Chapter 1

High-Dimensional Nonlinear Models

In this lecture, we will introduce supervised learning methods that induce data-driven interaction of the covariates. The interaction makes the covariates much more flexible to capture the subtle feature in the data. However, insufficient theoretical understanding sheds little light on these methods due to the complex nature, so they are often viewed by theorists as "black-box" methods. In real applications, when the machines are carefully tuned, they can achieve excellent performance. Caution must be exercised when we implement these methods in empirical economic analysis.

1.1 Series Estimation

Statisticians have long been working on nonparametric estimation. These nonparametric estimation methods pioneered modern machine learning. Suppose we are interested in the conditional mean $E[y|x]$. We solve the minimization problem

$$\min_m E[(y - m(x))^2].$$

Although it is common practice in econometrics to use the linear projection to approximate $m(x)$, the true conditional mean is nonlinear in general. If we do not know the underlying parametric estimation of (y, x), estimating $m(x)$ is a nonparametric problem.

Given a finite sample of n observations, the sample minimization problem is

$$\min_m \frac{1}{n} \sum_{i=1}^{n} (y_i - m(x_i))^2.$$

We must restrict the class of functions that we search for the minimizer. If we assume a smooth m, we can use a series expansion to approximate the function. Series expansion generates many additive terms. For example, any bounded, continuous, and differentiable function has a series representation $m(x) = \sum_{k=0}^{\infty} \beta_k \cos(\frac{k}{2}\pi x)$. In finite sample, we choose a finite K, usually much smaller than n, as a cut-off. Asymptotically $K \to \infty$ as $n \to \infty$ so that

$$m_K(x) = \sum_{k=0}^{K} \beta_k \cos\left(\frac{k}{2}\pi x\right) \to m(x).$$

Bias-variance tradeoff appears in this nonparametric regression. If K is too big, $m_K(x)$ will be too flexible and it can achieve even 100% of in-sample R-squared (when $K = n$). This is not useful for out-of-sample prediction. Such prediction will have large variance, but small bias. On the other extreme, a very small K will make $m_K(x)$ too rigid to approximate general nonlinear functions. It causes large bias but small variance. We must balance bias and variance.

Another way of regularization is to specify a sufficiently large K, and then add a penalty term to control the complexity of the additive series. Either ridge or Lasso can be used as a penalty.

1.2 Tree-based methods

Breiman (1984) deviates from the series regression by introducing **regression trees**. A regression tree recursively partitions the space of regressors. The algorithm is easy to describe: each time a covariate is split into two dummies, and the splitting criterion is the most aggressive reduction of the SSR. In the formulation of the SSR, the fitted value is computed as the average of the y_i's in a partition. You can search Youtube to see many videos that animate the growth of a tree.

The tuning parameter here is the depth of the tree, which is referred to as the number of splits. Given a dataset d and the depth of the tree, the fitted regression tree $\hat{r}(d)$ is completely determined by the data.

A major drawback of the regression tree is its instability. For independent datasets drawn from the same data generating process (DGP), the covariate chosen to be split and the splitting point can vary wildly and they heavily influence the path of the partition.

Bootstrap aggregation, or **bagging** for short (Breiman, 1996), was introduced to reduce the variance of the regression tree. Bagging grows a regression tree based on each **bootstrap** dataset, and then does a simple average. Let $d^{\star b}$ be the b-th bootstrap sample of the original data d, and then the bagging estimator is defined as

$$\hat{r}_{\text{bagging}} = B^{-1} \sum_{b=1}^{B} \hat{r}(d^{\star b}).$$

Bagging induces the number of bootstrap resampling B as another tuning parameter.

Random forest (Breiman, 2001) shakes up the regressors by randomly sampling s out of the total p covarites before each split of a tree during the process of bagging. Compared to bagging, random forest adds a third tuning parameter s to "de-correlation" the regressors.

Bagging and random forest are **ensemble learning** methods. They almost always use equal weights on each tree for the ensemble. Instead, **gradient boosting** takes a distinctive scheme to determine the ensemble weights. It is a deterministic approach that does not resample the original data.

The steps go as the following. (i) Use the original data $d^0 = (x_i, y_i)_{i=1}^n$ to grow a shallow tree $\hat{r}^0(d^0)$. Save the prediction $f_i^0 = \alpha \cdot \hat{r}^0(d^0, x_i)$ where $\alpha \in [0, 1]$ is the **learning rate** (α is a tuning parameter). Save the residual $e_i^0 = y_i - f_i^0$. Set $l = 1$. (ii) In the l-th iteration, use the data $d^l = (x_i, e_i^{l-1})$ to grow a shallow tree $\hat{r}^l(d^l)$. Save the prediction $f_i^l = f_i^{l-1} + \alpha \cdot \hat{r}^l(d, x_i)$.

Save the residual $e_i^l = y_i - f_i^l$. Update $l \leftarrow l + 1$. (iii) Repeat Step 2 until $l > L$ (L is a tuning parameter).

Three tuning parameters are involved in the boosting algorithm: the tree depth, the learning rate α, and the number of iterations M.

There are many variants of boosting algorithms. For example, L_2-boosting, componentwise boosting, and AdaBoosting, etc. Statisticians view boosting as a gradient descent algorithm to reduce the risk. The fitted tree in each iteration is the deepest descent direction, while the learning rate tames the fitting to avoid proceeding too aggressively.

1.3 Neural Network

Neural networks are the workhorses of AI. Statisticians view them as particular types of nonlinear models. The simplest architecture is a one-layer **feedforward neural network**, but in general there can be several layers to form a **deep neural network**. The transition from layer $k-1$ to layer k can be written as

$$\begin{aligned} z_l^{(k)} &= w_{l0}^{(k-1)} + \sum_{j=1}^{p_{k-1}} w_{lj}^{(k-1)} a_j^{(k-1)} \qquad (1.1) \\ a_l^{(k)} &= g^{(k)}(z_l^{(k)}), \end{aligned}$$

where $a_j^{(0)} = x_j$ is the input, $z_l^{(k)}$ is the k-th hidden layer, and all the w's are coefficients to be estimated. The above formulation shows that $z_l^{(k)}$ takes a linear form, while the **activation function** $g(\cdot)$ can be an identity function or a simple nonlinear function. Popular choices of the activation function are sigmoid $(1/(1+\exp(-x)))$ and rectified linear unit (ReLu, $z \cdot 1\{x \geq 0\}$), etc.

One of the important early contributions of the theoretical properties of NN came from econometricians. Hornik et al (1989) (Theorem 2.2) shows that a single hidden layer neural network, given enough many nodes, is a *universal approximator* for any measurable function.

A user has multiple tuning parameters to choose from when fitting a neural network: besides the activation function, one must decide the number of hidden layers and the number of nodes in each layer. Many (plain) parameters are induced by the multiple layers and multiple nodes, and estimation often employs regularization methods to penalize the L_1 or L_2 norms and/or the dropout rate, which require extra tuning parameters.

The nonlinear complex structure makes the optimization very challenging and the global optimizer is beyond guarantee. In particular, when the sample size is big, the de facto optimization algorithm is the stochastic gradient descent.

1.4 Stochastic Gradient Descent

In optimization we update the D-dimensional parameter

$$\beta_{k+1} = \beta_k + a_k p_k,$$

where $a_k \in \mathbb{R}$ is the step length and $p_k \in \mathbb{R}^D$ is a vector of directions. Use a Taylor expansion,

$$f(\beta_{k+1}) = f(\beta_k + a_k p_k) \approx f(\beta_k) + a_k \nabla f(\beta_k) p_k,$$

If in each step we want the value of the criterion function $f(x)$ to decrease, we need $\nabla f(\beta_k) p_k \leq 0$. A simple choice is $p_k = -\nabla f(\beta_k)$, which is called the deepest descent.[1]

When the sample size is huge and the number of parameters is also big, the evaluation of the gradient on the full dataset can be prohibitively expensive. **Stochastic gradient descent** (SGD) uses a small batch of the sample to evaluate the gradient in each iteration. It can significantly save computational time. It is the *de facto* optimization procedure in complex optimization problems such as training a neural network.

However, SGD involves tuning parameters (say, the batch size and the learning rate; learning rate replaces the step length a_k and becomes a regularization parameter) that can dramatically affect the outcome, in particular in nonlinear problems. Careful experiments must be carried out before serious implementation.

`Zhentao Shi.  April 10, 2025`

[1] Newton's method corresponds to $p_k = -(\nabla^2 f(\beta_k))^{-1} \nabla f(\beta_k)$, and BFGS uses a low-rank matrix to approximate $\nabla^2 f(\beta_k)$.