

T5 Modelos basados en ejemplos

Índice

1 Introducción

- 1.1 Modelos paramétricos
- 1.2 Modelos no paramétricos

2 K-vecinos más próximos

- 2.1 Definición
- 2.2 Resultados asintóticos
- 2.3 Ejemplo
- 2.4 Diagrama de Voronoi
- 2.5 K como hiperparámetro de regularización
- 2.6 The curse of dimensionality
 - 2.6.1 Concentración de distancias
 - 2.6.2 Pérdida de localidad
- 2.7 Reducción del coste computacional
- 2.8 Reconocimiento (de conjunto) abierto

3 Estimación con kernels densidad

- 3.1 Kernels densidad
- 3.2 Estimador de densidad ventana de Parzen
- 3.3 Elección del parámetro ancho de banda
- 3.4 De clasificación KDE a KNN
- 3.5 Regresión kernel

3.5.1 Nadaraya-Watson

3.5.2 Nadaraya-Watson Gaussiano con softmax

1 Introducción

1.1 Modelos paramétricos

Los modelos paramétricos pueden ser incondicionales, $p(\mathbf{y} \mid \boldsymbol{\theta})$, o condicionales, $p(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta})$, donde $\boldsymbol{\theta}$ es un vector de parámetros de dimensión fija. $\boldsymbol{\theta}$ se estima con datos $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^N$ que luego se tiran.

1.2 Modelos no paramétricos

A diferencia de los paramétricos, los no paramétricos mantienen los datos (en inferencia), por lo que el número efectivo de parámetros del modelo crece con \mathcal{D} . Consideraremos modelos definidos con la ayuda de una **medida de disimilitud** entre la entrada de test \mathbf{x} y cada dato de entrenamiento \mathbf{x}_n . Alternativamente, en lugar de una medida de disimilitud podemos emplear una **medida de disimilitud** o **función distancia** $d(\mathbf{x}, \mathbf{x}_n)$. Otros nombres con que se suelen usar son **modelos basados en ejemplos**, **instance-based learning** o **memory-based learning**.

2 K-vecinos más próximos

2.1 Definición

Clasificador (por los) K vecinos más próximos o K nearest neighbor (KNN): dada una entrada \mathbf{x} , busca K prototipos (datos) más cercanos a \mathbf{x} , $N_K(\mathbf{x}, \mathcal{D})$, y toma sus etiquetas para derivar una distribución sobre las salidas en \mathbf{x} :

$$p(y = c \mid \mathbf{x}, \mathcal{D}) = \frac{1}{K} \sum_{n \in N_K(\mathbf{x}, \mathcal{D})} \mathbb{I}(y_n = c)$$

Clasificador KNN en breve: retorna la etiqueta más votada (mayoritaria) si es única; si no, en caso de empate a votos entre dos o más clases, devuelve la etiqueta del prototipo más cercano entre los prototipos de las clases empatadas

Clasificador por el vecino más próximo o nearest neighbor (NN): KNN en el caso particular $K = 1$, con función predictiva delta:

$$p(y = c \mid \mathbf{x}, \mathcal{D}) = \delta(c, y_n) \quad \text{con conjunto unitario} \quad N_1(\mathbf{x}, \mathcal{D}) = \{n\}$$

Desempates a distancia decididos al azar: asumimos que la probabilidad de empate a distancia entre dos prototipos es insignificante por lo que, en caso de dos o más posibles conjuntos de K prototipos más cercanos a \mathbf{x} , escogemos uno de ellos al azar

Desempates a votos decididos por el NN: dado que la probabilidad de empate a votos no es insignificante (por ejemplo con $K = 2$), en este caso no decidimos al azar, sino que aplicamos NN entre los prototipos de las clases empatadas

Parámetros principales de KNN: tamaño del entorno local, K , y la distancia $d(\mathbf{x}, \mathbf{x}')$ con la que compara cualquier par de puntos en el espacio de representación de los datos, típicamente \mathbb{R}^D

Elección de la función distancia: se suele usar la Euclídea o, más generalmente, se introduce alguna función con parámetros a aprender; por ejemplo, la distancia de Mahalanobis

$$d_M(\mathbf{x}, \boldsymbol{\mu}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^t M (\mathbf{x} - \boldsymbol{\mu})} \quad \text{con} \quad M \succ 0$$

2.2 Resultados asintóticos

Análisis asintótico: KNN ha sido objeto de amplio estudio cuando $N \rightarrow \infty$

NN asintótico: su error de clasificación no es superior a dos veces el de Bayes

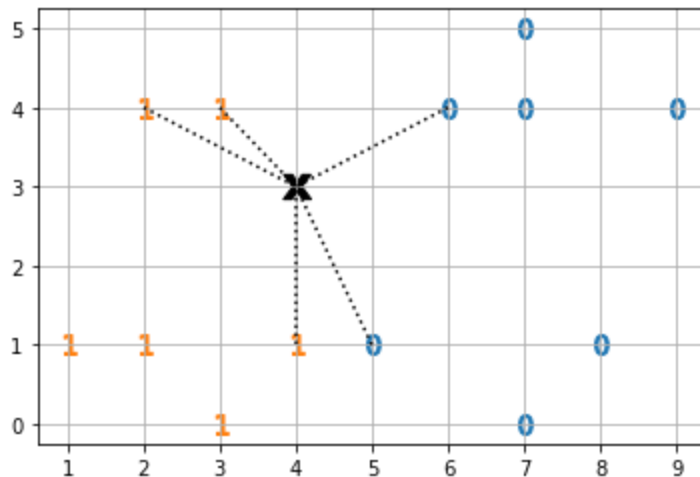
KNN asintótico: su error converge al de Bayes si K se escoge tal que $K \rightarrow \infty$ y $K/N \rightarrow 0$; por ejemplo, con $K = \sqrt{N}$

2.3 Ejemplo

En un problema de clasificación en dos clases, $y \in \{0, 1\}$, 5NN para un punto de test \mathbf{x} halla 3 vecinos de la clase 1 y 2 de la clase 0, por lo que la probabilidad de que \mathbf{x} pertenezca a la clase 1 se estima como:

$$p(y = 1 \mid \mathbf{x}, \mathcal{D}) = \frac{3}{5} = 0.6$$

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from sklearn.neighbors import NearestNeighbors
X0 = np.array([[5, 1], [6, 4], [7, 0], [7, 4], [7, 5], [8, 1], [9, 4]], dtype=float)
X1 = np.array([[1, 1], [2, 1], [2, 4], [3, 0], [3, 4], [4, 1]], dtype=float)
N0 = X0.shape[0]; N1 = X1.shape[0]
X = np.vstack((X0, X1))
y = np.vstack((np.zeros((N0, 1)), np.ones((N1, 1))))
x = np.array([[4, 3]]) # <<< test
fig, ax = plt.subplots()
ax.grid(); ax.scatter(*x, c='k', marker=r'$\mathbf{x}$', s=200)
ax.scatter(*X0.T, marker=r'$0$', s=100)
ax.scatter(*X1.T, marker=r'$1$', s=100)
K = 5 # <<< número de vecinos
KNN = NearestNeighbors(n_neighbors=K).fit(X)
_, KNN_ind = KNN.kneighbors([x])
lines = np.zeros((K, 2, 2))
lines[:, 0, :] = np.squeeze(X[KNN_ind, :])
lines[:, 1, :] = np.repeat([x], K, axis=0)
ax.add_collection(LineCollection(lines, colors='black', linestyle='dotted'));
```

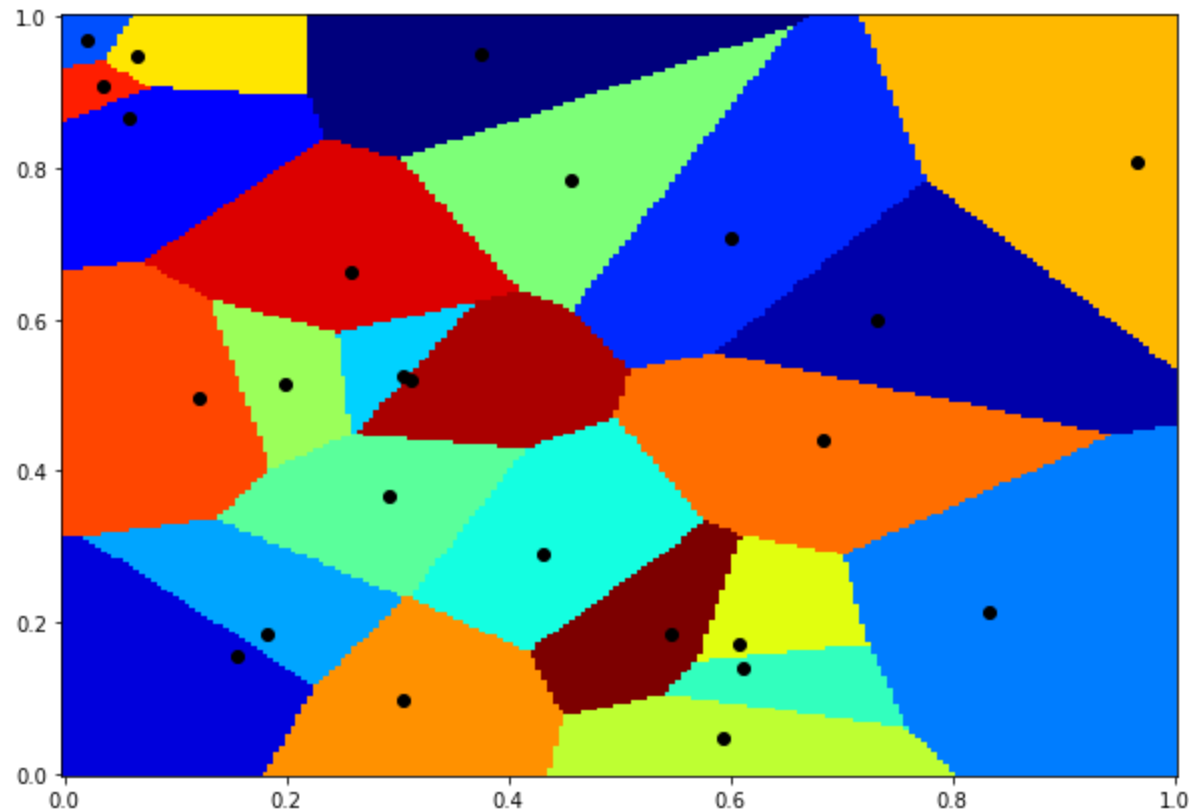


2.4 Diagrama de Voronoi

Partición del espacio inducida por NN: consta de una región $V(\mathbf{x}_n)$ por cada dato \mathbf{x}_n tal que todos los puntos en $V(\mathbf{x}_n)$ están más cerca de \mathbf{x}_n que de cualquier otro dato

Diagrama y celda de Voronoi: partición inducida por NN en el plano; cada región de la partición se denomina **celda de Voronoi**

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import KDTree, Voronoi
np.random.seed(42)
data = np.random.rand(25, 2)
fig, ax = plt.subplots(figsize=(10,7))
#voronoi_plot_2d(vor, ax=ax, show_vertices=False)
xlim = plt.xlim(); ylim = plt.ylim()
tree = KDTree(data)
x = np.linspace(xlim[0], xlim[1], 200)
y = np.linspace(ylim[0], ylim[1], 200)
xx, yy = np.meshgrid(x, y)
xy = np.c_[xx.ravel(), yy.ravel()]
plt.plot(data[:, 0], data[:, 1], 'ko')
plt.pcolormesh(x, y, tree.query(xy)[1].reshape(200, 200), cmap='jet');
```



2.5 K como hiperparámetro de regularización

Buenos resultados con K reducido: en general, los resultados teóricos y prácticos apoyan la idea de que un K reducido obtendrá resultados comparativamente buenos

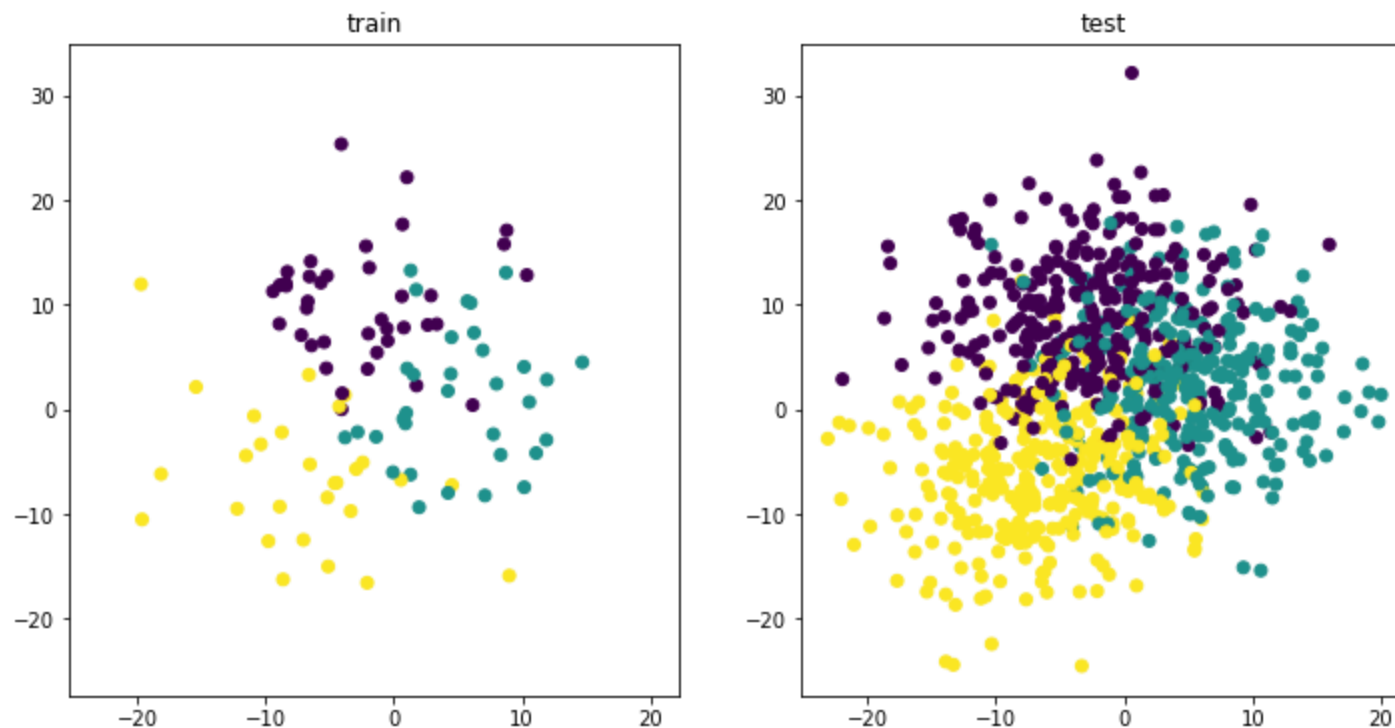
Interpretación de K como hiperparámetro de regularización: si K es muy pequeño, el modelo tenderá a sobre-ajustarse a los datos mientras que, si K es muy grande, se aproximará a las probabilidades a priori de las clases; por tanto, K puede interpretarse como un hiperparámetro de regularización con el que optimizar el grado de ajuste a los datos

```
In [3]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import cross_val_score
```

```

from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=1000, centers=3, n_features=2, cluster_std=6, random_state=42)
ntrain = 100; x_train = X[:ntrain]; y_train = y[:ntrain]; x_test = X[ntrain:]; y_test = y[ntrain:]
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
ax=axes[1]; ax.set_title('test'); ax.scatter(*x_test.T, c=y_test)
x_min, x_max = ax.get_xlim(); y_min, y_max = ax.get_ylim()
ax=axes[0]; ax.set_title('train'); ax.scatter(*x_train.T, c=y_train)
ax.set_xlim(x_min, x_max); ax.set_ylim(y_min, y_max);

```



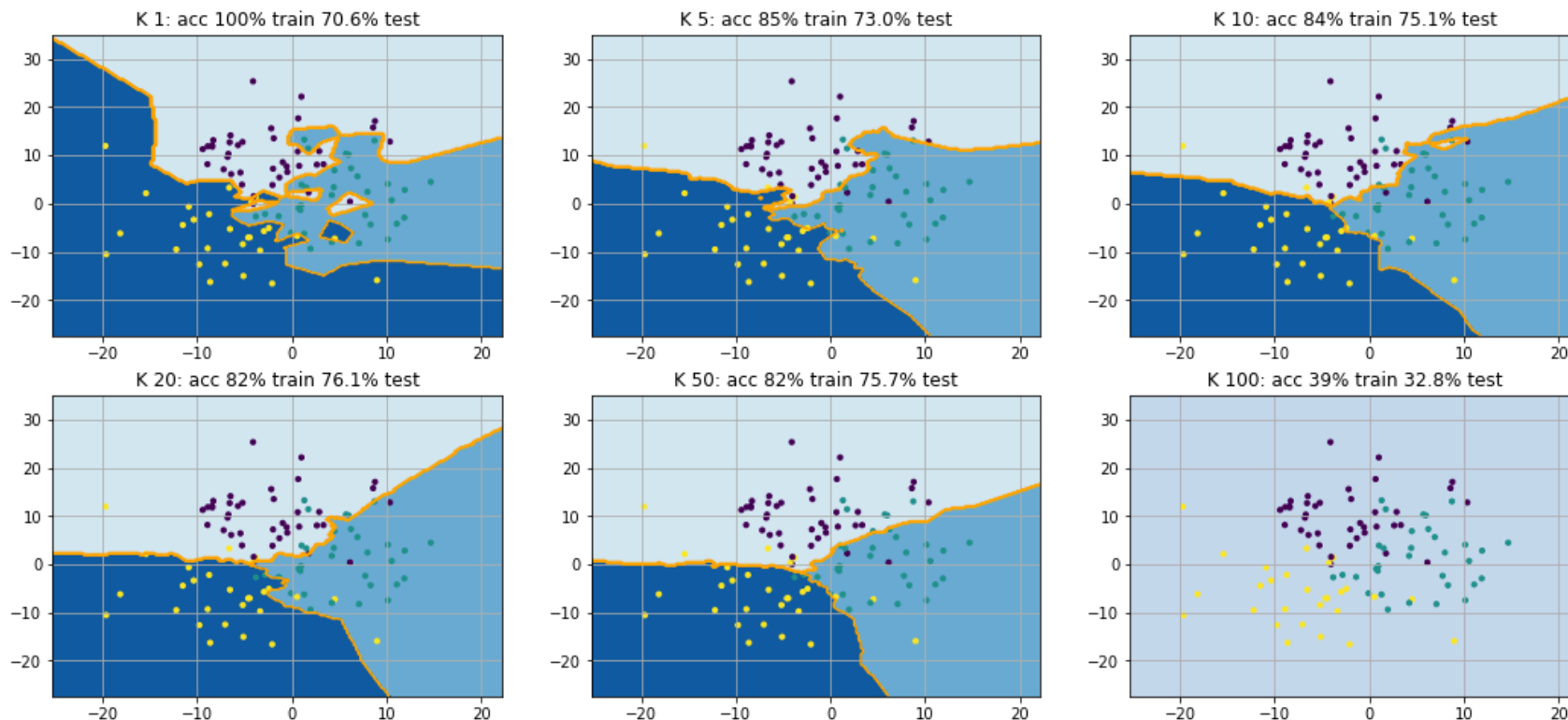
```

In [4]: from sklearn.neighbors import KNeighborsClassifier as KNN
from sklearn.metrics import accuracy_score
Ks = [1, 5, 10, 20, 50, ntrain]; nrows = 2; ncols = 3; n_Ks = len(Ks)
fig, axes = plt.subplots(nrows, ncols, figsize=(18, 8))
for i, K in enumerate(Ks):
    ax = axes.flat[i]
    clf = KNN(n_neighbors=K).fit(x_train, y_train)
    acc_train = accuracy_score(y_train, clf.predict(x_train))
    acc_test = accuracy_score(y_test, clf.predict(x_test))
    ax.grid(); ax.set_xlim(x_min, x_max); ax.set_ylim(y_min, y_max);

```

```
xx, yy = np.meshgrid(np.linspace(x_min, x_max, num=200), np.linspace(y_min, y_max, num=200))
zz = clf.predict(np.c_[xx.ravel(), yy.ravel()])
ax.contour(xx, yy, zz.reshape(xx.shape), 2, colors='orange', linestyle='solid')
ax.contourf(xx, yy, zz.reshape(xx.shape), 2, cmap='Blues'); ax.scatter(*x_train.T, c=y_train, s=10)
ax.set_title('K {}: acc {:.0%} train {:.1%} test'.format(K, acc_train, acc_test))
```

/tmp/ipykernel_59136/3721934215.py:13: UserWarning: No contour levels were found within the data range.
 ax.contour(xx, yy, zz.reshape(xx.shape), 2, colors='orange', linestyle='solid')



2.6 The curse of dimensionality

Maldición de la dimensionalidad: expresión bien conocida para referirse al hecho de que, por lo general, muchas técnicas clásicas como el clasificador KNN empeoran sensiblemente con entradas de alta dimensión

Maldición de la dimensionalidad y KNN: en el caso del clasificador KNN, la maldición de la dimensionalidad se explica fácilmente ya que, al aumentar la dimensión, las distancias se igualan y el NN se halla en un entorno

cada vez menos local

2.6.1 Concentración de distancias

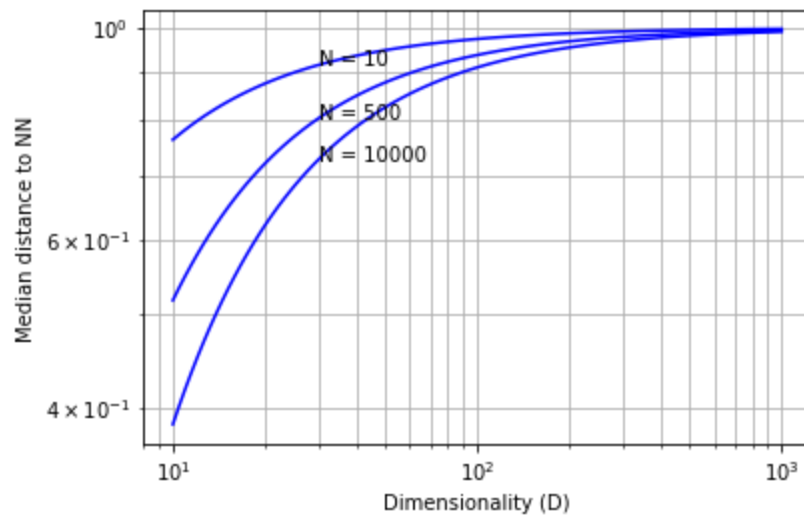
Concentración de distancias: fenómeno por el cual las distancias se igualan en altas dimensiones

Distancia mediana al NN en función de D : si tenemos N datos uniformemente distribuidos en una bola unitaria D -dimensional centrada en el origen (donde asumimos que se halla la muestra de test), se puede comprobar que la distancia mediana del origen a su vecino más cercano es (ver [solución al ejercicio 2.3 de HTF09](#))

$$d(D, N) = \left(1 - \left(\frac{1}{2} \right)^{1/N} \right)^{1/D}$$

- Para $N = 500$ y $D = 10$, $d(D, N) \approx 0.52$, esto es, más de la mitad de la distancia a la frontera
- En general, el vecino más cercano se aproxima a la frontera de la bola en altas dimensiones, donde también se hallarán el resto de datos a distancia (prácticamente) unitaria

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
D = np.logspace(1, 3, 100)
plt.xscale('log', base=10); plt.yscale('log', base=10); plt.grid(which='both')
for N in [10, 500, 10000]:
    d = ( 1 - .5 ** (1 / N) ) ** (1 / D)
    plt.plot(D, d, 'b-')
    plt.text(30, ( 1 - .5 ** (1 / N) ) ** (1 / 30), 'N = %d' % N)
plt.xlabel('Dimensionality (D)')
plt.ylabel('Median distance to NN')
```

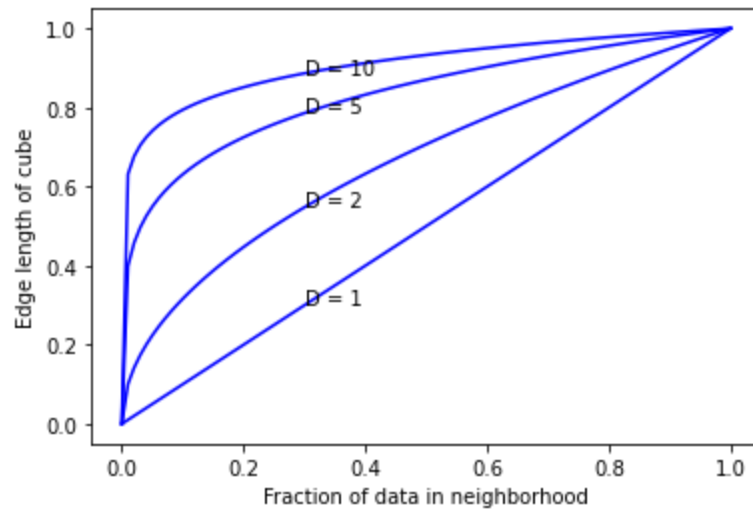


2.6.2 Pérdida de localidad

Pérdida de localidad: en general, con el aumento de la dimensión, tenemos que considerar entornos cada vez menos locales

- Supongamos que los datos se distribuyen uniformemente en un hipercubo unitario D -dimensional
- Consideremos un entorno hipercúbico centrado en la muestra de test que capture una fracción r de los datos
- Longitud esperada del lado del entorno que captura una fracción r del volumen unitario: $e_D(r) = r^{1/D}$
- Como $e_{10}(0.01) = 0.63$, debemos cubrir el 63% de cada variable para capturar el 1% de los datos

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
r = np.linspace(0, 1, 100)
for D in [1, 2, 5, 10]:
    e = r ** (1 / D)
    plt.plot(r, e, 'b-')
    plt.text(0.3, 0.3 ** (1 / D), 'D = %d' % D)
plt.xlabel('Fraction of data in neighborhood')
plt.ylabel('Edge length of cube')
```



2.7 Reducción del coste computacional

Elevado coste computacional de KNN: espacial y temporal, debido al mantenimiento de (todos) los datos en inferencia

Técnicas de reducción del coste espacial: eliminan prototipos que no afectan a las fronteras de decisión

Técnicas de reducción del coste temporal: búsqueda eficiente de K vecinos, exacta y aproximada (para $D > 10$)

- **K-d tree:** divide el espacio en regiones de lados paralelos a los ejes, o con algún método de clustering basado en puntos ancla
- **Locality sensitive hashing (LSH):** técnica popular de 1999; más recientemente se aprende la función de hashing

Librería popular para la búsqueda eficiente de vecinos: [FAISS](#)

2.8 Reconocimiento (de conjunto) abierto

Closed world assumption: asunción típica según la cual el conjunto de clases en un problema de clasificación se halla prefijado

Open set recognition: posibilidad de ampliar el conjunto de clases con muestras de test provenientes de nuevas categorías

KNN y open set recognition: en contraste con otras técnicas, los clasificadores KNN pueden adaptarse con relativa facilidad

Tareas típicas de open set recognition:

- **Novelty detection:** el sistema detecta que la muestra de test es de una clase no vista antes; por ejemplo una cara desconocida
- **Incremental learning, online learning, life-long learning o continual learning:** si el sistema detecta una nueva clase con éxito, pregunta por el id de la nueva clase y la añade a las existentes
- **Out-of-distribution (OOD) detection:** se detecta que la muestra de test no es de clase conocida ni desconocida, sino que procede de una distribución enteramente distinta; p.e., una foto sin cara
- **Few-shot classification:** tenemos pocos ejemplos (tal vez uno solo) de cada clase; caso muy adecuado para KNN y típico en online
- **Person re-identification o face verification:** se comprueba si la persona o cara de test es, con seguridad, de clase conocida
- **Entity resolution o linking:** se trata de determinar si cadenas diferentes (p.e. "John Smith" y "Jon Smith") se refieren a la misma entidad o no
- **Multi-object tracking:** cuando un sistema de radar detecta un nuevo "blip", debe decidir si se trata de un objeto ya en seguimiento o un nuevo objeto que ha entrado en el espacio aéreo

3 Estimación con kernels densidad

3.1 Kernels densidad

Kernel density estimation (KDE): aproximación no paramétrica a la estimación de densidades que define un modelo generativo $p(\mathbf{x})$

Kernel densidad: función no negativa $\mathcal{K} : \mathbb{R} \rightarrow \mathbb{R}^{\geq 0}$, normalizada y simétrica:

$$\int \mathcal{K}(x) dx = 1 \quad \text{y} \quad \mathcal{K}(-x) = \mathcal{K}(x)$$

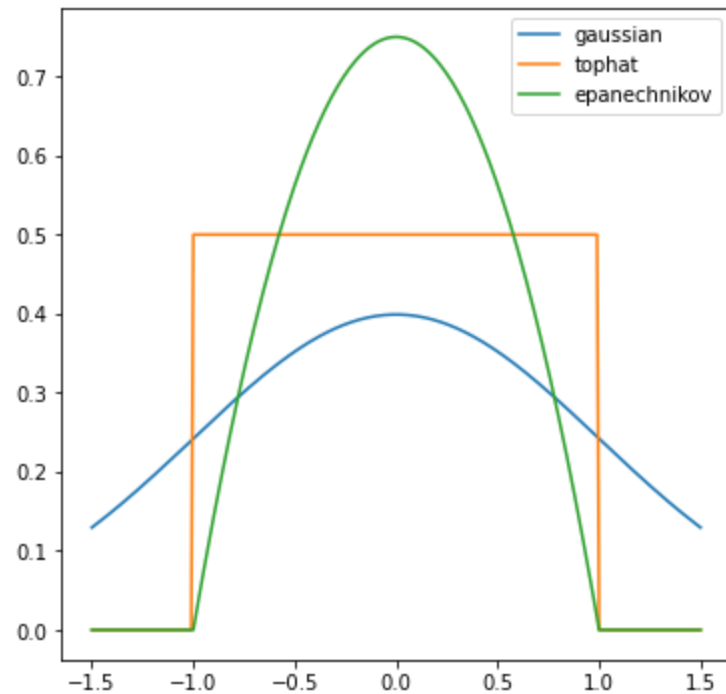
La simetría implica $\int x\mathcal{K}(x) dx = 0$ y, en general:

$$\int x\mathcal{K}(x - x_n) dx = x_n$$

Kernels populares:

Gaussiano:	$\mathcal{K}(x) = (2\pi)^{-\frac{1}{2}} e^{-x^2/2}$	diferenciable, pero no compacto
Tophat:	$\mathcal{K}(x) = \frac{1}{2} \mathbb{I}(x \leq 1)$	compacto, pero no diferenciable
Epanechnikov:	$\mathcal{K}(x) = \frac{3}{4} (1 - x^2) \mathbb{I}(x \leq 1)$	compacto y diferenciable salvo fronteras

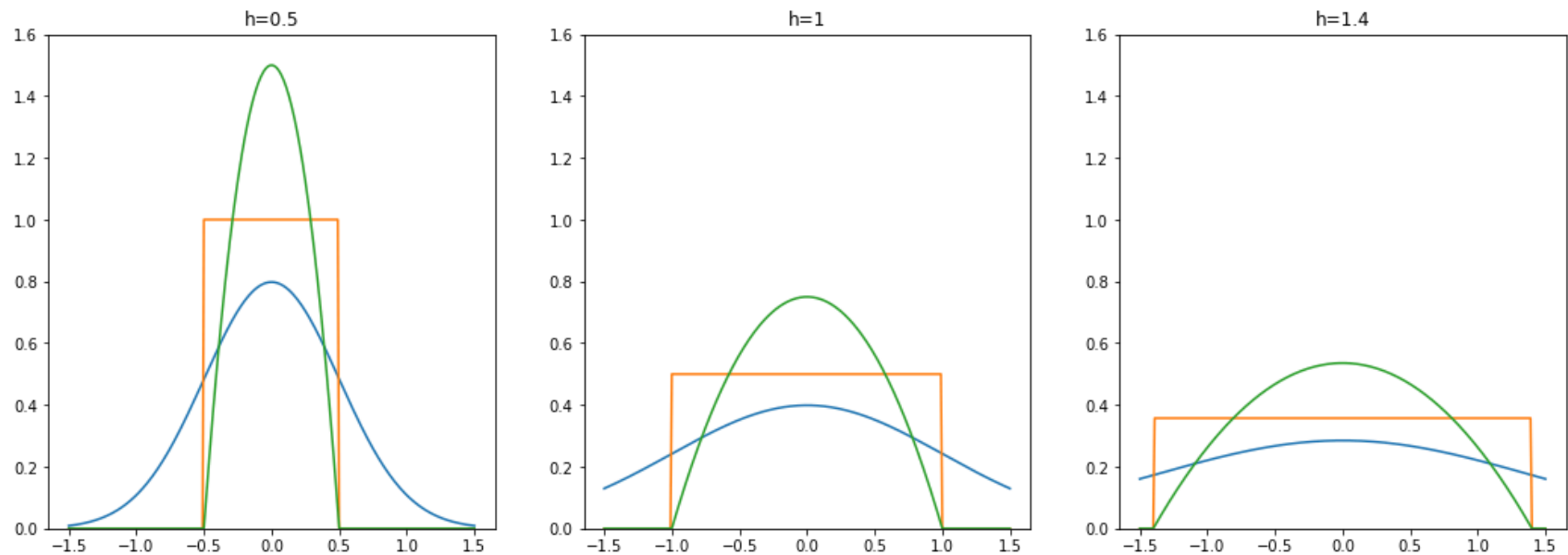
```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KernelDensity
X = np.arange(-1.5, 1.501, 0.01)[: , np.newaxis]
plt.figure(figsize=(6, 6));
for K in ['gaussian', 'tophat', 'epanechnikov']:
    kde = KernelDensity(kernel=K, bandwidth=1).fit([[.0]])
    log_dens = kde.score_samples(X)
    plt.plot(X, np.exp(log_dens), label=K)
plt.legend();
```



Ancho de banda o bandwidth: parámetro $h > 0$ que controla el ancho de un kernel

$$\mathcal{K}_h(x) = \frac{1}{h} \mathcal{K}\left(\frac{x}{h}\right)$$

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KernelDensity
X = np.arange(-1.5, 1.501, 0.01)[:, np.newaxis]
fig, axes = plt.subplots(1, 3, figsize=(18, 6))
for i, h in enumerate([.5, 1, 1.4]):
    axes[i].set_title(r'h={}'.format(h))
    axes[i].set_ylim(0, 1.6)
    for K in ['gaussian', 'tophat', 'epanechnikov']:
        log_dens = KernelDensity(kernel=K, bandwidth=h).fit([.0]).score_samples(X)
        axes[i].plot(X, np.exp(log_dens))
```



Radial basis function (RBF): kernel que generaliza los kernels 1d a vectores

$$\mathcal{K}_h(\mathbf{x}) \propto \mathcal{K}_h(\|\mathbf{x}\|)$$

RBF Gaussiano: h juega el papel de desviación típica (en cada dimensión)

$$\begin{aligned}
 \mathcal{K}_h(\mathbf{x}) &= \prod_{d=1}^D \mathcal{K}_h(x_d) \\
 &= \prod_{d=1}^D \frac{1}{h} \frac{1}{(2\pi)^{1/2}} \exp\left(-\frac{1}{2} \left[\frac{x_d}{h}\right]^2\right) \\
 &= \frac{1}{h^D (2\pi)^{D/2}} \prod_{d=1}^D \exp\left(-\frac{1}{2h^2} x_d^2\right) \\
 &= \frac{1}{h^D (2\pi)^{D/2}} \exp\left(-\frac{1}{2} \left[\frac{\|\mathbf{x}\|}{h}\right]^2\right)
 \end{aligned}$$

3.2 Estimador de densidad ventana de Parzen

Estimación con una mixtura de Gaussianas: aun suponiendo coeficientes uniformes y Gaussianas hiperesféricas de tamaño idéntico y conocido, la estimación del número de componentes y medias resulta problemática

$$p(\mathbf{x} \mid \boldsymbol{\theta}) = \frac{1}{K} \sum_{k=1}^K \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}, \sigma^2 \mathbf{I})$$

Mixtura con una Gaussiana por dato: solución trivial pues cada dato es la media de su componente

$$p(\mathbf{x} \mid \mathcal{D}) = \frac{1}{N} \sum_{n=1}^N \mathcal{N}(\mathbf{x} \mid \mathbf{x}_n, \sigma^2 \mathbf{I})$$

Parzen window kernel density estimator (KDE): generaliza la mixtura con una Gaussiana por dato a un kernel densidad por dato

$$p(\mathbf{x} \mid \mathcal{D}) = \frac{1}{N} \sum_{n=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_n)$$

KDE vs mixturas: KDE no requiere escoger el número de componentes ni estimar parámetros, salvo el ancho de banda; ahora bien, dado que requiere mantener todos los datos, su coste espacial y temporal es muy elevado

Ejemplo: KDE en 1d con 6 datos y kernel Gaussiano

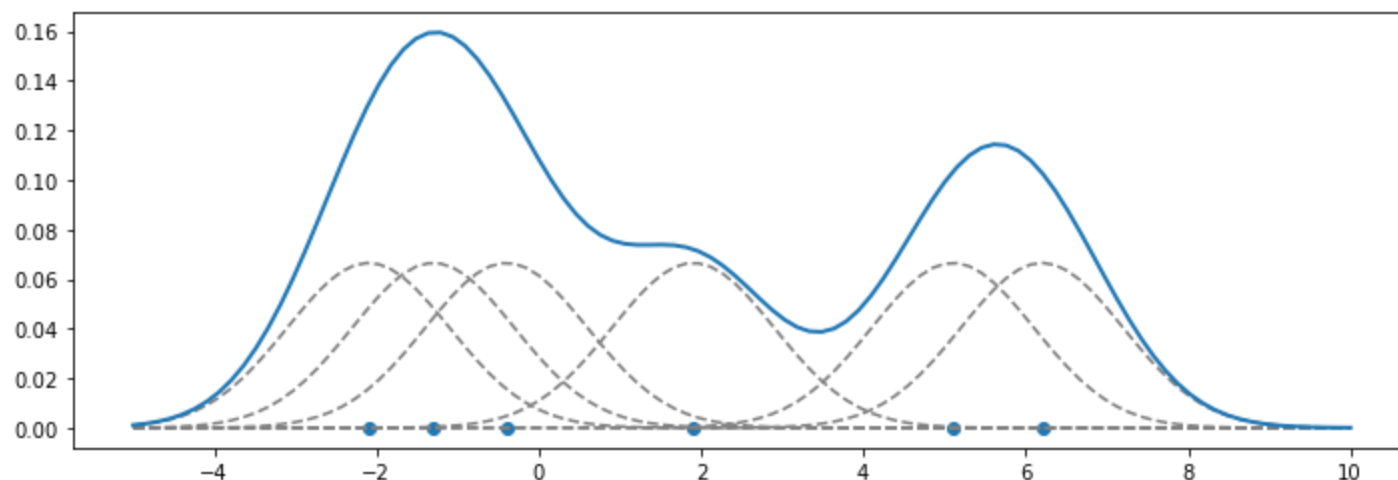
```
In [1]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KernelDensity
x_train = np.array([-2.1, -1.3, -0.4, 1.9, 5.1, 6.2])[:, np.newaxis]
N = x_train.shape[0]; h=1 # <<< bandwidth
x_test = np.linspace(-5, 10, 100)[:, np.newaxis]
plt.figure(figsize=(12, 4))
plt.scatter(x_train, np.zeros_like(x_train), marker="o")
for n in np.arange(N):
```



```

x = x_train[n].reshape((1, 1))
kde = KernelDensity(kernel='gaussian', bandwidth=h).fit(x)
log_dens = kde.score_samples(x_test)
plt.plot(x_test, np.exp(log_dens) / N, c="gray", linestyle="--")
kde = KernelDensity(kernel='gaussian', bandwidth=h).fit(x_train)
log_dens = kde.score_samples(x_test)
plt.plot(x_test, np.exp(log_dens), linewidth=2);

```



3.3 Elección del parámetro ancho de banda

Ancho de banda "óptimo" para el kernel Gaussiano 1d:

$$h = \sigma \left(\frac{4}{3N} \right)^{1/5}$$

Aproximación robusta (frente a la presencia de outliers) para estimar σ :

$$\hat{\sigma} = 1.4826 \text{ MAD}$$

donde MAD es la **median absolute deviation**:

$$\text{MAD} = \text{median}(|\mathbf{x} - \text{median}(\mathbf{x})|)$$

Caso multidimensional: si $D > 1$, estimamos h_d separadamente y tomamos

$$h = \left(\prod_{d=1}^D h_d \right)^{1/D}$$

3.4 De clasificación KDE a KNN

Problema de clasificación: sea \mathbf{x} una muestra a clasificar con base en conjunto de datos \mathcal{D} y sea $K \geq 1$

Densidad de una clase c en \mathbf{x} estimada con KDE balloon: hace crecer un volumen alrededor de \mathbf{x} , $V(\mathbf{x})$, hasta encontrar K datos

$$p(\mathbf{x} \mid y = c, \mathcal{D}) = \frac{\frac{N_c(\mathbf{x})}{N_c}}{V(\mathbf{x})}$$

donde $N_c(\mathbf{x})$ es el número de datos de la clase c en $V(\mathbf{x})$ y N_c es el total de datos de la clase c (en \mathcal{D})

Posterior de una clase c en \mathbf{x} : con priors de las clases estimadas como frecuencias relativas, coincide con la estimación KNN

$$p(y = c \mid \mathbf{x}, \mathcal{D}) = \frac{\frac{N_c(\mathbf{x})}{N_c V(\mathbf{x})} \frac{N_c}{N}}{\sum_{c'} \frac{N_{c'}(\mathbf{x})}{N_{c'} V(\mathbf{x})} \frac{N_{c'}}{N}} = \frac{N_c(\mathbf{x})}{\sum_{c'} N_{c'}(\mathbf{x})} = \frac{N_c(\mathbf{x})}{K} = \frac{1}{K} \sum_{n \in N_K(\mathbf{x}, \mathcal{D})} \mathbb{I}(y_n = c)$$

Clasificador KNN: puede verse como un clasificador generativo que estima las densidades condicionales KDE balloon

3.5 Regresión kernel

Regresión: como en clasificación, KDE se usa en regresión para construir modelos generativos que calculan la esperanza condicional

$$\mathbb{E}[y \mid \mathbf{x}, \mathcal{D}] = \int y p(y \mid \mathbf{x}, \mathcal{D}) dy = \frac{\int y p(\mathbf{x}, y \mid \mathcal{D}) dy}{\int p(\mathbf{x}, y \mid \mathcal{D}) dy}$$

Gaussiana multivariada para $p(\mathbf{x}, y \mid \mathcal{D})$?: modelo equivalente a regresión lineal que resulta bastante limitado

3.5.1 Nadaraya-Watson

Modelo Nadaraya-Watson: emplea KDE para aproximar la densidad conjunta

$$p(\mathbf{x}, y \mid \mathcal{D}) \approx \frac{1}{N} \sum_{n=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_n) \mathcal{K}_h(y - y_n)$$

Así, la esperanza condicional de la salida puede estimarse como una media ponderada de las salidas de los datos:

$$\begin{aligned} \mathbb{E}[y \mid \mathbf{x}, \mathcal{D}] &= \frac{\frac{1}{N} \sum_{n=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_n) \overbrace{\int y \mathcal{K}_h(y - y_n) dy}^{y_n}}{\frac{1}{N} \sum_{n=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_n) \underbrace{\int \mathcal{K}_h(y - y_n) dy}_1} \\ &= \frac{\sum_{n=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_n) y_n}{\sum_{n=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_n)} \\ &= \sum_{n=1}^N y_n w_n(\mathbf{x}) \quad \text{con} \quad w_n(\mathbf{x}) = \frac{\mathcal{K}_h(\mathbf{x} - \mathbf{x}_n)}{\sum_{n'=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_{n'})} \end{aligned}$$

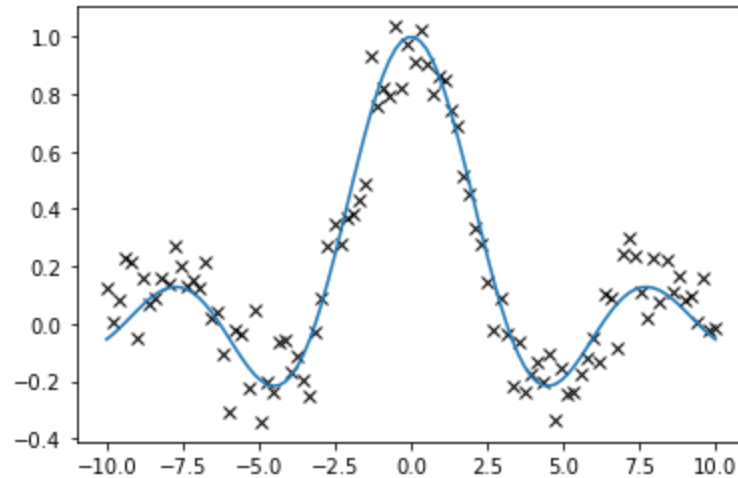
Ejemplo: regresión kernel en 1d con un kernel Gaussiano

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
N_train = 100; noise = 0.1; N_test = 100; np.random.seed(0)
x_train = 10 * (np.linspace(-1, 1, N_train).reshape(-1, 1))
```

```

y_train = np.divide(np.sin(np.abs(x_train)), np.abs(x_train))
y_train += noise * np.random.randn(N_train, 1)
x_test = 10 * (np.linspace(-1, 1, N_test).reshape(-1, 1))
y_test = np.divide(np.sin(np.abs(x_test)), np.abs(x_test))
plt.plot(x_train, y_train, 'kx'); plt.plot(x_test, y_test);

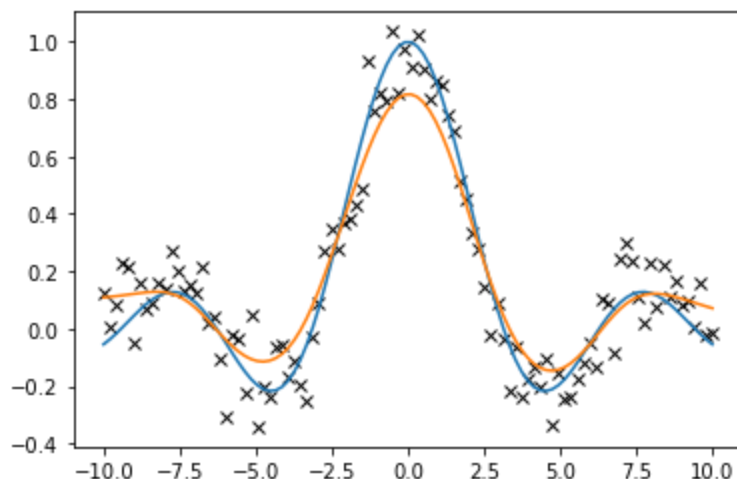
```



```

In [2]: from sklearn.neighbors import KernelDensity
y_pred = np.zeros_like(y_test); w = np.zeros_like(x_train);
kde = KernelDensity(kernel='gaussian', bandwidth=1).fit([[.0]])
for i in np.arange(N_test):
    x = x_test[i].reshape(1, 1)
    w = np.exp(kde.score_samples(x - x_train))
    w = w / w.sum()
    y_pred[i] = np.dot(y_train.T, w)
plt.plot(x_train, y_train, 'kx'); plt.plot(x_test, y_test);
plt.plot(x_test, y_pred);

```



3.5.2 Nadaraya-Watson Gaussiano con softmax

Dada una entrada \mathbf{x} , Nadaraya-Watson regresa una media ponderada de las salidas de los datos. Si se emplea un kernel Gaussiano, es fácil comprobar que los pesos son probabilidades obtenidas mediante normalización softmax de logits que miden la similitud (neg-distancia) entre \mathbf{x} y cada dato:

$$\begin{aligned}
 w_n(\mathbf{x}) &= \frac{\mathcal{K}_h(\mathbf{x} - \mathbf{x}_n)}{\sum_{n'=1}^N \mathcal{K}_h(\mathbf{x} - \mathbf{x}_{n'})} \\
 &= \frac{\frac{1}{h^D (2\pi)^{D/2}} \exp\left[-\frac{1}{2h^2} \|\mathbf{x} - \mathbf{x}_n\|^2\right]}{\sum_{n'=1}^N \frac{1}{h^D (2\pi)^{D/2}} \exp\left[-\frac{1}{2h^2} \|\mathbf{x} - \mathbf{x}_{n'}\|^2\right]} \\
 &= S\left(\left\{-\frac{1}{2h^2} \|\mathbf{x} - \mathbf{x}_{n'}\|^2\right\}_{n'=1}^N\right)_n
 \end{aligned}$$

Esta interpretación de Nadaraya-Watson con kernel Gaussiano se suele emplear en aprendizaje profundo como ejemplo ilustrativo de **mecanismo de atención (no paramétrico)**. Las entradas de los datos constituyen un **diccionario** de **claves**, sus salidas los **valores** asociados a las claves, y la entrada \mathbf{x} una **consulta (query)**. Los pesos que Nadaraya-Watson otorga a los datos tras la consulta se denominan **pesos de atención**, mientras que

sus logits correspondientes reciben el nombre de **scores de atención**. En estos términos, puede decirse que el mecanismo de atención es un modelo de regresión condicional que presta mayor atención a las claves que se parecen a la consulta. Los populares **transformers** se basan en el mecanismo de atención principalmente.

```
In [3]: h = 1.0; negsqL2 = lambda x: -0.5 * np.inner(x, x) / h
y_pred = np.zeros_like(y_test); w = np.zeros_like(x_train);
for i in np.arange(N_test):
    z = x_test[i] - x_train
    logits = np.apply_along_axis(negsqL2, 1, z)
    logits -= np.max(logits); w = np.exp(logits); w /= w.sum()
    y_pred[i] = np.dot(y_train.T, w)
plt.plot(x_train, y_train, 'kx'); plt.plot(x_test, y_test);
plt.plot(x_test, y_pred);
```

