Statistical Learning

Additional Notes

Homework-01 | Exercise 01

Curse of dimensionality vs Additive Models

We all know this now: the impact of high dimensionality on statistics is multiple. Numerical computations and optimizations in high-dimensional spaces, for example, can be overly intensive and, more importantly, high-dimensional spaces are vast and data points are isolated in their immensity: this point is *dramatically* relevant for all our estimation methods that rely on *local* averaging.

Let me illustrate this issue with an example. Consider a typical situation where we want to explain a response variable $Y \in \mathbb{R}$ by d variables $\boldsymbol{X} = [X_1, \dots, X_d]^\intercal \in [0,1]^d$. Assume, for the sake of this example, that each variable follows a uniform distribution on $[0,1]$. If these variables are independent, then the vector $\boldsymbol{X}$ follows a uniform distribution on the hypercube $[0,1]^d$. Now our data consist of n i.i.d. observations $\{(Y_i, \boldsymbol{X}_i)\}_{i=1}^n$ of the variables Y and $\boldsymbol{X}$. We model them with the classical regression equation

$$Y_i = m(\boldsymbol{X}_i) + \epsilon_i, \quad \text{with} \quad m : [0,1]^d \mapsto \mathbb{R} \quad \text{and} \quad \{\epsilon_i\}_i \text{ independent and centered.}$$

Assuming that the function $m(\cdot)$ is smooth, it is natural to estimate $m(\boldsymbol{x})$ by some (weighted) average of the Y_i associated to the $\boldsymbol{X}_i$ in the vicinity of the target $\boldsymbol{x}$. This makes perfect sense in low–dimensional settings and this is what most of the methods we've seen essentially do. Unfortunately, when the dimension d increases, the notion of "nearest points'' vanishes because for a fixed sample size n, as the dimension d increases, the observations get rapidly very isolated and local methods cannot work. To have a better, quantitative sense of this issue, let's try to answer the following question:

Question: how should the number of observations n increase with the dimension d so that for any fixed $\boldsymbol{x}_0 \in [0,1]^d$ we have at least one observation $\boldsymbol{X}_i$ out of n at distance less than 1 from it?

We can investigate this issue by computing a *lower bound* on the number n of points needed in order to fill the hypercube $[0,1]^d$ in such a way that at any fixed $\boldsymbol{x}_0 \in [0,1]^d$ there exists at least one point at distance less than 1 from $\boldsymbol{x}_0$. The math is simple (ask me if you wish) and the answer, summarized in the following table, is quite mind-boggling...

d	20	30	50	100	150
n	39	45630	$5.7 \cdot 10^{12}$	$42 \cdot 10^{39}$	$1.28 \cdot 10^{72}$

To put these numbers in perspective, please notice that within our observable universe it is believed there are between $1.2 \cdot 10^{23}$ and $3.0 \cdot 10^{23}$ stars but, looking a bit closer, it is estimated that there are between 10^{78} and 10^{82} atoms... quite comparable indeed! This number of points grows more than exponentially fast with d. If we come back to our regression setup, it means that if we want a local average estimator to work with observations uniformly distributed in $[0,1]^d$ with d larger than a few tens, then we would need a number n of observations which is hilariously big!

Take–Home Message: being able to sense simultaneously thousands of covariates sounds like a blessing, since we collect huge amounts of data. Yet, the information is awash in the noise of our high–dimensional setting.

In light of this discussion, the situation may appear hopeless. **Fortunately**, high–dimensional data are often much more low dimensional than they seem to be. Usually, they are not "uniformly'' spread in $\mathbb{R}^d$, but rather concentrated around some low–dimensional structures. The major issue with high–dimensional data is that these structures are usually *unknown*, and the main task will then be to identify these structures.

Now, coming back to our regression setup, when the covariate space is high–dimensional, it is sometimes more fruitful to turn to an **additive model** instead of desperately trying to estimate the full regression function $m : [0,1]^d \mapsto \mathbb{R}$. An **additive**

model is a model of the form

$$Y_i = \alpha + \sum_{j=1}^{d} m_j(x_{i,j}) + \epsilon_i, \quad i \in \{1, \ldots, n\},$$

where $m_1(\cdot), \ldots, m_d(\cdot)$ are **smooth** functions of a **single** variable. This model is not identifiable since we can add any constant to α and subtract the same constant from one of the $m_j(\cdot)$'s without changing the regression function. The easiest way to fix this problem is to set $\widehat{\alpha} = \bar{y}$ (the average response) and then regard the $m_j(\cdot)$'s as deviations from $\bar{y}$. In this case we require that, for each j, $\sum_{i=1}^{n} m_j(x_{i,j}) = 0$.

The additive model is clearly not as general as fitting $m(x_1, \ldots, x_d)$ but it is much simpler to compute and to interpret and so it is often a good starting point. They have been proven to be very useful as they naturally generalize the linear regression model and allow for an interpretation of marginal changes, i.e. for the effect of one variable on the mean function when holding all others constant. This kind of model structure is widely used in both theoretical economics and in econometric data analysis.

There is also a simple algorithm for turning any one–dimensional regression smoother into a method for fitting additive models. It is called **backfitting**[1]. This algorithm is essentially a coordinate descent method and is a particular example of the so called Gauss-Seidel iteration for solving large linear systems of equations. Since Gauss-Seidel is convergent, the backfitting algorithm is also convergent.

0. **Input:** Data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{n}$
1. **Initialize:** $\widehat{\alpha} = \bar{y}$, $\widehat{m}_j(\cdot) \equiv 0$, $\forall\, j \in \{1, \ldots, d\}$
2. **Iterate** until $\{\widehat{m}_j\}_j$ stop changing:
   - For each predictor $j \in \{1, \ldots, d\}$
     - Compute partial residuals: $r_{i,j} = y_i - \left(\widehat{\alpha} + \sum_{k \neq j} \widehat{m}_k(x_{i,k})\right)$, for $i \in \{1, \ldots, n\}$
     - Smooth the residuals: $\widehat{m}_j \leftarrow \texttt{smooth}\big(\{r_{i,j}\}_{i=1}^{n}\big)$
     - Recenter: $\widehat{m}_j \leftarrow \widehat{m}_j - \frac{1}{n}\sum_{i=1}^{n} \widehat{m}_j(x_{i,j})$
3. **Output:** Component functions $\widehat{m}_j(\cdot)$ and estimator $\widehat{m}(\boldsymbol{x}_i) = \widehat{\alpha} + \sum_j \widehat{m}_j(x_{i,j})$.

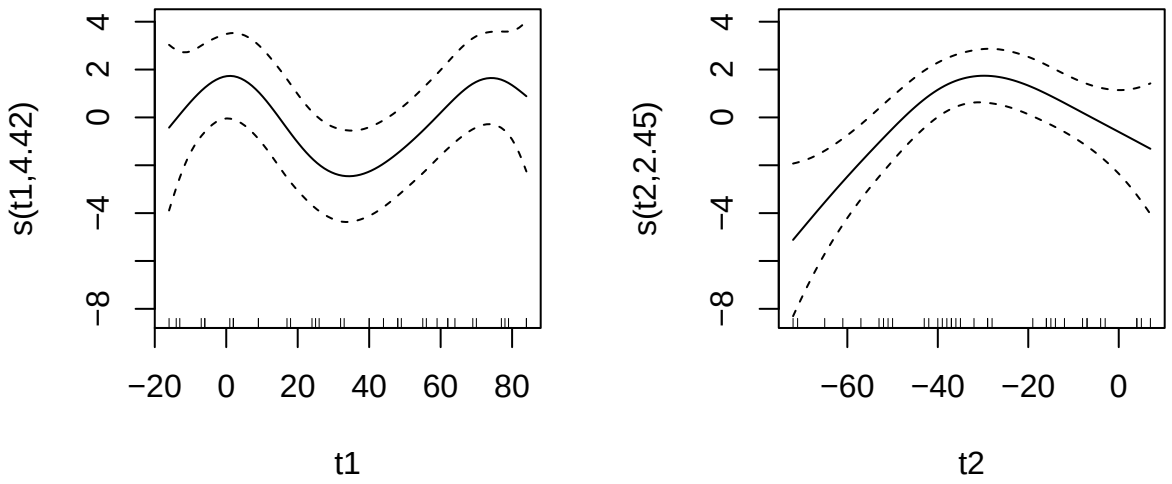
Note that any smoothing method can be used to do the univariate nonparametric regression at each stage. Usually we **scale** each covariate to have the same variance and then use a single common smoothing parameter, say h, for each covariate (to be set via a *global* `CV` round, for example). On the other hand, as in Example 5.129 (page 103) of the purple book, we could also try to perform `CV` to choose a bandwidth h_j for covariate x_j during each iteration of backfitting, although there's no theoretical backup that guarantees convergence if the smoothing parameters are changed this way.

The `gam()` function in the `mgcv` package does additive modelling using *smoothing splines* (**not** local polynomials as requested in this homework!) at each iteration. The following code demonstrates the elementary usage of the `gam()` function

```
# Data were collected from a mine in Cobar, NSW, Australia. At each of 38 sampling points,
# several measurements were taken, one of which is the 'true-width' of an ore-bearing rock
# layer. Also given are the coordinates t1 and t2 of of the data sites.
load("ore.RData")
str(ore)
library(mgcv)
ore.gam <- gam(width ~ s(t1) + s(t2), data = ore)
plot(ore.gam, pages = 1)
```

```
## 'data.frame':    38 obs. of  3 variables:
##  $ t1   : int  -16 -14 -13 -7 -6 -6 1 2 2 2 ...
##  $ t2   : int  -15 -4 4 5 -43 -36 -50 -39 -8 -51 ...
##  $ width: num  17 18 17.5 19 22 24 17.4 23 23.5 15 ...
```

[1] See also page 103 of our book.



As you can imagine, the first step is to implement the basic backfitting algorithm with any smoother you like. You may choose the "safe" single-bandwidth method, or the more (theoretically) risky route where in each iteration you pick an optimal smoother per dimension. To (hopefully) simplify your task a bit, the following is a slightly more explicit pseudo–code.
First of all some definitions:

- let $\boldsymbol{y} \in \mathbb{R}^n$ be the response vector;
- let $\mathbb{X} = [x_{i,j}]_{(i,j)}$ be the $(n \times d)$ design matrix. Each row $\boldsymbol{x}^i \in \mathbb{R}^d$ for $i \in \{1, \dots, n\}$ corresponds to an observation, whereas each column $\boldsymbol{x}_j \in \mathbb{R}^n$ for $j \in \{1, \dots, d\}$ to a covariate;
- let $\widehat{\boldsymbol{m}}_j \in \mathbb{R}^n$ be the generic vector of *in–sample predictions* based on the $j^{\texttt{th}}$ covariate and $\widehat{\mathbb{M}}$ be the $(n \times d)$ matrix that collects them.

Now the "expanded" algorithm[2]:

a. Set $\widehat{\alpha} = \texttt{mean}(\boldsymbol{y})$ and $\widehat{\boldsymbol{m}}_j = \boldsymbol{0}$ for $j \in \{1, \dots, d\}$.

b. Let $\texttt{rss}_0 = \texttt{sum}\Big(\big(\boldsymbol{y} - \widehat{\alpha} - \sum_{j=1}^{d} \widehat{\boldsymbol{m}}_j\big)^2\Big)$ be the *residual sum of squares* of the initial estimate.

c. Repeat the following backfitting loop *until* the `rss` of the current estimate doesn't change enough w.r.t. the previous value (always stored in the same variable called $\texttt{rss}_0$), or a maximum number of iteration `max.iter` is reached. More specifically, you can stop if $\texttt{abs}(\texttt{rss} - \texttt{rss0}) < \texttt{tol} * \texttt{rss}$ with $\texttt{tol} = \texttt{1e} - \texttt{6}$ for example.

- For $j \in \{1, \dots, d\}$:
  - Calculate partial residuals: $\boldsymbol{r}_j = \boldsymbol{y} - \widehat{\alpha} - \sum_{k \neq j} \widehat{\boldsymbol{m}}_k$.
  - Set $\widehat{\boldsymbol{m}}_j$ equal to the in–sample predictions obtained by smoothing the residuals $\boldsymbol{r}_j$ with respect to $\boldsymbol{x}_j$.
  - Recenter by subtracting the mean: $\widehat{\boldsymbol{m}}_j - \texttt{mean}(\widehat{\boldsymbol{m}}_j)$.
- Evaluate the `rss` of the current estimate and compare it with the previous value stored in $\texttt{rss}_0$.
- Update $\texttt{rss}_0$ and the stop–flag if needed.

---

[2] To be passed at each CV iteration.

Variable Importance: The LOCO

Being able to quantify the importance of a covariate in predicting a response of interest is crucial in most real applications... even more so nowadays, when the use of very complex, nonlinear, overparametrized models is the rule rather than the exception.

Despite this, the very idea of "importance" is slippery and need to be precisely framed, defined and handled (... yes, even when talking about linear models!). Here we'll focus on a very simple and general technique.

At page 32 of their paper, Lei and coauthors proposed a simple, general and, essentially assumptions–free idea for measuring variable importance, called **leave-one-covariate-out** (LOCO) inference. The algorithm is extremely simple.

Let $\ell(y, \widehat{y})$ be a suitable error measure/metric/loss for the learning task at hand. Then,

1. Randomly split the training data into two, non overlapping, parts: $\mathsf{D}_n = \mathsf{D}_{n_1}^{(1)} \cup \mathsf{D}_{n_2}^{(2)}$ with $n_1 + n_2 = n$.
2. Run any algorithm you like to compute an estimate $\widehat{f}_{n_1}(\cdot)$ on first part $\mathsf{D}_{n_1}^{(1)}$.
3. Select some variable $\boldsymbol{X}[j]$ of interest to you, and recompute $\widehat{f}_{n_1}^{-j}(\cdot)$ on $\mathsf{D}_{n_1}^{(1)}$ again (rerun algorithm without access to variable $\boldsymbol{X}[j]$).
4. Use $\mathsf{D}_{n_2}^{(2)}$ to construct finite-sample, distribution-free confidence interval (e.g., use non-parametric bootstrap or sign-test or Wilcoxon-test) for the following new (population) parameter[3]:

$$\theta_j\big(\mathsf{D}_{n_1}^{(1)}\big) = \text{median}_{(Y,\boldsymbol{X})}\Big(\ell\big(Y, \widehat{f}_{n_1}^{-j}(\boldsymbol{X})\big) - \ell\big(Y, \widehat{f}_{n_1}(\boldsymbol{X})\big) \,\Big|\, \mathsf{D}_{n_1}^{(1)}\Big), \quad j \in \{1, \dots, p\}. \tag{1}$$

   Since you're using the same dataset to build more than one confidence-interval, apply any reasonable correction to adjust for multiplicity (e.g. Bonferroni or Benjamini-Hochberg FDR).

θ_j has a very clear interpretation without resorting to linearity or any other *uncheckable* model assumption: it's just how much extra prediction error you would pay by not having access to variable $\boldsymbol{X}[j]$.

In addition, from a more technical point of view, this parameter is "smooth" enough (Hadamard differentiable) to guarantee the success of resampling techniques like the nonparametric bootstrap. In other words, confidence intervals, tests, etc for θ_j are easy to obtain no matter what is the underlying predictive model you picked (LASSO, LASSO + CV, Random Forest, Boosting, Deep nets, etc).

Of course there are also problems with the LOCO. More specifically:

1. It is **not** on an intuitive scale but we could fix this by rescaling.
2. Results are tied to our choice of the algorithm. In theory we could use a "meta–cross–validation" scheme that cycles over different candidate predictor classes (computational expensive but trivially parallelizable).
3. Results are also sensitive to: the ratio of training to test set sizes; the metric $\ell(\cdot, \cdot)$ selected; the correlation between features.
4. It is conditional on $\mathsf{D}_{n_1}^{(1)}$, meaning that it measures "*how important is variable X_j, to our algorithm's estimates on $\mathsf{D}_{n_1}^{(1)}$?*"...not obvious at all how to fix this...

[3] Alternatively, you could also consider the median of the ratio or log-ratio between the two losses.