Lecture 5: Kernel density estimation, regression and classification. Indexing large data sets

Marina Meilă
mmp@stat.washington.edu

Department of Statistics
University of Washington

October, 2018, April 2019

The Nearest-Neighbor predictor

- Main Idea The label of a point x is assigned as follows:
  1. find the example x^i that is nearest to x (in Euclidean distance)
  2. assign x the label y^i
- Practically, one uses the K nearest neighbors of x (with $K = 3, 5$ or larger), then
  - for classification $f(x) =$ the most frequent label among the K neighbors (well suited for multiclass)
  - for regression $f(x) = \frac{1}{K} \sum_{i \text{ neighbor of } x} y^i =$ mean of neighbors' labels
- No parameters to estimate! (But all data must be stored)

Kernel regression and classification

- Like the K-nearest neighbor but with "smoothed" neighborhoods

$$f(x) = \sum_{i=1}^{N} \beta_i b(x, x^i) y^i \tag{1}$$

  where β_i are coefficients
- Intuition: center a "bell-shaped" *kernel* function b on each data point, and obtain the prediction $f(x)$ as a weighted sum of the values y^i, where the weights are $\beta_i b(x, x^i)$
- Requirements for a kernel function $b(x, x')$
  1. non-negativity
  2. symmetry in the arguments x, x'
  3. optional: radial symmetry, bounded support, smoothness
- A typical kernel function is the **Gaussian kernel** (or **Radial Basis Function (RBF)**)

$$b_h(x, x') = e^{-\frac{||x-x'||^2}{2h^2}} \quad \text{with} \quad h = \textbf{the kernel width} \tag{2}$$

Regression example

A special case in wide use is the Nataraya-Watson regressor

$$f(x) = \frac{\sum_{i=1}^{N} b\left(\frac{||x-x^i||}{h}\right) y^i}{\sum_{i=1}^{N} b\left(\frac{||x-x^i||}{h}\right)}. \tag{3}$$

In this regressor, $f(x)$ is always a convex combination of the y^i's, and the weigths are proportional to $b_h(x, x^i)$.
The Nataraya-Watson regressor is biased if the density of P_X varies around x.

Local Linear Regression

To correct for the bias (to first order) one can estimate a regression line around x.

1. Given **query point** x
2. Compute kernel $b_h(x, x^i) = w_i$ for all $i = 1, \ldots N$
3. Solve weighted regression $\min_{\beta,\beta_0} \sum_{i=1}^{N} w_i \left(y^i - \beta^T x^i - \beta_0\right)^2$ to obtain β, β_0 (β, β_0 depend on x through w_i)
4. Calculate $f(x) = \beta^T x + \beta_0$

Exercise Show that Nataraya-Watson solves a local linear regression with fixed $\beta = 0$

Kernel binary classifiers

- obtained by setting y^i to ± 1.
- Note that the classifier can be written as the difference of two non-negative functions

$$f(x) \propto \sum_{i:y^i=1} b\left(\frac{||x - x^i||}{h}\right) - \sum_{i:y^i=-1} b\left(\frac{||x - x^i||}{h}\right). \tag{4}$$

Kernel density estimation

$$f(x) = \frac{1}{Nh^d} \sum_{i=1}^{N} b\left(\frac{||x - x^i||}{h}\right) \tag{5}$$

- $f(x)$ is the average of kernels placed at data points
- In all cases, h is a **smoothing parameter**

Neighbor search for large N

- Both K-nearest neighbor and kernel prediction involve scanning the whole data set for every single prediction
  - For K-nearest neighbor, predicting $f(x)$ for a single x involves computing N distances in n dimensions, a task that is $\sim Nn$.
  - For kernel methods, finding the data points in the support of the kernel, a ball of radius r, also involves computing the distances to all points.
- Neighbor search is (polynomial but) computationally expensive
- Can we be more efficient?
- **Yes,** if we **index** (i.e. preprocess) the data
  - indexing means organizing the data in a way that makes finding the neighbors of any point fast
  - in particular, with an index, finding neighbors does not require comparing with all N data points
- Indexing methods
  - K-D trees
  - Ball trees
  - A-D trees (for discrete data)
  - Locality Sensitive Hashing
  - . . . (many other methods with guarantees)

K-D Trees

A K-D tree is a "K-dimensional tree", whose nodes correspond to hyper-rectangular regions of the data space.

- Each node j stores:
  - a subset $\mathcal{D}_j$ of the data $\mathcal{D}$
  - an n-dimensional rectangle with $R_j = (r_{j,min}, r_{j,max}, j = 1:n)$, where $r_{j,min} = \min_{\mathcal{D}_j} x_j^i$, $r_{j,max} = \max_{\mathcal{D}_j} x_j^i$.
  - other statistics of $\mathcal{D}_j$, such as number of nodes, mean, variance

K-DTreeConstruction **Algorithm**

Input (labeled) training set $\mathcal{D}$ (labels are not used in the tree construction)

Initialize tree root R_0 with $\mathcal{D}_0 = \mathcal{D}$.

Repeat recursively until no leaf can be split

choose a leaf node j with $|\mathcal{D}_j| > N_0$ points

1. find the longest dimension of R_j, i.e $k = \underset{l=1:n}{\operatorname{argmax}} (r_{j,max} - r_{j,min})$ and set $r = (r_{k,max} - r_{k,min})/2$
2. split $\mathcal{D}_j$ into $\mathcal{D}_{j,left}, \mathcal{D}_{j,right}$ with $x^i \in \mathcal{D}_{j,left}$ iff $x_k^i \leq r$
3. create new leaves $R_{j,left}, R_{j,right}$ storing $\mathcal{D}_{j,left}, \mathcal{D}_{j,right}$ and their respective bounding boxes and other statistics.

Using a K-D tree to find the neighbors

Given a query point x, a search radius r, and a dataset $\mathcal{D}$ indexed by a K-D tree $\mathcal{T}$
Wanted find all the points in $\mathcal{D}$ that are in the ball $B_x(r)$ (i.e. the r-neighbors of x)

Basic remarks

- checking if $B_x(r)$ intersects with a hyper-rectangle R is fast
  - if $B_x(r) \cap R = \emptyset$, then no data point in R can be a neighbor
- checking if $B_x(r)$ contains a hyper-rectangle R is fast Exercise Think of an algorithm to do it!
  - if $B_x(r) \supset R$, then all data points in R are neighbors

K-D TreeNeighborRetrieval **Algorithm** $(x, r, \mathcal{T})$

Initialize $N_r = \emptyset$ set of neighbors, $R = root(\mathcal{T})$

call recursive function ProcessNode(x, r, R, N_r)

ProcessNode (x, r, R_j, N_r)

if $B_x(r) \cap R_j = \emptyset$ return — no neighbors in this box

else if $B_x(r) \supset R_j$ — all points in R_j are neighbors

1. $N_r \leftarrow N_r \cup R_j$
2. return

else if R_j is a leaf — make explicit comparisons

1. for $x^i \in \mathcal{D}_j$, if $||x^i - x|| < r$, $N_r \leftarrow N_r \cup \{x^i\}$
2. return

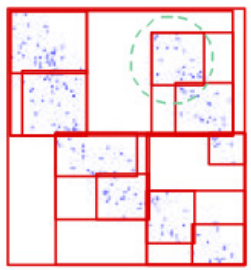
else — go to the next level

1. call ProcessNode $(x, r, R_{j,left}, N_r)$
2. call ProcessNode $(x, r, R_{j,right}, N_r)$
3. return

Ball Tree and K-D Tree

- K-D trees may become inefficient when data dimension n is large.
- In that case, we construct **Ball Trees**

K-D trees



- Each node of the tree stores bounding box, mean, variance of data points under it
- Hierarchically divide boxes along their longest dimension

Ball trees (M-trees)

- Like K-D trees but use balls instead of boxes
- Works in high dimensions if data non-uniform

Hash functions and hash codes

Let the data space be $\mathbb{R}^n$, and assume some fixed probability measure on this space.

- A **family of hash functions** is a set $\mathcal{H} = \{h : \mathbb{R}^n \to \{0,1\}\}$ with the following properties
  1. For each h, $Pr[h(x) = 1] \approx \frac{1}{2}$
  2. The binary random variables defined by the functions in $\mathcal{H}$ are mutually independent. (Or, if $\mathcal{H}$ is not finite, a "not too large" random sample of such random variables is mutually independent.)
- Let $h_{1:k}$ be a mutually independent subset of $\mathcal{H}$. We call

$$g(x) \;=\; [h_1(x)\, h_2(x)\, \ldots\, h_k(x)] \in \{0,1\}^k \tag{6}$$

  the **hash code** of x.
- Note that the codes $g(x)$ are (approximately) uniformly distributed; the probability of any $g \in \{0,1\}^k$ is about $\frac{1}{2^k}$.
- Useful hash functions must be fast to compute.

Hash tables

- A **hash table** $\mathcal{T}$ is a data structure in which points in $\mathbb{R}^n$ can be stored in such a way that
  1. All points with the same code g are in the same **bin** denoted by $\mathcal{T}_g$. The table need not use space for empty bins.
  2. Given any value $g \in \{0,1\}^k$, we can obtain a point in $\mathcal{T}_g$ or find if $\mathcal{T}_g = \emptyset$ in constant time (independent of the number of points N stored in $\mathcal{T}$).
     Some versions of hash tables return all points in $\mathcal{T}_g$, e.g., as a list, in constant time.
  3. It is usually assumed that storing a point x with given code $g(x)$ in a hash table is also constant time.
- Hence, using a hash table to store an x or to retrieve something, involves computing k hash functions, then a constant-time access to $\mathcal{T}$.
- When $x' \neq x$ and $g(x') = g(x)$ we call this a **collision**. In some applications (not of interest to us), collisions are to be avoided.

Locality Sensitive Hash Functions and Codes

- A hash function h is **locality sensitive** iff for any $x, x' \in \mathbb{R}^n$

$$Pr[h(x) = h(x')] \geq p_1 \quad \text{when } ||x - x'|| \leq r \tag{7}$$

$$Pr[h(x) = h(x')] \leq p_2 \quad \text{when } ||x - x'|| \geq cr \tag{8}$$

  with p_1, p_2, r and $c > 1$ fixed parameters (of the family $\mathcal{H}$) and $p_1 > p_2$.
- W.l.o.g., we set $p_1 = p_2^\rho$ for some $\rho < 1$.

- A locality sensitive h makes a weak distinction between points that are close in space vs. points that are far away. A hash code g from locality sensitive hash functions sharpens this distinction, in the sense that the probability of far away points colliding can be made arbitrarily small.

$$p_{bad} \;=\; Pr[g(x) = g(x') \,|\, ||x - x'|| > cr] \;\leq\; p_2^k \tag{9}$$

- Assume x is not in $\mathcal{T}$; for any $x' \in \mathcal{D}$ which is far from x,the probability that x' collides with x is $\leq p_{bad}$.
- We construct $\mathcal{T}$ so that $p_{bad} \leq \frac{1}{N}$ for N the sample size. For this we need Exercise (in Homework 1)

$$k \;=\; \frac{\ln N}{-\ln p_2} \quad \Rightarrow \quad p_{bad} \leq \frac{1}{N} \tag{10}$$

- Suppose $x' \in \mathcal{T}$ is "close" to x. What is the probability that $g(x') = g(x)$?

$$p_{good} \;=\; p_1^k \;=\; p_2^{\rho k} \;=\; \frac{1}{N^\rho} \tag{11}$$

  This is the probability that the bin $\mathcal{T}_{g(x)}$ contains x'.

Approximate r-neighbor retrival by LSH

Input $\mathcal{D}$ set of N points, L mutually independent hash codes $g_{1:L}$ of dimension k.

Indexing Construct L hash tables $\mathcal{T}^{1:L}$, each storing $\mathcal{D}$.

Retrieval Given x

1. compute $g(x)$
2. for $j = 1, 2, \ldots L$
   if the bin $\mathcal{T}^j_{g(x)} \neq \emptyset$
   - 2.1 return some (all) x' from it.
   - 2.2 stop if a single neighbor is wanted.

Some analysis. We set $L = N^\rho$

- Indexing time $\propto kN^{\rho+1}$
- Retrieval time $\propto kN^\rho$
- Space used $\propto kN^{\rho+1}$

- For each $x' \in \mathcal{D}$ close to x, the probability that x' is NOT returned for any $j \in 1 : L$ is

$$(1 - \frac{1}{N^\rho})^{N^\rho} \approx \frac{1}{e} \tag{12}$$

  This can be made arbitrarily small by multiplying L with a constant.
- For each $x' \in \mathcal{D}$ far from x, the probability that x' is NOT returned for any $j \in 1 : L$ is

$$(1 - \frac{1}{N})^{N^\rho} \approx \left(\frac{1}{e}\right)^{1/N^{1-\rho}} \approx \frac{1}{e^0} = 1 \tag{13}$$

- Hence, we are almost sure not to return a far point, and have a significant probability to return a close point when one exists, if no points neither far nor close are in the data. This is why this algorithm is **approximate**: it may also return points with $r < ||x' - x|| \leq cr$.

How to find **good** hash functions?

- We need large families of h functions
- that are easy to generate randomly
- and fast to compute for a given x

- Generic method to obtain them: random projections

Projecting on a random vector

- Data are $x \in \mathbb{R}^n$ as usual.
- Define $h_{a,b} : \mathbb{R}^n \to \mathbb{Z}$ by

$$h_{a,b}(x) = \lfloor \frac{a^T x + b}{w} \rfloor \tag{14}$$

  with $w > 0$ a width parameter, $a \in \mathbb{R}^n, b \in [0, w)$.
- Intuitively, x is "projected" on a[1], then the result is quantized into bins of width w, with a grid origin given by b.

- The family of hash functions is $\mathcal{H}_w = \{h_{a,b},\ a \in \mathbb{R}^n, b \in [0, w)\}$.
- Sampling $\mathcal{H}_w$: $a \sim \mathit{Normal}(0, I_n)$, $b \sim \text{uniform}[0, w)$.
  - Because the Normal distribution is a stable distribution, this ensures that $a^T x$ is distributed as $\mathit{Normal}(0, ||x||^2)$. Exercise Verify this
  - Hence $a^T x - a^T x'$ is distributed as $\mathit{Normal}(0, ||x - x'||^2)$. Exercise Verify this
  - Moreover, if hash functions are sampled independently from $\mathcal{H}_w$,(and nothing is known about x) then $h_{a,b}(x), h_{a',b'}(x)$ are independent random variables. Exercise Prove this

- This type of hash functions are being widely used by approximate neighbor search algorithms.

[1] a is not necessarily unit length