# lecture9-density-estimation

October 28, 2021

## 1 Lecture 9: Density Estimation

#### 1.0.1 Applied Machine Learning

**Volodymyr Kuleshov**Cornell Tech

## 2 Part 1: Density Estimation

Density estimation is the problem of estimating a probability distribution from data.

As a first step, we will introduce probabilistic models for unsupervised learning.

## 3 Review: Unsupervised Learning

We have a dataset *without* labels. Our goal is to learn something interesting about the structure of the data: * Clusters hidden in the dataset. * Outliers: particularly unusual and/or interesting datapoints. * Useful signal hidden in noise, e.g. human speech over a noisy phone.

## 4 Components of an Unsupervised Learning Problem

At a high level, an unsupervised machine learning problem has the following structure:

$$\underbrace{\text{Dataset}}_{\text{Attributes}} + \underbrace{\text{Learning Algorithm}}_{\text{Model Class + Objective + Optimizer }} \to \text{Unsupervised Model}$$

The unsupervised model describes interesting structure in the data. For instance, it can identify interesting hidden clusters.

## 5 Review: Data Distribution

We will assume that the dataset is sampled from a probability distribution $P_{\text{data}}$, which we will call the *data distribution*. We will denote this as

$$x \sim P_{\text{data}}.$$

The dataset $\mathcal{D} = \{x^{(i)} \mid i = 1, 2, ..., n\}$ consists of *independent and identicaly distributed* (IID) samples from $P_{\text{data}}$.

## 6 Unsupervised Probabilistic Models

An unsupervised probabilistic model is a probability distribution

$$P_\theta(x) : \mathcal{X} \to [0, 1].$$

This model can approximate the data distribution $P_{\text{data}}$.

Probabilistic models often have *parameters* $\theta \in \Theta$.

## 7 Why Use Probabilistic Models?

There are many tasks that we can solve with a good model $P_\theta$. 1. Generation: sample new objects from $P_\theta$, such as images. 2. Structure learning: find interesting structure in $P_{\text{data}}$ 3. Density estimation: approximate $P_\theta \approx P_{\text{data}}$ and use it to solve any downstream task (generation, clustering, outlier detection, etc.).

We are going to be interested in the latter.

## 8 Maximum Likelihood Estimation

We can fit any unsupervised model $P_\theta(x)$ by optimizing the *maximum log-likelihood* objective

$$\max_\theta \ell(\theta) = \max_\theta \frac{1}{n} \sum_{i=1}^{n} \log P_\theta(x^{(i)}).$$

This asks that $P_\theta$ assign a high probability to the training instances in the dataset $\mathcal{D}$.

Maximizing likelihood is closely related to minimizing the Kullback-Leibler (KL) divergence $D(\cdot\|\cdot)$ between the model distribution and the data distribution.

$$D(P_{\text{data}} \| P_\theta) = \sum_{\mathbf{x}} P_{\text{data}}(\mathbf{x}) \log \frac{P_{\text{data}}(\mathbf{x})}{P_\theta(\mathbf{x})}.$$

The KL divergence is always non-negative, and equals zero when $P_{\text{data}}$ and $P_\theta$ are identical. This makes it a natural measure of similarity that's useful for comparing distributions.

## 9 Example: Flipping a Random Coin

How should we choose $P_\theta(x)$ if 3 out of 5 coin tosses are heads? Let's apply maximum likelihood learning.

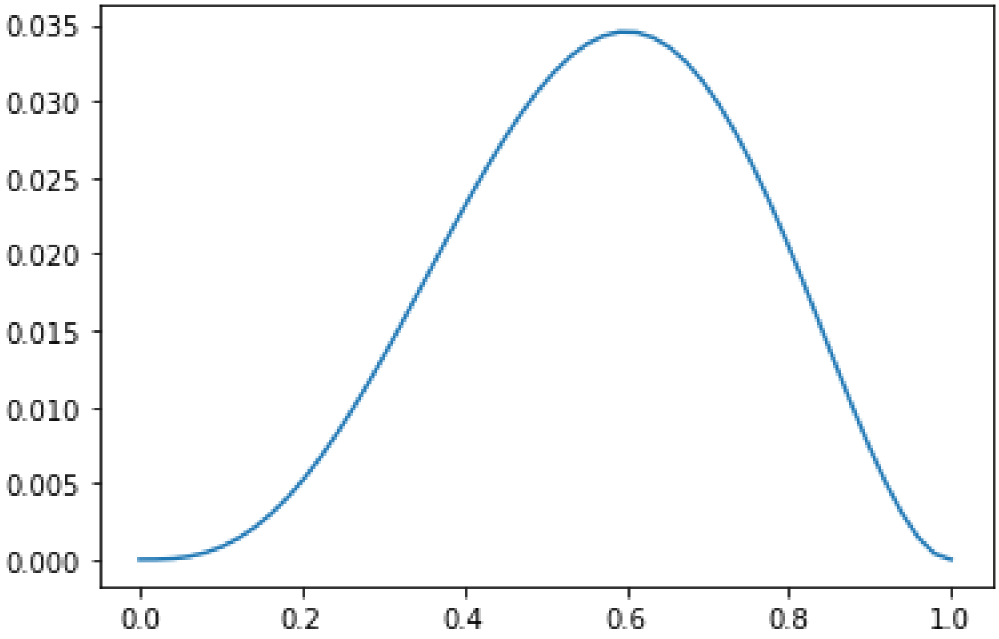
- Our model is $P_\theta(x = H) = \theta$ and $P_\theta(x = T) = 1 - \theta$
- Our data is: $\{H, H, T, H, T\}$
- The likelihood of the data is $\prod_i P_\theta(x_i) = \theta \cdot \theta \cdot (1 - \theta) \cdot \theta \cdot (1 - \theta)$.

We optimize for $\theta$ which makes the data most likely. What is the solution in this case?

[1]:
```
%matplotlib inline
import numpy as np
```

```python
from matplotlib import pyplot as plt

# our dataset is {H, H, T, H, T}; if theta = P(x=H), we get:
coin_likelihood = lambda theta: theta*theta*(1-theta)*theta*(1-theta)

theta_vals = np.linspace(0,1)
plt.plot(theta_vals, coin_likelihood(theta_vals))
```

```
[1]: [<matplotlib.lines.Line2D at 0x1168f7390>]
```



# Part 2: Kernel Density Estimation

Next, let's look at a first example of probabilistic models and how they are used to perform density estimation.

## 10 Density Estimation

The problem of density estimation is to approximate the data distribution $P_{\text{data}}$ with the model $P$.

$$P \approx P_{\text{data}}.$$

It's also a general learning task. We can solve many downstream tasks using a good model $P$: * Outlier and novelty detection * Generating new samples $x$ * Visualizing and understanding the structure of $P_{\text{data}}$

## 11 Histogram Density Estimation

Perhaps the simplest approach to density estimation is by forming a histogram.

A histogram partitions the input space $x$ into a $d$-dimensional grid and counts the number of points in each cell.

This is best illustrated by an example.

Let's start by creating a simple 1D dataset coming from a mixture of two Gaussians:

$$P_{\text{data}}(x) = 0.3 \cdot \mathcal{N}(x; \mu = 0, \sigma = 1) + 0.7 \cdot \mathcal{N}(x; \mu = 5, \sigma = 1)$$

[23]:

```
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_kde_1d.html
import numpy as np
np.random.seed(1)

N = 20 # number of points
# concat samples from two Gaussians:
X = np.concatenate((
    np.random.normal(0, 1, int(0.3 * N)),
    np.random.normal(5, 1, int(0.7 * N))
))[:, np.newaxis]
bins = np.linspace(-5, 10, 10) # locations of the bins

# print out X
print(X.flatten())
```

```
[ 1.62434536 -0.61175641 -0.52817175 -1.07296862  0.86540763 -2.3015387
  6.74481176  4.2387931   5.3190391   4.75062962  6.46210794  2.93985929
  4.6775828   4.61594565  6.13376944  3.90010873  4.82757179  4.12214158
  5.04221375  5.58281521]
```

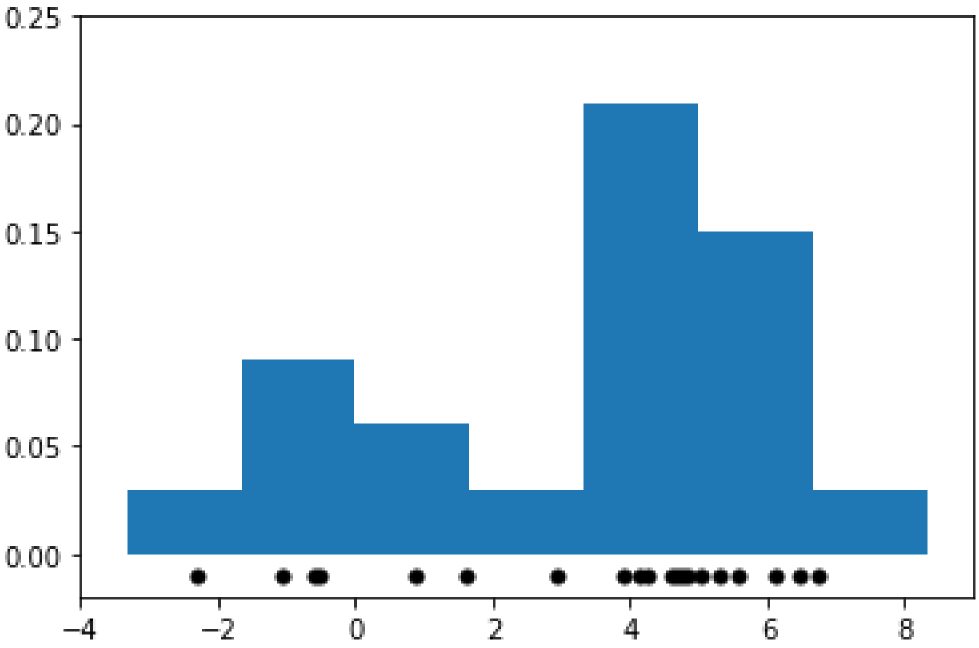
We can now estimate the density using a histogram.

[41]:

```
import matplotlib.pyplot as plt

plt.hist(X[:, 0], bins=bins, density=True) # plot the histogram
plt.plot(X[:, 0], np.full(X.shape[0], -0.01), '.k', markersize=10) # plot the␣
 ↪points in X
plt.xlim(-4, 9)
plt.ylim(-0.02, 0.25)
```

[41]:

```
(-0.02, 0.25)
```



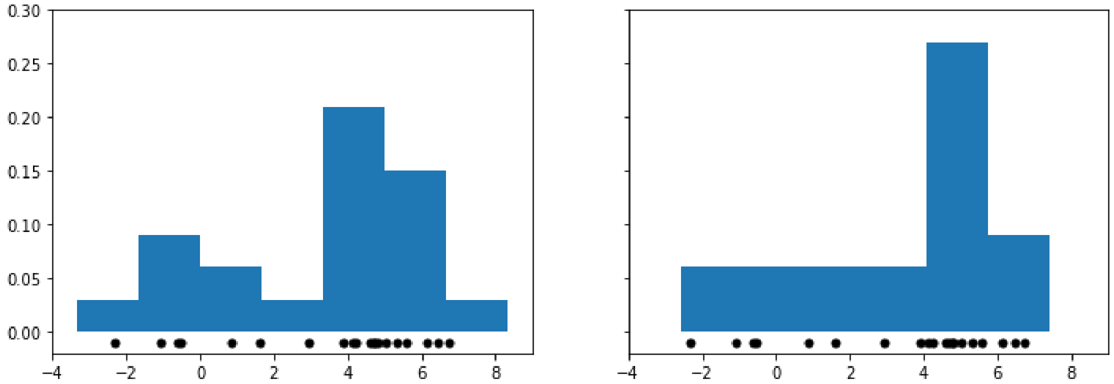
## 12 Limitations of Histograms

Histogram-based methods have a number of shortcomings. * The number of grid cells increases exponentially with dimension $d$. * The histogram is not "smooth". * The shape of the histogram depends on the bin positions.

We will now try to address the last two limitations.

Let's also visualize what we mean when we say that shape of the histogram depends on the histogram bins.

[38]:
```
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(12,4))
ax[0].hist(X[:, 0], bins=bins, density=True) # plot the histogram
ax[1].hist(X[:, 0], bins=bins+0.75, density=True) # plot the histogram with bin␣
 ↪centers shifted by 0.75
for axi in ax.ravel():
    axi.plot(X[:, 0], np.full(X.shape[0], -0.01), '.k', markersize=10) # plot␣
 ↪the points in X
    axi.set_xlim(-4, 9)
    axi.set_ylim(-0.02, 0.3)
```



## 13 Kernel Density Estimation: Idea

Kernel density estimation (KDE) is a different approach to histogram estimation. * A histogram has $b$ bins of width $\delta$ at fixed positions. * KDE effectively places a bin of with $\delta$ at each $x \in \mathcal{X}$. * To obtain $P(x)$, we count the % of points that fall in the bin centered at $x$.

## 14 Tophat Kernel Density Estimation

The simplest form of this strategy (Tophat KDE) assumes a model of the form

$$P_\delta(x) = \frac{N(x;\delta)}{n},$$

where

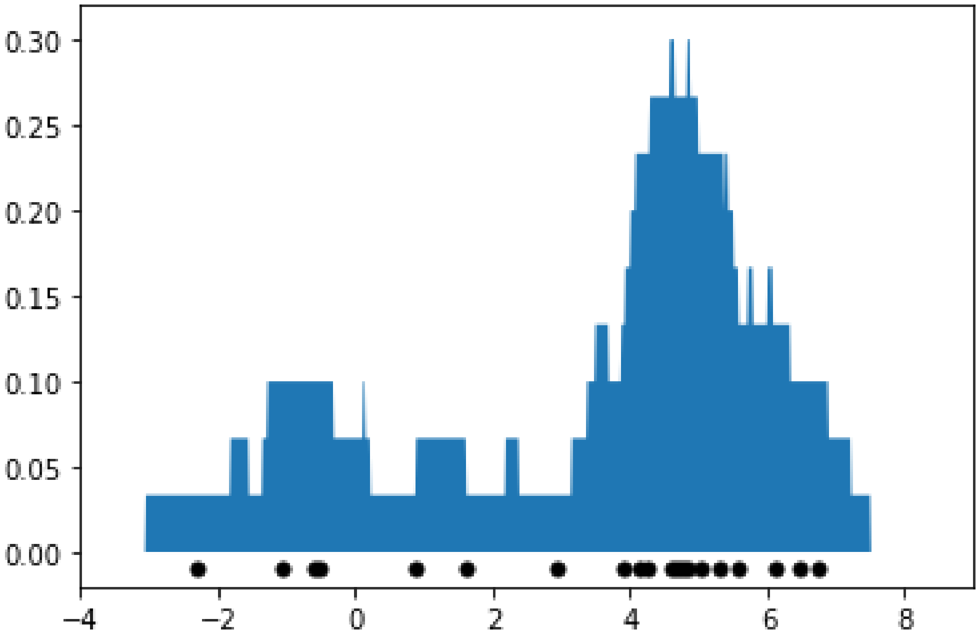
$$N(x;\delta) = |\{x^{(i)} : ||x^{(i)} - x|| \leq \delta/2\}|,$$

is the number of points that are within a bin of with $\delta$ centered at $x$.

This is best understood via a picture.

[47]:
```
from sklearn.neighbors import KernelDensity

kde = KernelDensity(kernel='tophat', bandwidth=0.75).fit(X) # fit a KDE model
x_ticks = np.linspace(-5, 10, 1000)[:, np.newaxis] # choose 1000 points on␣
 ↪x-axis
log_density = kde.score_samples(x_ticks) # compute density at 1000 points

plt.fill(x_ticks[:, 0], np.exp(log_density)) # plot the density estimate
plt.plot(X[:, 0], np.full(X.shape[0], -0.01), '.k', markersize=10) # plot the␣
 ↪points in X
plt.xlim(-4, 9)
plt.ylim(-0.02, 0.32)
```

[47]:
```
(-0.02, 0.32)
```



The above algorithm still has the problem of producing a density estimate that is not smooth.

We are going to resolve this by replacing histogram counts with weighted averages.

## 15 Kernels

A *kernel function* $K : \mathcal{X} \times \mathcal{X} \to [0, \infty]$ maps pairs of vectors $x, z \in \mathcal{X}$ to a real-valued score $K(x, z)$.

- A kernel represents the similarity between $x$ and $z$.
- We will see many ways of defining "similarity"; they will all fit the framework that follows.

## 16 Kernel Density Estimation

A kernelized density model $P$ takes the form:

$$P(x) \propto \sum_{i=1}^{n} K(x, x^{(i)}).$$

This can be interpreted in several ways: * We count the number of points "near" $x$, but each $x^{(i)}$ has a weight $K(x, x^{(i)})$ that depends on similarity between $x, x^{(i)}$. * We place a "micro-density" $K(x, x^{(i)})$ at each $x^{(i)}$; the final density $P(x)$ is their sum.

## 17 Types of Kernels

We have seen several types of kernels in the context of support vector machines.

There are additional kernels that are popular for density estimation.

The following kernels are available in `scikit-learn`. * Gaussian kernel $K(x, z; \delta) \propto \exp(-||x - z||^2/2\delta^2)$ * Tophat kernel $K(x, z; \delta) = 1$ if $||x - z|| \leq \delta/2$ else 0. * Epanechnikov kernel $K(x, z; \delta) \propto 1 - ||x - z||^2/\delta^2$ * Exponential kernel $K(x, z; \delta) \propto \exp(-||x - z||/\delta)$ * Linear kernel $K(x, z; \delta) \propto (1 - ||x - z||/\delta)^+$

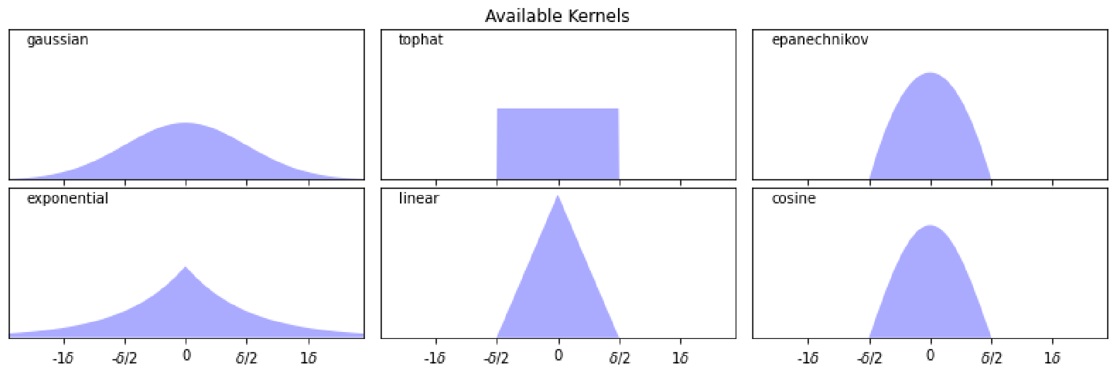
It's easier to understand these kernels by looking at a figure.

[81]:

```python
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_kde_1d.html
X_plot = np.linspace(-6, 6, 1000)[:, None]
X_src = np.zeros((1, 1))

fig, ax = plt.subplots(2, 3, sharex=True, sharey=True, figsize=(12,4))
fig.subplots_adjust(left=0.05, right=0.95, hspace=0.05, wspace=0.05)

def format_func(x, loc):
    if x == 0:
        return '0'
    elif x == 1:
        return '$\delta/2$'
    elif x == -1:
        return '-$\delta/2$'
    else:
        return '%i$\delta$' % (int(x/2))

for i, kernel in enumerate(['gaussian', 'tophat', 'epanechnikov',
                            'exponential', 'linear', 'cosine']):
    axi = ax.ravel()[i]
    log_dens = KernelDensity(kernel=kernel).fit(X_src).score_samples(X_plot)
    axi.fill(X_plot[:, 0], np.exp(log_dens), '-k', fc='#AAAAFF')
    axi.text(-2.6, 0.95, kernel)

    axi.xaxis.set_major_formatter(plt.FuncFormatter(format_func))
    axi.xaxis.set_major_locator(plt.MultipleLocator(1))
    axi.yaxis.set_major_locator(plt.NullLocator())

    axi.set_ylim(0, 1.05)
    axi.set_xlim(-2.9, 2.9)

ax[0, 1].set_title('Available Kernels')
```

[81]:

```
Text(0.5, 1.0, 'Available Kernels')
```



## 18 Kernel Density Estimation: Example

Let's look at an example in the context of the 1D points we have seen earlier.

We will fit a model of the form

$$P(x) = \sum_{i=1}^{n} K(x, x^{(i)})$$

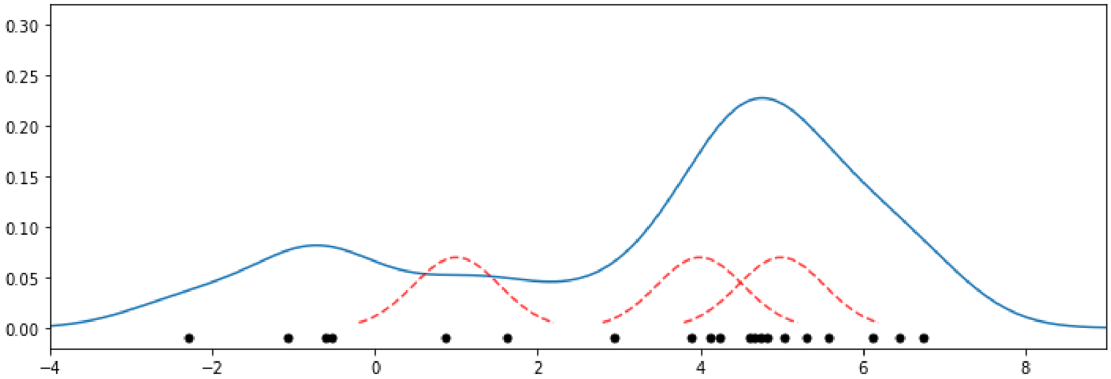
with a Gaussian kernel $K(x, z; \delta) \propto \exp(-||x - z||^2/2\delta^2)$.

[77]:

```
from sklearn.neighbors import KernelDensity

kde = KernelDensity(kernel='gaussian', bandwidth=0.75).fit(X) # fit a KDE model
x_ticks = np.linspace(-5, 10, 1000)[:, np.newaxis] # choose 1000 points on␣
 ↪x-axis
log_density = kde.score_samples(x_ticks) # compute density at 1000 points
gaussian_kernel = lambda z : lambda x: np.exp(-np.abs(x-z)**2/(0.75**2)) #␣
 ↪gaussian kernel
kernel_linspace = lambda x : np.linspace(x-1.2,x+1.2,30)

plt.figure(figsize=(12,4))
plt.plot(x_ticks[:, 0], np.exp(log_density)) # plot the density estimate
plt.plot(X[:, 0], np.full(X.shape[0], -0.01), '.k', markersize=10) # plot the␣
 ↪points in X
plt.plot(kernel_linspace(4), 0.07*gaussian_kernel(4)(kernel_linspace(4)), '--',␣
 ↪color='r', alpha=0.75)
plt.plot(kernel_linspace(5), 0.07*gaussian_kernel(5)(kernel_linspace(5)), '--',␣
 ↪color='r', alpha=0.75)
plt.plot(kernel_linspace(1), 0.07*gaussian_kernel(1)(kernel_linspace(1)), '--',␣
 ↪color='r', alpha=0.75)
plt.xlim(-4, 9)
plt.ylim(-0.02, 0.32)
```

[77]:

```
(-0.02, 0.32)
```



## 19 KDE in Higher Dimensions

In priciple, kernel density estimation also works in higher dimensions.

However, the number of datapoints needed for a good fit incrases expoentially with the dimension, which limits the applications of this model in high dimensions.

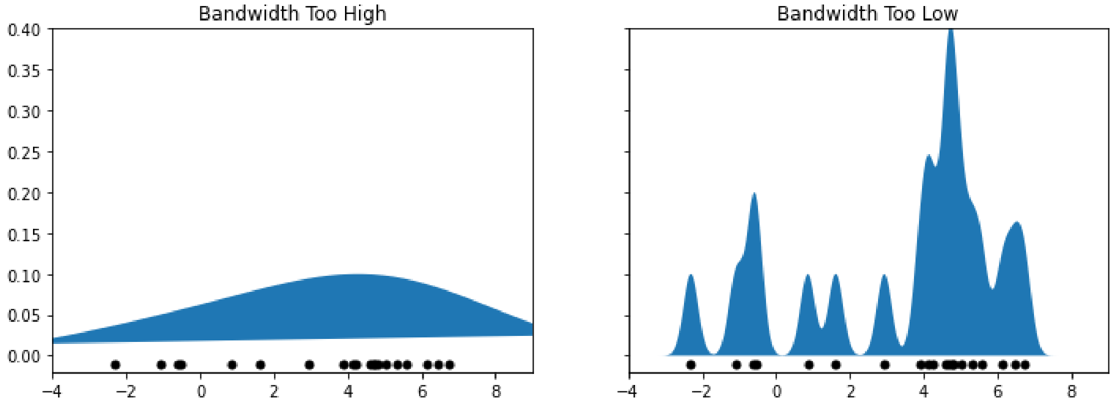
## 20 Choosing Hyperparameters

Each kernel has a notion of "bandwidth" $\delta$. This is a hyperparameter that controls the "smoothness" of the fit. * We can choose it using inspection or heuristics like we did for $K$ in $K$-Means. * Because we have a probabilistic model, we can also estimate likelihood on a holdout dataset (more on this later!)

Let's illustrate how the bandwidth affects smoothness via an example.

[91]:

```
from sklearn.neighbors import KernelDensity

kde1 = KernelDensity(kernel='gaussian', bandwidth=3).fit(X) # fit a KDE model
kde2 = KernelDensity(kernel='gaussian', bandwidth=0.2).fit(X) # fit a KDE model

fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(12,4))
ax[0].fill(x_ticks[:, 0], np.exp(kde1.score_samples(x_ticks))) # plot the␣
 ↪density estimate
ax[1].fill(x_ticks[:, 0], np.exp(kde2.score_samples(x_ticks))) # plot the␣
 ↪density estimate
ax[0].set_title('Bandwidth Too High')
ax[1].set_title('Bandwidth Too Low')

for axi in ax.ravel():
    axi.plot(X[:, 0], np.full(X.shape[0], -0.01), '.k', markersize=10) # plot␣
 ↪the points in X
    axi.set_xlim(-4, 9)
    axi.set_ylim(-0.02, 0.4)
```



## 21 Algorithm: Kernel Density Estimation

- **Type**: Unsupervised learning (density estimation).
- **Model family**: Non-parametric. Sum of $n$ kernels.
- **Objective function**: Log-likelihood to choose optimal bandwidth.
- **Optimizer**: Grid search.

## 22 Pros and Cons of KDE

Pros: * Can approximate any data distribution arbtrarily well.

Cons: * Need to store entire dataset to make queries, which is computationally prohibitive. * Number of data needed scale exponentially with dimension (“curse of dimensionality”).

# Part 3: Nearest Neighbors

The ideas from density estimation also extend to supervised learning via an algorithm called Nearest Neighbors.

## 23 Review: Classification

Consider a training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$.

We distinguish between two types of supervised learning problems depnding on the targets $y^{(i)}$.

1. **Regression**: The target variable $y \in \mathcal{Y}$ is continuous: $\mathcal{Y} \subseteq \mathbb{R}$.
2. **Classification**: The target variable $y$ is discrete and takes on one of $K$ possible values: $\mathcal{Y} = \{y_1, y_2, \ldots y_K\}$. Each discrete value corresponds to a *class* that we want to predict.

## 24 A Simple Classification Algorithm: Nearest Neighbors

Suppose we are given a training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(n)}, y^{(n)})\}$. At inference time, we receive a query point $x'$ and we want to predict its label $y'$.

A really simple but suprisingly effective way of returning $y'$ is the *nearest neighbors* approach. * Given a query datapoint $x'$, find the training example $(x, y)$ in $\mathcal{D}$ that's closest to $x'$, in the sense that $x$ is "nearest" to $x'$ * Return $y$, the label of the "nearest neighbor" $x$.

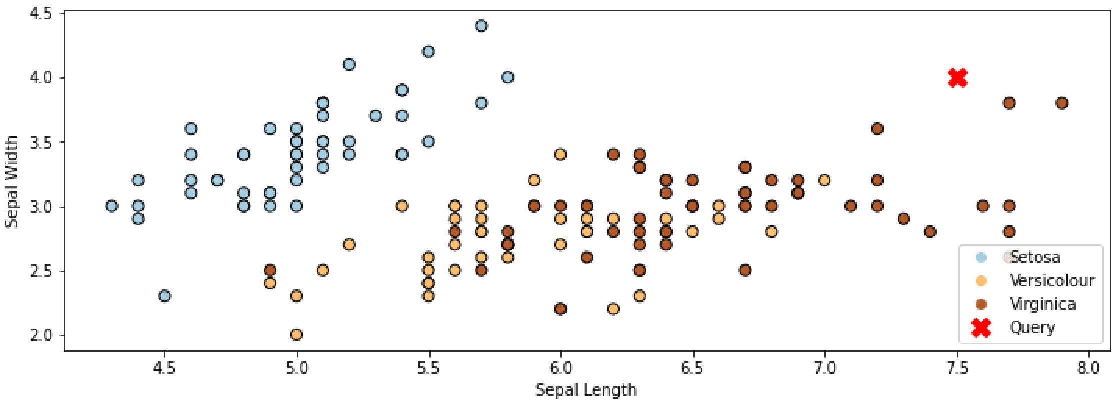
In the example below on the Iris dataset, the red cross denotes the query $x'$. The closest class to it is "Virginica". (We're only using the first two features in the dataset for simplicity.)

[6]:

```python
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [12, 4]

# Plot also the training points
p1 = plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y,
            edgecolor='k', s=50, cmap=plt.cm.Paired)
p2 = plt.plot([7.5], [4], 'rx', ms=10, mew=5)
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.legend(['Query Point', 'Training Data'], loc='lower right')
plt.legend(handles=p1.legend_elements()[0]+p2, labels=['Setosa', 'Versicolour',␣
 ↪'Virginica', 'Query'], loc='lower right')
```

[6]:

```
<matplotlib.legend.Legend at 0x12982b4e0>
```



## 25 Choosing a Distance Function

How do we select the point $x$ that is the closest to the query point $x'$? There are many options:

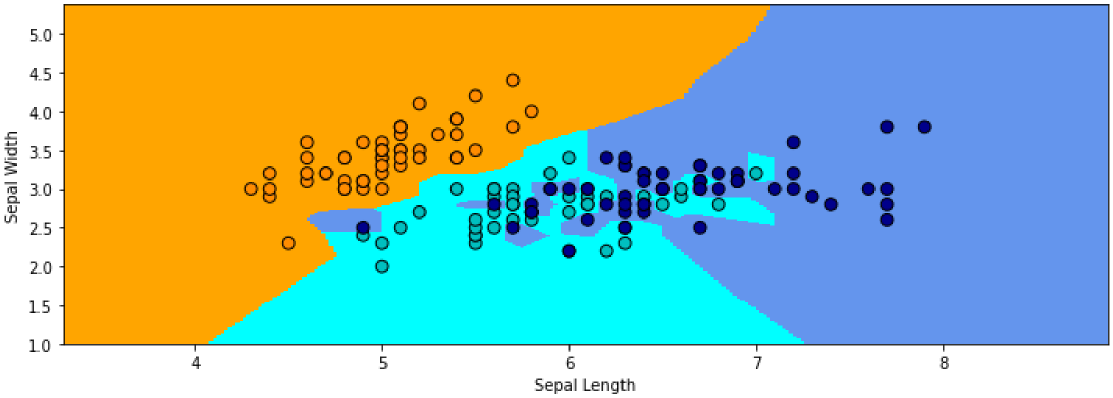
- The Euclidean distance $||x - x'||_2 = \sqrt{\sum_{j=1}^d |x_j - x_j'|^2)}$ is a popular choice.

- The Minkowski distance $||x - x'||_p = (\sum_{j=1}^{d} |x_j - x'_j|^p)^{1/p}$ generalizes the Euclidean, L1 and other distances.
- The Mahalanobis distance $\sqrt{x^\top V x}$ for a positive semidefinite matrix $V \in \mathbb{R}^{d \times d}$ also generalizes the Euclidean distnace.
- Discrete-valued inputs can be examined via the Hamming distance $|\{j : x_j \neq x'_j\}|$ and other distances.

Let's apply Nearest Neighbors to the above dataset using the Euclidean distance (or equiavalently, Minkowski with $p = 2$)

[23]:
```
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.
 ↪html
from sklearn import neighbors
from matplotlib.colors import ListedColormap

# Train a Nearest Neighbors Model
clf = neighbors.KNeighborsClassifier(n_neighbors=1, metric='minkowski', p=2)
clf.fit(iris_X.iloc[:,:2], iris_y)

# Create color maps
cmap_light = ListedColormap(['orange', 'cyan', 'cornflowerblue'])
cmap_bold = ListedColormap(['darkorange', 'c', 'darkblue'])

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = iris_X.iloc[:, 0].min() - 1, iris_X.iloc[:, 0].max() + 1
y_min, y_max = iris_X.iloc[:, 1].min() - 1, iris_X.iloc[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                     np.arange(y_min, y_max, 0.02))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

# Plot also the training points
plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y, cmap=cmap_bold,
            edgecolor='k', s=60)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
```

[23]:
```
Text(0, 0.5, 'Sepal Width')
```



In the above example, the regions of the 2D space that are assigned to each class are highly irregular. In areas where the two classes overlap, the decision of the boundary flips between the classes, depending on which point is closest to it.

## 26 K-Nearest Neighbors

Intuitively, we expect the true decision boundary to be smooth. Therefore, we average $K$ nearest neighbors at a query point.

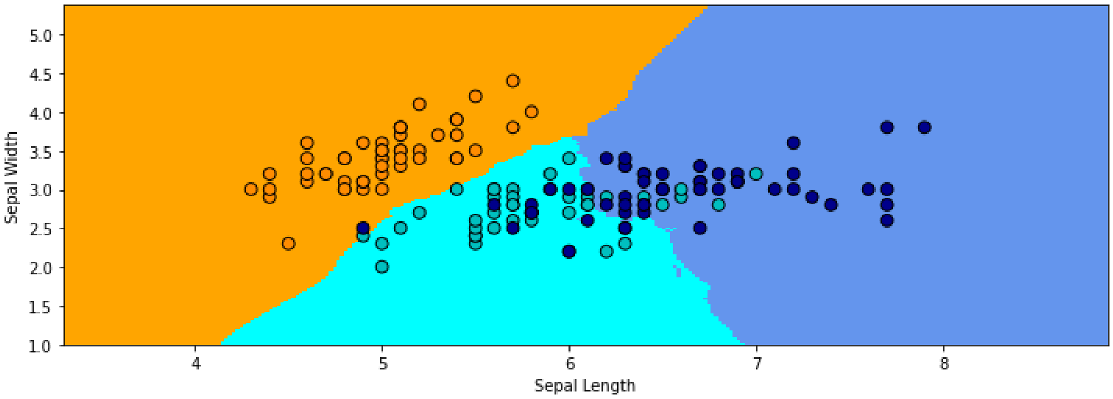
- Given a query datapoint $x'$, find the $K$ training examples $\mathcal{N} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(K)}, y^{(K)})\} \subseteq D$ that are closest to $x'$.
- Return $y_{\mathcal{N}}$, the consensus label of the neighborhood $\mathcal{N}$.

The consesus $y_{\mathcal{N}}$ can be determined by voting, weighted average, etc.

Let's look at Nearest Neighbors with a neighborhood of 30. The decision boundary is much smoother than before.

[8]:
```
# https://scikit-learn.org/stable/auto_examples/neighbors/plot_classification.
 ↪html
# Train a Nearest Neighbors Model
clf = neighbors.KNeighborsClassifier(n_neighbors=30, metric='minkowski', p=2)
clf.fit(iris_X.iloc[:,:2], iris_y)
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure()
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

# Plot also the training points
plt.scatter(iris_X.iloc[:, 0], iris_X.iloc[:, 1], c=iris_y, cmap=cmap_bold,
            edgecolor='k', s=60)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
```

```
[8]: Text(0, 0.5, 'Sepal Width')
```



## 27 KNN Estimates Data Distribution

Suppose that the output $y'$ of KNN is the average target in the neighborhood $\mathcal{N}(x')$ around the query $x'$. Observe that we can write:

$$y' = \frac{1}{K} \sum_{(x,y)\in\mathcal{N}(x')} y \approx \mathbb{E}[y \mid x'].$$

- When $x \approx x'$ and when $\mathbb{P}$ is reasonably smooth, each $y$ for $(x, y) \in \mathcal{N}(x')$ is approximately a sample from $\mathbb{P}(y \mid x')$ (since $\mathbb{P}$ doesn't change much around $x'$, $\mathbb{P}(y \mid x') \approx \mathbb{P}(y \mid x)$).
- Thus $y'$ is essentially a Monte Carlo estimate of $\mathbb{E}[y \mid x']$ (the average of $K$ samples from $\mathbb{P}(y \mid x')$).

## 28 Algorithm: K-Nearest Neighbors

- **Type**: Supervised learning (regression and classification)
- **Model family**: Consensus over $K$ training instances.
- **Objective function**: Euclidean, Minkowski, Hamming, etc.
- **Optimizer**: Non at training. Nearest neighbor search at inference using specialized search algorithms (Hashing, KD-trees).
- **Probabilistic interpretation**: Directly approximating the density $P_{\text{data}}(y|x)$.

# 29 Non-Parametric Models

Nearest neighbors is an example of a *non-parametric* model. Parametric vs. non-parametric are is a key distinguishing characteristic for machine learning models.

A parametric model $f_\theta(x) : \mathcal{X} \times \Theta \to \mathcal{Y}$ is defined by a finite set of parameters $\theta \in \Theta$ whose dimensionality is constant with respect to the dataset. Linear models of the form

$$f_\theta(x) = \theta^\top x$$

are an example of a parametric model.

In a non-parametric model, the function $f$ uses the entire training dataset (or a post-proccessed version of it) to make predictions, as in $K$-Nearest Neighbors.

In other words, the complexity of the model increases with dataset size.

Non-parametric models have the advantage of not loosing any information at training time.

However, they are also computationally less tractable and may easily overfit the training set.

# 30 Pros and Cons of KNN

Pros: * Can approximate any data distribution arbtrarily well.

Cons: * Need to store entire dataset to make queries, which is computationally prohibitive. * Number of data needed scale exponentially with dimension ("curse of dimensionality").