parzen_window_from_scratch

September 12, 2025

1 Parzen Window (Kernel Density Estimation) — from scratch

Implementing Parzen window (a.k.a. kernel density estimation) step-by-step. This notebook focuses on intuition, a clean numpy implementation, visual checks, and small applications (anomaly detection, simple classification).

1.1 What this notebook contains

- A minimal, readable Parzen (KDE) implementation in numpy
- 1D and 2D demos with plots
- Common questions and troubleshooting notes
- Basic application: anomaly detection using low-density threshold

Style: educational, with inline comments and Q&A blocks for typical implementation questions.

[ ]:
```
# Standard imports
import numpy as np
import matplotlib.pyplot as plt

# helper for 2D plotting
from matplotlib import cm

# set a default random seed for reproducibility
np.random.seed(0)
```

1.2 Parzen estimator (1D and 2D)

We'll implement a flexible Parzen estimator that supports Gaussian and uniform kernels. The Parzen estimate at a point x is the averaged kernel values centered at each sample.

[ ]:
```
def gaussian_kernel(dist_sq, h):
    # dist_sq: squared Euclidean distance
    # h: bandwidth (scalar)
    # returns kernel value (unnormalized for multidimensional case we divide␣
 ↪appropriately)
    d = 1  # we'll divide by (sqrt(2*pi)*h)**d later in 1D/2D cases explicitly
    return np.exp(-0.5 * dist_sq / (h**2))

def uniform_kernel(dist_sq, h):
```

```python
    # dist_sq: squared Euclidean distance
    # uniform kernel: 1 inside radius h, 0 outside
    return (dist_sq <= h**2).astype(float)

def parzen_pdf(X_train, X_eval, h=0.5, kernel='gaussian'):
    """Estimate density at points X_eval given training points X_train using Parzen windows.
    X_train: (n_samples, d)
    X_eval: (m_points, d)
    h: bandwidth (scalar)
    kernel: 'gaussian' or 'uniform'
    returns: (m_points,) density estimates
    """
    X_train = np.asarray(X_train)
    X_eval = np.asarray(X_eval)
    n, d = X_train.shape
    m = X_eval.shape[0]
    out = np.zeros(m, dtype=float)
    if kernel == 'gaussian':
        # Normalizing constant for d dimensions: (2*pi)^(d/2) * h^d
        norm = (2 * np.pi)**(d/2) * (h**d)
        for i in range(m):
            diff = X_train - X_eval[i]  # (n, d)
            dist_sq = np.sum(diff**2, axis=1)
            k = gaussian_kernel(dist_sq, h)
            out[i] = np.sum(k) / (n * norm)
    elif kernel == 'uniform':
        # Volume of d-dim ball of radius h: for 1D it's 2h, for 2D it's pi*h^2
        if d == 1:
            vol = 2*h
        elif d == 2:
            vol = np.pi * (h**2)
        else:
            # keep general but warn: uniform kernel normalization for high-d is rarely used here
            vol = (2*np.pi)**(d/2) * (h**d)  # fallback
        for i in range(m):
            diff = X_train - X_eval[i]
            dist_sq = np.sum(diff**2, axis=1)
            k = uniform_kernel(dist_sq, h)
            out[i] = np.sum(k) / (n * vol)
    else:
        raise ValueError('Unknown kernel: choose gaussian or uniform')
    return out
```

1.2.1 1D demo: visualize Parzen estimate vs true distribution

We'll sample from a mixture of Gaussians and compare the estimated density for different bandwidths. Questions to keep in mind: How does bandwidth h affect smoothness? What happens when h is too small or too large?

[ ]:
```
# generate 1D data: mixture of gaussians
np.random.seed(1)
n = 200
x1 = np.random.normal(-2, 0.5, size=n//2)
x2 = np.random.normal(2, 0.8, size=n//2)
X = np.concatenate([x1, x2])[:, None]  # shape (n,1)

# evaluation points
xs = np.linspace(-6, 6, 400)[:, None]

for h in [0.1, 0.4, 1.0]:
    p_hat = parzen_pdf(X, xs, h=h, kernel='gaussian')
    plt.plot(xs[:,0], p_hat, label=f'h={h}')
# plot train points (rug)
plt.scatter(X[:,0], np.zeros_like(X[:,0]) - 0.002, marker='|', color='k', s=40)
plt.legend()
plt.title('Parzen density estimate (1D) - effect of bandwidth')
plt.xlabel('x')
plt.ylabel('density')
plt.show()
```

1.2.2 2D demo: contour estimate

Use a mixture of 2D Gaussians and visualize contour of estimated density. This is a common check to see modes and smoothing.

[ ]:
```
# 2D demo
np.random.seed(2)
n = 300
# two 2D Gaussian clusters
c1 = np.random.multivariate_normal(mean=[-2, -1], cov=[[0.6, 0.2],[0.2, 0.6]],␣
 ↪size=n//2)
c2 = np.random.multivariate_normal(mean=[2, 2], cov=[[0.8, -0.3],[-0.3, 0.8]],␣
 ↪size=n//2)
X2 = np.vstack([c1, c2])

# grid
xx, yy = np.meshgrid(np.linspace(-6,6,120), np.linspace(-6,6,120))
grid = np.column_stack([xx.ravel(), yy.ravel()])

# estimate density on grid
p_grid = parzen_pdf(X2, grid, h=0.7, kernel='gaussian')
```

```python
p_grid = p_grid.reshape(xx.shape)

plt.figure(figsize=(7,5))
plt.contourf(xx, yy, p_grid, levels=30, cmap=cm.viridis)
plt.scatter(X2[:,0], X2[:,1], s=10, edgecolor='k', alpha=0.6)
plt.title('Parzen (KDE) contour (2D) - Gaussian kernel, h=0.7')
plt.xlabel('x1'); plt.ylabel('x2')
plt.show()
```

1.3 Basic application: anomaly detection

Flag points as anomalies if their estimated density is below a chosen threshold. This is simple but effective for low-dimensional data.

[ ]:
```python
# create normal points and some anomalies
np.random.seed(3)
clean = np.random.normal(0, 1, size=(300,1))
anoms = np.array([[5.0],[ -5.0], [3.5]])  # clear outliers
X_ad = np.vstack([clean, anoms])

# estimate densities
p_est = parzen_pdf(clean, X_ad, h=0.6, kernel='gaussian')

# choose threshold as low quantile of clean densities
thr = np.quantile(parzen_pdf(clean, clean, h=0.6), 0.02)
labels = p_est < thr

print('Threshold (2nd percentile):', thr)
for xi, pi, is_anom in zip(X_ad, p_est, labels):
    print(f'x={xi[0]:.2f}, p~={pi:.3e}, anomaly={bool(is_anom)}')
```

1.4 Common implementation questions (FAQ)

Q: How do I select bandwidth h? A: There's no free lunch. For Gaussian kernels, rules of thumb (Silverman's rule) exist for 1D; cross-validation is practical. Inspect plots for under/oversmoothing.

Q: Why does Parzen get worse in high dimensions? A: Curse of dimensionality: required samples grow exponentially. Bandwidth selection and kernel choice become fragile.

Q: Can I use this for classification? A: Yes — estimate class-conditional densities p(x|y) with Parzen for each class and apply Bayes rule to classify.

Q: Performance tips - Use vectorized linear algebra and broadcast where possible - For large data, use FFT-based KDE or libraries (scipy, sklearn) or approximate methods (KD-trees)

1.5 Next steps / exercises

- Implement Parzen with Gaussian kernel but using vectorized broadcasting for speed
- Add cross-validation to choose h
- Try class-conditional Parzen for simple classification on synthetic labeled data