Tyler Singleton
February 16, 2025

Problem 1:
With the Nadaraya–Watson estimator defined by:

$$\hat{f}(x_0) = \frac{\sum_{i=1}^{n} K_\lambda(x_0, x_i)\, y_i}{\sum_{i=1}^{n} K_\lambda(x_0, x_i)}$$

Which has the scaled kernel:

$$K_\lambda(x_0, x) = \frac{1}{\lambda} K\Big(\frac{x - x_0}{\lambda}\Big)$$

Where K is nonnegative, typically integrates to one, is centered at zero, and decays with increasing $|x - x_0|$. When the kernel is Gaussian, it takes the form:

$$K_\lambda(x_0, x) = \frac{1}{\lambda} \frac{1}{\sqrt{2\pi}} \exp\Big(-\frac{1}{2}\Big(\frac{x - x_0}{\lambda}\Big)^2\Big)$$

This is strictly positive for all x and infinitely differentiable in x_0. Since both the numerator $\sum_{i=1}^{n} K_\lambda(x_0, x_i) y_i$ and the denominator $\sum_{i=1}^{n} K_\lambda(x_0, x_i)$ are finite sums of smooth functions and the denominator never vanishes, the ratio $\hat{f}(x_0)$ inherits infinite differentiability with respect to x_0. In contrast, when the Epanechnikov kernel is used with a fixed bandwidth, it is given by:

$$K(t) = \frac{3}{4}(1 - t^2) \quad \text{for } |t| \le 1, \quad \text{and 0 otherwise}$$

So that:

$$K_\lambda(x_0, x) = \frac{1}{\lambda} \begin{cases} \frac{3}{4}\Big(1 - \Big(\frac{x - x_0}{\lambda}\Big)^2\Big) & |x - x_0| \le \lambda \\ 0 & |x - x_0| > \lambda \end{cases}$$

So, within the interval $|x - x_0| \le \lambda$ the kernel is a quadratic polynomial, which is infinitely differentiable; however, at the boundary $|x - x_0| = \lambda$ the kernel transitions abruptly to zero. Although the function remains continuous at the boundary, its first derivative experiences a discontinuous jump since the derivative inside the interval is nonzero while it is identically zero outside. Because the estimator is constructed as a ratio of finite sums of such piecewise functions, $\hat{f}(x_0)$ is continuous overall but exhibits non-differentiable behavior (corners) at those points where one or more kernel contributions switch on or off. When λ is allowed to depend on x_0, it becomes less smooth, and is defined as the distance from x_0 to its kth nearest neighbor. In this adaptive setting, $\lambda(x_0)$ may change discontinuously as the identity of the kth neighbor shifts with x_0. Such abrupt changes in the bandwidth imply that both the support of the kernel and the weights $K_{\lambda(x_0)}(x_0, x_i)$ may change suddenly, causing $\hat{f}(x_0)$ to exhibit even stronger non-smooth behavior; it may develop sharper corners and, in some cases, even discontinuities. So, under a fixed bandwidth with a Gaussian kernel the Nadaraya–Watson estimator is infinitely differentiable. A fixed Epanechnikov kernel remains continuous but loses differentiability at the boundaries of the kernel's support. With an adaptive Epanechnikov bandwidth the estimator can fail to be continuously smooth, with the potential for discontinuities.

Problem 2:

Local polynomial regression of degree k at a target point x_0, where the estimator is given by $\hat{f}(x_0) = \sum_{i=1}^{N} l_i(x_0)\, y_i$. The key observation is that the weights $l_i(x_0)$ are determined solely by the predictors, the target point, the kernel function, and the polynomial degree; consequently, they possess the property of reproducing any polynomial of degree at most k exactly. If the responses are generated by a polynomial $p(x)$ of degree no greater than k, so that $y_i = p(x_i)$, the estimator must satisfy $\hat{f}(x_0) = p(x_0)$. This can be written as:

$$p(x_0) = \sum_{i=1}^{N} l_i(x_0)\, p(x_i)$$

So, choosing $p(x) = 1$ shows immediately that $\sum_{i=1}^{N} l_i(x_0) = 1$. Likewise, if one takes $p(x) = x - x_0$ so that $p(x_0) = 0$, it follows that $\sum_{i=1}^{N} (x_i - x_0)\, l_i(x_0) = 0$. Typically, for any integer j with $1 \leq j \leq k$, setting $p(x) = (x - x_0)^j$ (which is a polynomial of degree $j \leq k$) yields $p(x_0) = 0$ and hence $\sum_{i=1}^{N} (x_i - x_0)^j\, l_i(x_0) = 0$. These moment conditions ensure that the estimator exactly reproduces polynomials of degree at most k. When analyzing the bias of the estimator via a Taylor expansion of a smooth function f around x_0, we have:

$$f(x_i) = f(x_0) + f'(x_0)(x_i - x_0) + \frac{f''(x_0)}{2!}(x_i - x_0)^2 + \cdots$$

Multiplying by $l_i(x_0)$ and summing over i, the reproduction properties force the contributions from the constant and all terms up to $(x_i - x_0)^k$ to match exactly, so that the only bias arises from the $(k+1)$th and higher order terms. In particular, for local linear regression ($k = 1$), the bias is free of constant and linear components and the leading bias term is proportional to $f''(x_0)$. Then, these conditions formally establish that $\sum_{i=1}^{N} l_i(x_0) = 1$ and $\sum_{i=1}^{N} (x_i - x_0)^j\, l_i(x_0) = 0$ for $j = 1, \ldots, k$, thereby explaining why the local polynomial estimator is unbiased for all polynomial terms up to degree k and why its leading bias depends on the $(k+1)$th derivative of f.

Problem 3:

Given a multinomial response $G \in \{1, 2, \ldots, J\}$ with features $X \in \mathbb{R}^p$ and class probabilities $\theta_j(x_0) = \Pr(G = j \mid X = x_0)$ that vary with x_0 but are assumed to be constant within a local neighborhood, so that no local linear or higher-order adjustments are introduced in the log-odds. For any given point x_0, each observation (x_i, g_i) contributes to the local log-likelihood with weight $K_\lambda(x_0, x_i)$. Then, the local negative log-likelihood to be minimized is:

$$-\ell\big(\theta(x_0)\big) = -\sum_{i=1}^{N} K_\lambda(x_0, x_i)\, \log\big(\theta_{g_i}(x_0)\big)$$

subject to $\sum_{j=1}^{J} \theta_j(x_0) = 1$ and $\theta_j(x_0) \geq 0$ for all j. Introducing a Lagrange multiplier α to enforce the equality constraint, we form the Lagrangian:

$$\mathcal{L}\big(\{\theta_j(x_0)\}, \alpha\big) = -\sum_{i=1}^{N} K_\lambda(x_0, x_i)\, \log\big(\theta_{g_i}(x_0)\big) + \alpha\Big(\sum_{j=1}^{J} \theta_j(x_0) - 1\Big)$$

Observing that the function $\theta_{g_i}(x_0)$ depends on $\theta_k(x_0)$ only when $g_i = k$, we differentiate

with respect to $\theta_k(x_0)$ to obtain

$$\frac{\partial \mathcal{L}}{\partial \theta_k(x_0)} = -\sum_{i:g_i=k} \frac{K_\lambda(x_0, x_i)}{\theta_k(x_0)} + \alpha = 0$$

Rearranging, we have:

$$\theta_k(x_0) = \frac{\sum_{i:g_i=k} K_\lambda(x_0, x_i)}{\alpha}$$

Then, summing over k and invoking the constraint $\sum_{k=1}^{J} \theta_k(x_0) = 1$ yields:

$$\sum_{k=1}^{J} \theta_k(x_0) = \frac{1}{\alpha} \sum_{k=1}^{J} \sum_{i:g_i=k} K_\lambda(x_0, x_i) = \frac{1}{\alpha} \sum_{i=1}^{N} K_\lambda(x_0, x_i) = 1$$

With $\alpha = \sum_{i=1}^{N} K_\lambda(x_0, x_i)$. Substituting:

$$\theta_k(x_0) = \frac{\sum_{i:g_i=k} K_\lambda(x_0, x_i)}{\sum_{i=1}^{N} K_\lambda(x_0, x_i)}$$

This is equivalent to taking the kernel-weighted average of the indicator function $\mathbf{1}(g_i = k)$. So, the maximum likelihood estimator for the locally constant multinomial logit model reduces to performing a Nadaraya–Watson kernel regression on each binary indicator variable.

Problem 5:

For this analysis, five values of lambda were pre-selected, with a fixed nearest neighbor window size of 50. This was chosen to significantly improve performance, as LDA over the full dataset exceeded the compute runtime limitations of Google Colab. The results for the five values are given below, where larger values of lambda clearly performed better on the test dataset.

```
Lambda: 0.1 | Train: 0.0000 | Test: 0.3473
Lambda: 0.2 | Train: 0.0000 | Test: 0.1829
Lambda: 0.5 | Train: 0.0000 | Test: 0.0563
Lambda: 0.75 | Train: 0.0000 | Test: 0.0558
Lambda: 1.0 | Train: 0.0000 | Test: 0.0508
```

Python code to perform this analysis follows:

```
import numpy as np
import pandas as pd
from sklearn.neighbors import KDTree

def local_lda_predict(x0, X_train, y_train, bandwidth, reg=1e-3):
    diff = X_train - x0
    squared_dists = np.sum(diff**2, axis=1)
    weights = np.exp(-squared_dists / (2 * bandwidth**2))
    # gaussian kernel weights

    # esp to avoid div by zero errors
```

```
    eps = np.finfo(np.float64).eps
    total_weight = np.sum(weights) + eps

    # indicator matrix
    classes = np.unique(y_train)
    K = classes.shape[0]
    n_train, d = X_train.shape
    M = (y_train.reshape(-1, 1) == classes.reshape(1, -1)).astype(np.float64)

    # weighted sums
    weighted_sum = M.T @ (weights[:, None] * X_train)
    sum_w_per_class = M.T @ weights

    # weighted means per class w/ eps for div by zeero errors
    weighted_means = np.zeros((K, d))
    nonzero = sum_w_per_class > 0
    weighted_means[nonzero] = weighted_sum[nonzero] /
    sum_w_per_class[nonzero, None]

    # local priors
    class_priors = sum_w_per_class / total_weight

    # class means
    class_idx = np.searchsorted(classes, y_train)
    mu_per_sample = weighted_means[class_idx]

    # diff and cov matrix
    diff_per_sample = X_train - mu_per_sample
    weighted_diff = diff_per_sample * weights[:, None]
    cov = (weighted_diff.T @ weighted_diff) / total_weight
    cov += reg * np.eye(d)

    inv_cov = np.linalg.inv(cov)

    # discriminate scoring
    term1 = (weighted_means @ inv_cov.T) @ x0
    term2 = 0.5 * np.sum(weighted_means * (weighted_means @ inv_cov.T), axis=1)
    with np.errstate(divide='ignore'):
        term3 = np.where(class_priors > 0, np.log(class_priors), -np.inf)

    scores = term1 - term2 + term3
    best_class = classes[np.argmax(scores)]
    return best_class

def local_lda_predict_with_nn(x0, tree, X_train, y_train,
bandwidth, k=50, reg=1e-3):
    _, ind = tree.query([x0], k=k)
    X_local = X_train[ind[0]]
```

```
    y_local = y_train[ind[0]]
    return local_lda_predict(x0, X_local, y_local, bandwidth, reg)

def local_lda_predict_many_with_nn(X_query, tree, X_train,
y_train, bandwidth, k=50, reg=1e-3):
    n_query = X_query.shape[0]
    preds = np.empty(n_query, dtype=y_train.dtype)
    for i in range(n_query):
        preds[i] = local_lda_predict_with_nn(X_query[i], tree,
        X_train, y_train, bandwidth, k, reg)
    return preds

# since the dataset is not a friendly format...
train_df = pd.read_csv("zip.train", header=None, sep='\s+')
test_df = pd.read_csv("zip.test", header=None, sep='\s+')

# to limit dataset sizes (for debugging with faster run times,
not for final run)
train_df = train_df.iloc[:]
test_df = test_df.iloc[:]

# define the labels and features
y_train = train_df.iloc[:, 0].values
X_train = train_df.iloc[:, 1:].values
y_test  = test_df.iloc[:, 0].values
X_test  = test_df.iloc[:, 1:].values

#print(f"train shape: {X_train.shape} | test shape: {X_test.shape}") # debugging

# for k nearest neighbors
tree = KDTree(X_train)

# define params to sweep
bandwidth_values = [0.1, 0.2, 0.5, 0.75, 1.0]
results = []
k_neighbors = 50  # window size

for lam in bandwidth_values:

    y_train_pred = local_lda_predict_many_with_nn(X_train,
    tree, X_train, y_train, lam, k=k_neighbors)
    train_error = np.mean(y_train_pred != y_train)

    y_test_pred = local_lda_predict_many_with_nn(X_test,
    tree, X_train, y_train, lam, k=k_neighbors)
    test_error = np.mean(y_test_pred != y_test)

    print(f"Lambda: {lam} | Train: {train_error:.4f} | Test: {test_error:.4f}")
```

```
results.append({
    "lambda": lam,
    "train_error": train_error,
    "test_error": test_error
})
```