Kernels and Smoothers

Muchang Bahng

Spring 2025

Contents

1 K Nearest Neighbors Classification

Question 1.1 (To Do)

Maybe similar like a kernel regression?

Given a bunch of points in a metric space $(\mathcal{X}, d)$ that have classification labels, we want to label new datapoints $\hat{\mathbf{x}}$ based on the labels of other points that already exist in our dataset. One way to look at it is to look for close points within the dataset and use their labels to predict the new ones.

Definition 1.1 (Closest Neighborhood)

Given a dataset $\mathcal{D} = \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}$ and a point $\hat{\mathbf{x}} \in (\mathcal{X}, d)$, let the **k closest neighborhood** of $\hat{\mathbf{x}}$ be $N_k(\hat{\mathbf{x}}) \subset [N]$ defined as the indices i of the k points in $\mathcal{D}$ that is closest to $\hat{\mathbf{x}}$ with respect to the distance metric $d_{\mathcal{X}}$.

Definition 1.2 (K Nearest Neighbors)

The **K Nearest Neighbors (KNN)** is a discriminative nonparametric supervised learning algorithm that doesn't have a training phase. Given a new point $\hat{\mathbf{x}}$, we look at all points in its k closest neighborhood, and $h(\hat{\mathbf{x}})$ will be equal to whatever the majority class will be in. Let us one-hot encode the labels $\mathbf{y}^{(i)}$ into $\mathbf{e}_i$'s, and the number of data point in the ith class can be stored in the variable

$$a_i = \sum_{i \in N_k(\hat{\mathbf{x}})} 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_i\}} \tag{1}$$

which results in the vector storing the counts of labels in the k closest neighborhood

$$\mathbf{a} = (a_1, a_2, \dots, a_K) = \left(\sum_{i \in N_k(\hat{\mathbf{x}})} 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_1\}}, \sum_{i \in N_k(\hat{\mathbf{x}})} 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_2\}}, \dots, \sum_{i \in N_k(\hat{\mathbf{x}})} 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_K\}} \right) \tag{2}$$

and take the class with the maximum element as our predicted label.

The best choice of K depends on the data:

1. Larger values of K reduces the effect of noise on the classification, but make boundaries between classes less distinct. The number of misclassified data points (error) increases.
2. Smaller values are more sensitive to noise, but boundaries are more distinct and the number of misclassified data points (error) decreases.

Too large of a K value may increase the error too much and lead to less distinction in classification, while too small of a k value may result in us overclassifying the data. Finally, in binary (two class) classification problems, it is helpful to choose K to be odd to avoid tied votes.

This is an extremely simple algorithm that may not be robust. For example, consider $K \geq 3$, and we are trying to label a point $\hat{\mathbf{x}}$ that happens to be exactly where one point is on our dataset $\mathbf{x}^{(i)}$. Then, we should do $h(\hat{\mathbf{x}}) = y^{(i)}$, but this may not be the case if there are no other points with class $y^{(i)}$ in the k closest neighborhood of $\mathbf{x}^{(i)}$. Therefore, we want to take into account the distance of our new points from the others.

Definition 1.3 (Weighted Nearest Neighbor Classifier)

Let us define a monotinically decreasing function $\omega : \mathbb{R}_0^+ \mapsto \mathbb{R}_0^+$. Given a point $i \in N_k(\hat{\mathbf{x}})$, we can construct the weight of our matching label as inversely proportional to the distance: $\omega_i[d(\hat{\mathbf{x}}, \mathbf{x}^{(i)})]$

and store them as

$$\mathbf{a} = (a_1, a_2, \dots, a_{\mathcal{K}}) = \left(\sum_{i \in N_k(\hat{\mathbf{x}})} \omega_i 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_1\}}, \sum_{i \in N_k(\hat{\mathbf{x}})} \omega_i 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_2\}}, \dots, \sum_{i \in N_k(\hat{\mathbf{x}})} \omega_i 1_{\{\mathbf{y}^{(i)} = \mathbf{e}_{\mathcal{K}}\}} \right) \tag{3}$$

and again take the class with the maximum element.

One caveat of KNN is in high dimensional spaces, as its performance degrades quite badly due to the curse of dimensionality.

Example 1.1 (Curse of Dimensionality in KNN)

Consider a dataset of N samples uniformly distributed in a d-dimensional hypercube. Now given a point $x \in [0,1]^d$, we want to derive the expected radius r_k required to encompass its k nearest neighbors. Let us define this ball to be $B_{r_k} \coloneqq \{z \in \mathbb{R}^d \mid ||z - x||_2 \leq r_k\}$. Since thse N points are uniformly distributed, the expected number of points contained in $B_{r_k}(x)$ is simply the proportion of the volume that $B_{r_k}(x)$ encapsulates in the box, multiplied by N. Therefore, for some fixed x and r, let us denote $Y(x, y)$ as the random variable representing the number of points contained within $B_r(x)$. By linearity of expectation and summing over the expectation for whether each point will be in the ball, we have

$$\mathbb{E}[Y(x,r)] = N \cdot \frac{\mu(B_r(x) \cap [0,1]^d)}{\mu([0,1]^d)}$$

where μ is the Lebesgue measure of $\mathbb{R}^d$. Let us assume for not that we don't need to worry about cases where the ball is not fully contained within the cube, so we can just assume that Y is only dependent on r: $Y(r)$. Also, since the volume of the hypercube is 1, $\mu([0,1]^d) = 1$ and we get

$$\mathbb{E}[Y(r)] = N \cdot C_d \cdot r^d$$

which we set equal to k and evaluate for r. C_d is a constant such that the volume of the hypersphere of radius r can be derived as $V = C_d \cdot r^d$. We therefore get

$$N \cdot C_d \cdot r_k^d = k \implies r_k = \left(\frac{k}{NC_d} \right)^{1/d}$$

It turns out that C_d decreases exponentially, so the radius r_k explodes as d grows. Another way of looking at this is that in high dimensions, the ℓ_2 distance between all the pairwise points are close in every single dimension, so it becomes harder to distinguish points that are close vs those that are far.

1.1 Approximate K Nearest Neighbors

2 K Nearest Neighbors Regression

When we want to do nonparametric regression, i.e. when dealing with nonlinear functions, we can construct a function that uses local averaging of its nearby points.

Example 2.1 (Local Averaging)

Say that we want to fit some function through a series of datapoints in simple regression (one covariate). Then, what we can do is take some sliding window and our vale of the function at a point x is the average of all values in the window $[x - \delta, x + \delta]$.

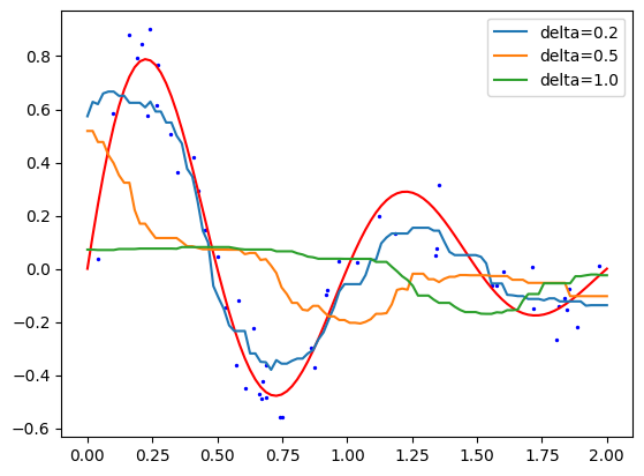

Figure 1: K means smoother

Code 2.1 (MWS of K Nearest Neighbor Regression in scikit-learn)

Local averaging is implemented as the K nearest neighbor regressor in scikit learn. It is slightly different in the way that it doesn't use the points within a certain δ away but rather the K nearest points. Either way, a minimal working example of this is

```
X = [[0], [1], [2], [3]]
y = [0, 0, 1, 1]
from sklearn.neighbors import KNeighborsRegressor
neigh = KNeighborsRegressor(n_neighbors=2)
neigh.fit(X, y)
print(neigh.predict([[1.5]]))
```

Note that since $\hat{f}$ is a combination of step functions, this makes it discontinuous at points.

3 Kernel Regression and Linear Smoothers

K nearest neighbor regression puts equal weights on both near and far points, as long as they are in the window. This may not be ideal, so a simple modification is to *weigh* these points according to their distance from the middle x. We can do this with a kernel, as the name suggests. Now this is not the same thing as a Mercer kernel in RKHS, so to distinguish that I will call it a *local averaging kernel*.

Definition 3.1 (Local Averaging Kernel)

A **kernel** is any smooth, symmetric, and non-negative function $K : \mathbb{R} \to \mathbb{R}$.

Definition 3.2 (Kernel Regression)

Given some datapoints, X, our fitted regressor is of form

$$\hat{f}(X) = \frac{\sum_i Y_i K\left(\frac{||X-X_i||}{h}\right)}{\sum_i K\left(\frac{||X-X_i||}{h}\right)} \tag{4}$$

where h is the **bandwidth** and the denominator is made sure so that the coefficients sum to 1. To get a clearer picture, we are really taking the weighted average of the Y_i's.

$$\hat{f}(X) = \sum_i Y_i \ell_i(X) \text{ where } \sum_i \ell_i(X) = 1 \tag{5}$$

Denoting $Y = (Y_1, \dots, Y_n) \in \mathbb{R}^n$ and the vector $f(X) = (f(X_1), \dots, f(X_n))$, if we can write the kernel function as $\hat{Y} = \hat{f}(X) = SY$, which in matrix form, is

$$\begin{bmatrix} \hat{Y}_1 \\ \vdots \\ \hat{Y}_n \end{bmatrix} = \begin{bmatrix} \hat{f}(X_1) \\ \vdots \\ \hat{f}(X_n) \end{bmatrix} = \begin{bmatrix} \ell_1(X_1) & \cdots & \ell_n(X_1) \\ \vdots & \ddots & \vdots \\ \ell_1(X_n) & \cdots & \ell_n(X_n) \end{bmatrix} \begin{bmatrix} Y_1 \\ \vdots \\ Y_n \end{bmatrix} \tag{6}$$

then we say that we have a **linear smoother**, with stochastic matrix S being our **smoothing matrix**.

The reason we'd like to have the weights to sum to 1 is that if we had data that was constant (i.e. all Y_i's are the same), then the fitted function should be constant at that value as well. Furthermore, the theme of linearity is important and will be recurring. The kernel estimator is defined for all X, but it's important to see its behavior at the training points X_i. The estimator $\hat{Y} = \hat{f}(X)$ is a linear combination of the Y_i's, and the coefficients $\ell_i(X_j)$ depend on the values of X_j. Therefore, we have $\hat{Y} = SY$, which is very similar to the equation $\hat{Y} = HY$ in linear regression, where H is the hat matrix that projects Y onto the column space of X. Nonparametric regression has the same form, but rather than being a projection, it is a linear smoothing matrix. Therefore, this theme unifies both linear regression and nonparametric regression. Linear smoothers, spline smoother, Gaussian processes, are all just different choices of the smoothing matrix S. However, not all nonparametric estimators are linear smoothers, as we will see later.

Here are some popular kernels.

Definition 3.3 (Gaussian Kernel)

The **Gaussian kernel** is defined as

$$K(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \tag{7}$$

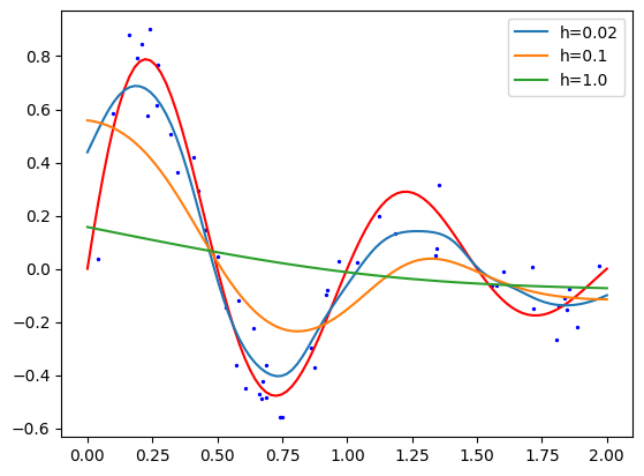

Figure 2: Gaussian kernel.

Definition 3.4 (Box-Car Kernel)

The **Box-Car kernel** is defined as

$$K(x) = \frac{1}{2}\mathbb{1}(|x| \leq 1) \tag{8}$$

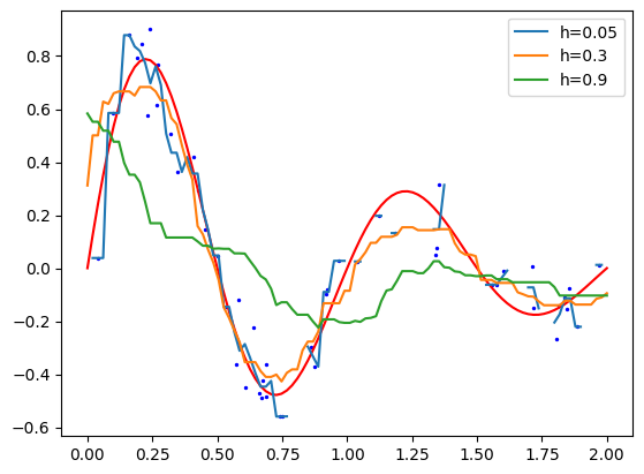

Figure 3: Boxcar kernel.

Definition 3.5 (Epanechnikov Kernel)

The **Epanechnikov kernel** is defined as

$$K(x) = \frac{3}{4}(1 - x^2)\mathbb{1}(|x| \leq 1) \tag{9}$$

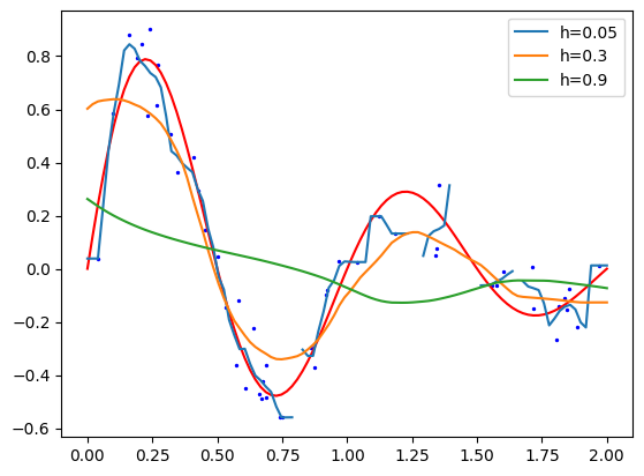

Figure 4: Epanechnikov kernel.

It turns out that from a theoretical point of view, the choice of the kernel doesn't really matter. What really matters is the bandwidth h since that is what determines the bias variance tradeoff. To see why, if $h = 0$, then it will simply interpolate the points and variance is extremely high, and if $h = \infty$, then the fitted function will be constant at $\bar{Y}$, leading to high bias. The following theorem formalizes this.

Theorem 3.1 (Bias Variance Tradeoff of Kernel Regression)

Suppose that $d = 1$ and that m'' is bounded. Also suppose that X has a nonzero, differentiable density p and that the support is unbounded. Then, the risk is

$$R_n = \frac{h_n^4}{4}\left(\int x^2 K(x)\right)^2 \int \left(m''(x) + 2m'(x)\frac{p'(x)}{p(x)}\right)^2 dx \tag{10}$$

$$+ \frac{\sigma^2 \int K^2(x)\,dx}{nh_n} \int \frac{dx}{p(x)} + o\left(\frac{1}{nh_n}\right) + o(h_n^4) \tag{11}$$

The first term is the squared bias and the second term is the variance.

Proof.

We first denote

$$\hat{f}(X) = \frac{\frac{1}{nh}\sum_{i=1}^n K\left(\frac{X - X_i}{h}\right) Y_i}{\frac{1}{nh}\sum_{i=1}^n K\left(\frac{X - X_i}{h}\right)} \tag{12}$$

where the denominator is the kernel density estimator $\hat{p}(X)$. Therefore, we rewrite

$$\hat{f}(x) - f(x) = \frac{\hat{a}(x)}{\hat{p}(x)} - f(x) \tag{13}$$

$$= \left(\frac{\hat{a}(x)}{\hat{p}(x)} - f(x)\right)\left(\frac{\hat{p}(x)}{p(x) + 1 - \frac{\hat{p}(x)}{p(x)}}\right) \tag{14}$$

$$= \frac{\hat{a}(x) - f(x)\hat{p}(x)}{p(x)} + \frac{(\hat{f}(x) - f(x))(p(x) - \hat{p}(x))}{p(x)} \tag{15}$$

as $n \to \infty$ both $\hat{f}(x) - f(x)$ and $p(x) - \hat{p}(x)$ going to 0, and since they're multiplied in the second

term, it will go to 0 very fast. So the dominant term is the first term, and we can write the above as approximately

$$\hat{f}(x) - f(x) \approx \frac{\hat{a}(x) - f(x)\hat{p}(x)}{p(x)} \tag{16}$$

TBD continued. Wasserman lecture 6, 10:00.

From the theorem above, we can see that if the bandwidth is small, then h^4 is small and the bias decreases. However, there is a h term in the denominator of the variance term, which also trades it off. We can furthermore see that the bias is sensitive to $p'/p(x)$. This means that if the density is steep, then the bias will be high. This is known as *design bias*, which is an underlying weakness in smoothing kernel regression. Another problem that is not contained in the theorem is the *boundary bias*, which states that if you're near the boundary of the distribution (i.e. near the boundary of its support), then the bias also explodes. This happens to be very nasty especially in high dimensions where most of the data tends to be near the boundary. It turns out that this can be easily fixed with local polynomial regression, which gets rid of this term in the bias without any cost to variance. This means that this is unnecessary bias.

Then you can apply regularization on this to get kernel ridge regression.

Code 3.1 (MWS of Kernel Ridge Regression in scikit learn)

```
from sklearn.kernel_ridge import KernelRidge
import numpy as np
n_samples, n_features = 10, 5
rng = np.random.RandomState(0)
y = rng.randn(n_samples)
X = rng.randn(n_samples, n_features)
krr = KernelRidge(alpha=1.0)
krr.fit(X, y)
```

4 Local Polynomial Regression

Now another way to think about the kernel estimator is as such. Suppose that you're doing linear regression on a bunch of points and you want to choose a c that minimizes the loss.

$$\sum_i (Y_i - c)^2 \tag{17}$$

You would just pick $c = \hat{Y}$. But if you are given some sort of locality condition, that the value of c should depend more on the values closer to it, you're really now minimizing

$$\sum_i (Y_i - c(x))^2 K\left(\frac{X_i - x}{h}\right) \tag{18}$$

Minimizing this by setting the derivative equal to 0 and solving gives us the kernel estimator. Therefore you're fitting some sort of local constant at a point X. But why fit a local constant, when you can fit a local line or polynomial? This is the idea behind local polynomial regression.

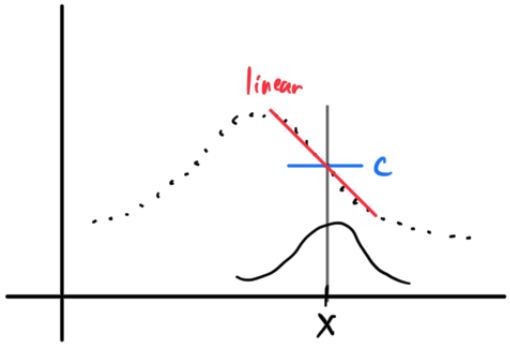

Figure 5: Rather than using a local constant, we can use a local linear estimator.

Therefore, we can minimize the modified loss.

Definition 4.1 (Local Polynomial Estimator)

The **local polynomial estimator** is a local linear kernel smoother that estimates the function $\hat{f}$ that minimizes the following loss.

$$\underset{\beta}{\text{argmin}} \sum_i K\left(\frac{X_i - x}{h}\right)\left(Y_i - (\beta_0(x) - \beta_1(x)(x - X_i) + \ldots + \beta_k(x)(x - X_i)^k)\right) \tag{19}$$

So we can fit a line

$$f(\mu) \approx \hat{\beta}_0(x) + \hat{\beta}_1(x)(\mu - x) \tag{20}$$

and simply remove the intercept term to get the local linear estimator.

$$\hat{f}(x) = \hat{\beta}_0(x) \tag{21}$$

Note that this is not the same as taking the constant estimate. We are extracting the fitted intercept term and so $\hat{\beta}_0(x) \neq c(x)$.

Theorem 4.1 (Weighted Least Squares)

The solution to the local linear estimator is similar to the weighted least squares solution.

$$\hat{\beta}(x) = \begin{pmatrix} \hat{\beta}_0(x) \\ \hat{\beta}_1(x) \end{pmatrix} = (X^T W X)^{-1} X^T W Y \tag{22}$$

where

$$X = \begin{pmatrix} 1 & X_1 - x \\ \vdots & \vdots \\ 1 & X_n - x \end{pmatrix} \qquad W = \begin{pmatrix} K\left(\frac{X_1 - x}{h}\right) & 0 & \cdots & 0 \\ 0 & K\left(\frac{X_2 - x}{h}\right) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & K\left(\frac{X_n - x}{h}\right) \end{pmatrix} \tag{23}$$

Computationally, it's similar to kernel regression and it gets rid of both the boundary and design bias.

5 Regularized: Spline Smoothing

This is not local, but it's a linear smoother.

6 Regularized: RKHS Regression

This is not local, but it's a linear smoother.

7 Additive Models

In the most general case, we want to create nonparametric regression functions of the form

$$Y = f(x_1, \dots, x_d) + \epsilon \tag{24}$$

We've done this for one dimensional case, but we can extend this to multiple dimensions through additive models of the form

$$Y = \sum_j f_j(x_j) + \epsilon \tag{25}$$

This gives us very interpretable models where we can clearly see the effect of each covariate on Y. Clearly, this is not as flexible as the previous model since they can't capture dependencies, but we can create sub-dependency functions and replace the form above to something like

$$Y = \sum_{i,j} f_{i,j}(x_i, x_j) + \epsilon \tag{26}$$

giving us more flexible models.

8 Nonlinear Smoothers, Trend Filtering

Tough example of the Dobbler function (like topologists sine curve). It's a pretty good fit but it's not too good since it's using a linear smoother (homogeneous). So we might need to fit it with nonlinear smoothers.

Bibliography