DSA5105 Principles of Machine Learning

AY2024/25 Sem1 By Zhao Peiduo

Supervised Learning

Lecture 1

Supervised Learning Supervised learning is the most common type of machine learning problem, where the goal is to make predictions and learn a function that maps an input to an output based on example input-output pairs.

Problem Setup Given a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$, where x_i are the inputs and y_i are the corresponding outputs or labels, the goal is to learn the mapping $f : X \to Y$ such that $f(x_i) \approx y_i$ for all i.

Hypothesis Space The **oracle** $f*$ is unknown to us except through the dataset. The function f is chosen from a hypothesis space $\mathcal{H}$, which is a set of candidate functions. For example, in linear regression, the hypothesis space might be $\mathcal{H} = \{f : f(x) = w_0 + w_1 x \mid w_0, w_1 \in \mathbb{R}\}$.

Three Paradigms of Supervised Learning

- **Approximation:** Analyze the breadth and depth of the hypothesis space $\mathcal{H}$ to determine if it contains, or closely approximates, the optimal function. (How large is our hypothesis space?)
- **Optimization:** Design and implement efficient algorithms to address the empirical risk minimization problem and find or closely approximate the best function within $\mathcal{H}$. (How can we find or get close to an approximation $\hat{f}$ of f^*?)
- **Generalization:** Evaluate whether the optimized model can effectively generalize to new, unseen data, focusing on the interplay between data size and model complexity. This is done through minimizing the population risk: $R_{\text{pop}}(f) = \mathbb{E}_{(x,y)\sim P}[L(y, f(x))]$.(Can the $\hat{f}$ found generalized to unseen examples?)

Empirical Risk Minimization (ERM) The learning process aims to find a function $f \in \mathcal{H}$ that minimizes the empirical risk, $R_{\text{emp}}(f) = \frac{1}{N}\sum_{i=1}^N L(y_i, f(x_i))$, where $y_i = f^*(x_i)$.

Loss Function To quantify how well a function f fits the data, we use a loss function $L(y, \hat{y})$, where y is the true output, and $\hat{y} = f(x)$ is the predicted output. Common loss functions include the mean squared error (MSE) for regression tasks:$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$

Ordinary Least Squares Formula The formula for Ordinary Least Squares in a simple linear regression context is given by: $\beta = (X^T X)^{-1} X^T y$. In 1D context, the empericial risk minization problem can be defined as $\min_{w_0,w_1} \frac{1}{2N}\sum_{i=1}^N (w_0 + w_1 x_i - y_i)^2$. By setting the partial derivatives to zero, the ordinary least sqaures formula 1D is given by: $\hat{w}_0 = \bar{y} - \hat{w}_1 \bar{x}$ $\hat{w}_1 = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2}$ $\bar{x} = \frac{1}{N}\sum_i x_i$ $\bar{y} = \frac{1}{N}\sum_i y_i$

Huber Loss The Huber loss, used for robust regression, is defined as: $L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \le \delta \\ \delta|y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$

General Ordinary Least Squares Formula Consider $x \in \mathbb{R}^d$ and the new hypothesis space $\mathcal{H}_M = \left\{f : f(x) = \sum_{j=0}^{M-1} w_j \varphi_j(x)\right\}$ Each $\varphi_j : \mathbb{R}^d \to \mathbb{R}$ is called a **basis function** or **feature map**.

We can rewrite the ERM $\min_{w\in\mathbb{R}^M} \frac{1}{2N}\sum_{i=1}^N \left(\sum_{j=0}^{M-1} w_j \varphi_j(x_i) - y_i\right)^2$ into $\min_{w\in\mathbb{R}^M} \frac{1}{2N}\|\Phi w - y\|^2$. Solving by setting $\nabla R_{\text{emp}}(\hat{w}) = 0$, we have $\hat{w} = (\Phi^\top \Phi)^{-1}\Phi^\top y$, given invertible $\Phi^T\Phi$.

For cases where $\Phi^T\Phi$ is not invertible, the formula using the Moore-Penrose pseudoinverse is: $\hat{w}(u) = \Phi^\dagger y + \left(I - \Phi^\dagger\Phi\right)u \quad u \in \mathbb{R}^M$

Overfitting: Overfitting occurs when the hypothesis space $\mathcal{H}$ is too large, allowing the model to fit the noise in the training data. This results in poor generalization to new data.

Regularization: To prevent overfitting, regularization techniques add a penalty to the loss function: $\min_{w\in\mathbb{R}^M} \frac{1}{2N}\|\Phi w - y\|^2 + \lambda C(w)$ $\frac{1}{2N}\|\Phi w - y\|^2 + \lambda C(w)$. Minimizing the ERM we get $\hat{w} = \left(\Phi^\top\Phi + \lambda I_M\right)^{-1}\Phi^\top y$ which is always invertible for positive λ.

Common regularization terms:

- **L_2 (Ridge) regularization:** $\lambda\sum_{j=1}^p w_j^2$
- **L_1 (Lasso) regularization:** $\lambda\sum_{j=1}^p |w_j|$

Softmax Function For a multi-class classification problem with K classes, the softmax function is defined as:$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^K ;\exp(z_j)}$

Cross-Entropy Loss Commonly used in classification tasks, the cross-entropy loss is: $L(y, p) = -\sum_i y_i \log(p_i)$

Lecture 2

Another view of feature maps One can also view feature maps as implicitly defining some sort of **similarity measure**.Consider two vectors u and v. Then, $u^T v$ measures how similar they are. Feature maps define a **similarity** between two samples x, x' by computing the dot product in **feature space**.

Reformulation of Ridge Regression We rewrite the regularized least squares solution in another way:

$\hat{w} = \left(\Phi^\top\Phi + \lambda I_M\right)^{-1}\Phi^\top y = \Phi^\top\left(\Phi\Phi^\top + \lambda I_N\right)^{-1} y$

Proof:

$\left(\Phi^\top\Phi + \lambda I_M\right)\Phi^\top = \Phi^\top\left(\Phi\Phi^\top + \lambda I_N\right)^{-1}$

$= \Phi^\top\left(\Phi\Phi^\top + \lambda I_N\right)^{-1} y$

$= \Phi^\top\Phi^\top y\left(\Phi^\top + \lambda I_N\right)^{-1}$

$\lambda I_N^{-1} y = \left(\Phi^\top\Phi + \lambda I_M\right)^{-1}\Phi^\top y$

Reformulation of Ridge Regression

$\hat{f}(x) = \sum_{i=1}^N \alpha_i \varphi(x_i)^\top \varphi(x)$

$\alpha = (G + \lambda I_N)^{-1} y$ where $G_{ij} = \varphi(x_i)^\top\varphi(x_j)$ is the gram matrix

Kernel Ridge Regression Essentially, the reformulation computes the similarity score between x and x', which can be replaced by a kernel function $K(x_i, x_j)$, allowing computation in high-dimensional feature spaces without explicit feature transformations. The solution to kernel ridge regression is: $f(x) = \sum_{i=1}^N \alpha_i K(x_i, x)$ where α_i are coefficients determined based on the training data and the kernel.

Kernel Construction To build a kernel, we can write an expression in the form of a dot product of the same function with variable x and x', but this method does not always work.

Mercer's Theorem and SPD Kernels Suppose k is a SPD kernel. Then, there exists a feature space $\mathcal{H}$ and a feature map $\varphi : \mathbb{R}^d \to \mathcal{H}$ such that $k(x, x') = \varphi(x)^\top\varphi(x')$

SPD kernels properties:

- **Symmetry:** $K(x, x') = K(x', x)$
- **Positive Semi-Definiteness:** $\sum_{i,j}\alpha_i\alpha_j K(x_i, x_j) \ge 0$ for any α_i

Examples of SPD Kernels

- **Linear Kernel:** $K(x, x') = x^\top x'$
- **Polynomial Kernel:** $K(x, x') = (1 + x^\top x')^d$
- **Gaussian RBF Kernel:** $K(x, x') = \exp\left(-\frac{\|x-x'\|^2}{2\sigma^2}\right)$

Constructing kernels Given valid kernels $k_1(x, x')$ and $k_2(x, x')$, the following new kernels will also be valid:

$k(x, x') = ck_1(x, x')$ $k(x, x') = f(x)k_1(x, x')f(x')$

$k(x, x') = q(k_1(x, x'))$ $k(x, x') = \exp(k_1(x, x'))$

$k(x, x') = k_1(x, x') + k_2(x, x')$ $k(x, x') = k_1(x, x')k_2(x, x')$

$k(x, x') = k_3(\varphi(x), \varphi(x'))$ $k(x, x') = x^\top Ax'$

$k(x, x') = k_a(x_a, x'_a) + k_b(x_b, x'_b)$ $k(x, x') = k_a(x_a, x'_a)k_b(x_b, x'_b)$

Lecture 3

Support Vector Machines (SVM)

Support Vector Machines (SVMs) are designed for binary classification. They find a hyperplane that separates two classes with the maximum margin, defined as the minimum distance between the separating hyperplane and the data points.

Optimization Problem $\min_{w,b}\frac{1}{2}\|w\|^2$ subject to: $y_i(w^\top x_i + b) \ge 1 \quad \forall i$. Introducing Lagrange multipliers $\alpha_i \ge 0$, the Lagrangian is: $\mathcal{L}(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^N \alpha_i[y_i(w^\top x_i + b) - 1]$

Karush-Kuhn-Tucker (KKT) Conditions: Define the Lagrangian $\mathcal{L}(z, \mu) = F(z) + \mu^T G(z)$. Then, under technical conditions, for each locally optimal $\hat{z}$, there exists Lagrange multipliers $\hat{\mu} \in \mathbb{R}^m$ such that:

stationarity $\nabla_z\mathcal{L}(\hat{z}, \hat{\mu}) = 0$
Primal Feasibility $G(\hat{z}) \le 0$
Dual Feasibility $\hat{\mu} \ge 0$
Complementary Slackness $\hat{\mu}_j G_j(\hat{z}_j) = 0$

Dual Problem $\max_{\mu\ge 0}\tilde{F}(\mu)$ where $\tilde{F}(\mu) = \min_z \mathcal{L}(z, \mu)\mathcal{L}(z, \mu) = F(z) + \mu^T G(z)$

KKT conditions for SVM

1. From Stationarity $\hat{w} = \sum_{i=1}^N \hat{\mu}_i y_i x_i, \quad 0 = \sum_{i=1}^N \hat{\mu}_i y_i$

2. From Dual Feasibility $\hat{\mu}_i \ge 0$ for $i = 1, \ldots, N$

3. From Complementary Slackness$\hat{\mu}_i = 0$ or $y_i(\hat{w}^T x_i + \hat{b}) = 1$

4. The multipliers $\hat{\mu}$ can be found by the dual problem $\max_{\mu\in\mathbb{R}^N}\sum_{i=1}^N \mu_i - \frac{1}{2}\sum_{i,j}\mu_i\mu_j y_i y_j (x_i^T x_j)$

Dual formulation of SVM

$\hat{\mu} = \arg\max_{\mu\in\mathbb{R}^N}\sum_{i=1}^N \mu_i - \frac{1}{2}\sum_{i,j}\mu_i\mu_j y_i y_j (x_i^T x_j)$

Subject to: $\hat{\mu} \ge 0$ and $\sum_{i=1}^N \hat{\mu}_i y_i = 0$

Decision function: $\hat{f}(x) = \text{sgn}\left(\sum_{i=1}^N \hat{\mu}_i y_i x_i^T x + \hat{b}\right)$

Complementary slackness: $\hat{\mu}_i = 0$ or $1 = y_i(\hat{w}^T x_i + \hat{b})$

Kernel Support Vector Machines

$\hat{\mu} = \arg\max_{\mu\in\mathbb{R}^N}\sum_{i=1}^N \mu_i - \frac{1}{2}\sum_{i,j=1}^N \mu_i\mu_j y_i y_j k(x_i, x_j)$

Subject to: $\hat{\mu} \ge 0$ and $\sum_{i=1}^N \hat{\mu}_i y_i = 0$

Decision function: $\hat{f}(x) = \text{sgn}\left(\sum_{i=1}^N \hat{\mu}_i y_i k(x_i, x) + \hat{b}\right)$

Only support vectors satisfying $1 = y_i(\hat{w}^T\varphi_i(x) + \hat{b})$ matter for predictions. This is a **sparse kernel method.**;

Lecture 4

Decision Trees Decision trees are very simple and useful ways to build models. Key ideas:

- **Stratify** the input space into distinct, non-overlapping regions
- Assign a chosen, **constant** prediction to each region

Universal approximation theorem In general, any (sufficiently regular) oracle function $f^* : \mathcal{X} \to \mathbb{R}$ can be approximated with a decision tree with an arbitrary small precision $\epsilon > 0$. i.e. there exists a decision tree T (with some finite depth) such that $|T(x) - f^*(x)| < \epsilon$. Want ϵ to be small $\Rightarrow$ choose larger L

Classification and Regression Trees Suppose that the input space is $\mathcal{X}$. A **partition** of $\mathcal{X}$ is a collection of subsets $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_J$ such that $\mathcal{R}_i \cap \mathcal{R}_j = \emptyset$ for $i \neq j$ and $\bigcup_{j=1}^{J} \mathcal{R}_j = \mathcal{X}$

The general decision tree hypothesis space is: $\mathcal{H} = \left\{ f : f(x) = \sum_{j=1}^{J} a_j \mathbb{I}_{x \in \mathcal{R}_j}, \{\mathcal{R}_j\} \text{ is a partition of } \mathcal{X}, a_j \in \mathcal{Y} \right\}$

where $\mathbb{I}_{x \in \mathcal{R}_j} = \begin{cases} 1, & \text{if } x \in \mathcal{R}_j \\ 0, & \text{otherwise} \end{cases}$

Learning Decision Trees

A decision tree model $f(x) = \sum_{j=1}^{J} a_j \mathbb{I}_{x \in \mathcal{R}_j}$ depends on both a_j and $\mathcal{R}_j$. Given $\{\mathcal{R}_j\}$, $\{a_j\}$ are easy to fix:

- **Regression**: we take the **average** label values $a_j = y_j = \dfrac{\sum_i y_i \mathbb{I}_{x \in \mathcal{R}_j}}{\sum_i \mathbb{I}_{x \in \mathcal{R}_j}}$
- **Classification**: we take the **modal** label values $a_j = \text{mode}\{y_i : x_i \in \mathcal{R}_j\}$

Suppose we are dealing with regression, then we can fix a_j as before and solve the following empirical risk minimization: $\min_{\mathcal{R}_j} \frac{1}{2} \sum_{i=1}^{N} \left(\sum_{j=1}^{J} a_j \mathbb{I}_{x \in \mathcal{R}_j} - y_i \right)^2$. However this is very hard to solve (NP-hard).

Recursive Binary Splitting Greedy algorithm which essentially repeats the following two steps and grows the tree by adding two leaf nodes at a time:Pick a dimension of the input space, then find the **best** value θ to split this input dimension into two parts and assign new constant values to these new regions.

Decision Trees for Classification The greedy algorithm can be carried out analogously, except that we need to define a proper loss function:

Entropy: $-\sum_{k=1}^{K} \sum_{j=1}^{J} p_{jk} \log p_{jk}$

Gini Impurity: $\sum_{k=1}^{K} \sum_{j=1}^{J} p_{jk}(1 - p_{jk})$

Where p_{jk} is the proportion of samples in $\mathcal{R}_j$ belonging to class k.

Advantages and Disadvantages of Decision Trees

Advantages: 1.Can readily visualize and understand predictions; 2.Implicit feature selection via analyzing contribution of splits to reduction of error/impurity; 3.Robust to data types, supervised learning tasks, and nonlinear relationships

Disadvantages: 1.Greedy algorithms may find sub-optimal solutions; 2.Sensitive to data variation and balancing; 3.Prone to overfitting.

Model Ensembling Bagging reduces the variance by aggregating predictions from multiple models trained on different random subsamples of the data:

- **Regression** $\bar{f}(x) = \frac{1}{m} \sum_{j=1}^{m} f_j(x)$
- **Classification** $\bar{f}(x) = \text{Mode}\{f_j(x) : j = 1, \ldots, m\}$

Boosting works by training weak learners sequentially, where each learner focuses on correcting the mistakes of the previous ones. The combined model is a weighted sum of the weak learners: $f(x) = \sum_{t=1}^{T} \alpha_t f_t(x)$ where α_t are coefficients based on each learner's performance. Boosting helps reduce bias.

Key Ideas of AdaBoost 1. Initialize with uniform weight across all training samples
2. Train a classifier/regressor f_1
3. Identify the samples that f_1 got wrong (classification) or has large errors (regression)
4. Weight these samples more heavily and train f_2 on this reweighted dataset
5. Repeat steps 3-5

AdaBoost Implementation

- **Data:** $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$
- Initialize $w_i^{(1)} = \frac{1}{N}$ for all $i = 1, \ldots, N$;
- For $j = 1, \ldots, m$ do
  1. Obtain f_j from: $f_j = \arg\min_{f \in \mathcal{H}} \sum_{i=1}^{N} w_i^{(j)} \mathbb{I}_{y_i \neq f(x_i)}$
  2. Compute combination coefficients: $\delta_j = \dfrac{\sum_{i=1}^{N} w_i^{(j)} \mathbb{I}_{y_i \neq f_j(x_i)}}{\sum_{i=1}^{N} w_i^{(j)}}$ $\alpha_j = \log\left(\frac{1-\delta_j}{\delta_j}\right)$
  3. Update weights:$w_i^{(j+1)} = w_i^{(j)} \exp\left(\alpha_j \mathbb{I}_{y_i \neq f_j(x_i)}\right)$
- End loop
- Return: $\bar{f}(x) = \text{Sign}\left(\sum_{j=1}^{m} \alpha_j f_j(x)\right)$

Cross-Validation Cross-validation is essential for tuning model hyperparameters. In k-fold cross-validation, the data is split into k subsets. The model is trained on $k-1$ subsets and validated on the remaining one. This process is repeated k-times, and the average performance is evaluated. Cross-validation ensures model generalizability.

Lecture 5

Underfitting A strongly biased solution with low variance.

Overfitting No strong bias but with very high variance.

Bias The difference between expected prediction and the true value: It measures how well you expect to represent the true solution and decreases with more complex models. $\text{bias}^2 = \int_x \{\mathbb{E}_D[f(x)] - y\}^2 \, p(x) \, dx$

Variance The difference between what you expect to learn, i.e., $\bar{f}$, and what you learn from a particular dataset. It measure how sensitive the learner is to a specific dataset and decreases with simpler models. $\text{variance} = \int_x \mathbb{E}_D \left[(f(x) - \bar{f}(x))^2 \right] p(x) \, dx$

Bias-Variance Tradeoff The bias-variance tradeoff is critical for understanding model performance:
$\text{MSE} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$

Neural Networks The neural network (NN) hypothesis space is quite like linear basis models: $f(x) = \sum_{j=1}^{M} v_j \, \phi_j(x)$, where $\phi_j(x) = \sigma(w_j^T x + b_j)$.

$w_j \in \mathbb{R}^d$: the trainable weights of the hidden layer
$b_j \in \mathbb{R}$: the trainable biases of the hidden layer
$v_j \in \mathbb{R}$: the trainable weights of the output layer
$\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function.

Activation Functions

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$, $[0, 1]$

Tanh: $\sigma(z) = \tanh(z)$, $[-1, 1]$

Rectified Linear Unit (ReLU): $\sigma(z) = \max(0, z)$, $[0, \text{inf}]$

Leaky-ReLU: $\sigma(z) = \begin{cases} z & \text{if } z \geq 0 \\ \delta z & \text{if } z < 0 \end{cases}$, $[-\text{inf}, inf]$

Universal Approximation Theorem Any continuous function f^* on a compact domain can be approximated by neural networks to arbitrary precision, provided there are enough neurons (M large enough).

Curse of Dimensionality: Expotential increase in number of neurons required as the dimensionality increases. Baron(1993) showed that for any continuous function $f^* : [0, 1]^d \to \mathbb{R}$ (with other conditions), there exists a width-M neural network f_M such that $\|f^* - f_M\|^2 \leq \mathcal{O}(M^{-1})$

The Significance of Data-Dependent Feature Maps

Functions behave just like vectors:

- Each ϕ_j is like a coordinate axis. They play the role of e_j. **Important difference**: there are an **infinite** number of them.
- The oracle function f^* plays the role of u.

Writing $f^*(x) = \sum_{j=1}^{M} v_j \phi_j(x)$ is like expanding a vector into its components, but we can't have all components since M is finite. If we get to choose which components to have in the sum **after** seeing some information on f^*, we can usually do much better.

Gradient Descent $w^{(t+1)} = w^{(t)} - \eta \frac{\partial L}{\partial w}$ When η is too small, updates are **slow**; When η is too large, the updates may become **unstable**.

Stochastic Gradient Descent (SGD): $\theta_{k+1} = \theta_k - \eta \frac{1}{B} \sum_{i \in I_B} \nabla \Phi_i(\theta_k)$, this diminishes the probability of stucking at local minima as the main drawback for GD.

Back-propagation Algorithm

- Initialize $x_0 = x \in \mathbb{R}^d$.
- For $t = 0, 1, \ldots, T$: $x_{t+1} = g_t(x_t, W_t) = \sigma(W_t x_t)$
- Set $p_{T+1} = \nabla_x L(x_{T+1}, y)$.
- For $t = T, T-1, \ldots, 1$:
  - $\nabla_{W_t} \Phi = p_{t+1}^T \nabla_W g_t(x_t, W_t)$
  - $p_t = [\nabla_x g_t(x_t, W_t)]^T p_{t+1}$
- Return $\{\nabla_{W_t} \Phi : t = 0, \ldots, T\}$.

Unsupervised Learning

Lecture 6

Principal Component Analysis Two formulations:

- Find the direction that captures the most variance, where the order in magnitude of the normalized sample covariance matrix S corresponds to the order of eigenvalues. λ
- Find the direction that minimizes projection error by partitioning the set of eigenvectors and choosing the exclusion set (m+1 to d) to be the eigenvector subset corresponding to the smallest (d - m - 1) eigenvalues.

PCA Algorithm Implementation

- **Data**: $\mathcal{D} = \{x_i\}_{i=1}^{N}, x_i \in \mathbb{R}^d$ for all i.
- **Hyper-parameters**: m (reduction dimension).
- Compute sample covariance matrix $S = \frac{1}{N} \sum_{i=1}^{N} x_i x_i^T$.
- Compute the first m eigenvectors $\{u_1, \ldots, u_m\}$ and eigenvalues $\{\lambda_1, \ldots, \lambda_m\}$.
- Form $d \times m$ matrix U_m whose j^{th} column is u_j.
- Compute $Z_m = X U_m$.
- Return Principal component scores Z_m, Eigenvalues, and eigenvectors λ_j, u_j for $j = 1, \ldots, m$.