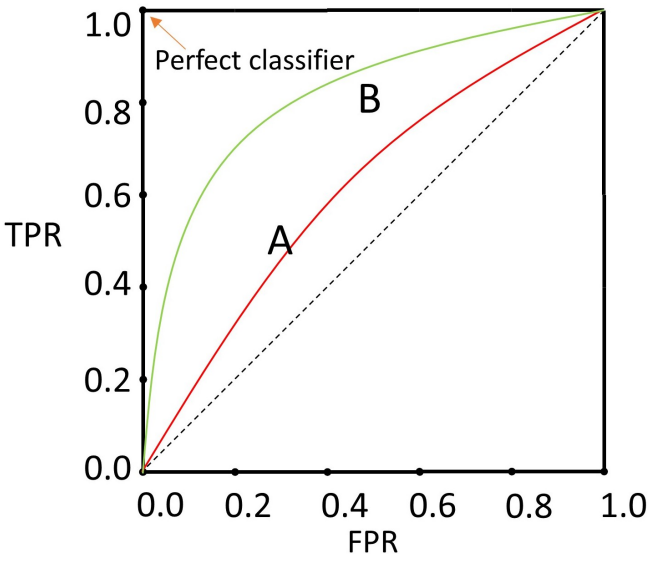

Figure 22.3.3: Scheme for ROC curves diagram. A and B demote ROC curves of different model.

Another similar curve is precision recall curve (PRC), which plots precision and recall at different thresholds.

22.3.2.4 *Metrics for imbalanced data*

In the imbalanced classification problem, the distribution of examples across different classes is severely biased or skewed. For example, example in the minority classes are hundreds or thousands, but there are millions of examples in the majority classes. Imbalanced classification are common in rare event predictions, such as fraud detection, and terrorist detection.

Imbalanced classifications pose a challenge for predictive modeling as most of the machine learning algorithms used for classification are designed around the assumption of an equal number of examples for each class.

Accuracy can be a misleading metric for imbalanced data sets. Consider a sample with 95 negative and 5 positive values. Classifying all values as negative in this case gives 0.95 accuracy score. An improved accuracy measure is the **balanced accuracy**.

Definition 22.3.7 (classification metrics for imbalanced data). *Let w_i denote the sample weight for sample i (usually $w_i = 1$), the balanced accuracy is given by*

$$\textit{balanced accuracy} = \frac{1}{\sum \hat{w}_i} \sum_i 1\left(\hat{y}_i = y_i\right) \hat{w}_i$$

where

$$\hat{w}_i = \frac{w_i}{\sum_j 1\left(y_j = y_i\right) w_j}$$

Note that for imbalanced data, ROC curve can also give misleading results when the true-labeled examples are dominant classes. Precision and recall could provide useful information on classification performance on each class. Consider the following example. The average metrics of precision, recall and f1-score are all quite high, while the scores specific to the minor class are poor.

	precision	recall	f1-score	sample number
0	0.96	0.99	0.98	1884
1	0.73	0.41	0.53	116
avg	0.95	0.96	0.95	2000

22.4 Model selection methods

As we search for an optimal hypothesis in the hypothesis space, we need to carefully control model complexity to avoid overfitting/underfitting or to achieve better variance bias trade-off. The parameters controlling the model complexity are generally referred to as **hyperparameters**. Hyperparameters determine the high level characteristics of a model. Example hyperparameters in different hypothesis space include

- The degree of order in polynomial regression.
- The penalization coefficient in penalized linear regression.
- The maximum depth of decision tree.
- The number of weak learners in ensemble learning.
- The architecture(number of neurons, layers)in artificial neural networks.

Unlike model parameter that can be optimized via well developed optimization algorithms such as gradient descent, optimization of hyperparameters usually relies on quite empirical, ad-hoc, and brute force search strategies.

22.4.1 The training-validation-testing idea

To prevent overfitting in supervised learning algorithms, it is common practice to split the data to training set and testing data where the model parameters are estimated based on the training data and the performance evaluation is conducted on the testing data set.

For a set of models parameterized by hyperparameters (e.g., the regularizer parameter in the penalized linear regression), using performance evaluation on the testing data set to determine the optimal hyperparameters has the risk of overfitting on the testing data set (i.e., the model might still perform bad for samples never seen before.). This is because the during the hyperparameter searching process, knowledge about the testing data set can 'leak' into the model and evaluation metrics no longer represents generalization performance. To solve this problem, common practice is to divide the data set into **training set, validation set, and testing set**. In such way, training proceeds on the training set, after which hyperparameters tuning is done on the validation set. In the end, final evaluation of the model is done on the test set.

22.4.2 Cross-validation

One of most popular implementation of the training-validation-testing idea is cross-validation. In the most popular, basic approach, called K-fold CV[Figure 22.4.1], we divide

the training data into K folds and train a model using $k-1$ of the folds as training data; the resulting model is validated on the remaining part of the data.

This approach can be computationally expensive, as we have to train and test the model for k times, but it utilizes data in a quite economical way, which is a critical advantage in problems with scarce data.

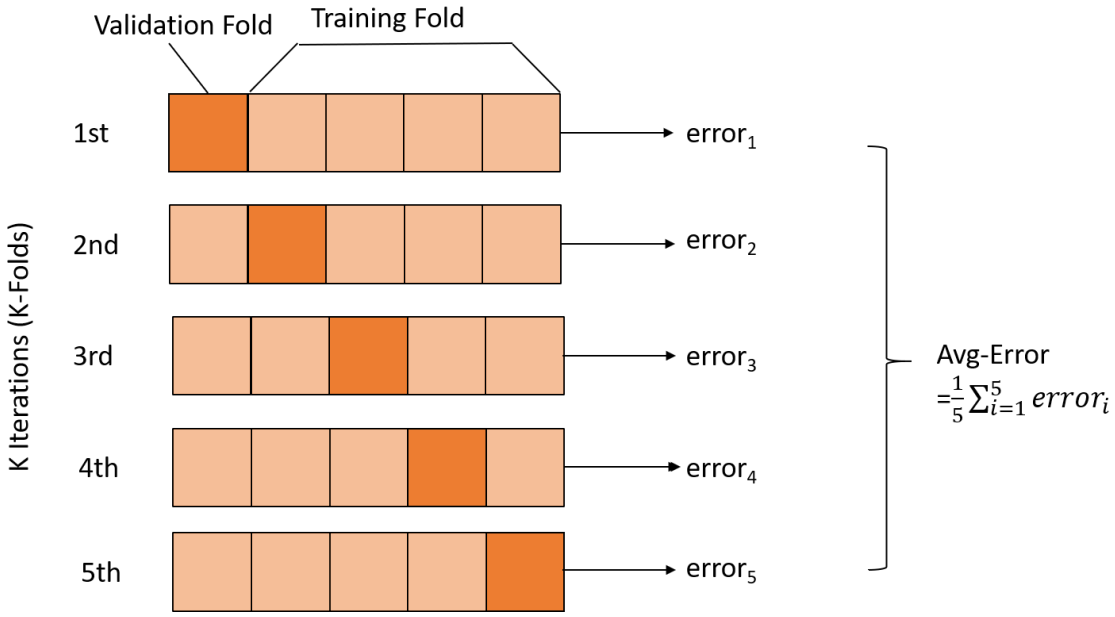

Figure 22.4.1: Scheme for cross-validation error calculation procedure.

For each model hyperparameter θ, the **cross-validation error (CV error)** is then measured by

$$\mathrm{CV}(\theta) = \frac{1}{n}\sum_{k=1}^{K}\sum_{i\in F_k}\left(y_i - \hat{f}_\theta^{-k}(x_i)\right)^2.$$

And we can select hyperparameter based on the CV error. We have following summary on calculating CV error.

Methodology 22.4.1 (K-fold CV error calculation procedure). *Suppose we are given a training data set $D = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$. Our model estimate parameterized by hyper-parameter θ is denote by $\tilde{f}_\theta$. The K fold CV has the following procedures:*

- *Divide the index set $\{1, 2, ..., n\}$ into K subsets $F_1, F_2, ..., F_K$.*

- *For each $k = 1, ..., K$, train the model on the set $\{(x_i, y_i) \notin F_k\}$ and validate the model on the validation set $\{(x_i, y_i) \in F_k\}$. Let $\tilde{f}_\theta^{(k)}$ denote the trained model, and the validation error is given by*

$$e_k = \sum_{i \in F_k} (y_i - \tilde{f}_\theta^{(k)}(x_i))^2$$

- *For each hyper-parameter θ, compute average error over all folds,*

$$CV(\theta) = \frac{1}{n} \sum_{k=1}^{K} e_k(\theta)$$

Methodology 22.4.2 (minimal error rule). *The minimal error rule select the model parameters minimizing the cross-validation error, i.e.,*

$$\hat{\theta} = \underset{\theta \in \{\theta_1, ... \theta_m\}}{\operatorname{argmin}} \ \mathrm{CV}(\theta)$$

Remark 22.4.1 (how to choose fold number K).

- Large K value requires more computational time. K can go to as large as $K = n$ (i.e., leave-one-out cross-validation).
- The choice of K affects the quality of our cross-validation error estimates for model assessment. For example, if $K = 2$, the CV error estimates are going to be biased upwards, because half the data each time is used to train; if $K = n$, the CV error will have high variance since we are averaging a set of highly correlated quantities.
- In practice, $K = 3, 5, 10$ are common choices.

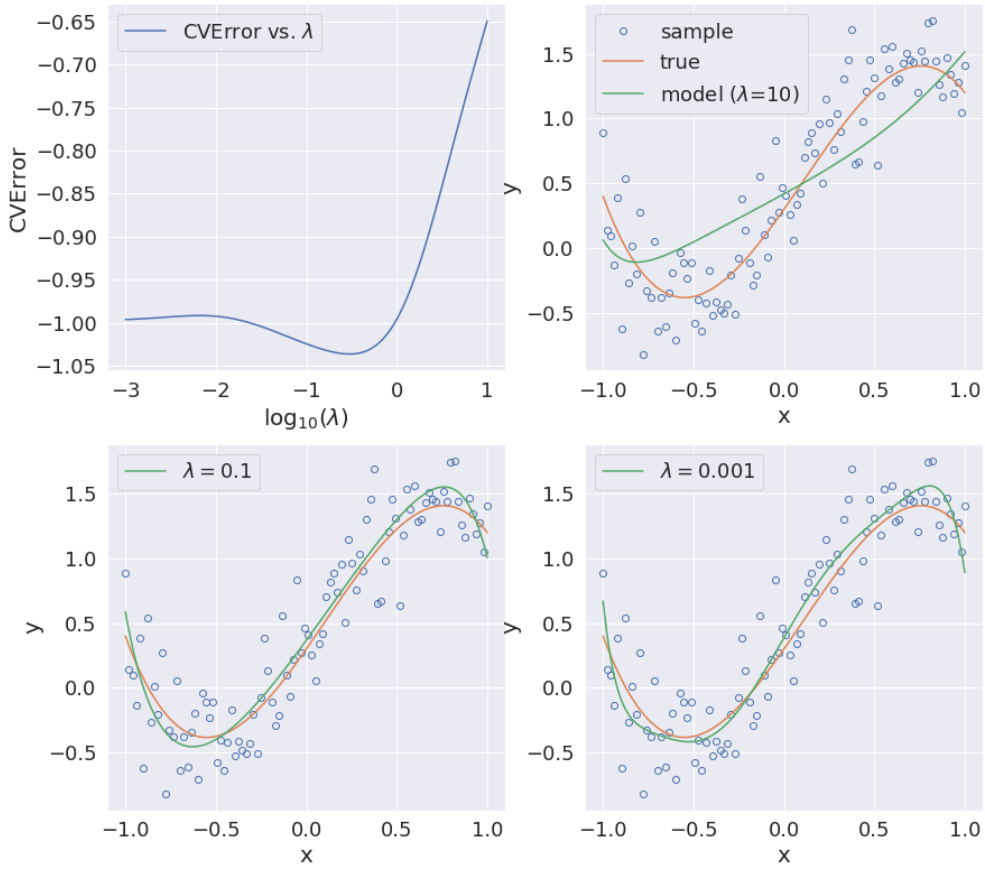

Figure 22.4.2: Hyperparameter search via turning regularization parameter λ. At large λ, heavy regularization causes underfitting; at small λ, insufficient regularization causes overfitting.

Considering the minimal CV error rule could be too aggressive and causes overfitting, another popular, more conservative rule is the one-standard-error rule:

Methodology 22.4.3 (one-standard-error rule). *In the **one-standard-error rule**, optimal model parameter is selected via*

$$\hat{\theta} = \underset{\theta \in \{\theta_1, \ldots \theta_m\}}{\operatorname{argmin}} \ \mathrm{CV}(\theta)$$

where $SE[\hat{\theta}]$ is the standard deviation of the CV error. And we increase regularization strength via θ until

$$\text{CV}(\theta) \leq \text{CV}(\hat{\theta}) + \text{SE}(\hat{\theta})$$

in other words, we select the simplest model whose model error is within one standard error of the minimal error. Therefore, the model selected via one-standard-error rule is usually more restricted than models selected via the minimal-error rule.

22.5 Data and feature engineering

22.5.1 Data preprocessing

22.5.1.1 *Data standardization*

Date standardization is one of mostly widely applied data preprocessing step in machine learning. Standardized data will have feature-wise zero mean and unit variance, which can significantly speed up learning process (especially true for complex models like neural networks). Data standardization is also the requirement for regularized learning algorithm (L1, L2, and others) such that all feature variables is fairly affected by shrinkage procedures.

We have the following summary:

Methodology 22.5.1 (input data standardization). *Given the input data set, $X = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}, x^{(i)} \in \mathbb{R}^p$, consisting n inputs and p features.*

Let $\mu = (\mu_1, ..., \mu_p)$ be the estimated feature mean

$$\mu_k = \sum_{i=1}^{n} \frac{1}{n} X_k^{(i)}.$$

Let $\sigma = (\sigma_1, ..., \sigma_p)$ be the estimated feature standard variance

$$\sigma_k = \sum_{i=1}^{n} \frac{1}{n-1} (X_k^{(i)} - \mu_k)^2.$$

*Then the **standardized input data** set is given by $\tilde{x}^{(i)} = \frac{x^{(i)} - \mu}{\sigma}$, where we implicitly use element-wise multiplication and division.*

After standardization, data $\{\tilde{x}^{(1)}, \tilde{x}^{(2)}, ..., \tilde{x}^{(n)}\}$ has zero mean and unit variance for each feature.

Remark 22.5.1 (practical use in predictive modeling). In predictive modeling, we use training data to determine the means and variances used for standardization for both training and testing input data.

Remark 22.5.2 (dealing with outliers). If the input data contains many outliers, standardization process can yield very poor results in the testing stage. Robust estimation of mean and variance [subsubsection 13.1.2.6], detecting and discarding outliers are needed for data standardization.

22.5.1.2 *Data normalization*

Different from data standardization, data normalization is the process of scaling individual samples to have unit norm. The normalized data sample can be used to compute dot-product or any other kernel to quantify the similarity of any pair of samples. Application examples include computing similarity scores between text documents for vector space document models.

We can summarize as data normalization in the following.

Methodology 22.5.2 (sample-wise data normalization,L_p norm). *Given the input data set, $X = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}, x^{(i)} \in \mathbb{R}^p$, consisting n inputs and p features.*

For each sample, compute $\left\| x^{(i)} \right\|_p$, then the normalized input data set is given by

$$\tilde{x}^{(i)} = \frac{x^{(i)}}{\left\| x^{(i)} \right\|_p}.$$

22.5.1.3 *Handle categorical data*

Most of machine learning algorithms cannot directly handle categorical variable. For example, a feature called martial status takes three distinct values of *married*, *single* and *divorced*. To enable machine learning algorithm to cope with such feature, we can convert them to multiple binary features, for example,

Categorical Value	x_1	x_2	x_3
Married	1	0	0
Single	0	1	0
Divorced	0	0	1

More formally, we use the following methodology summary

Methodology 22.5.3 (Encoding categorical variables). *Assume the categorical variable is taking $K, K \geq 3$ discrete values.*

- *Suppos the variable takes values from 1 to 6. Then we can design five additional binary features with the following rule*

$$x_i = \begin{cases} 1, if\ value\ i \\ 0, others \end{cases}, i = 1, 2, ..., 5.$$

  If $x_i = 0, \forall i = 1, 2, ..., 5$*, then the categorical value is 6.*
- *Note that we cannot design six features since they are not linearly independent; that is, when value of five features are known, the six is known.*

22.5.1.4 *Handle missing values*

For various reasons, many real world datasets contain missing values, often encoded as blanks, NaNs, or other placeholders. Such datasets however are incompatible with most of the machine learning algorithms.

One basic strategy to handle incomplete data X is to discard discard the example, and remove the feature from all samples. Either approach comes at the price of losing data which may be valuable (even though incomplete). A better strategy is to impute the missing values, i.e., to infer them from the known part of the data based on some assumed distributions on missing values.

In practice, some ad hoc strategy is to impute missing values using either the mean, the median, or the most frequent value.

22.5.1.5 *Dimensional reduction*

In some real-world machine learning problems, the feature vector can have very high dimensionality (e.g., image data) and leads to the curse of dimensionality for many machine learning algorithms. Common methods to reduce the data dimensionality include unsupervised learning approaches such as PCA and manifold learning [chapter 29], and supervised learning approaches like linear discriminate analysis [section 24.3].

22.5.1.6 *Centering kernel matrix*

When we apply kernel methods [section 22.6], oftentimes we need to center the kernel matrix, which aims to remove the mean in the implicitly mapped high-dimensional feature space, we can apply the follow methodology:

Methodology 22.5.4 (centering the feature vectors). *Given a finite subset* $S = \{x_1, x_2, ..., x_n\}$ *of an input space* X *and a kernel* $k : X^2 \to \mathbb{R}$ *satisfying* $k(x, x') = \langle \phi(x), \phi(x') \rangle$ *for some feature map* $\phi : X \to H$*: we can centering* $K_{ij} = \left\langle \phi(x_i), \phi(x_j) \right\rangle$ *to* $K'_{ij} = \left\langle \phi(x_i) - \phi_C, \phi(x_j) - \phi_C \right\rangle$ *via*

$$K' = HKH, H = I - 1_n 1_n^T / n,$$

where $\phi_C = \frac{1}{n} \sum_i \phi(x_i)$.

For proof, see Lemma 22.6.3.

22.5.2 Feature engineering I: basic routines

Feature engineering is the process of using domain knowledge of the data to create features that make machine learning algorithms work. Feature engineering is fundamental to the application of machine learning, and is both difficult and expensive.

Here we covers some basic strategies for feature engineering. See a more systematic treatment, see [3][4].

22.5.2.1 *Nonlinear transformation*

Nonlinear transformations directly apply a nonlinear transformation to the feature. Common transformation includes logarithmic, exponential, logit, etc.

22.5.2.2 *Polynomial features*

The idea of polynomial features is to increase interaction and high order features to capture nonlinear effects. For example, features (X_1, X_2) can be used to generate features $(1, X_1, X_2, X_1 X_2, X_1^2, x_2^2)$.

22.5.2.3 *Binning*

Feature binning is a method of turning continuous variables into categorical values. The main motivation of binning is to make the model more robust to outliers (e.g., extreme values will be binned with other normal data) and prevent overfitting; however, it has a cost to the performance. Binning can be viewed as a regularization technique applied to the data level.

Binning continuous variables that span a large value range to significantly increases the total number of bins and ultimately makes learning challenges. In these cases, one apply logarithmic or quantile binning strategies. For logarithmic strategy, binning is applied to logarithm of the original data; for quantile strategy, the bin breaks are chosen from quantiles, which are values dividing the data into roughly equal-sized subsets.

22.5.3 Feature engineering II: feature selection

22.5.3.1 *Filtering methods*

One simple yet naive method to select 'important' features is to remove features that have low variance. It is based on the assumption features with a higher variance may contain more useful information. Clearly, such filtering method does not consider the interaction among features. Further, the feature with low variance might be the key deciding factor for the outcome.

Methodology 22.5.5 (feature selection via variance filtering).

- *Compute the variance of each feature.*
- *Select the subset of features whose variance is greater than a threshold.*

Example 22.5.1. When each raw pixel is used as a feature for some image classification problems, some features can have near zero variance (e.g., background part) and should be removed.

It is possible that some features have small variance but still contribute significantly to the prediction. It is therefore necessary to run a more informative filtering rather than simply running a variance filtering will remove these feature. Inspired by the Bayesian statistics approach to classification problems, we can filter out features based on the distribution conditioning on the class label.

Methodology 22.5.6 (feature selection via conditional distribution filtering).

- *Compute the variance of each feature*
- *Remove features can that have similar conditional probability density $Pr(X_i|y)$ based on similarity measure*

$$d(X_i) = Pr(X_i|y=0) \cdot Pr(X_i|y=1)$$

22.5.3.2 *Recursive elimination methods*

Filtering methods usually ignore that fact that interactions among features might contribute to the predicative output. As a result, some useful features could be eliminated and ultimately damage model performance.

Here we introduce a more informative, principled methods to select features. Because some models, such linear regression model and tree methods, can reveal the importance of each feature after we fit a model. We can evaluate the feature importance from the fitted model and recursively eliminate un-importance feature until models of desired simplicity is obtained.

Methodology 22.5.7 (recursive elimination method for linear models). *Consider a linear regression/classification model.*

- *Fit model to a data set with candidate features.*
- *Eliminate un-important features by inspecting fitted model parameters (e.g, features associated with the smallest coefficients)*
- *Repeat previous steps until desired number of features is left.*

22.5.3.3 *Regularization methods*

Feature selection can also be achieved automatically in the model parameter estimation process, known as regularization methods. One most widely used regularization methods to induce sparsity as part of selecting features is to apply L_1 regularization on models.

Applying L_1 regularization on linear regression model yield lasso [subsection 23.2.2]. The majority of linear classification models such as logistic regression also accepts L_1 regularization.

22.5.4 Feature engineering III: feature extraction

In practice, data rarely comes in the form of ready-to-use matrices, particularly for applications in text, image, audio, etc. These tasks begin with feature extraction.In the following, we survey some of the popular types of data from which features can be extracted.

22.5.4.1 *Text analytics*

Text is a type of data requires substantial efforts for feature engineering from all levels from single word to documents. There are a wealth of feature extraction methods for text analytic, and here we only review the most critical ones. For a practical treatment on this topic, see [5].

One popular feature engineering commonly used in simple text analytics is **one-hot encoding**, every word is represented as an $\mathbb{R}^{|V|}$ vector with all 0s and one 1 at the location corresponding to the index of word, where $|V|$ is the size of the dictionary. To meet the needs of more complex applications, one-hot encoding is usually mapped to low-dimensional embedding to capture the semantic relationship between words.

On the document level feature extraction, one commonly used feature for document clustering and sentiment classification is TF-IDF.

Definition 22.5.1 (TF-IDF). *Term frequency $tf(t,d)$ is the count of a term t in a document d. Inverse document frequency $idf(t,\mathcal{D})$ is defined as the logarithmically scaled inverse fraction of the documents that contain the terms t, given by*

$$idf(t,\mathcal{D}) \triangleq \log \frac{|\mathcal{D}|}{|\{d \in \mathcal{D} : t \in d|}$$

where $\mathcal{D}$ is the set of all documents.

TF-IDF has following interpretations:

- a word appears frequently in one document but not the other documents will have a large tf value and large idf value. Then the $tf \times idf$ value will indicates that this word is an import feature of the document.
- a word appears frequently in all documents, like stop words 'the', 'a', 'is', etc, will have a large tf value but small idf value. Then $tf \times idf$ value will be a small value indicating that this word is not an import feature of the document.

22.5.4.2 *Image*

A raw image data with pixel level data is a high-dimensional data format. Before the remarkable success led by deep learning. High-level features, such as edges and corners are extracted via a number of traditional feature detection algorithms for applications in recognization, detection, segmentation, etc. The algorithms to extract these features are results of decades' efforts from human experts. Please refer to [6][7] for a comprehensive overview .

The employment of deep neural networks, particularly convolutional neural networks, to achieve automatically learning of high-level features from data, has been a disruptive force in the realm of traditional methods. The layer-by-layer structure allows direct input of raw image data and perform feature extraction from low-level pixels to high-level features[8]. The learned features in neural networks can be used in other machine learning modules. See [7] for all details.

22.5.4.3 *Time series*

Time series data modeling and prediction arise from countless applications, from stock market to radar signal to biomedical measurement. In applications of time series classification and regression, traditional machine learning algorithm rarely directly take raw time series data as input, but takes unique features extracted from in time series data including trends, periodicity, stationarity, seasonarity, etc. Common features are reviewed in [9]. Notably, feature extraction software are also available[1].

More recently, deep recurrent neural networks are applied in machine learning tasks in a end-to-end manner, where feature extraction are performed automatically by learning from data.

1 https://github.com/blue-yonder/tsfresh

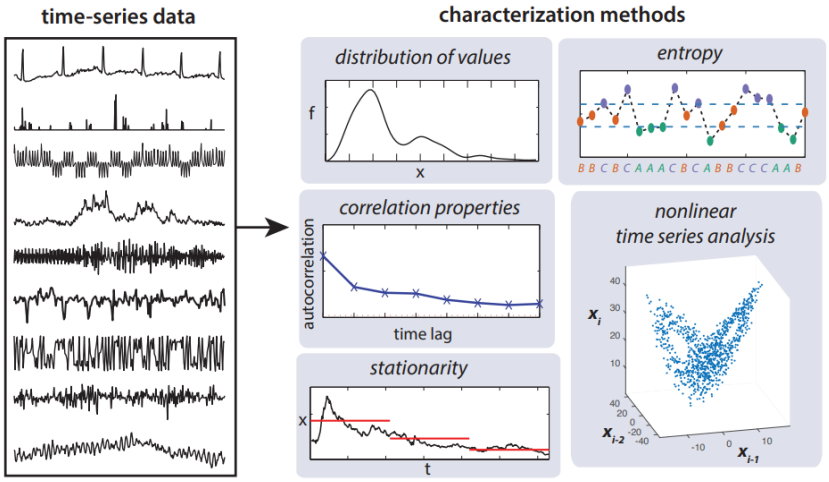

Figure 22.5.1: Left: A sample of nine real-world time series reveals a diverse range of temporal patterns [4, 5]. Right: Examples of different classes of methods for quantifying the different types of structure, such as those seen in time series on the left: (i) distribution (the distribution of values in the time series, regardless of their sequential ordering); (ii) autocorrelation properties (how values of a time series are correlated to themselves through time); (iii) stationarity (how statistical properties change across a recording); (iv) entropy (measures of complexity or predictability of the time series quantified using information theory); and (v) nonlinear time-series analysis (methods that quantify nonlinear properties of the dynamics).[9]

22.5.5 Imbalanced data

22.5.5.1 *Motivations*

Imbalanced data issues prevail in domains such as security and financial crime where the events we want to detect and prevent rarely occur. Machine learning methods not taking such imbalance into account could produce misleading results.

For instance, in a task of detecting fraudulent transactions in the field of financial crime, fraudulent transactions usually only make up less than 2% total transactions, and they are significantly infrequent than normal transactions.

Direct application of machine learning algorithms (with loss functions primarily applied to balanced data) tend to predict the majority class, although accurate prediction of minority class are most needed. For example, if the majority class makes up 98% of all data, then unanimously predicting the majority class label will yield an accuracy rate of 98%. The apparent high accuracy is pointless in the sense of business purpose.

In this section, we will overview different methodologies from different perspectives, including data resampling and algorithm enhancement.

22.5.5.2 *Data resampling: undersampling*

One simple approach to address data imbalance is to randomly eliminate majority class examples, i.e., undersampling, provided that the elimination will not negatively affect the sample diversity to a significant degree. The elimination is typically carried out until the data are balanced.

One severe potential drawback is losing potentially important information that could enable the construction of a more interpretable classifier. Apart from randomly elimination of abundant samples, sample removal can be conducted in a more informed way in which similar and repeated (based on some similarity measure) samples are removed.

22.5.5.3 *Data resampling: upsampling*

The opposite of undersampling is upsampling, meaning increasing the number of samples in the minority class by randomly replicating them. Increased presence of data samples will encourage algorithms to learn classifiers that focuses on detecting minority classes.

Random upsampling can increase the risk of overfitting since it replicates the minority class events, which tend to be more of noises than underlying signals. Alternative way is the Synthetic Minority Over-sampling Technique (SMOTe)[2]. In SMOTe, synthetic observations of the minority class are generated by the following two procedures:

- Finding the k-nearest-neighbors for minority class observations (based on some similarity measures)
- For a minority sample, randomly selecting one of the k-nearest-neighbors and create a new sample by perturbing the selected neighbor.

In the actual training algorithms, we can upsample via the either of the following approaches

- Split the original training data into training and validation set; then use SMOTe to augment the training data only.
- Directly augment the original training data, then do the split.

2 For software implementation, see https://imbalanced-learn.readthedocs.io/en/stable/

22.5.5.4 *Choice of loss functions, algorithms, and metrics*

For most linear classification algorithms like logistic regression and SVM, the loss functions used for gradient steps in general assign equal weights to each individual sample. The resulting classifies are trained to primarily identify the majority class. A simple way to overcome data imbalance is to increase the weight associated with minority samples, which aims to penalize more on misclassification of minority samples. Mathematically, increasing the weight is equivalent to randomly upsample minority samples, therefore it brings the risk of overfitting.

Some algorithms like logistic regression and SVM, are inappropriate for imbalanced data without additional modification on loss function. Some other algorithms, like decision trees and ensemble learning[10], with proper loss weight adjustment can be very robust when facing imbalanced data. Take decision tree as an example. The splitting criterion usually be designed to be robust to class imbalances and enable both classes to be addressed.

In terms of performance metrics, accuracy is inappropriate for an imbalanced dataset as it places significantly less weight on minority class. Alternatively, suitable metrics include

- Precision/Specificity: how many selected instances are relevant.
- Recall/Sensitivity: how many relevant instances are selected.
- F1 score: harmonic mean of precision and recall.
- MCC: correlation coefficient between the observed and predicted binary classifications.

22.6 Kernel methods

22.6.1 Basic concepts of kernels and feature maps

In many applications, some data types (such as text document, material structure) are not easy to represent the data as a vector in $\mathbb{R}^d$. On the other hand, we have some way to measure the similarity between different input examples. Here we introduce **kernel methods**, an important machine learning methodology, that enables the design of algorithms using similarity as input for classification and regression tasks.

We first introduce the concept of kernel and their connections to feature mapping.

Definition 22.6.1 (kernel). *A function $K : X \times X \to \mathbb{F}$ is called a kernel over X.*

Definition 22.6.2 (feature map, feature, feature space). *In machine learning, a **feature map** is usually an embedding:*

$$\phi : R^D \to \mathbb{R}^M$$

*with $M \gg D$. The point $\phi(x) \in \mathbb{R}^M$ is called the **feature** of data point $x \in \mathbb{R}^D$. The space $\mathbb{R}^M$ is called the **feature space**.*

Remark 22.6.1 (relations between kernel and feature map).

- For general kernels, usually there is no connection between the two.
- When the kernel satisfies certain conditions, say Mercer's condition, one can show that the kernel is implicitly associated with a feature map. **This association is significant, it enables us to use some implicitly complex feature maps to do machine learning tasks by only doing calculation using relatively 'simple' kernels.**

Example 22.6.1 (linear kernel). If the original data is already in $\mathbb{R}^d$, we can use $\phi(x) = x, k(x, x') = x^T x'$. **This kernel is useful only when (1) x is high dimensional (2) every individual dimension is informative(providing useful information)**. In the example of image recognition, linear kernel will be a bad choice since not every pixel is informative.

Remark 22.6.2. More complex feature(high-dimensional) space usually enable us to distinguish distributions with different means.

22.6.2 Mercer's theorem

Definition 22.6.3 (positive definite symmetric kernel). *A kernel $k : X^2 \to \mathbb{F}$ is said to be positive definite symmetric(PDS) if for any $x_1, x_2, ..., x_m \in X$, the matrix $K_{ij} = k(x_i, x_j)$ is symmetric semi-positive definite matrix. The matrix K is called **Gram matrix** of k with respect to $x_1, x_2, ..., x_m$.*

Lemma 22.6.1 (Cauchy-Schwarz inequality for PDS kernel). *If k is a PDS kernel, then*

$$\left|k(x_1, x_2)\right| \leq k(x_1, x_1)k(x_2, x_2)$$

Proof. It could be showed by from $detK \geq 0$, where K is the Gram matrix of x_1, x_2. □

Remark 22.6.3 (interpretation). In studying approximation theory in vector space using basis functions, we also define Gram matrix with entries being the dot product of basis function. Here, we view the **input data vectors as the basis of the input vector space**.

Definition 22.6.4 (Mercer's condition). *A real-valued function $k : X \times X \to \mathbb{R}$ is said to fulfill Mercer's condition if for **all square integrable functions** $g(x)$ one has*

$$\int_X \int_X g(x)K(x, y)g(x)dxdy \geq 0$$

Or in discrete case, the k is replaced by a matrix $K \in \mathbb{R}^{N \times N}$, such that for all vectors $g \in \mathbb{R}^N$ such that

$$\left\langle g, Kg \right\rangle = g^T Kg \geq 0$$

Theorem 22.6.1 (Mercer's theorem). *[11, p. 64] Suppose $k : X \times X \to \mathbb{R}$ is a continuous symmetric non-negative definite kernel, that is, it satisfies Mercer's condition. Then there is an orthonormal basis $e_i(x)$ of $L^2(X)$ such that*

$$k(s, t) = \sum_{j=1}^{\infty} \lambda_i e_j(s) e_j(t)$$

where $\lambda_i \geq 0$

Remark 22.6.4 (interpretation & significance).

- The mercer's theorem enables us to interpret: for symmetric semi-positive definite kernels, there exist feature maps given as $\phi(x) = \sum_i \sqrt{\lambda_i} e_i(x)$ such that $k(s, t) = \left\langle \sum_i \sqrt{\lambda_i} e_i(s), \sum_i \sqrt{\lambda_i} e_i(s) \right\rangle$ note that it would be difficult to recover the underlying functions $e_i(x)$ and λ_i.

- The mercer's theorem enables us to connect kernel to feature maps when mercer's condition holds. **Note that for an arbitrary kernel, the concept of kernel and feature map might be unrelated.**

Remark 22.6.5 (the underlying feature map). Given a kernel function k satisfying Mercer's condition,**the feature map and the feature space are implicitly defined**, and it is usually hard to find the underlying feature map $\phi(x)$ is(because of the uniqueness issue for example). But if we are given a feature map $\phi(x)$, calculating a 'canonical' kernel k is straight forward via $k(x, x') = \left\langle \phi(x), \phi(x') \right\rangle$.

Remark 22.6.6 (projection onto subspaces spanned by input vectors). Given a new input vector x, we want to measure how similar x is to the training input vectors $x_1, x_2, ..., x_m$. We can achieve this by solving coefficients $p \in \mathbb{R}^m$, via $Kp = q$, where $K \in \mathbb{R}^{m \times m}, K_{ij} = k(x_i, x_j), q_i = k(x, x_i)$. As a result, the projection coefficient is given as

$$p = K^{-1}q.$$

Note that we can calculate the projection with merely usage of kernel function.

22.6.3 Common kernels

Definition 22.6.5 (linear kernel). *Linear kernel $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is defined as*

$$k(x_i, x_j) = \left\langle x_i, x_j \right\rangle.$$

Definition 22.6.6 (cosine similarity kernel). *Cosine similarity kernel $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ of degree k is defined as*

$$k(x_i, x_j) = \frac{\left\langle x_i, x_j \right\rangle}{\|x_i\|_2 \left\|x_j\right\|_2}.$$

Remark 22.6.7. Cosine similarity kernel can be viewed as the normalized version of linear kernel. Cosine similarity kernel is commonly used in text analytics to compute the similarity between tf-idf vectors.

Definition 22.6.7 (polynomial kernel). *Polynomial kernel $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ of degree k is defined as*

$$k(x_i, x_j) = (\left\langle x_i, x_j \right\rangle + c)^d = \sum_{s=0}^{d} \binom{d}{s} c^{d-s} \left\langle x_i, x_j \right\rangle^s$$

where $c \in \mathbb{R}$

Remark 22.6.8 (adjust weighting on feature map). We can see that by changing the value of c, we can adjust weight on terms of the associated polynomial. The larger value of c, the heavier weight is on higher order terms.

Lemma 22.6.2 (dimensionality of induced feature space). *[11, p. 293] The dimensionality of the feature space for the polynomial kernel of degree d is $\binom{n+d}{d}$ where n is the dimension of the input space.*

Proof. For $n = 1$, the number is correctly computed as $d = 1$ from the binomial expansion. The correctness of the expression can be proved by induction. Note that for $n = 2$, the inner product $\langle x, z \rangle = (x_1, x_2)^T (z_1, z_2) = x_1 z_1 + x_2 z_2$, and the power on this will significantly increase number of terms and the dimension of feature space. □

Definition 22.6.8 (Gaussian kernel). *Gaussian kernel $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is defined as*

$$k(x_i, x_j) = \exp(-\frac{\left\| x_i - x_j \right\|^2}{2\sigma^2})$$

Remark 22.6.9 (infinite dimensionality of induced feature space). If we take $\sigma = 1$, the we can expand the Gaussian kernel (via Taylor series) as

$$k(x_i, x_j) = \exp(-\left\| x_i - x_j \right\|^2) = \exp(-x_i^2) \exp(-x_j^2) \sum_{j=0}^{\infty} \underbrace{\frac{2^k (x_i)^k (x_j)^k}{k!}}_{\exp(2x_i x_j)}.$$

Therefore, we can view the feature map is an infinite dimensional map such that

$$\phi(x) = (\frac{2^0 \exp(-x^2) x^0}{0!}, \frac{2^1 \exp(-x^2) x}{1!}, ..., \frac{2^k \exp(-x^2) x^k}{k!}, ...).$$

Remark 22.6.10 (when to use Gaussian kernel). Gaussian kernel has the effect of penalizing large dissimilarity between samples and only maintains similarity measure resulting from 'close/nearby' samples in the feature space.

22.6.4 Kernel trick and elementary algorithms using kernels

"kernel trick"

Given an algorithm formulated in terms of **positive definite kernel** k, then one can construct an alternative algorithm by replacing k by another positive definite kernel k'.

Remark 22.6.11 (dot product to kernel). If the original k is the dot product (which is a linear kernel) between input vectors, then we can replace the dot product by other positive definite kernel(for example, nonlinear kernel).

22.6.5 Elementary algorithms

Theorem 22.6.2 (elementary algorithms). *[11, p. 114] Given a finite subset $S = \{x_1, x_2, ..., x_n\}$ of an input space X and a kernel $k : X^2 \to \mathbb{R}$ satisfying $k(x, x') = \langle \phi(x), \phi(x') \rangle$ for some feature map $\phi : X \to H$: we are able to calculate the following quantity using kernels alone:*

- *The norm of feature vector:* $\|\phi(x)\|_2 = k(x, x)$
- *The norm of linear combination of feature vector:*

$$\left\| \sum_i \alpha_i \phi(x_i) \right\|^2 = \sum_i \sum_j \alpha_i \alpha_j k(x_i, x_j)$$

- *Distance between feature vectors:* $\|\phi(x) - \phi(z)\| = k(x, x) + k(z, z) - 2k(x, z)$
- *The norm of the center mass:*

$$\|\phi_C\|_2^2 = \frac{1}{n^2} \sum_i \sum_j k(x_i, x_j)$$

  where $\phi_C = \frac{1}{n} \sum_i \phi(x_i)$.
- *The average distance from the center mass*

$$\frac{1}{n} \sum_{i=1}^{n} \|\phi(x_C) - \phi_C\|^2 = \frac{1}{n} \sum_{i=1}^{n} k(x_i, x_i) - \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} k(x_i, x_j),$$

  where $x_C = \frac{1}{n} \sum_{i=1}^{n} x_i$.

Proof. These can be proved directly by replacing $\langle \phi_i, \phi_j \rangle = k(x_i, x_j)$. □

Lemma 22.6.3 (centering the feature vectors). *[12, p. 496][11, p. 115]Given a finite subset $S = \{x_1, x_2, ..., x_n\}$ of an input space X and a kernel $k : X^2 \to \mathbb{R}$ satisfying $k(x, x') = \langle \phi(x), \phi(x') \rangle$ for some feature map $\phi : X \to H$: we can centering $K_{ij} = \left\langle \phi(x_i), \phi(x_j) \right\rangle$ to $K'_{ij} = \left\langle \phi(x_i) - \phi_C, \phi(x_j) - \phi_C \right\rangle$ via*

$$K' = HKH, H = I - 1_n 1_n^T / n,$$

where $\phi_C = \frac{1}{n} \sum_i \phi(x_i)$.

Proof. Note that

$$\begin{aligned} K'_{ij} &= \left\langle \phi_i - \phi_C, \phi_j - \phi_C \right\rangle \\ &= \left\langle \phi_i, \phi_j \right\rangle - \langle \phi_i, \phi_S \rangle - \left\langle \phi_j, \phi_C \right\rangle + \langle \phi_C, \phi_C \rangle \\ &= k(x_i, x_j) - \frac{1}{n} \sum_k k(x_i, x_k) - \frac{1}{n} \sum_k k(x_j, x_k) + \frac{1}{n^2} \sum_k \sum_l k(x_k, x_l) \end{aligned}$$

□

22.7 Note on bibliography

For excellent coverage on machine learning, see [13][12][14][2][13].

For kernel method, see [11].

For feature engineering, see [3][4][15] and link. Particularly, see [5].

For imbalanced data learning, see [16].