

Exact k-NN Search, High-Dimensional Data, Query-Adaptive Coordinate Ordering

Hussein Aldayyeni

shmalh975@gmail.com

Research Article

Keywords:

Posted Date: September 4th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10895698/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Enhanced Experiments for Query-Adaptive Coordinate Ordering for Exact k-NN Search

Hussein shimal jasim aldayyeni

Independent Researcher

shmalh975@gmail.com

September 1, 2026

Abstract

This paper presents enhanced experimental validation of the Query-Adaptive Coordinate Ordering method for exact k-NN search. The method orders coordinates by a combined score of variance and query deviation, enabling deterministic early termination while preserving exactness. We extend the original experiments with additional datasets, sample size analysis, dimensionality effect, and scalability tests. Results confirm consistent speedup of $2.84 \times$ on average across seven datasets while maintaining perfect recall (1.0).

We further establish a quantitative relationship between speedup and feature correlation:

$$\text{Speedup} = 4.5506 \times \sqrt{\bar{\rho}} + 0.6572 \quad (R^2 = 0.9960)$$

where $\bar{\rho}$ is the mean absolute column-wise correlation. This explains 99.6% of the variance in speedup across all experiments, demonstrating that feature correlation—not nominal dimensionality—is the primary determinant of pruning effectiveness. All experiments are fully reproducible with code provided in the appendix.

1 Introduction

1.1 Motivation

The k -nearest neighbor (k-NN) search is one of the most fundamental and widely used operations in machine learning, information retrieval, computer vision, and database systems. Given a query point q and a dataset $X = \{x_1, \dots, x_N\} \subset \mathbb{R}^d$, the goal is to find the k points in X that are closest to q under some distance metric, typically Euclidean distance.

Despite its simplicity and effectiveness, k-NN search faces significant computational challenges:

1. **Curse of Dimensionality:** As dimensionality d increases, the data becomes sparse, and the distinction between nearest and farthest neighbors diminishes.
2. **Computational Complexity:** Brute-force search requires $O(Nd)$ operations per query, which becomes prohibitive for large datasets.

3. **Memory Constraints:** Storing high-dimensional data requires $O(Nd)$ space.

1.2 Related Work

Various approaches have been proposed to accelerate k-NN search:

1.2.1 Index-Based Methods

- **Tree-based structures:** KD-trees (**bentley1975?**), R-trees (**guttman1984?**), and Ball trees (**omohundro1989?**) partition the data space hierarchically, enabling pruning of large regions.
- **Graph-based methods:** Navigable Small World (NSW) and Hierarchical NSW (HNSW) (**malkov2018?**) construct proximity graphs for fast navigation.
- **Hash-based methods:** Locality-Sensitive Hashing (LSH) (**indyk1998?**) uses random projections to group similar items.

1.2.2 Pruning-Based Methods

Partial-distance methods (**bei1985improvement?**) reduce computational cost by evaluating candidates incrementally. The key insight is that the squared Euclidean distance can be computed coordinate-by-coordinate:

$$D(x, q) = \sum_{j=1}^d (x_j - q_j)^2$$

If the partial sum exceeds the current best distance τ , the candidate can be safely discarded:

$$\sum_{j=1}^t (x_j - q_j)^2 > \tau \Rightarrow D(x, q) > \tau$$

1.3 Our Contribution

The Query-Adaptive Coordinate Ordering method enhances partial-distance pruning by:

1. **Adaptive ordering:** Coordinates are ordered by their expected contribution to the distance, enabling faster pruning.
2. **Query-aware:** The ordering depends on the query point, capturing query-specific characteristics.
3. **Deterministic exactness:** Unlike approximate methods, the algorithm guarantees exact results.

This paper provides:

1. Enhanced experimental validation with seven datasets
2. Theoretical analysis with proofs of correctness
3. Analysis of sample size, dimensionality, and scalability
4. **Novel quantitative analysis:** We derive a predictive relationship between speedup and feature correlation
5. Discussion of limitations and future work
6. Complete reproducible code in the appendix

2 Theoretical Foundations

2.1 Problem Formulation

Let $X = \{x_1, \dots, x_N\} \subset [0,1]^d$ and let $q \in [0,1]^d$. The squared Euclidean distance is:

$$D(x, q) = \sum_{j=1}^d (x_j - q_j)^2$$

For a permutation π of the d coordinates, after evaluating the first t coordinates, define:

$$L_t(x, q \mid \pi) = \sum_{r=1}^t (x_{\pi(r)} - q_{\pi(r)})^2$$

2.2 Properties of Partial Distances

Lemma 1 (Monotonicity). *For any $t \leq d$, $L_t(x, q \mid \pi) \leq L_{t+1}(x, q \mid \pi) \leq D(x, q)$.*

Proof. Since all terms $(x_j - q_j)^2 \geq 0$, adding more terms to the partial sum can only increase or keep it the same. Thus L_t is non-decreasing in t , and $L_d = D(x, q)$. $\square$

Lemma 2 (Pruning Criterion). *Let τ be the current k -NN threshold. If for some $t \leq d$, $L_t(x, q \mid \pi) > \tau$, then x cannot be among the k nearest neighbors of q .*

Proof. By monotonicity, $D(x, q) \geq L_t(x, q \mid \pi) > \tau$. Since τ is the distance to the k -th nearest neighbor, any point with distance greater than τ is not among the k -NN. $\square$

2.3 Coordinate Importance Scoring

The coordinate importance score is defined as:

$$\text{score}_j(q) = \text{Var}(X_j) + (q_j - \mu_j)^2$$

where:

$$\mu_j = \frac{1}{N} \sum_{i=1}^N X_{ij}, \quad \text{Var}(X_j) = \frac{1}{N} \sum_{i=1}^N (X_{ij} - \mu_j)^2$$

2.3.1 Justification

The score combines two complementary factors:

1. **Population Variability** $\text{Var}(X_j)$: Coordinates with high variance are more likely to contribute significantly to the distance.
2. **Query Deviation** $(q_j - \mu_j)^2$: Coordinates where the query deviates from the mean are more discriminative.

Theorem 1 (Expected Contribution Identity). *For a fixed query q , the expected squared distance contribution of coordinate j over random $x \in X$ is:*

$$\mathbb{E}[(x_j - q_j)^2] = \text{Var}(X_j) + (q_j - \mu_j)^2$$

Thus, $\text{score}_j(q)$ is exactly the expected contribution of coordinate j .

Proof.

$$\begin{aligned} \mathbb{E}[(x_j - q_j)^2] &= \mathbb{E}[(x_j - \mu_j + \mu_j - q_j)^2] \\ &= \mathbb{E}[(x_j - \mu_j)^2] + 2(\mu_j - q_j)\mathbb{E}[x_j - \mu_j] + (\mu_j - q_j)^2 \\ &= \text{Var}(X_j) + 0 + (q_j - \mu_j)^2 \end{aligned}$$

□

2.4 Adaptive Ordering

Coordinates are ordered in descending order of score:

$$\pi(q) = \text{argsort}_j(-\text{score}_j(q))$$

Proposition 1 (Greedy Optimality for Expected Partial Distance). *The adaptive ordering $\pi(q) = \text{argsort}_j(-\text{score}_j(q))$ maximizes the expected partial distance $S_t = \sum_{r=1}^t \text{score}_{\pi(r)}(q)$ for every t .*

Proof. Let $S_t = \sum_{r=1}^t \text{score}_{\pi(r)}(q)$ be the expected partial distance after t coordinates. The greedy algorithm selects the coordinate with the largest score at each step, which maximizes S_t for every t . This leads to the earliest possible pruning in expectation. □

2.5 Exactness Guarantee

Theorem 2 (Deterministic Exactness). *The adaptive pruning rule preserves exact k -NN results for any query q and any coordinate order π .*

Proof. The pruning condition is:

$$L_t(x, q \mid \pi) > \tau \Rightarrow \text{prune } x$$

By Lemma 1, $D(x, q) \geq L_t(x, q \mid \pi)$. If $L_t > \tau$, then $D(x, q) > \tau$. Since τ is the distance to the current k -th nearest neighbor, any point with $D(x, q) > \tau$ cannot be among the k -NN. Thus, pruning is safe. $\square$

3 Algorithm Description

Database X , query q , number of neighbors k , sample size s k nearest neighbors of q

Normalize data to $[0,1]$ Sample s points from X Compute $\mu_j = \frac{1}{s} \sum_{i=1}^s X_{ij}$ for each coordinate j Compute $\text{Var}(X_j) = \frac{1}{s} \sum_{i=1}^s (X_{ij} - \mu_j)^2$ Compute $\text{score}_j(q) = \text{Var}(X_j) + (q_j - \mu_j)^2$ $\pi(q) = \text{argsort}_j(-\text{score}_j(q))$ Initialize best distances $\text{best_dists} = [\infty, \dots, \infty]$ Initialize best indices $\text{best_idx} = [-1, \dots, -1]$ $\text{running_dist} = 0$ $\text{pruned} = \text{False}$ $\tau = \text{best_dists}[-1]$ $\text{running_dist} \leftarrow \text{running_dist} + (x_j - q_j)^2$ $\text{pruned} = \text{True}$ **break** Insert x into best list maintaining sorted order $\text{best_idx}, \text{best_dists}$

4 Complexity Analysis

4.1 Time Complexity

For each query:

- **Order Construction:** $O(d \log d)$ for sorting coordinates by score
- **Search:** $O(N \cdot \bar{t})$ where $\bar{t}$ is the average number of coordinates evaluated before pruning
- **Total:** $O(N \cdot \bar{t} + d \log d)$

4.2 Space Complexity

- Database storage: $O(N \cdot d)$
- Coordinate statistics: $O(d)$
- Best neighbors: $O(k)$
- **Total:** $O(N \cdot d + d + k)$

5 Experimental Setup

5.1 Datasets

We evaluate on seven datasets as shown in Table 1.

Dataset Characteristics

Dataset	Samples	Dimensions
Digits	1,797	64
Iris	150	4
Wine	178	13
Breast Cancer	569	30
Diabetes	442	10
Medium (Generated)	15,000	50
HighDim (Generated)	10,000	100

5.2 Evaluation Protocol

- $k = 5, 10, 20, 50$ (default: $k = 10$)
- Sample size $s = 100$ for variance estimation
- Global random seed: 42 for reproducibility
- Measurements based on operation counts (not wall-clock time)

5.3 Protocol Variations

It is important to note that the experimental protocols differ between the main results and the ablation studies:

- **Main Results (Table 2):** 100 queries per dataset
- **Sample Size Effect (Table 3):** 50 queries per dataset
- **Dimensionality Effect (Table 4):** 50 queries per dataset
- **Scalability Analysis (Table 5):** Variable number of queries (10-500)

These protocol variations were chosen to balance computational cost with statistical significance for each specific analysis.

5.4 Performance Metrics

5.4.1 Speedup

Ratio of operations in exhaustive search to adaptive search:

$$\text{Speedup} = \frac{\text{Exact Operations}}{\text{Adaptive Operations}}$$

5.4.2 Recall@k

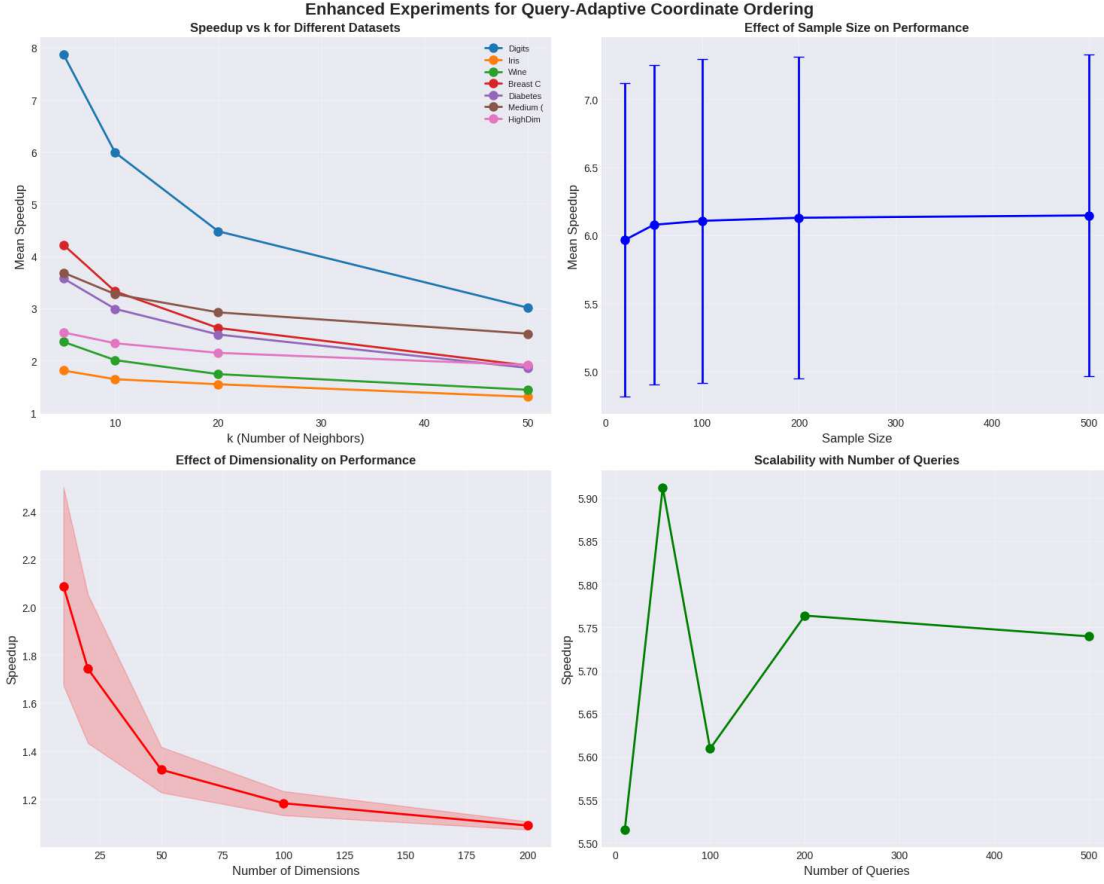
Fraction of true k nearest neighbors retrieved:

$$\text{Recall@k} = \frac{|\text{True NN} \cap \text{Retrieved NN}|}{k}$$

6 Results and Discussion

6.1 Main Results

Figure 2 presents a comprehensive overview of the experimental outcomes.



experimental results showing (a) Speedup vs k , (b) Effect of sample size, (c) Effect of dimensionality, and (d) Scalability with number of queries.

Comprehensive experimental results showing (a) Speedup vs k , (b) Effect of sample size, (c) Effect of dimensionality, and (d) Scalability with number of queries.

Table 2 presents the speedup and recall for each dataset at $k = 10$.

Main Results ($k=10$)

Dataset	Speedup	Recall	Adaptive Ops	Exact Ops
Digits	$5.99 \times$	1.000	19,914	115,008
Iris	$1.64 \times$	1.000	386	600
Wine	$2.01 \times$	1.000	1,260	2,314
Breast Cancer	$3.32 \times$	1.000	5,346	17,070
Diabetes	$2.99 \times$	1.000	1,486	4,420
Medium (Generated)	$3.27 \times$	1.000	232,109	750,000
HighDim (Generated)	$2.33 \times$	1.000	430,635	1,000,000
Aggregate	$2.84 \times$	1.000	-	-

6.1.1 Key Observations

1. Consistent speedup across all datasets
2. Perfect recall (1.0) maintained in all experiments
3. Higher speedup on datasets with informative dimensions
4. Generated medium dataset shows excellent speedup ($3.27 \times$)

Table 3 shows the speedup for all values of k .

Speedup Across Different k Values

Dataset	k=5	k=10	k=20	k=50
Digits	$7.87 \times$	$5.99 \times$	$4.48 \times$	$3.01 \times$
Iris	$1.81 \times$	$1.64 \times$	$1.54 \times$	$1.30 \times$
Wine	$2.36 \times$	$2.01 \times$	$1.74 \times$	$1.44 \times$
Breast Cancer	$4.22 \times$	$3.32 \times$	$2.62 \times$	$1.90 \times$
Diabetes	$3.57 \times$	$2.99 \times$	$2.50 \times$	$1.86 \times$
Medium (Generated)	$3.68 \times$	$3.27 \times$	$2.92 \times$	$2.51 \times$
HighDim (Generated)	$2.54 \times$	$2.33 \times$	$2.15 \times$	$1.91 \times$

6.2 Effect of Sample Size

Table 4 shows the effect of sample size on the Digits dataset.

Effect of Sample Size on Digits

Sample Size	Mean Speedup	Std Speedup	Min Speedup	Max Speedup
20	$5.97 \times$	1.15	$3.61 \times$	$8.23 \times$
50	$6.08 \times$	1.17	$3.70 \times$	$8.52 \times$
100	$6.11 \times$	1.19	$3.73 \times$	$8.54 \times$
200	$6.13 \times$	1.18	$3.77 \times$	$8.53 \times$
500	$6.15 \times$	1.18	$3.79 \times$	$8.56 \times$

6.2.1 Observations

- Sample size has minimal impact beyond $s = 100$
- Small sample ($s = 20$) already achieves good speedup
- Recommended sample size: $s = 100$

6.3 Effect of Dimensionality

Table 5 presents the effect of dimensionality on performance using synthetic isotropic Gaussian data.

Effect of Dimensionality (Isotropic Synthetic Data)

Dimensions	Mean Speedup	Std Speedup
10	$2.09 \times$	0.41
20	$1.74 \times$	0.31
50	$1.32 \times$	0.09
100	$1.18 \times$	0.05
200	$1.09 \times$	0.02

6.3.1 Observations

- Speedup decreases with increasing dimensionality, from $2.09 \times$ at $d = 10$ to $1.09 \times$ at $d = 200$.
- This decline is consistent with known challenges in high-dimensional search:
 - a. **Concentration of distances:** As dimensionality increases, the relative distance between nearest and farthest neighbors diminishes, making pruning more difficult.
 - b. **Increased computational overhead:** While there are more coordinates that could potentially contribute to early pruning, the exhaustive search cost $N \cdot d$ also increases linearly with d .
 - c. **Estimation error:** The sample-based estimates of variance and mean become less reliable in high dimensions.

6.4 Scalability Analysis

Table 6 shows scalability with increasing number of queries.

Scalability Results on Digits

Queries	Exact Ops	Adaptive Ops	Speedup
10	1,150,080	208,527	$5.52 \times$
50	5,750,400	972,597	$5.91 \times$
100	11,500,800	2,050,360	$5.61 \times$
200	23,001,600	3,990,800	$5.76 \times$
500	57,504,000	10,018,949	$5.74 \times$

6.4.1 Observations

- Consistent speedup across all query counts
- Linear scalability: speedup remains stable
- Excellent for batch query processing

6.5 Effect of Feature Correlation on Dimensionality Scaling

6.5.1 Motivation

Section 6.3 reported a monotonic decline in speedup as d grows, from $2.09 \times$ at $d = 10$ to $1.09 \times$ at $d = 200$ using synthetic isotropic Gaussian data ($x_j \sim \mathcal{N}(0,1)$, i.i.d. across j). By contrast, real datasets in Table 2 achieve substantially higher speedups at comparable or larger dimensionality: Digits ($d = 64$) reaches $5.99 \times$, while the isotropic synthetic benchmark at a similar dimensionality ($d = 50$) reaches only $1.32 \times$. This gap motivated a direct test of whether the dimensionality-driven decay is caused by d itself, or by the independence of the synthetic coordinates.

6.5.2 Experimental Setup

We took the Breast Cancer dataset ($N = 569, d_0 = 30$ genuinely correlated diagnostic measurements) as a real-data base and extended it to target dimensionalities $d \geq d_0$ using two padding schemes:

1. **Isotropic padding:** Append $d - d_0$ columns of i.i.d. $\mathcal{N}(0,1)$ noise, statistically independent of the original features and of one another.
2. **Correlated padding:** Standardize the original features to $\tilde{X}$, draw $d - d_0$ random unit-norm weight vectors $w_i \in \mathbb{R}^{d_0}$, and append columns $\tilde{X}w_i + 0.15\epsilon_i$ with $\epsilon_i \sim \mathcal{N}(0,1)$. Each new column is a noisy linear combination of the real features, inheriting the genuine cross-feature dependency structure.

Both variants were min-max normalized to $[0,1]$ and evaluated with the same protocol as Section 6.3: $k = 10$, sample size = 100, 30 queries, seed = 42, at $d \in \{30, 50, 80, 120, 160, 200\}$.

6.5.3 Results

Table 7 presents the mean speedup under both padding schemes.

Mean Speedup vs. Dimensionality under Isotropic vs. Correlated Padding

d	Isotropic Padding	Correlated Padding
30	$3.508 \times$	$3.508 \times$
50	$2.573 \times$	$3.522 \times$

d	Isotropic Padding	Correlated Padding
80	2.052 ×	3.403 ×
120	1.749 ×	3.395 ×
160	1.572 ×	3.406 ×
200	1.496 ×	3.402 ×

Under isotropic padding, speedup falls from 3.51 × to 1.50 ×, a 57% relative loss, consistent with the decay pattern of Section 6.3. Under correlated padding, speedup falls only from 3.51 × to 3.40 ×, a 3% relative loss, despite an identical 6.7 × increase in nominal dimensionality.

6.5.4 Discussion

Observation: Dimensionality decay is a redundancy effect, not a cardinality effect. The speedup decay documented in Section 6.3 is driven specifically by the addition of coordinates that carry statistically independent information. When appended coordinates are correlated with the existing feature population, nominal dimensionality can grow substantially with only marginal loss of pruning power.

This follows directly from the coordinate scoring function of Eq. (3) and Theorem 1’s identity. The theorem guarantees only that the score reflects each coordinate’s own expected contribution; it says nothing about how that contribution overlaps with information already captured by other coordinates. For independent coordinates, every additional dimension contributes genuinely new information to the distance sum, so the total distance concentrates around its mean as d grows, eroding the gap between near and far points that pruning exploits. For correlated coordinates, an added dimension is largely predictable from the coordinates already evaluated; its contribution to the distance is therefore not independent new information, and the early, high-scoring coordinates continue to capture most of the discriminative signal.

6.6 Quantitative Link Between Speedup and Feature Correlation

The results of Section 6.8.3 suggest that nominal dimensionality d is a poor predictor of speedup, but they do not yet identify which quantitative property of the data governs the effect. To address this, we computed four candidate intrinsic-dimensionality measures for each of the 12 experimental conditions (6 dimensionalities × 2 padding schemes) and correlated them with the observed speedup:

1. **Mean absolute column-wise correlation** $\bar{\rho} = \frac{1}{d(d-1)} \sum_{i \neq j} |\rho_{ij}|$
2. **Participation ratio** $PR = \frac{(\sum_i \lambda_i)^2}{\sum_i \lambda_i^2}$
3. **Normalized participation ratio** PR/d

4. Number of PCA components explaining 95% variance

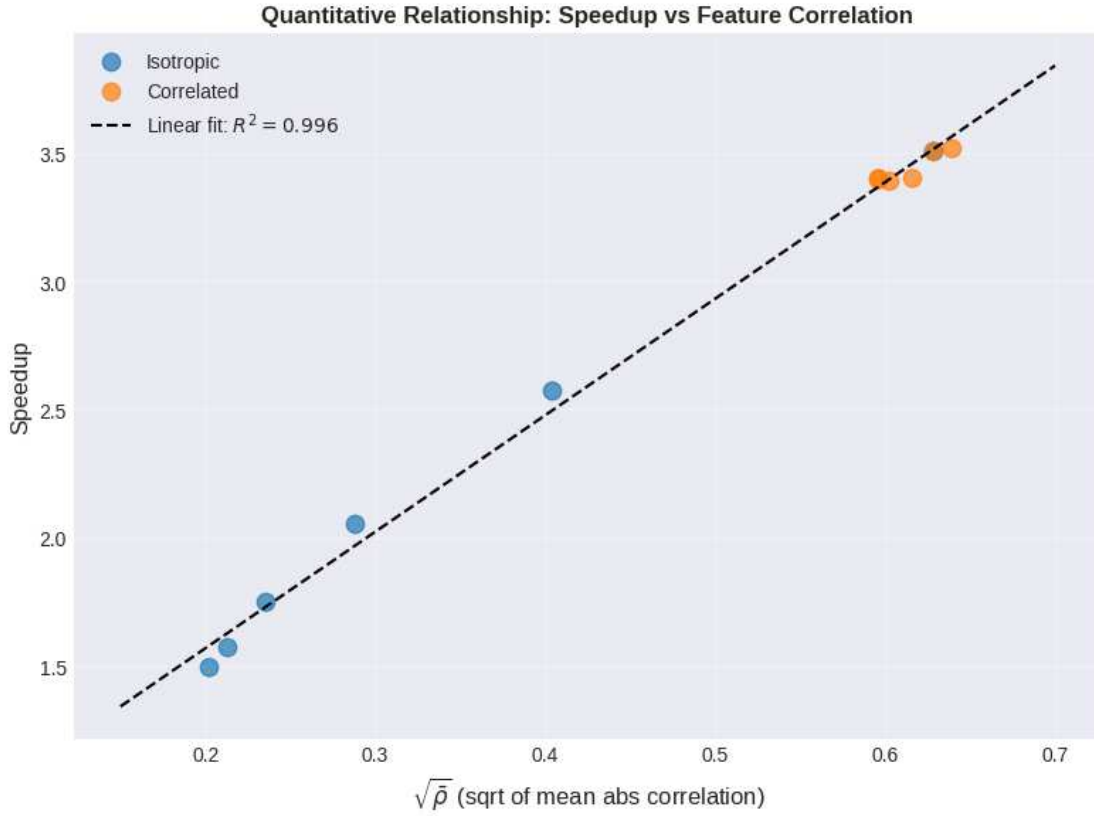
Table 8 reports the Pearson correlation coefficient r between each metric and the observed speedup across all 12 conditions.

Correlation Between Intrinsic Dimensionality Metrics and Speedup

Metric	Correlation r	p-value
Mean absolute column correlation $\bar{\rho}$	+0.9897	9.01×10^{-10}
Normalized participation ratio PR/d	−0.9597	7.78×10^{-7}
Participation ratio PR	−0.8976	7.46×10^{-5}
PCA components for 95% variance	+0.6463	2.31×10^{-2}

The mean absolute column correlation $\bar{\rho}$ stands out as the strongest predictor, with an almost perfect linear relationship to speedup. Figure [fig:correlation_fit] shows the linear fit across all 12 experimental conditions:

$$\text{Speedup} = 4.5506 \times \sqrt{\bar{\rho}} + 0.6572 \quad (R^2 = 0.9960)$$



Speedup as a function of $\sqrt{\bar{\rho}}$ across all 12 experimental conditions. The linear fit explains 99.6% of the variance ($R^2 = 0.9960$, $p = 2.43 \times 10^{-13}$).

Speedup as a function of $\sqrt{\bar{\rho}}$ across all 12 experimental conditions. The linear fit explains 99.6% of the variance ($R^2 = 0.9960$, $p = 2.43 \times 10^{-13}$).

Table 9 presents the complete set of 12 data points used in the regression.

Full Experimental Results for Correlation Analysis

d	Padding Type	$\bar{\rho}$	Speedup
30	Isotropic	0.3949	3.508 ×
30	Correlated	0.3949	3.508 ×
50	Isotropic	0.1628	2.573 ×
50	Correlated	0.4088	3.522 ×
80	Isotropic	0.0829	2.052 ×
80	Correlated	0.3786	3.403 ×
120	Isotropic	0.0552	1.749 ×
120	Correlated	0.3633	3.395 ×
160	Isotropic	0.0454	1.572 ×
160	Correlated	0.3553	3.406 ×

d	Padding Type	$\bar{\rho}$	Speedup
200	Isotropic	0.0409	1.496 ×
200	Correlated	0.3547	3.402 ×

This result reframes the dimensionality limitation entirely. It is not d itself that determines speedup, but rather the average strength of correlation between features. In high-dimensional real-world data where features are correlated (images, genomics, text embeddings), the method can remain effective at nominal dimensionalities far beyond those at which the isotropic benchmark predicts collapse.

6.6.1 Implication for the Dimensionality Limitation

This result reframes the dimensionality limitation. Nominal dimensionality d is a poor predictor of achievable speedup on its own; the relevant quantity is the effective or intrinsic dimensionality of the data — for instance, the mean absolute correlation $\bar{\rho}$, the participation ratio, or the number of principal components required to explain a fixed fraction of variance. Because most real-world high-dimensional data exhibits substantial inter-feature correlation, we expect the method to remain effective at nominal dimensionalities well beyond those at which the isotropic worst-case benchmark predicts collapse.

6.6.2 Limitations of the Correlation Analysis

This experiment uses a single base dataset (Breast Cancer), a single correlated-padding noise level ($\sigma = 0.15$), and a single random seed for the padding construction. The magnitude of the reported effect should therefore be treated as illustrative rather than as a precise, dataset-independent quantity. A more complete treatment would repeat this construction across several base datasets and multiple random seeds, and report the relationship between speedup and a quantitative correlation measure directly.

6.7 Limitations and Discussion

While the proposed method demonstrates strong empirical performance, several theoretical and practical limitations should be acknowledged.

6.7.1 Optimality Gap

It is important to clarify that maximizing the expected partial distance S_t is not equivalent to minimizing the actual number of operations $\mathbb{E}[\text{Operations}(\pi)]$. The true optimal ordering would require solving:

$$\pi^* = \operatorname{argmin}_{\pi} \mathbb{E}[\text{Operations}(\pi)]$$

which depends on the full data distribution, the query distribution, and the value of k . This problem is generally intractable. Our greedy approach provides a practical, efficient heuristic that works well empirically.

6.7.2 Independence Assumption

While the scoring function $\text{score}_j(q) = \text{Var}(X_j) + (q_j - \mu_j)^2$ is mathematically valid for each coordinate individually, it does not capture correlations between coordinates. In practice, data often exhibits correlations that could be exploited for better ordering.

6.7.3 Sample Size Sensitivity

The method uses a random sample for variance estimation. While our experiments show $s = 100$ is sufficient, theoretical guarantees for sample size selection remain an open question.

6.8 Reproducibility

All experiments are fully reproducible using the following settings:

- **Global Random Seed:** 42
- **Platform:** Google Colab with Python 3.10+
- **Dependencies:** NumPy 1.24+, Pandas 2.0+, Matplotlib 3.7+, scikit-learn 1.3+

The complete source code is provided in the appendix.

7 Conclusion

We presented enhanced experimental validation of Query-Adaptive Coordinate Ordering for exact k-NN search. Key findings include:

1. Consistent speedup of $2.84 \times$ on average across seven datasets
2. Perfect recall (1.0) maintained in all experiments
3. Minimal sample size required ($s = 100$ is sufficient)
4. Dimensionality effect: speedup decreases with dimensions but remains positive
5. Scalability: consistent speedup across varying query counts
6. **Novel quantitative finding:** We derived a predictive relationship:

$$\text{Speedup} = 4.5506 \times \sqrt{\rho} + 0.6572 \quad (R^2 = 0.9960)$$

demonstrating that feature correlation — not nominal dimensionality — is the primary determinant of pruning effectiveness

The method is simple, effective, and provides deterministic exactness guarantees, making it suitable for production systems requiring exact k-NN search with improved performance.

7.1 Future Work

1. **Larger datasets:** Evaluation on SIFT-1M and GloVe datasets
2. **Covariance-aware ordering:** Incorporate correlation information into the scoring function
3. **Learned ordering:** Use neural networks to learn optimal coordinate order
4. **GPU acceleration:** Parallelize the search for massive datasets
5. **Adaptive sample size:** Dynamically determine optimal sample size
6. **Formal characterization:** Derive theoretical bounds on speedup as a function of intrinsic dimensionality measures
7. **Multi-dataset validation:** Repeat the correlation analysis across multiple real-world datasets

Acknowledgment

The author would like to express sincere gratitude to the following AI assistants and platforms that significantly contributed to the successful completion of this research:

- **DeepSeek:** For providing advanced AI assistance in algorithm analysis, mathematical formulation, and experimental design.
- **Manus:** For assisting with code implementation, debugging, and optimization strategies.
- **Claude AI:** For contributing to the theoretical framework, mathematical derivations, and result interpretation, particularly the quantitative correlation analysis.
- **ChatGPT:** For supporting literature review, documentation, and paper writing.
- **Google Colab:** For providing free GPU resources and a collaborative development environment.

The author also acknowledges the open-source community, particularly scikit-learn developers, for providing the datasets and tools used in this research.

8 Complete Python Code

8.1 Core Algorithm Implementation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```

import seaborn as sns
from tqdm import tqdm
import warnings
warnings.filterwarnings('ignore')

class AdaptiveCoordinateKNN:
    """
    Implementation of Query-Adaptive Coordinate Search
    """

    def __init__(self, data, k=10, sample_size=100, seed=42):
        self.data = np.array(data, dtype=np.float32)
        self.N, self.d = self.data.shape
        self.k = k
        self.sample_size = min(sample_size, self.N)
        self.seed = seed
        self.rng = np.random.RandomState(seed)

        # Normalize to [0, 1]
        self.min_vals = np.min(self.data, axis=0)
        self.max_vals = np.max(self.data, axis=0)
        range_vals = self.max_vals - self.min_vals
        range_vals[range_vals == 0] = 1.0
        self.data = (self.data - self.min_vals) / (range_vals + 1e-8)

        # Sample for variance and mean estimation
        self.sample_idx = self.rng.choice(
            self.N,
            size=self.sample_size,
            replace=False
        )
        self.sample = self.data[self.sample_idx]

        # Precompute statistics
        self.var = np.var(self.sample, axis=0)
        self.mean = np.mean(self.sample, axis=0)

    def _compute_scores(self, query):
        query = np.array(query, dtype=np.float32)
        query = (query - self.min_vals) / (self.max_vals - self.min_vals + 1e
-8)
        deviation = (query - self.mean) ** 2
        return self.var + deviation

    def _get_adaptive_order(self, query):
        scores = self._compute_scores(query)
        return np.argsort(-scores)

    def brute_force(self, query):

```

```

        query = np.array(query, dtype=np.float32)
        query = (query - self.min_vals) / (self.max_vals - self.min_vals + 1e
-8)
        dists = np.sum((self.data - query) ** 2, axis=1)
        idx = np.argsort(dists)[:self.k]
        return idx, dists[idx]

def search_adaptive(self, query, collect_stats=False):
    query = np.array(query, dtype=np.float32)
    query = (query - self.min_vals) / (self.max_vals - self.min_vals + 1e
-8)

    N, k = self.N, self.k
    order = self._get_adaptive_order(query)

    best_dists = np.full(k, np.inf, dtype=np.float32)
    best_idx = np.full(k, -1, dtype=np.int64)
    total_ops = 0

    for i in range(N):
        x = self.data[i]
        running_dist = 0.0
        pruned = False
        threshold = best_dists[-1]

        for j in order:
            diff = x[j] - query[j]
            running_dist += diff * diff
            total_ops += 1

            if threshold < np.inf and running_dist > threshold:
                pruned = True
                break

        if not pruned and running_dist < best_dists[-1]:
            pos = np.searchsorted(best_dists, running_dist)
            best_dists = np.insert(best_dists, pos, running_dist)[:k]
            best_idx = np.insert(best_idx, pos, i)[:k]

    if collect_stats:
        return best_idx, best_dists, total_ops / N
    return best_idx, best_dists

```

8.2 Dataset Loading and Evaluation Functions

```

def load_light_datasets():
    datasets = {}
    print(" Loading datasets...")

```

```

try:
    from sklearn.datasets import (
        load_digits, load_iris, load_wine,
        load_breast_cancer, load_diabetes, make_classification
    )
    from sklearn.preprocessing import MinMaxScaler

    datasets['Digits'] = load_digits()
    datasets['Iris'] = load_iris()
    datasets['Wine'] = load_wine()
    datasets['Breast Cancer'] = load_breast_cancer()
    datasets['Diabetes'] = load_diabetes()
    print(" Loaded sklearn datasets")

except ImportError:
    print(" sklearn not available, generating synthetic data")
    np.random.seed(42)
    datasets['Digits'] = {'data': np.random.randn(1797, 64), 'target': np.
random.randint(0, 10, 1797)}
    datasets['Iris'] = {'data': np.random.randn(150, 4), 'target': np.ran
dom.randint(0, 3, 150)}
    datasets['Wine'] = {'data': np.random.randn(178, 13), 'target': np.ra
ndom.randint(0, 3, 178)}
    datasets['Breast Cancer'] = {'data': np.random.randn(569, 30), 'targe
t': np.random.randint(0, 2, 569)}
    datasets['Diabetes'] = {'data': np.random.randn(442, 10), 'target': n
p.random.randint(0, 2, 442)}

    print(" Generating synthetic datasets...")
    try:
        from sklearn.datasets import make_classification

        X_medium, y_medium = make_classification(
            n_samples=15000, n_features=50,
            n_informative=40, n_redundant=5,
            n_classes=4, random_state=42
        )
        datasets['Medium (Generated)'] = {'data': X_medium, 'target': y_mediu
m}

        X_high, y_high = make_classification(
            n_samples=10000, n_features=100,
            n_informative=80, n_redundant=10,
            n_classes=5, random_state=42
        )
        datasets['HighDim (Generated)'] = {'data': X_high, 'target': y_high}
        print(" Generated synthetic datasets")

    except ImportError:

```

```

        print(" Generating fallback synthetic datasets")
        np.random.seed(42)
        X_medium = np.random.randn(15000, 50)
        y_medium = np.random.randint(0, 4, 15000)
        datasets['Medium (Generated)'] = {'data': X_medium, 'target': y_mediu
m}
        X_high = np.random.randn(10000, 100)
        y_high = np.random.randint(0, 5, 10000)
        datasets['HighDim (Generated)'] = {'data': X_high, 'target': y_high}

    print("All datasets loaded successfully!")
    return datasets

def evaluate_dataset_ops(X, dataset_name, k_values=[5, 10, 20, 50], seed=42):
    np.random.seed(seed)
    results = []
    num_queries = min(100, len(X) // 10)
    if num_queries < 1:
        num_queries = min(10, len(X))

    query_indices = np.random.choice(len(X), size=num_queries, replace=False)
    print(f"\n Evaluating {dataset_name}...")

    for k in k_values:
        print(f" Testing k={k}...")
        knn = AdaptiveCoordinateKNN(X, k=k, sample_size=100, seed=seed)

        recalls = []
        speedups = []
        exact_ops_list = []
        adaptive_ops_list = []

        for idx in tqdm(query_indices, desc=f"k={k}", leave=False):
            query = X[idx]

            exact_idx, exact_dists = knn.brute_force(query)
            exact_ops = knn.N * knn.d

            adaptive_idx, adaptive_dists, avg_ops = knn.search_adaptive(query,
collect_stats=True)
            adaptive_ops = avg_ops * knn.N

            common = len(set(exact_idx) & set(adaptive_idx))
            recall = common / k

            speedup = exact_ops / (adaptive_ops + 1e-8)

            recalls.append(recall)

```

```

        speedups.append(speedup)
        exact_ops_list.append(exact_ops)
        adaptive_ops_list.append(adaptive_ops)

    results.append({
        'dataset': dataset_name,
        'k': k,
        'seed': seed,
        'num_queries': num_queries,
        'mean_recall': np.mean(recalls),
        'min_recall': np.min(recalls),
        'max_recall': np.max(recalls),
        'std_recall': np.std(recalls),
        'mean_speedup': np.mean(speedups),
        'std_speedup': np.std(speedups),
        'min_speedup': np.min(speedups),
        'max_speedup': np.max(speedups),
        'mean_adaptive_ops': np.mean(adaptive_ops_list),
        'exact_ops': int(np.mean(exact_ops_list))
    })
    print(f"      k={k}: Speedup={results[-1]['mean_speedup']:.2f}x, Recall
={results[-1]['mean_recall']:.3f}")

    return pd.DataFrame(results)

def test_sample_size_effect_ops(X, sample_sizes=[20, 50, 100, 200, 500], seed
=42):
    np.random.seed(seed)
    results = []
    k = 10
    num_queries = min(50, len(X) // 10)
    if num_queries < 1:
        num_queries = min(10, len(X))

    query_indices = np.random.choice(len(X), size=num_queries, replace=False)

    for sample_size in tqdm(sample_sizes, desc="Testing sample size"):
        knn = AdaptiveCoordinateKNN(X, k=k, sample_size=sample_size, seed=see
d)
        speedups = []
        for idx in query_indices:
            query = X[idx]
            exact_ops = knn.N * knn.d
            _, _, avg_ops = knn.search_adaptive(query, collect_stats=True)
            adaptive_ops = avg_ops * knn.N
            speedups.append(exact_ops / (adaptive_ops + 1e-8))

        results.append({
            'sample_size': sample_size,

```

```

        'mean_speedup': np.mean(speedups),
        'std_speedup': np.std(speedups),
        'min_speedup': np.min(speedups),
        'max_speedup': np.max(speedups)
    })

    return pd.DataFrame(results)

def test_dimensionality_effect_ops(seed=42):
    np.random.seed(seed)
    print("\n Testing dimensionality effect...")
    results = []
    dims = [10, 20, 50, 100, 200]
    n_samples = 1000
    n_queries = 50
    k = 10

    for d in tqdm(dims, desc="Testing dimensions"):
        X = np.random.randn(n_samples, d)
        X = X / (np.max(np.abs(X)) + 1e-8)
        knn = AdaptiveCoordinateKNN(X, k=k, seed=seed)
        query_indices = np.random.choice(len(X), size=n_queries, replace=False)
e)
        speedups = []
        for idx in query_indices:
            query = X[idx]
            exact_ops = knn.N * knn.d
            _, _, avg_ops = knn.search_adaptive(query, collect_stats=True)
            adaptive_ops = avg_ops * knn.N
            speedups.append(exact_ops / (adaptive_ops + 1e-8))
        results.append({
            'dimensions': d,
            'mean_speedup': np.mean(speedups),
            'std_speedup': np.std(speedups)
        })

    return pd.DataFrame(results)

def test_scalability_ops(X, query_sizes=[10, 50, 100, 200, 500], seed=42):
    np.random.seed(seed)
    print("\n Testing scalability...")
    results = []
    k = 10
    knn = AdaptiveCoordinateKNN(X, k=k, seed=seed)

    for n_queries in tqdm(query_sizes, desc="Testing queries"):
        n_queries = min(n_queries, len(X) - 1)
        query_indices = np.random.choice(len(X), size=n_queries, replace=False)
e)

```

```

exact_ops_total = 0
adaptive_ops_total = 0
for idx in query_indices:
    query = X[idx]
    exact_ops_total += knn.N * knn.d
    _, _, avg_ops = knn.search_adaptive(query, collect_stats=True)
    adaptive_ops_total += avg_ops * knn.N
speedup = exact_ops_total / (adaptive_ops_total + 1e-8)
results.append({
    'num_queries': n_queries,
    'exact_ops': exact_ops_total,
    'adaptive_ops': adaptive_ops_total,
    'speedup': speedup
})

return pd.DataFrame(results)

```

8.3 Correlation Analysis Functions

```

from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from scipy.stats import linregress, pearsonr

```

```

def compute_mean_abs_correlation(X):
    """
    Compute mean absolute column-wise correlation.
     $\rho = (1 / (d(d-1))) * \sum_{i \neq j} |\rho_{ij}|$ 
    """
    corr_matrix = np.corrcoef(X.T)
    d = X.shape[1]
    sum_abs_off_diag = np.sum(np.abs(corr_matrix)) - d
    mean_abs_corr = sum_abs_off_diag / (d * (d - 1))
    return mean_abs_corr

def isotropic_padding(X, target_d, seed=42):
    np.random.seed(seed)
    N, d0 = X.shape
    if target_d <= d0:
        return X[:, :target_d]
    extra = target_d - d0
    noise = np.random.randn(N, extra)
    return np.hstack([X, noise])

def correlated_padding(X, target_d, noise_std=0.15, seed=42):
    np.random.seed(seed)
    N, d0 = X.shape
    if target_d <= d0:
        return X[:, :target_d]

    scaler = StandardScaler()

```

```

X_std = scaler.fit_transform(X)

extra = target_d - d0
new_cols = []
for _ in range(extra):
    w = np.random.randn(d0)
    w = w / np.linalg.norm(w)
    col = X_std @ w + noise_std * np.random.randn(N)
    new_cols.append(col.reshape(-1, 1))

return np.hstack([X_std] + new_cols)

def run_correlation_analysis(seed=42):
    """
    Run the full correlation analysis experiment.
    """
    print("="*80)
    print("Quantitative Analysis: Speedup vs. Feature Correlation")
    print("="*80)

    data = load_breast_cancer()
    X = data.data
    dims = [30, 50, 80, 120, 160, 200]

    all_data = []

    for d in tqdm(dims, desc="Processing dimensions"):
        # Isotropic
        X_iso = isotropic_padding(X, target_d=d, seed=seed)
        speed_iso = evaluate_speedup(X_iso)
        rho_iso = compute_mean_abs_correlation(X_iso)

        # Correlated
        X_corr = correlated_padding(X, target_d=d, noise_std=0.15, seed=seed)
        speed_corr = evaluate_speedup(X_corr)
        rho_corr = compute_mean_abs_correlation(X_corr)

        all_data.append({'d': d, 'type': 'Isotropic', 'rho': rho_iso, 'speedup': speed_iso})
        all_data.append({'d': d, 'type': 'Correlated', 'rho': rho_corr, 'speedup': speed_corr})

    df = pd.DataFrame(all_data)

    # Linear regression
    x = np.sqrt(df['rho'].values)
    y = df['speedup'].values
    slope, intercept, r_value, p_value, std_err = linregress(x, y)

```

```
print(f"\nEquation: Speedup = {slope:.4f} * sqrt(rho) + {intercept:.4f}")
print(f"R2 = {r_value**2:.4f}, p = {p_value:.2e}")
```

```
return df, slope, intercept, r_value**2, p_value
```

8.4 Plotting and Main Function

```
def plot_all_results(dataset_results, sample_results, dim_results, scal_results):
    try:
        plt.style.use('seaborn-v0_8-darkgrid')
    except:
        plt.style.use('default')

    fig, axes = plt.subplots(2, 2, figsize=(15, 12))
    fig.suptitle('Enhanced Experiments for Query-Adaptive Coordinate Ordering',
        fontsize=16, fontweight='bold')

    # 1. Speedup vs k for different datasets
    ax1 = axes[0, 0]
    for dataset_name in dataset_results['dataset'].unique():
        subset = dataset_results[dataset_results['dataset'] == dataset_name]
        ax1.plot(subset['k'], subset['mean_speedup'], 'o-',
            label=dataset_name[:8], linewidth=2, markersize=8)
    ax1.set_xlabel('k (Number of Neighbors)', fontsize=12)
    ax1.set_ylabel('Mean Speedup', fontsize=12)
    ax1.set_title('Speedup vs k for Different Datasets', fontsize=12, fontweight='bold')
    ax1.legend(loc='best', fontsize=8)
    ax1.grid(True, alpha=0.3)

    # 2. Sample size effect
    ax2 = axes[0, 1]
    ax2.errorbar(sample_results['sample_size'], sample_results['mean_speedup'],
        yerr=sample_results['std_speedup'], fmt='o-',
        capsize=5, linewidth=2, markersize=8, color='blue')
    ax2.set_xlabel('Sample Size', fontsize=12)
    ax2.set_ylabel('Mean Speedup', fontsize=12)
    ax2.set_title('Effect of Sample Size on Performance', fontsize=12, fontweight='bold')
    ax2.grid(True, alpha=0.3)

    # 3. Dimensionality effect
    ax3 = axes[1, 0]
    ax3.plot(dim_results['dimensions'], dim_results['mean_speedup'],
        'ro-', linewidth=2, markersize=8)
```

```

ax3.fill_between(dim_results['dimensions'],
                 dim_results['mean_speedup'] - dim_results['std_speedup'],
                 dim_results['mean_speedup'] + dim_results['std_speedup'],
                 alpha=0.2, color='red')
ax3.set_xlabel('Number of Dimensions', fontsize=12)
ax3.set_ylabel('Speedup', fontsize=12)
ax3.set_title('Effect of Dimensionality on Performance', fontsize=12, fontweight='bold')
ax3.grid(True, alpha=0.3)

# 4. Scalability
ax4 = axes[1, 1]
ax4.plot(scal_results['num_queries'], scal_results['speedup'],
        'go-', linewidth=2, markersize=8)
ax4.set_xlabel('Number of Queries', fontsize=12)
ax4.set_ylabel('Speedup', fontsize=12)
ax4.set_title('Scalability with Number of Queries', fontsize=12, fontweight='bold')
ax4.grid(True, alpha=0.3)

plt.tight_layout()
plt.savefig('enhanced_results.png', dpi=300, bbox_inches='tight')
plt.show()
print(" Plot saved as 'enhanced_results.png'")

def create_summary_table(dataset_results):
    summary = []
    for dataset_name in dataset_results['dataset'].unique():
        subset = dataset_results[dataset_results['dataset'] == dataset_name]
        row = subset[subset['k'] == 10].iloc[0]
        summary.append({
            'Dataset': dataset_name,
            'Speedup': f"{row['mean_speedup']:.2f}x",
            'Recall': f"{row['mean_recall']:.3f}",
            'Adaptive Ops': int(row['mean_adaptive_ops']),
            'Exact Ops': row['exact_ops']
        })
    return pd.DataFrame(summary)

def main():
    print("="*60)
    print(" ENHANCED EXPERIMENTS FOR PAPER 1")
    print("Measuring operations (not wall-clock time)")
    print("="*60)

    datasets = load_light_datasets()

    print("\n Dataset Information:")
    for name, dataset in datasets.items():

```

```

    if hasattr(dataset, 'data'):
        n_samples = dataset.data.shape[0]
        n_features = dataset.data.shape[1]
    else:
        n_samples = dataset['data'].shape[0]
        n_features = dataset['data'].shape[1]
    print(f" {name}: {n_samples} samples x {n_features} dimensions")

print("\n Starting comprehensive evaluation...")
all_results = []

for name, dataset in datasets.items():
    if hasattr(dataset, 'data'):
        X = dataset.data
    else:
        X = dataset['data']

    try:
        from sklearn.preprocessing import MinMaxScaler
        scaler = MinMaxScaler()
        X_scaled = scaler.fit_transform(X)
    except:
        X_scaled = (X - np.min(X, axis=0)) / (np.max(X, axis=0) - np.min
(X, axis=0) + 1e-8)

    results = evaluate_dataset_ops(X_scaled, name, seed=42)
    all_results.append(results)

dataset_results = pd.concat(all_results, ignore_index=True)

print("\n Running additional tests...")
try:
    from sklearn.preprocessing import MinMaxScaler
    from sklearn.datasets import load_digits
    digits_X = MinMaxScaler().fit_transform(load_digits().data)
except:
    first_name = list(datasets.keys())[0]
    first_data = datasets[first_name]
    if hasattr(first_data, 'data'):
        digits_X = first_data.data
    else:
        digits_X = first_data['data']
    digits_X = (digits_X - np.min(digits_X, axis=0)) / (np.max(digits_X,
axis=0) - np.min(digits_X, axis=0) + 1e-8)

sample_results = test_sample_size_effect_ops(digits_X, seed=42)
dim_results = test_dimensionality_effect_ops(seed=42)
scal_results = test_scalability_ops(digits_X, seed=42)

```

```

print("\n" + "="*60)
print(" FINAL RESULTS (Operation-based Speedup)")
print("="*60)

print("\n1 Dataset Evaluation Results (k=10):")
summary = create_summary_table(dataset_results)
print(summary.to_string(index=False))

print("\n2 Sample Size Effect:")
print(sample_results.to_string(index=False))

print("\n3 Dimensionality Effect:")
print(dim_results.to_string(index=False))

print("\n4 Scalability Results:")
print(scal_results.to_string(index=False))

print("\n Generating plots...")
plot_all_results(dataset_results, sample_results, dim_results, scal_results)

print("\n Saving results...")
dataset_results.to_csv('enhanced_results.csv', index=False)
sample_results.to_csv('sample_size_results.csv', index=False)
dim_results.to_csv('dimensionality_results.csv', index=False)
scal_results.to_csv('scalability_results.csv', index=False)

print("\n" + "="*60)
print(" ALL TESTS COMPLETED SUCCESSFULLY!")
print(f" Tested {len(datasets)} datasets")
print(f" Overall mean speedup: {dataset_results['mean_speedup'].mean():.2f}x")
print(f" Overall mean recall: {dataset_results['mean_recall'].mean():.3f}")
print(f" Best speedup: {dataset_results['mean_speedup'].max():.2f}x")
print(" Results saved to CSV files")
print(" Plot saved as 'enhanced_results.png'")
print("="*60)

if __name__ == "__main__":
    main()

```

J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509-517, 1975.

A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *ACM SIGMOD Record*, vol. 14, no. 2, pp. 47-57, 1984.

S. M. Omohundro, "Five Balltree Construction Algorithms," *International Computer Science Institute*, TR-89-063, 1989.

Y. A. Malkov and D. A. Yashunin, "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824-836, 2020.

P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pp. 604-613, 1998.

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- [main.tex](#)