Nonparametric Statistics: Homework 7

蒋翌坤 20307100013

Solution to Problem 1

From definition, we have $L_{ij} = l_j(x_i)$ and for any x, $\sum_{j=1}^{n} l_j(x) = 1$

$$\begin{aligned} Y_i - \hat{r}_{(-i)}(x_i) &= Y_i - \Big(\sum_{j=1}^{n} \frac{l_j(x_i)}{\sum_{k\neq i} l_k(x_i)} Y_j - \frac{l_i(x_i)}{\sum_{k\neq i} l_k(x_i)} Y_i\Big) \\ &= \Big(1 + \frac{L_{ii}}{1 - L_{ii}}\Big) Y_i - \sum_{j=1}^{n} \frac{L_{ij}}{1 - L_{ii}} Y_j \\ &= \frac{Y_i - \hat{r}_n(x_i)}{1 - L_{ii}} \end{aligned}$$

Therefore, we have proved Theorem (5.34)

$$\hat{R}(h) = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{r}_{(-i)}(x_i))^2 = \frac{1}{n} \sum_{i=1}^{n} \Big(\frac{(Y_i - \hat{r}_n(x_i))}{(1 - L_{ii})}\Big)^2$$

■

Solution to Problem 2

Several assumptions and methods used to solve the problem need pointing out.

- The chosen kernel $K(x)$ is the Gaussian kernel.
- Generalized cross validation is used to avoid the case where $L_{ii} = 1$.
- Optimal smoothing parameter is chosen by minimizing generalized cross validation score $\mathrm{GCV}(h)$ through analyzing h-$\mathrm{GCV}(h)$ plot (see Figure A.1 in the appendix) as well as using functions to find $\arg\min \mathrm{GCV}(h)$.
- In the spline estimator, we use each sample x points as knots and optimize λ. If we encounter x that has the same value, we take the average of y corresponding to those x and treat the sample points as a single point.
- Code for constructing estimator and producing figures can be found in the appendix.

A summary of four estimators is shown in Table 2.1.

Estimator	Optimal smoothing parameter	Variance estimation	c (for calculating CB)
Regressogram	$m = 30$	5.543	1.96
Kernel	$h = 0.0418$	5.775	3.38
Local Linear	$h = 0.0442$	5.769	3.39
Spline	$\lambda = 2.176 \times 10^{-6}$	4.982	3.58

Table 2.1: Summary of four estimators

Confidence band and estimation of each estimator is shown in Figure 2.1.

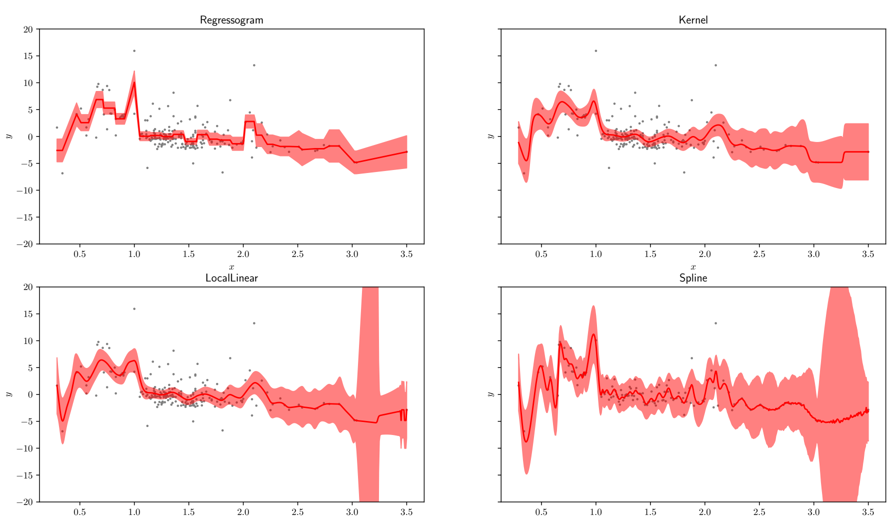

Figure 2.1: Confidence band and estimation of each estimator

A visual comparison of four estimators is shown in Figure 2.2.

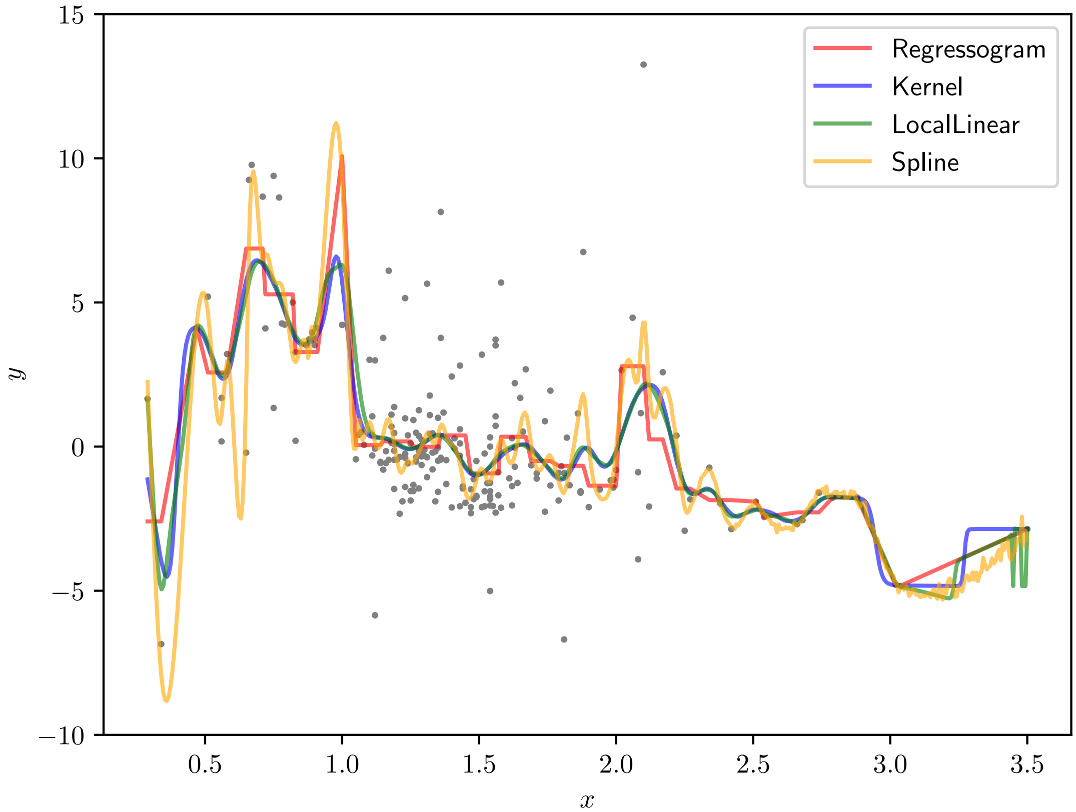

Figure 2.2: Comparison among four estimators

Appendix

Figures

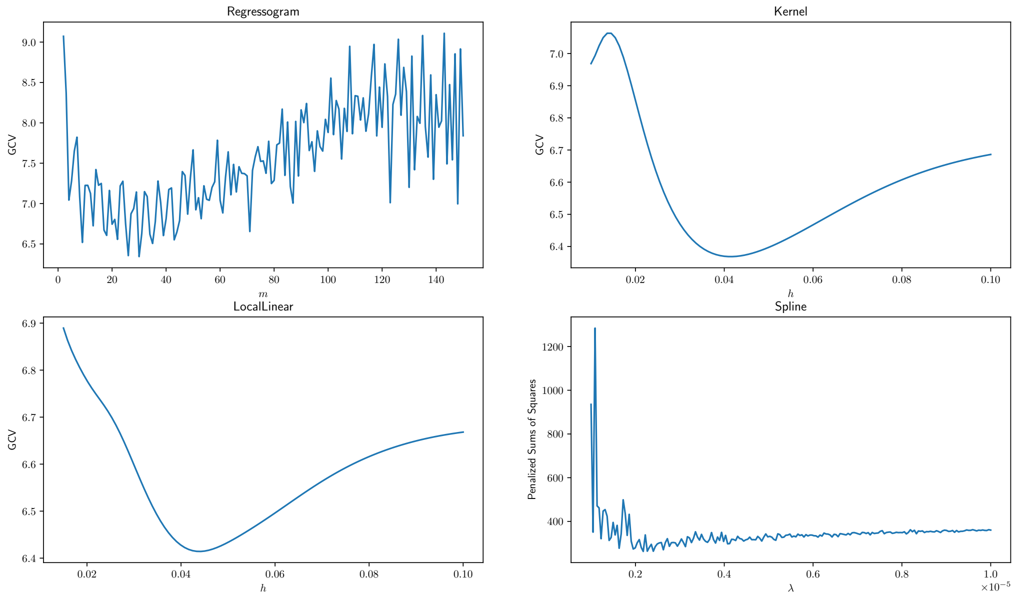

Figure A.1: GCV or Penalized sum of squares of four different estimators

Code

Some explainations about the code.

- I use python to implement the code instead of R.
- Estimators are calculated using basic functions. Note that derivative and integral are computed using modules `numdifftools` and `scipy.integrate`.
- Estimators are constructed using class for better readability and convenience.
- In the spline estimator, we use nartual cubic spline basis instead of B-spline basis for simplier code implementation. Nartual cubic spline basis is roughly defined as the following:

$$f(x) = \sum_{k=1}^{n} \beta_k N_k(x), \quad N_1(x) = 1, \quad N_2(x) = x, \quad N_{k+2}(x) = d_k(x) - d_{n-1}(x)$$

  where

$$d_k(x) = \frac{(x-\xi_k)_+^3 - (x-\xi_n)_+^3}{\xi_n - \xi_k} \quad \xi_k \text{ are knots}$$

- Note that some of the codes are not effective. Running all of the code may take as long as one hour.

- Code can also be found in https://thisiskunmeng.github.io/nonparametric/hw7.html.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.stats import norm
import scipy.integrate as integrate
from scipy.optimize import fsolve
import numdifftools as nd
from tqdm import tqdm


class Estimate:
    """
    Base Class for nonparametric regression estimator
    """

    def __init__(self, y: pd.Series, x: pd.Series):
        self.y = y
        self.x = x
        self.n = self.x.shape[0]
        self.parameter = None
        self.optimal_parameter = None
        self.effective_kernel = None  # L
        self.v = None  # trace of L
        self.v_hat = None  # trace of L^T L
        self.kappa0 = None  # use to calculate confidence interval coefficient
        self.c = None  # confidence interval coefficient
        self.method = None  # The method used to estimate the parameter

    def l_x(self, i) -> np.ndarray:
        pass

    def get_effective_kernel(self):
        self.effective_kernel = np.array([self.l_x(i) for i in self.x])
        self.v = np.trace(self.effective_kernel)
        self.v_hat = np.trace(self.effective_kernel.T @ self.effective_kernel)

    def set_optimal_parameter(self, para):
        self.optimal_parameter = para
        self.set_parameter(para)
        self.get_effective_kernel()

    def set_parameter(self, para):
        self.parameter = para
        self.get_effective_kernel()

    def estimate(self, x) -> float | np.ndarray:
        """
        :return: .. math:: \hat{r}_n(x)
        """
        if isinstance(x, np.ndarray):
            if x.ndim == 0:
                return np.sum(self.y.values * self.l_x(x))
            return np.array([self.estimate(i) for i in x])
        return np.sum(self.y.values * self.l_x(x))

    def plot_cv_score(self, low, high, num=100):
```

```
        h = np.linspace(low, high, num)
        score_f = self.cross_validation()
        y = [score_f(i) for i in tqdm(h, desc=self.method + " cross validation score")]
        plt.plot(h, y)
        return np.array([h, y])

    def get_kappa0(self):
        self.kappa0 = integrate.quad(self._get_tx_norm, self.x.min(), self.x.max())[0]

    def get_c(self, alpha=0.05):
        self.get_kappa0()
        self.c = fsolve(self._solve_c, np.array(1.96))[0]

    def _solve_c(self, x):
        return 2 * (1 - norm.cdf(x)) + self.kappa0 / np.pi * np.exp(-x ** 2 / 2) - 0.05

    def _get_tx_norm(self, x):
        return np.linalg.norm(self._get_t_derivative_x(x))

    def _get_tx(self, x):
        l_ix = self.l_x(x)
        t_ix = l_ix / np.linalg.norm(l_ix)
        return t_ix

    def _get_t_derivative_x(self, x):
        d = nd.Derivative(self._get_tx)
        return d(x)

    def cross_validation(self):
        def score(h):
            self.set_parameter(h)
            r = np.sum(
                ((self.y.values - self.estimate(self.x.values)) / (1 - self.v / self.n)) ** 2
            ) / self.n
            self.set_optimal_parameter(h)
            return r

        return score

    def variance_estimate(self) -> float:
        sigma = np.sum((self.y - self.estimate(self.x.values)) ** 2) / (self.n - 2 * self.v + self.v_hat)
        return sigma

    def confidence_band(self):
        self.get_c()
        sigma = np.sqrt(self.variance_estimate())

        def band(x):
            s = self.c * np.sqrt(sigma) * np.linalg.norm(self.l_x(x))
            return [self.estimate(x) - s, self.estimate(x) + s, self.estimate(x)]

        return band

    def draw_confidence_band(self, flag=True, ax=None):
        if ax is None:
            ax = plt
        band = self.confidence_band()
        if flag:
            x = np.linspace(self.x.min(), self.x.max(), 500)
```

```python
        else:
            x = self.x.sort_values()
        y = np.array([band(i) for i in tqdm(x, desc=self.method + " confidence band")])
        ax.plot(self.x, self.y, "o", color="grey", ms=2)
        ax.plot(x, y[:, 0], color="red", label="lower")
        ax.plot(x, y[:, 1], color="green", label="upper")
        ax.plot(x, y[:, 2], color="blue", label="estimate")
        ax.legend()
        return np.array([x, y[:, 0], y[:, 1]])

    @staticmethod
    def _k(x):
        """
        Gaussian kernel
        """
        return np.exp(-x ** 2 / 2) / np.sqrt(2 * np.pi)


class Regressogram(Estimate):
    def __init__(self, y: pd.Series, x: pd.Series):
        super().__init__(y, x)
        self.range = self.x.max() - self.x.min()
        self.method = "Regressogram"

    def get_effective_kernel(self):
        if self.parameter is None:
            raise ValueError("bins is not set")

        def i_th_effective_kernel(i):
            x_v = self.x.values
            ek = np.zeros(self.n)
            w = np.arange(self.x.min(), self.x.max(), self.range / self.parameter)
            w = np.append(w, self.x.max())
            for j in range(len(w) - 1):
                if w[j] <= self.x[i] <= w[j + 1]:
                    if j == len(w) - 2:
                        ek[(x_v >= w[j]) & (x_v <= w[j + 1])] = 1 / len(self.x[(self.x >= w[j]) & (self.x <= w[j + 1])])
                    else:
                        ek[(x_v >= w[j]) & (x_v < w[j + 1])] = 1 / len(self.x[(self.x >= w[j]) & (self.x < w[j + 1])])
            return ek

        self.effective_kernel = np.array([i_th_effective_kernel(i + 1) for i in range(self.n)]).T
        self.v = np.trace(self.effective_kernel)
        self.v_hat = np.trace(self.effective_kernel.T @ self.effective_kernel)

    def estimate(self, x):
        if isinstance(x, np.ndarray):
            return np.array([self.estimate(i) for i in x])
        w = np.arange(self.x.min(), self.x.max(), self.range / self.parameter)
        w = np.append(w, self.x.max())
        if x < w[1]:
            return self.y[self.x <= w[1]].mean()
        if x > w[-2]:
            return self.y[self.x >= w[-2]].mean()
        for j in range(len(w) - 1):
            if w[j] <= x < w[j + 1]:
                return self.y[(self.x >= w[j]) & (self.x < w[j + 1])].mean()

    def plot_cv_score(self, low=2, high=150, num=149):
```

```
        h = np.linspace(low, high, num)
        score_f = self.cross_validation()
        y = [score_f(i) for i in h]
        plt.plot(h, y)
        return np.array([h, y])

    def confidence_band(self):
        self.get_c()
        sigma = np.sqrt(self.variance_estimate())

        def band(x):
            w = np.arange(self.x.min(), self.x.max(), self.range / self.parameter)
            w = np.append(w, self.x.max())
            l_x = np.zeros(self.n)
            l_x_norm = 0
            if x < w[1]:
                l_x[self.x <= w[1]] = 1 / len(self.x[self.x <= w[1]])
                l_x_norm = np.linalg.norm(l_x)
            if x > w[-2]:
                l_x[self.x >= w[-2]] = 1 / len(self.x[self.x >= w[-2]])
                l_x_norm = np.linalg.norm(l_x)
            for j in range(len(w) - 1):
                if w[j] <= x < w[j + 1]:
                    l_x[(self.x >= w[j]) & (self.x < w[j + 1])] = 1 / len(
                        self.x[(self.x >= w[j]) & (self.x < w[j + 1])])
                    l_x_norm = np.linalg.norm(l_x)
            s = self.c * np.sqrt(sigma) * l_x_norm
            return [self.estimate(x) - s, self.estimate(x) + s, self.estimate(x)]

        return band

    def get_kappa0(self):
        self.kappa0 = 0

    def get_c(self, alpha=0.05):
        self.c = norm.ppf(1 - alpha / 2)


class Kernel(Estimate):
    def __init__(self, y: pd.Series, x: pd.Series):
        super().__init__(y, x)
        self.method = "Kernel"

    def l_x(self, x) -> np.ndarray:
        return self._k((x - self.x) / self.parameter) / self._k((x - self.x) / self.parameter).sum()


class LocalLinear(Estimate):
    def __init__(self, y: pd.Series, x: pd.Series):
        super().__init__(y, x)
        self.method = "Local Linear"

    def l_x(self, x):
        b_x = self._b_x(x)
        s = np.sum(b_x)
        l_ix = b_x / s
        return l_ix

    def _s_nj(self, x, j: {1, 2}):
```

```
        return np.sum(self._k((self.x - x) / self.parameter) * (self.x - x) ** j)

    def _b_x(self, x):
        return self._k((self.x - x) / self.parameter) * (self._s_nj(x, 2) - (self.x - x) * self._s_nj(x, 1))


class Spline(Estimate):
    def __init__(self, y: pd.Series, x: pd.Series):
        super().__init__(y, x)
        self.data = None
        self.process_same_value()
        self.method = "Spline"
        self._G()
        self._Omega()
        self.inv_np = None
        self.beta = None

    def process_same_value(self):
        self.data = pd.DataFrame({"x": self.x, "y": self.y})
        self.x = pd.Series(self.data["x"].value_counts().index.sort_values().values)
        self.y = pd.Series([self.data[self.data["x"] == i]["y"].mean() for i in self.x])
        self.n = len(self.x)

    def set_parameter(self, para):
        self.parameter = para
        self._get_inv()
        self.get_effective_kernel()

    def l_x(self, x) -> np.ndarray:
        return self._g_x(x)[0:self.n].T @ self.inv_np @ self.G.T

    def _get_inv(self):
        A = self.G.T @ self.G + self.parameter * self.Omega
        self.inv_np = np.linalg.inv(A)
        self.beta = self.inv_np @ self.G.T @ self.y.values.T

    def _g_x(self, x):
        g = np.zeros(self.n + 2)
        d_K_1 = self._g_x_i(x, self.x.values[-2])
        g[0] = 1
        g[1] = x
        g[2:] = np.array(
            [
                self._g_x_i(x, i) - d_K_1 if i < x else 0 for i in self.x
            ])
        return g

    def _g_x_i(self, x, i):
        if x < i:
            temp = 0
        elif x < self.x.values[-1]:
            temp = ((x - i) ** 3) / (self.x.values[-1] - i)
        elif x >= self.x.values[-1]:
            temp = ((x - i) ** 3 - (x - self.x.values[-1]) ** 3) / (self.x.values[-1] - i)
        return temp

    def d2_g_x(self, x):
        g = np.zeros(self.n + 2)
        d2d_K_1 = self._d2_g_x_i(x, self.x.values[-2])
```

```
        g[0] = 0
        g[1] = 0
        g[2:] = np.array(
            [
                self._d2_g_x_i(x, i) - d2d_K_1 if i < x else 0 for i in self.x
            ])
        return g[0:self.n]

    def _d2_g_x_i(self, x, i):
        if x < i:
            temp = 0
        elif x < self.x.values[-1]:
            temp = (6 * (x - i)) / (self.x.values[-1] - i)
        elif x >= self.x.values[-1]:
            temp = (6 * ((x - i) - (x - self.x.values[-1]))) / (self.x.values[-1] - i)
        return temp

    def _G(self):
        self.G = np.array([self._g_x(i) for i in self.x])[0:self.n:, 0:self.n]

    def _Omega(self):
        self.Omega = np.array([self._omega_i(i) for i in tqdm(range(self.n), desc=self.method + " Omega")])

    def _omega_i(self, i):
        return integrate.quad_vec(lambda x: self.d2_g_x(x) * self.d2_g_x(x)[i], self.x.min(), self.x.max())[0]

    def cross_validation(self):
        def score(h):
            self.set_parameter(h)
            mse = np.sum(
                (self.y.values - self.estimate(self.x.values)) ** 2)
            penalty = self.parameter * \
                      integrate.quad(lambda x: np.sum(self.beta * self.d2_g_x(x)), self.x.min(), self.x.max())[0]
            return mse + penalty
        return score
```

```
df = pd.read_csv("https://www.stat.cmu.edu/~larry/all-of-nonpar/=data/glass.dat", sep="\s+", header=0)
target = df["RI"]
x_al = df["Al"]

reg = Regressogram(target, x_al)
bin_cv = reg.plot_cv_score()
plt.savefig("./fig/Regressogram_cv_score.pdf")
plt.clf()

reg.set_optimal_parameter(int(bin_cv[0, np.argmin(bin_cv[1])]))

bin_cb = reg.draw_confidence_band(flag=False)
plt.savefig("./fig/Regressogram_confidence_band.pdf")
plt.clf()

print(f"optimal smoothing parameter: {np.argmin(bin_cv[1]) + 2}")
print(f"Regressogram variance estimation: {reg.variance_estimate()}")
print(f"Regressogram c: {reg.c}")

ker = Kernel(target, x_al)
ker_cv = ker.plot_cv_score(0.01, 0.1)
```

```
plt.savefig("./fig/Kernel_cv_score.pdf")
plt.clf()

ker.set_optimal_parameter(ker_cv[0, np.argmin(ker_cv[1])])

ker_cb = ker.draw_confidence_band()
plt.savefig("./fig/Kernel_confidence_band.pdf")
plt.clf()

print(f"optimal smoothing parameter: {ker_cv[0, np.argmin(ker_cv[1])]}")
print(f"Kernel variance estimation: {ker.variance_estimate()}")
print(f"Kernel c: {ker.c}")

print(f"Kernel c: {ker.c}")

ll = LocalLinear(target, x_al)
ll_cv = ll.plot_cv_score(0.015, 0.1)
plt.savefig("./fig/LocalLinear_cv_score.pdf")
plt.clf()

ll.set_optimal_parameter(ll_cv[0, np.argmin(ll_cv[1])])

ll_cb = ll.draw_confidence_band()
plt.savefig("./fig/LocalLinear_confidence_band.pdf")
plt.clf()

print(f"optimal smoothing parameter: {ll_cv[0, np.argmin(ll_cv[1])]}")
print(f"LocalLinear variance estimation: {ll.variance_estimate()}")
print(f"LocalLinear c: {ll.c}")

spline = Spline(target, x_al)
spline_cv = spline.plot_cv_score(1e-06, 1e-05, 200)
plt.savefig("./fig/Spline_cv_score.pdf")
plt.clf()

spline.set_optimal_parameter(spline_cv[0, np.argmin(spline_cv[1])])

spline_cb = spline.draw_confidence_band()
plt.savefig("./fig/Spline_confidence_band.pdf")
plt.clf()

print(f"optimal smoothing parameter: {spline_cv[0, np.argmin(spline_cv[1])]}")
print(f"Spline variance estimation: {spline.variance_estimate()}")
print(f"Spline c: {spline.c}")
```

```
optimal smoothing parameter: 30
Regressogram variance estimation: 5.5430334733388245
Regressogram c: 1.959963984540054

optimal smoothing parameter: 0.0441919191919192
LocalLinear variance estimation: 5.768970862437912
LocalLinear c: 3.3862804226633734

optimal smoothing parameter: 2.175879396984925e-06
Spline variance estimation: 4.981818961988229
Spline c: 3.578015201230727
```

```
fig, ax = plt.subplots(2, 2, figsize=(16, 9))
ax[0, 0].set_title('Regressogram')
ax[0, 1].set_title('Kernel')
ax[1, 0].set_title('LocalLinear')
ax[1, 1].set_title('Spline')
ax[0, 0].set_xlabel(r'$m$')
ax[0, 1].set_xlabel(r'$h$')
ax[1, 0].set_xlabel(r'$h$')
ax[1, 1].set_xlabel(r'$\lambda$')
ax[0, 0].set_ylabel('GCV')
ax[0, 1].set_ylabel('GCV')
ax[1, 0].set_ylabel('GCV')
ax[1, 1].set_ylabel('Penalized Sums of Squares')
ax[0, 0].plot(bin_cv[0], bin_cv[1])
ax[0, 1].plot(ker_cv[0], ker_cv[1])
ax[1, 0].plot(ll_cv[0], ll_cv[1])
ax[1, 1].plot(spline_cv[0], spline_cv[1])
plt.savefig('./fig/cv_score.pdf')
```

```
fig, ax = plt.subplots(2, 2, figsize=(16, 9), sharey='all')
ax[0, 0].set_ylim(-20, 20)
ax[0, 0].set_title('Regressogram')
ax[0, 1].set_title('Kernel')
ax[1, 0].set_title('LocalLinear')
ax[1, 1].set_title('Spline')
ax[0, 0].set_xlabel(r'$x$')
ax[0, 1].set_xlabel(r'$x$')
ax[1, 0].set_xlabel(r'$x$')
ax[1, 1].set_xlabel(r'$x$')
ax[0, 0].set_ylabel(r'$y$')
ax[0, 1].set_ylabel(r'$y$')
ax[1, 0].set_ylabel(r'$y$')
ax[1, 1].set_ylabel(r'$y$')
ax[0, 0].scatter(reg.x, reg.y, s=2, color='grey')
ax[0, 1].scatter(ker.x, ker.y, s=2, color='grey')
ax[1, 0].scatter(ll.x, ll.y, s=2, color='grey')
ax[1, 1].scatter(spline.x, spline.y, s=2, color='grey')
ax[0, 0].fill_between(bin_cb[0], bin_cb[1], bin_cb[2], alpha=.5, color='red')
ax[0, 0].plot(bin_cb[0], reg.estimate(bin_cb[0]), color='red')
ax[0, 1].fill_between(ker_cb[0], ker_cb[1], ker_cb[2], alpha=.5, color='red')
ax[0, 1].plot(ker_cb[0], ker.estimate(ker_cb[0]), color='red')
ax[1, 0].fill_between(ll_cb[0], ll_cb[1], ll_cb[2], alpha=.5, color='red')
ax[1, 0].plot(ll_cb[0], ll.estimate(ll_cb[0]), color='red')
ax[1, 1].fill_between(spline_cb[0], spline_cb[1], spline_cb[2], alpha=.5, color='red')
ax[1, 1].plot(spline_cb[0], spline.estimate(spline_cb[0]), color='red')
plt.savefig('./fig/confidence_band.pdf')
```

```
fig, ax = plt.subplots()
ax.set_ylim(-10, 15)
ax.scatter(x_al, target, s=2, color='grey')
ax.plot(bin_cb[0], reg.estimate(bin_cb[0]), color='red', alpha=0.6, label='Regressogram')
ax.plot(ker_cb[0], ker.estimate(ker_cb[0]), color='blue', alpha=0.6, label='Kernel')
ax.plot(ll_cb[0], ll.estimate(ll_cb[0]), color='green', alpha=0.6, label='LocalLinear')
ax.plot(spline_cb[0], spline.estimate(spline_cb[0]), color='orange', alpha=0.6, label='Spline')
```

```python
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.legend()
plt.savefig('./fig/compare.pdf')
```