MATH5836: Data and Machine Learning

Week 8: Ensemble Learning

Sarat Moka *UNSW, Sydney*

Key Topics

Book:

(A) Data Science and Machine Learning: Mathematical and Statistical Methods, by Kroese, Botev, Taimre, and Vaisman. Click here to download a pdf copy.

(B) (Original paper): Chen and Guestrin (2016) *XGBoost: A Scalable Tree Boosting System*, Click here.

8.1 Introduction to Ensemble Methods

- Ensemble methods are a powerful technique in machine learning that combine multiple base models to produce a single, more robust predictive model.
- The fundamental idea is that by aggregating the predictions of several models, the ensemble can achieve better generalisation performance than any individual model.
- This approach leverages the diversity among the base models to reduce variance, bias, or both.

Underlying Principle

- **Base Models (Learners):** Let $h_1, h_2, \ldots, h_M$ denote M base models trained on the same dataset $\mathcal{D}$. Each model h_i makes a prediction $h_i(x)$ for a given input x.
- **Ensemble Model:** The ensemble model H combines the predictions of the M base models. The way these predictions are combined defines the type of ensemble method. Formally, the prediction of the ensemble model $H(x)$ can be represented as:

$$H(x) = \mathcal{F}(h_1(x), h_2(x), \ldots, h_M(x)),$$

where $\mathcal{F}$ is a function that aggregates the base model predictions.

Advantages of Ensemble Methods

- **Improved Performance:** By combining multiple models, ensemble methods often achieve higher accuracy and robustness compared to single models.
- **Reduction in Overfitting:** Techniques like bagging reduce overfitting by averaging out the predictions, which helps to smooth out the errors made by individual models.
- **Flexibility:** Ensemble methods can be applied to a wide range of base models, including decision trees, neural networks, and more.

Theoretical Insights

- **Bias-Variance Trade-off:** Ensemble methods, particularly bagging and boosting, help to manage the bias-variance trade-off. Bagging primarily reduces variance, while boosting can reduce both bias and variance by focusing on difficult-to-predict instances.
- **Error Decomposition:** The prediction error of an ensemble model can be decomposed into three components: bias, variance, and noise. The goal of ensemble methods is to reduce the bias and variance components, thus minimizing the overall prediction error.
- **Diversity:** The success of ensemble methods heavily relies on the diversity among the base models. Diverse models make different errors, which can be averaged out to improve overall performance.

8.2 Types of Ensemble Methods

There are four major types of ensemble methods: bagging, pasting, boosting, and stacking.

Bagging (Bootstrap Aggregating)

- Bagging aims to reduce variance by training each base model on a different bootstrap sample (a random sample with replacement) of the original dataset. The final prediction is typically an average (for regression) or majority vote (for classification) of the base model predictions.
- **Mathematical Formulation:**
$$H(x) = \frac{1}{M}\sum_{i=1}^{M} h_i(x)$$
for regression, or
$$H(x) = \text{mode}\{h_1(x), h_2(x), \ldots, h_M(x)\}$$
for classification.

Example 8.2.1

Random Forest: The random forests, which we studied in last week's lecture, is quintessential an ensemble method that perfectly illustrates the principles of bagging (described later) and feature randomness.

- **Base Models (h_i):** The base learners, $h_1, h_2, \ldots, h_M$, are a collection of individual **decision trees**. The total number of trees, M, is a key hyperparameter denoted in exercises as 'n_estimators'.
- **Achieving Diversity:** The power of the random forests comes from ensuring the decision trees are different from one another. This is achieved in two primary ways:
  1. **Bagging:** Each tree h_i is not trained on the full dataset $\mathcal{D}$. Instead, it is trained on a bootstrap sample $\mathcal{D}_i$, which is created by sampling N data points from $\mathcal{D}$ *with replacement.*
  2. **Feature Randomness:** When constructing each tree, at every split point, the algorithm only considers a random subset of the available (randomly chosen) features to find the best split. This prevents strong predictors from dominating all trees.
- By combining many de-correlated trees, the Random Forest significantly reduces the high variance of individual decision trees, leading to a robust and accurate model with excellent generalisation performance.

Pasting

- Pasting is a method similar to bagging, but it involves creating subsets of the data without replacement. This means each subset contains unique instances from the original dataset, and no instance is repeated within a single subset.
- **Mathematical Formulation:** Given a dataset $D = \{(x_i, y_i)\}_{i=1}^N$, pasting generates M subsets $D_1, D_2, \ldots, D_M$, each created by sampling without replacement. Each base learner h_i is trained on D_i, and the final prediction is an average (for regression) or majority vote (for classification) of the base learners:

  $$H(x) = \frac{1}{M} \sum_{i=1}^{M} h_i(x)$$

  for regression, or

  $$H(x) = \text{mode}\{h_1(x), h_2(x), \ldots, h_M(x)\}$$

  for classification.

Boosting

- Boosting focuses on reducing bias by sequentially training base models. Each new model is trained to correct the errors made by the previous models. The final prediction is a weighted sum of the base model predictions.
- **Mathematical Formulation:**

  $$H(x) = \sum_{i=1}^{M} \alpha_i h_i(x),$$

  where α_i are the weights assigned to each base model, often based on their performance.

Stacking

- Stacking involves training a meta-model to combine the predictions of multiple base models. The base models are trained on the original dataset, and the meta-model is trained on the predictions of the base models.
- **Mathematical Formulation:** Let $\mathbf{H}(x) = (h_1(x), h_2(x), \ldots, h_M(x))$ be the vector of base model predictions. The meta-model g takes this vector as input:

  $$H(x) = g(\mathbf{H}(x)).$$

8.3 Out-of-Bag (OOB) Approach in Ensemble Methods

Recall the Out-of-Bag (OOB) approach from the previous week. This is a technique used to evaluate the performance of ensemble methods, specifically in bagging. It leverages the bootstrap sampling process to provide an internal estimate of the model's performance without the need for a separate validation set.

Bagging and Bootstrap Sampling

Recall that in bagging, multiple base models are trained on different bootstrap samples of the original dataset. A bootstrap sample is created by randomly selecting data points from the original dataset with replacement. Consequently, some data points are included multiple times in a bootstrap sample, while others are excluded.

Out-of-Bag Samples

For each bootstrap sample, the data points not selected are referred to as out-of-bag (OOB) samples. These OOB samples can be used to evaluate the performance of the model trained on the corresponding bootstrap sample.

Mathematically, for a dataset $D = \{(x_i, y_i)\}_{i=1}^N$:

1. Generate M bootstrap samples $D_1, D_2, \ldots, D_M$ by sampling N instances with replacement from D.
2. For each bootstrap sample D_m, identify the out-of-bag samples D_m^{OOB} which are the data points not included in D_m.
3. Train the m-th base model h_m on the bootstrap sample D_m.

OOB Prediction and Error

To estimate the performance of the ensemble using OOB samples, follow these steps:

- **OOB Predictions:** For each data point $(x_i, y_i) \in D$, collect the predictions from all base models for which (x_i, y_i) was an OOB sample. Let $\mathcal{M}_i$ be the set of models for which (x_i, y_i) is OOB.

  The OOB prediction for x_i is:

$$\hat{y}_i^{\text{OOB}} = \frac{1}{|\mathcal{M}_i|} \sum_{m \in \mathcal{M}_i} h_m(x_i)$$

where $|\mathcal{M}_i|$ is the number of models in $\mathcal{M}_i$.

- **OOB Error:** Calculate the OOB error by comparing the OOB predictions $\hat{y}_i^{\text{OOB}}$ with the true values y_i. For regression, the OOB error can be measured using Mean Squared Error (MSE):

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

For classification, the OOB error can be measured using the misclassification rate:

$$\text{Error}_{\text{OOB}} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}(\hat{y}_i^{\text{OOB}} \neq y_i)$$

where $\mathbb{I}(\cdot)$ is the indicator function, which is 1 if the condition is true and 0 otherwise.

Advantages of OOB Approach

- **No Need for a Separate Validation Set:** The OOB approach provides an internal estimate of model performance, eliminating the need to set aside a separate validation set.
- **Efficient Use of Data:** Since all data points are used both for training and validation (but in different bootstrap samples), the OOB approach makes efficient use of the available data.
- **Unbiased Estimate:** The OOB error is an unbiased estimate of the true error, as each data point is evaluated on models that have not seen it during training.

Example 8.3.1

Numerical Example: Consider a tiny dataset with 4 instances for demonstration:

$$D = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$$

Step 1: Generate Bootstrap Samples (with $M = 3$ trees)

- **Tree 1**: $D_1 = \{(x_1, y_1), (x_2, y_2), (x_2, y_2), (x_4, y_4)\}$ (OOB: (x_3, y_3))
- **Tree 2**: $D_2 = \{(x_1, y_1), (x_3, y_3), (x_3, y_3), (x_4, y_4)\}$ (OOB: (x_2, y_2))
- **Tree 3**: $D_3 = \{(x_2, y_2), (x_3, y_3), (x_4, y_4), (x_4, y_4)\}$ (OOB: (x_1, y_1))

Step 2: Track OOB Samples for Each Instance

Instance	**Models Where OOB**
(x_1, y_1)	Tree 3
(x_2, y_2)	Tree 2
(x_3, y_3)	Tree 1
(x_4, y_4)	None (appears in all bootstrap samples)

Step 3: Compute OOB Predictions

- For x_1: Only Tree 3's prediction $h_3(x_1)$ is used
- For x_2: Only Tree 2's prediction $h_2(x_2)$ is used
- For x_3: Only Tree 1's prediction $h_1(x_3)$ is used
- For x_4: No OOB prediction (excluded from error calculation)

Step 4: Calculate OOB Error (Classification Example) Suppose:

- True labels (suppose): $y_1 = 1, y_2 = 0, y_3 = 1$
- Predictions (suppose): $h_1(x_3) = 0$, $h_2(x_2) = 0$, $h_3(x_1) = 1$

$$\text{Error}_{\text{OOB}} = \frac{1}{3} \left(\mathbb{I}(1 \neq 1) + \mathbb{I}(0 \neq 0) + \mathbb{I}(0 \neq 1) \right) = \frac{0 + 0 + 1}{3} \approx 33.3\%$$

Key Observations:

- OOB evaluation uses *unseen* data for each model (like a built-in validation set)
- About 36.8% of data is OOB on average for large datasets ($\lim_{N\to\infty}(1 - \frac{1}{N})^N = 1/e$)
- More trees $\Rightarrow$ more reliable OOB error estimate

8.4 Boosting Methods

- Boosting is a popular ensemble learning technique that combines multiple weak learners (typically decision trees) to create a strong learner. It works by sequentially training new models, where each model focuses on correcting the errors made by its predecessors. The final prediction is made by aggregating the predictions of all models, typically using a weighted sum.
- Boosting was initially designed for binary classification tasks, but it can be easily adapted for general classification and regression problems.
- While boosting shares similarities with bagging in that both methods utilize an ensemble of predictive models, there is a key difference between them. In bagging, predictive models are fitted to bootstrapped subsets of the data independently. In contrast, boosting involves a sequential learning process where each model is trained using information from the previous models.

Boosting Algorithm

- Similar to other ensemble methods, such as the bagging method, we have a collection of models, denoted as $h_1, \ldots, h_M$ such that the final model is a function of all the models.
- The idea is to start with a simple model (weak learner) h_1 for the given training data $D = (x_i, y_i)_{i=1}^n$, and then to improve or "boost" this learner to a learner

  $$g_2 := h_1 + \gamma_2 h_2.$$

  Here, the function h_2 is found by minimizing the training loss for $h_1 + h$ over all functions h in some class of functions $\mathcal{H}$. For example, $\mathcal{H}$ could be the set of prediction functions that can be obtained via a decision tree of maximal depth 3.
- The parameter γ_2 helps in reducing overfitting.
- Given a loss function Loss, the function h_2 is thus obtained as the solution to the optimization problem

  $$h_2 = \arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \mathsf{Loss}(y_i, h_1(x_i) + h(x_i)).$$
- This process is repeated to get $g_3 = g_2 + \gamma_3 h_3 = h_1 + \gamma_2 h_2 + \gamma_3 h_3$ via

  $$h_3 = \arg\min_{h \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \mathsf{Loss}(y_i, g_2(x_i) + h(x_i)).$$

  This is done until we get $g_M = g_{M-1} + \gamma_M h_M = \sum_{m=1}^{M} \gamma_m h_m$, with $\gamma_1 = 1$.

Algorithm 1: Regression Boosting with Squared-Error Loss

```
Input: Dataset $\{(x_1, y_1), \ldots, (x_n, y_n)\}$,
       Number of boosting rounds $M$, and
       shrinkage parameters $1 = \gamma_1, \gamma_2, \ldots, \gamma_M$
Output: Boosted prediction function $g_M = \sum_{m=1}^{M} \gamma_m h_m$

Set $g_1(x) = h_1(x) \leftarrow \frac{1}{n} \sum_{i=1}^{n} y_i$.
for $m = 2, \ldots, M$ do
    Set $\epsilon_i^{(m)} \leftarrow y_i - g_{m-1}(x_i)$ for all $i = 1, \ldots, n$
    Let $D_m \leftarrow \{(x_i, \epsilon_i^{(m)})\}_{i=1}^{n}$
    Fit a model $h_m$ using the new dataset $D_m$ and mean-squared error as loss
    Set $g_m \leftarrow g_{m-1} + \gamma_m h_m$.
end
Return $g_M$
```

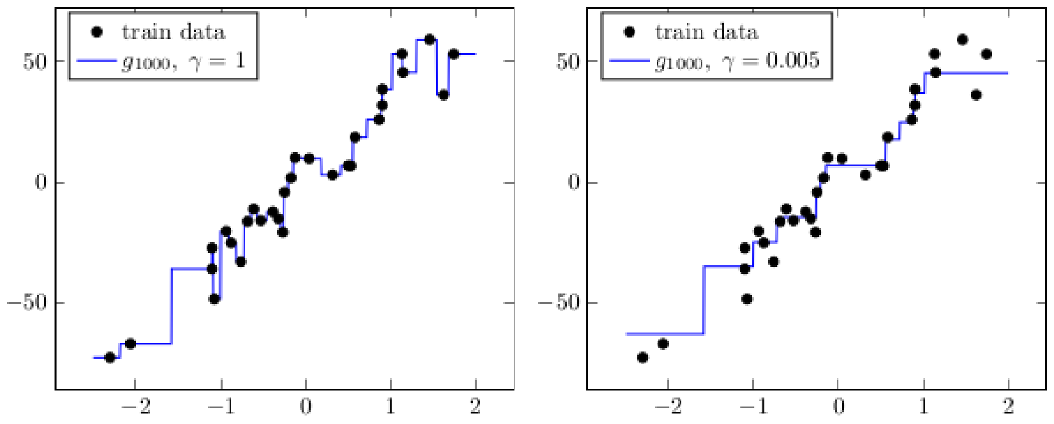

Figure 8.1: The left and the right panels show the fitted boosting regression model g_{1000} with $\gamma = 1.0$ and $\gamma = 0.005$, respectively. Note the overfitting on the left.

Remark

The γ parameters controls the speed of the fitting process. Suppose assume that $\gamma_2 = \gamma_3 = \cdots = \gamma_M = \gamma$. For small values of γ, boosting takes smaller steps towards the training loss minimization. The step-size γ is of great practical importance, since it helps the boosting algorithm to avoid overfitting. This phenomenon is demonstrated in Figure 8.1.

Gradient Boosting

- In the above algorithm, the parameter γ_m can be viewed as a step size made in the direction of the negative gradient of the squared-error training loss during the m-th update.

- To see this, for squared-error loss,

$$-\frac{\partial \mathsf{Loss}(y_i, z)}{\partial z}\Big|_{z=g_{m-1}(x_i)} = -\frac{\partial (y_i - z)^2}{\partial z}\Big|_{z=g_{m-1}(x_i)}$$
$$= 2(y_i - g_{m-1}(x_i)) = 2\epsilon_i^{(m)}.$$

- Since h_m is trained to predict $\epsilon_i^{(m)}$, we have

$$g_m(x_i) = g_{m-1}(x_i) + \gamma_m h_m(x_i)$$
$$\approx g_{m-1}(x_i) + \gamma_m \epsilon_i^{(m)}.$$

  Alternatively.

$$g_m(x_i) \approx g_{m-1}(x_i) - \frac{\gamma_m}{2}\frac{\partial \mathsf{Loss}(y_i, z)}{\partial z}\Big|_{z=g_{m-1}(x_i)},$$

  which is similar to update in gradient descent optimization method.

- One of the significant breakthroughs in boosting theory was the realization that a gradient descent method could be applied to any differentiable loss function. This led to the development of the algorithm known as *gradient boosting*.

- In particular, in gradient boosting, for any differentiable loss function Loss, in the m-th iteration, we compute

$$r_i^{(m)} = -\frac{\partial \mathsf{Loss}(y_i, z)}{\partial z}\Big|_{z=g_{m-1}(x_i)}, \quad i = 1, \dots, n,$$

  and obtain h_m as

$$h_m = \arg\min_{h \in \mathcal{H}} \frac{1}{n}\sum_{i=1}^{n}(r_i^{(m)} - h(x_i))^2.$$

- In practice, it is common to introduce a small *shrinkage* (learning-rate) $\nu \in (0, 1]$:

$$g_m(x) \;=\; g_{m-1}(x) \;+\; \nu\, \gamma_m\, h_m(x),$$

  which often yields better generalisation (at the cost of more boosting rounds).

Remark 8.4.1

Using the logistic deviance $\mathsf{Loss}(y, z) = \ln\bigl(1+e^{-yz}\bigr)$ for the loss, one obtains the popular *LogitBoost* classifier.

Example 8.4.1

Consider a tiny regression dataset with 4 instances:

$$D = \{(1, 2.0), (2, 3.5), (3, 4.0), (4, 5.5)\} \quad \text{(where } (x_i, y_i)\text{)}$$

Initialisation: Start with the mean prediction $g_0(x_i) = \bar{y} = \frac{2.0+3.5+4.0+5.5}{4} = 3.75$ for all x_i.
Iteration 1 ($m = 1$):

1. Compute residuals (negative gradients): $r_i^{(1)} = y_i - g_0(x_i) = \{-1.75, -0.25, 0.25, 1.75\}$.
2. Fit weak learner h_1 (e.g., decision stump) to residuals:
$$h_1(x) = \begin{cases} 0.5 & \text{if } x \leq 2.5 \\ 1.5 & \text{if } x > 2.5 \end{cases}$$
3. Compute step size γ_1 via line search (minimizing loss):
$$\begin{aligned} \gamma_1 &= \frac{\sum(y_i - g_0(x_i))h_1(x_i)}{\sum h_1(x_i)^2} \\ &= \frac{(-1.75)(0.5) + (-0.25)(0.5) + (0.25)(1.5) + (1.75)(1.5)}{0.5^2 + 0.5^2 + 1.5^2 + 1.5^2} \approx 0.8. \end{aligned}$$
4. Update model:
$$g_1(x_i) = g_0(x_i) + \gamma_1 h_1(x_i) = \{3.75 + 0.8 \times 0.5, \dots\} = \{4.15, 4.15, 4.95, 4.95\}.$$

Iteration 2 ($m = 2$):

1. New residuals: $r_i^{(2)} = y_i - g_1(x_i) = \{-2.15, -0.65, -0.95, 0.55\}$
2. Fit h_2 to residuals:
$$h_2(x) = \begin{cases} -1.2 & \text{if } x \leq 1.5 \\ 0.3 & \text{if } x > 1.5 \end{cases}$$
3. New step size $\gamma_2 \approx 0.5$ (via line search)
4. Updated model:
$$g_2(x_i) = g_1(x_i) + \gamma_2 h_2(x_i) = \{4.15 + 0.5 \times (-1.2), \dots\} = \{3.55, 4.30, 5.10, 5.10\}$$

Key Observations:

- Each iteration fits a weak learner to the *negative gradients* (residuals for squared error)
- The step size γ_m controls how aggressively we update predictions
- After 2 iterations, predictions are closer to true values:

$$\text{MSE improved from 1.56 (initial)} \rightarrow \text{0.61 (Iteration 1)} \rightarrow \text{0.29 (Iteration 2)}$$

AdaBoost (Adaptive Boosting)

- AdaBoost is another popular boosting algorithm. It assigns higher weights to the misclassified samples in each iteration, allowing subsequent weak learners to focus more on the difficult-to-classify instances.
- The idea of AdaBoost is similar to the one presented in the regression setting, that is, AdaBoost fits a sequence of prediction functions $g_1 = h_1, g_2 = h_1 + h_2, \dots$ with final prediction function
$$g_M = \sum_{m=1}^{M} h_m,$$
where each weak learner h_m is of the form
$$h_m(x) = \gamma_m c_m(x),$$
with c_m is a proper classifier from a class of weaker classifiers $\mathcal{C}$. To get this classifier, we solve
$$(\gamma_m, c_m) = \arg\min_{\gamma \geq 0, c \in \mathcal{C}} \frac{1}{n} \sum_{i=1}^{n} \mathsf{Loss}(y_i, g_{m-1}(x_i) + \gamma c(x_i)).$$

Remark 8.4.2

The inventors of the AdaBoost method considered a binary classification problem, where the response variable belongs to the $-1, 1$ set. In this case the loss function is defined as
$$\mathsf{Loss}(y, \hat{y}) = \exp(-y\hat{y}).$$

XGBoost (Extreme Gradient Boosting)

An efficient and scalable implementation of gradient boosting with tree learners. Besides the usual stagewise boosting steps, XGBoost adds:

- **Regularized Objective:**
$$\mathcal{L}^{(m)} = \sum_{i=1}^{N} \mathsf{Loss}\big(y_i, \hat{y}_i^{(m-1)} + f_m(x_i)\big) \; + \; \Omega(f_m),$$
where each tree f_m incurs
$$\Omega(f_m) = \gamma T + \tfrac{1}{2}\lambda \sum_{j=1}^{T} w_j^2,$$
with T the number of leaves and w_j their weights.

- **Second-order Approximation:** Expand the loss around the current predictions:

$$\mathsf{Loss}\big(y_i, \hat{y}_i^{(m-1)} + f(x_i)\big) \approx \mathsf{Loss}\big(y_i, \hat{y}_i^{(m-1)}\big) + g_i^{(m)} f(x_i) + \tfrac{1}{2}\, h_i^{(m)}\, f(x_i)^2,$$

  where

$$g_i^{(m)} = \frac{\partial \mathsf{Loss}(y_i, \hat{y})}{\partial \hat{y}}\Big|_{\hat{y}=\hat{y}_i^{(m-1)}}, \quad h_i^{(m)} = \frac{\partial^2 \mathsf{Loss}(y_i, \hat{y})}{\partial \hat{y}^2}\Big|_{\hat{y}=\hat{y}_i^{(m-1)}}.$$

- **Optimal Leaf Weight:** For a leaf j containing instances I_j,

$$w_j^* = -\frac{\sum_{i \in I_j} g_i^{(m)}}{\sum_{i \in I_j} h_i^{(m)} + \lambda}.$$

- **Split-Gain Criterion:** Splitting node $I \to I_{\text{left}} + I_{\text{right}}$ yields

$$\text{Gain} = \tfrac{1}{2}\Big[\frac{(\sum_{i \in I_{\text{left}}} g_i)^2}{\sum_{i \in I_{\text{left}}} h_i + \lambda} + \frac{(\sum_{i \in I_{\text{right}}} g_i)^2}{\sum_{i \in I_{\text{right}}} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda}\Big] - \gamma.$$

  A split is kept only if Gain $> \gamma$.

- **Sparsity and Missing-Value Handling:** Uses a default direction for missing features and a specialised weighted-quantile sketch for fast split finding on sparse data.

- **Subsampling & Parallelism:** `subsample` (rows), `colsample_bytree` (columns), and an efficient `hist` or GPU method implementation for massive speedups.

- **Shrinkage (Learning Rate):** After fitting f_m, scale by $\eta \in (0, 1]$: $\hat{y}^{(m)} = \hat{y}^{(m-1)} + \eta\, f_m(x)$.

With these additions—second-order updates, regularisation, sparsity-aware splits, subsampling, and highly optimised internals—XGBoost often outperforms a vanilla GradientBoosting implementation in both speed and generalisation.