Lecture 2, Jan 12, 2024

Supervised Learning

- Denote the input (aka features) $\boldsymbol{x} = \{ x_1, x_2, \ldots, x_D \}^T \in \mathcal{X} \subseteq \mathbb{R}^D$
  - Extraction of relevant features from data is often necessary (*feature engineering*)
- Denote the target (aka labels) $y \in \mathcal{Y}$
  - Regression: $\mathcal{Y} \subseteq \mathbb{R}$ or $\mathbb{R}^K$
  - Binary classification: $\mathcal{Y} = \{ -1, +1 \}$
  - Multi-class classification: $\mathcal{Y} = \{ 1, 2, \ldots, K \}$
    - This can be decomposed into a sequence of binary classification problems
    - We usually use a one-hot encoding (a K-dimensional target with a 1 in the desired class and 0s elsewhere)
- The probability distribution that the targets and inputs are sampled from is denoted $\mathcal{P}(X, Y), \Pr(X, Y)$ or $p(\boldsymbol{x}, y) = p(x_1, \ldots, x_d, y)$ (joint density of features and targets)
- The expectation is denoted $\mathbb{E}_{\boldsymbol{x}}[g(\boldsymbol{x})] = \int p(\boldsymbol{x}) g(\boldsymbol{x}) \, \mathrm{d}x, \mathbb{E}_y[g(y)|\boldsymbol{x}] = \int p(y|\boldsymbol{x}) g(y) \, \mathrm{d}y$

Definition

Supervised learning: Given the training dataset

$$\mathcal{D} := \{ (\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), \ldots, (\boldsymbol{x}^{(N)}, y^{(N)}) \}$$

with

$$y^{(i)} = f(\boldsymbol{x}^{(i)}) + \epsilon$$

where f is the function we wish to learn and ϵ is some measurement noise; the goal is to find a function $\hat{f} \colon \mathcal{X} \mapsto \mathcal{Y}$ such that

$$\hat{f}(\boldsymbol{x}^{(i)}) \approx f^{(i)} \forall (\boldsymbol{x}^{(i)}, y^{(i)})$$

for both points in $\mathcal{D}$ and outside.

- To make this problem tractable, we need additional assumptions
  - This is due to the No Free Lunch theorem: no single model is best for all problems!
  - We restrict $\hat{f}$ to a set of possible functions that we refer to the *hypothesis class* $\mathcal{H}$ (*model structure specification*)
  - We attempt to solve $\hat{f} = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \mathcal{L}(h)$ where $\mathcal{L} \colon \mathcal{H} \mapsto \mathbb{R}$
- If the hypothesis class is parameterized by $\boldsymbol{w}$, we are essentially solving $\hat{\boldsymbol{w}} = \underset{\boldsymbol{w}}{\operatorname{argmin}} \mathcal{L}(h_{\boldsymbol{w}})$
  - Hypothesis classes can be e.g. neural networks, polynomials, etc
  - e.g. $\mathcal{H}_{\boldsymbol{w}} := \left\{ w_0 + \sum_{i=1}^{D} w_i x_i, \boldsymbol{w} \in \mathbb{R}^{D+1} \right\}$
- The loss function is an expectation with respect to the joint distribution of inputs and outputs: $\mathcal{L}(h) = \mathbb{E}[l(h(\boldsymbol{x}), y)]$
  - Examples:
    - Squared loss: $l(y, y') = (y - y')^2$
    - Absolute loss: $l(y, y') = |y - y'|$
      - This is less affected by outliers compared to squared loss
    - Huber loss: $l(y, y') = \begin{cases} \frac{1}{2}(y - y')^2 & |y - y'| \leq \delta \\ \delta(|y - y'| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$
    - ϵ-insensitive loss: $l(y, y') = \begin{cases} 0 & |y - y'| \leq \epsilon \\ |y - y'| - \epsilon & \text{otherwise} \end{cases}$
      - Useful in regression tasks where some degree in error tolerance is acceptable

  - For classification, y' is the raw output of the classifier (not the output label, so this can include confidence):
    * Zero-one loss: $l(y, y') = \begin{cases} 1 & y \neq y' \\ 0 & \text{otherwise} \end{cases}$
      - Correctly classified points that are far from the decision boundary (i.e. very confident) are not penalized
      - i.e. this doesn't care how confident we are
    * Hinge loss: $l(y, y') = \max(0, 1 - yy')$
      - Correctly classified points that are close to the decision boundary are penalized
- The actual loss function we want to minimize is the generalized $\mathcal{L}(h) = \mathbb{E}_{(\boldsymbol{x},y)\sim\Pr(\boldsymbol{x},y)}[l(h(\boldsymbol{x}), y)] = \mathcal{L}_{\text{gen}}(h)$
  - Consider the squared error loss; it can be rewritten as $\mathbb{E}_{\boldsymbol{x}}[\mathbb{E}_y[h(\boldsymbol{x})^2 + y^2 - 2h(\boldsymbol{x})y|\boldsymbol{x}]]$
  - We can rearrange the inner expectation as $(h(\boldsymbol{x}) - \mathbb{E}[y|\boldsymbol{x}])^2 + \text{Var}(y|\boldsymbol{x})$
  - The first term can be minimized by choosing $h(\boldsymbol{x}) = \mathbb{E}[y|\boldsymbol{x}]$
  - The second term does not depend on $h(\boldsymbol{x})$; it cannot be minimized because it is the intrinsic variance of the outputs
    * This is the *Bayes error*
  - An algorithm that achieves the Bayes error is *Bayes optimal*; this is not something we can do in practice (if we had all the information about the distribution, we wouldn't need to learn in the first place)
- Let the optimal predictor be $h_*(\boldsymbol{x}) = \mathbb{E}[y|\boldsymbol{x}]$; for any dataset we can run our learning algorithm to get a particular $h(\boldsymbol{x}; \mathcal{D})$
  - Let's take the expectation of the error over all choices of datasets
  - We can rewrite $\mathbb{E}_{\mathcal{D}}[(h(\mathcal{D}) - h_*)^2 + \text{Var}(y)] = (\mathbb{E}_{\mathcal{D}}[h] - h_*)^2 + \text{Var}(h) + \text{Var}(y)$
  - So $\mathbb{E}_{\mathcal{D}}[\mathbb{E}_y[(h(\boldsymbol{x}; \mathcal{D}) - y)^2|\boldsymbol{x}]] = \underbrace{(\mathbb{E}_{\mathcal{D}}[h] - h_*)^2}_{\text{bias}} + \underbrace{\text{Var}(h)}_{\text{variance}} + \underbrace{\text{Var}(y)}_{\text{Bayes error}}$
  - The loss is now decomposed into 3 terms:
    * The *bias* indicates how the average prediction over all datasets differs from the optimal predictor
    * The *variance* indicates how sensitive $h(\boldsymbol{x})$ is to the choice of a particular dataset
    * The *Bayes error* is irreducible noise that is intrinsic to the data generation process
  - There is often a tradeoff between bias and variance; lower bias usually result in high variance and vice-versa
    * Often high bias and high variance are used as synonyms for underfitting and overfitting (even though this technically only applies for squared loss)
- However we can't actually compute $\mathcal{L}_{\text{gen}}(h)$ since we don't know the underlying distribution; we can approximate it using the *empirical loss*: $\mathcal{L}(h) \approx \frac{1}{N}\sum_{i=1}^{N} l(h(\boldsymbol{x}^{(i)}), y^{(i)}) = \mathcal{L}_{\text{emp}}(h)$
  - Minimization of the empirical loss is known as *empirical risk minimization*
  - Convergence in the limit $N \to \infty$ holds due to the weak law of large numbers
- The empirical loss measures performance only on the training set $\mathcal{D}$; this means that the training error can be reduced to zero simply by memorizing the entire training dataset
  - Such a "memorizer" model is useless for predicting on new data, so it's undesirable
  - To guarantee that the out-of-sample error (i.e. the error for data not in $\mathcal{D}$) is low, we need to minimize the generalized loss
  - But we can't actually compute the generalized loss, so much of the focus of theoretical research is on bounding the generalized loss in terms of the empirical loss
- An overly flexible model will memorize irrelevant details of the training set (*overfitting*) whereas simpler models don't have enough degrees of freedom to approximate the underlying function (*underfitting*)
  - Generally if two models fit the data equally well, the simpler model probably generalizes better
- To prevent overfitting, standard practice splits $\mathcal{D}$ into training, validation, and testing sets
  - This works when we have a large amount of data so we can afford to reduce the training dataset
  - The best way to partition is dependent on the problem and size of the dataset available

  - The training set is used to fit the model parameters
  - The validation set is used to select model complexity (i.e. hyperparameters)
  - The testing set is used to estimate the generalization performance
- For smaller datasets, another popular approach is ν-fold cross-validation
  - $\mathcal{D}$ is spit into ν equal partitions/folds
  - For $i = 1, \ldots, \nu$, train models on data in all folds except the i-th, and test on the i-th fold
  - For each model the average loss over all ν folds is calculated
  - Large values of ν typically lead to a large increase in computational cost, but this can be parallelized
  - The choice of ν is dictated by the bias-variance tradeoff; typical values are 5 or 10
    * Increasing ν leads to less bias (since we are averaging over more terms)
    * For $\nu = N$ this is called *leave-one-out error estimation*
- Other approaches to improving generalization:
  - Prior knowledge (feature engineering)
  - Getting more data
  - Fake more data (e.g. adding noise)
  - Ensembles (e.g. bootstrap)
  - Bayesian approaches

Ensembles: Bootstrap Aggregation (Bagging)

- Take the dataset and generate M new datasets by sampling N training examples from $\mathcal{D}$ with replacement
- Train the model separately on each of the M datasets to get models h_i, $i = 1, \ldots, M$
- Average the predictions of models trained on each of the M datasets: $h_{\text{bootstrap}}(\boldsymbol{x}) = \dfrac{1}{M} \displaystyle\sum_{i=1}^{M} h_i(\boldsymbol{x})$
- The bias remains unchanged: $E_{\mathcal{D}} \left[\dfrac{1}{M} \displaystyle\sum_{i=1}^{M} h_i \right] = \dfrac{1}{M} \displaystyle\sum_{i=1}^{M} \mathbb{E}_{\mathcal{D}}[h_i] = E_{\mathcal{D}}[h]$
- The variance can be shown to be $\text{Var}(h) = \dfrac{1}{M}(1 - \rho)\sigma^2 + \rho\sigma^2$ where ρ is the pairwise correlation coefficient between models and σ^2 is the variance
  - If we can reduce the correlation between models, i.e. $\rho \to 0$, we can approach a variance reduction of $\dfrac{1}{M}$
  - This is difficult to do but even if we can't reduce ρ to 0, bootstrapping generally still reduces the variance