Nature-Inspired Dimensionality Reduction Algorithms

This document provides a comprehensive overview of nature-inspired algorithms for dimensionality reduction, including their implementation details, time complexity analysis, and comparison with traditional techniques like PCA and t-SNE.

Introduction

Dimensionality reduction is a critical preprocessing step in data visualization and analysis that projects high-dimensional data into a lower-dimensional space while preserving as much of the original structure as possible. This document focuses on nature-inspired algorithms for dimensionality reduction and compares them with traditional methods.

Traditional Methods

PCA

Principal Component Analysis (PCA) is a linear dimensionality reduction technique that identifies the directions (principal components) along which the data varies the most.

python

```python
def pca_visualize(X, y=None, n_components=2, n_neighbors=10):
    X_red = PCA(n_components=n_components).fit_transform(X)
    labels = y if y is not None else KMeans(n_clusters=3, n_init=10).fit_predict(X_red)
    sil_score = silhouette_score(X_red, labels)
    tw_score = trustworthiness(X, X_red, n_neighbors=n_neighbors)
    plt.scatter(X_red[:, 0], X_red[:, 1], c=labels, cmap="viridis", s=20)
    plt.xlabel("PC-1")
    plt.ylabel("PC-2")
    plt.title(f"PCA 2-D | Silhouette={sil_score:.3f} | Trustworthiness={tw_score:.3f}")
    plt.tight_layout()
    plt.show()
    print(f"Silhouette Score: {sil_score:.3f}")
    print(f"Trustworthiness : {tw_score:.3f}")
```

Time Complexity: $O(\min(n^2d, nd^2))$ where n is the number of samples and d is the number of features.

- Computing covariance matrix: $O(nd^2)$

- Eigenvalue decomposition: $O(d^3)$

Key Characteristics:

- Linear technique
- Fast and computationally efficient
- Preserves global structure but may fail to preserve local structure
- Focuses on variance maximization

t-SNE

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a nonlinear dimensionality reduction technique that focuses on preserving local structure.

python

```python
def tsne_visualize(X, y=None, n_components=2, n_neighbors=10):
    X_red = TSNE(n_components=n_components, random_state=0, init="random", learning_rate="auto"
    labels = y if y is not None else KMeans(n_clusters=3, n_init=10).fit_predict(X_red)
    sil_score = silhouette_score(X_red, labels)
    tw_score = trustworthiness(X, X_red, n_neighbors=n_neighbors)
    plt.scatter(X_red[:, 0], X_red[:, 1], c=labels, cmap="viridis", s=20)
    plt.xlabel("t-SNE-1")
    plt.ylabel("t-SNE-2")
    plt.title(f"t-SNE 2-D | Silhouette={sil_score:.3f} | Trustworthiness={tw_score:.3f}")
    plt.tight_layout()
    plt.show()
    print(f"Silhouette Score: {sil_score:.3f}")
    print(f"Trustworthiness : {tw_score:.3f}")
```

Time Complexity: $O(n^2 \log n)$ where n is the number of samples.

- Computing pairwise distances: $O(n^2)$
- Gradient descent optimization: $O(n^2 \times \text{iterations})$

Key Characteristics:

- Nonlinear technique
- Computationally intensive
- Excellent at preserving local structure
- May distort global structure

- Stochastic algorithm with random initialization

Nature-Inspired Algorithms

Self-Organizing Maps (SOM)

Self-Organizing Maps (SOM) is a type of artificial neural network that produces a low-dimensional representation of the input space, preserving topological properties.

Implementation Details:

```python
def evaluate_som_on_wine_data(grid_size=7, pca_components=5, som_iterations=10000, random_state
    # Apply PCA
    pca = PCA(n_components=pca_components)
    X_pca = pca.fit_transform(X)

    # Initialize SOM
    som = MiniSom(grid_size, grid_size, pca_components, sigma=3.0, learning_rate=0.5, random_se
    som.train_random(X_pca, som_iterations)

    # Compute BMUs (Best Matching Units)
    bmu_indices = [som.winner(x) for x in X_pca]

    # Remaining code for visualization and evaluation
```

Time Complexity: $O(n \times \text{iterations} \times \text{grid\_size}^2 \times d)$ where:

- n: number of samples
- iterations: number of training iterations
- grid_size^2: number of neurons in the grid
- d: dimensionality of input data after PCA preprocessing

Key Components:

- Initialization: Random weight initialization for each neuron in the grid
- Training: Competitive learning where neurons compete to respond to input stimuli
- Neighborhood Function: Defines the influence of winning neuron on its neighbors
- Learning Rate: Controls the magnitude of weight updates during training
- Best Matching Unit (BMU): Neuron with weights most similar to an input vector

Advantages:

- Preserves topological relationships in the data
- Visualization through U-matrix shows cluster boundaries
- Can handle non-linear relationships
- Combines clustering and projection

Limitations:

- Requires parameter tuning (grid size, iterations, learning rate)
- Fixed grid structure may not be optimal for all datasets
- May converge to local optima

Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) adapts the foraging behavior of ants to find optimal paths, here applied to dimensionality reduction.

Implementation Details:

```python
def run_aco(num_iterations, num_ants, evaporation_rate):
    K_attract, K_repulse = 30, 50
    nbrs = NearestNeighbors(n_neighbors=K_attract + 1).fit(X)
    indices = nbrs.kneighbors(X, return_distance=False)
    nbrs_repulse = NearestNeighbors(n_neighbors=K_repulse + 1).fit(X)
    indices_repulse = nbrs_repulse.kneighbors(X, return_distance=False)

    Y = np.random.randn(n_points, dim_low)
    pheromone = np.ones((n_points, dim_low))
    repulsion_strength, attraction_strength = 0.1, 0.5
    alpha = 0.5
    centering_interval = 20

    # Main iterations and ant movement logic
```

Time Complexity: O(iterations × ants × n^2 × k) where:

- iterations: number of algorithm iterations
- ants: number of artificial ants
- n: number of data points

- k: number of nearest neighbors considered

Key Components:

- Initialization: Random initial projections and pheromone matrix
- Attraction and Repulsion: Based on neighborhood relationships in high-dimensional space
- Pheromone Update: Reinforces good projections through evaporation and deposition
- Move Scale: Decreases over iterations for convergence
- Centering and Scaling: Periodically applied for stability

Advantages:

- Balances local and global structure preservation
- Adaptive search based on collective intelligence
- Can escape local optima through stochastic exploration

Limitations:

- Sensitive to parameter settings
- May require many iterations for convergence
- Computationally intensive for large datasets

Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) adapts the social behavior of bird flocking or fish schooling, here used for feature selection and dimensionality reduction.

Implementation Details:

python

```python
def pso_select(pop_size=90, max_iter=90, n_clusters=3, alpha=0.5, w=0.8, c1=1.2, c2=1.2, k_trus
    # Sigmoid and binary position conversions
    sigmoid = lambda z: 1 / (1 + np.exp(-z))
    bin_pos = lambda z: (sigmoid(z) > .5).astype(int)

    # Fitness function combining silhouette score and trustworthiness
    def fit_val(p):
        m = bin_pos(p)
        if m.sum() != 2:
            return np.inf
        Xs = X[:, m == 1]
        try:
            sil = silhouette_score(Xs, KMeans(n_clusters, n_init=1, random_state=42).fit_predic
        except:
            return np.inf
        return -alpha*sil - (1-alpha)*trust(X, Xs)

    # Particle class and main PSO iteration logic
```

Time Complexity: O(iterations × particles × n × d) where:

- iterations: number of PSO iterations
- particles: number of particles in the swarm
- n: number of data points
- d: number of features in the original dataset

Key Components:

- Particle Representation: Real-valued vector corresponding to feature selection probability
- Velocity: Controls how particles move through the search space
- Personal Best: Best position found by each particle
- Global Best: Best position found by any particle
- Fitness Function: Combines silhouette score and trustworthiness

Advantages:

- Performs feature selection and dimensionality reduction simultaneously
- Balances between clustering quality and trustworthiness

- Simple to implement and parallelize

Limitations:

- Binary feature selection limits to existing dimensions
- May get trapped in local optima
- Performance depends on initialization and parameter settings

Bat Algorithm (BA)

Bat Algorithm (BA) mimics the echolocation behavior of bats to find optimal solutions in the search space.

Implementation Details:

```python
def BAT_ALGORITHM(
    X, y=None, n_components=2, n_bats=40, max_iter=100,
    f_min=0.0, f_max=2.0, alpha=0.9, gamma=0.9, A0=1.0, r0=0.5,
    n_neighbors=10, n_clusters=3, return_embed=False):

    n_samples, n_features = X.shape
    dim = n_features * n_components
    pos   = np.random.uniform(-1, 1, (n_bats, dim))
    vel   = np.zeros_like(pos)
    loud  = np.full(n_bats, A0)
    pulse = np.full(n_bats, r0)

    # Projection, fitness evaluation, and main iteration logic
```

Time Complexity: O(iterations × bats × n × (d + k)) where:

- iterations: number of iterations
- bats: number of bats in the population
- n: number of data points
- d: number of features
- k: number of nearest neighbors for trustworthiness calculation

Key Components:

- Position: Each bat represents a potential projection matrix

- Velocity: Rate and direction of movement in the search space
- Frequency: Controls the step size of movement
- Loudness: Decreases as bats find good solutions
- Pulse Rate: Increases as bats converge to solutions
- Fitness Function: Based on trustworthiness

Advantages:

- Balances exploration and exploitation through frequency and pulse rate
- Can handle non-linear transformations
- Automatic parameter adjustment through loudness and pulse rate

Limitations:

- Many parameters to tune
- May converge slowly for complex datasets
- Computationally intensive projection calculation

Artificial Bee Colony (ABC)

Artificial Bee Colony (ABC) simulates the foraging behavior of honeybees to optimize solutions.

Implementation Details:

```python
def run_abc(X, y, num_iterations=100, num_bees=20, limit=5):
    # Parameters
    K_attract = 30
    n_points, dim_high = X.shape
    dim_low = 2

    # Nearest neighbors in high dimensional space for local attraction guidance
    nbrs = NearestNeighbors(n_neighbors=K_attract + 1).fit(X)
    indices = nbrs.kneighbors(X, return_distance=False)

    # Initialize bees randomly
    bee_positions = [np.random.randn(n_points, dim_low) for _ in range(num_bees)]
    trial_counters = np.zeros(num_bees)

    # Fitness function and main iteration logic with employed, onlooker, and scout phases
```

Time Complexity: O(iterations × bees × n^2 × k) where:

- iterations: number of algorithm iterations
- bees: number of bees in the colony
- n: number of data points
- k: number of nearest neighbors for local attraction

Key Components:

- Employed Bees: Explore current solutions and share information
- Onlooker Bees: Select promising solutions based on fitness
- Scout Bees: Abandon poor solutions and explore new areas
- Local Neighborhood Guidance: Uses high-dimensional structure to guide low-dimensional movement
- Trial Counter: Tracks solution improvement attempts

Advantages:

- Balance between exploitation and exploration
- Adaptive search with local guidance
- Can escape local optima through scout phase

Limitations:

- Parameter sensitivity
- Potential slow convergence
- Memory-intensive for large datasets

Firefly Algorithm (FA)

Firefly Algorithm (FA) is inspired by the flashing behavior of fireflies, where brightness attracts other fireflies.

Implementation Details:

```python
def run_firefly(X, y, num_iterations=40, num_fireflies=20, beta0=1.0, gamma=1.0, alpha=0.2):
    n_points, dim_original = X.shape
    dim_low = 2

    # Nearest neighbors for attractive and repulsive forces
    K_attract, K_repulse = 30, 50
    nbrs = NearestNeighbors(n_neighbors=K_attract + 1).fit(X)
    indices = nbrs.kneighbors(X, return_distance=False)
    nbrs_repulse = NearestNeighbors(n_neighbors=K_repulse + 1).fit(X)
    indices_repulse = nbrs_repulse.kneighbors(X, return_distance=False)

    # Initialize fireflies
    fireflies = [np.random.randn(n_points, dim_low) for _ in range(num_fireflies)]

    # Main algorithm loop with attraction/repulsion logic
```

Time Complexity: O(iterations × fireflies^2 × n^2 × k) where:

- iterations: number of algorithm iterations
- fireflies: number of fireflies in the population
- n: number of data points
- k: number of nearest neighbors for local forces

Key Components:

- Brightness/Fitness: Based on variance of the projection
- Attraction: Fireflies move toward brighter ones
- Distance-dependent Attraction: Decreases with distance (gamma parameter)
- Random Movement: Controlled by alpha parameter
- Neighborhood Forces: Local attraction and repulsion based on high-dimensional structure

Advantages:

- Automatic subdivision of population based on brightness
- Can handle multimodal optimization landscapes
- Incorporates both global movement and local guidance

Limitations:

- Quadratic complexity in number of fireflies
- May require fine-tuning of parameters
- Convergence speed depends on initial positions

Performance Comparison

Self-Organizing Maps (SOM)

Time Complexity: O(n × iterations × grid_size^2 × d)

Seed	Silhouette Score	Trustworthiness
42	0.0202	0.9080
142	-0.0281	0.9158
254	0.0004	0.9154
444	0.0053	0.9190
698	-0.0013	0.9163
Average	**-0.0007**	**0.9149**

Ant Colony Optimization (ACO)

Time Complexity: O(iterations × ants × n^2 × k)

Seed	Silhouette Score	Trustworthiness
6872	0.6775	0.8782
42	0.5112	0.8291
120	0.5432	0.8263
33333	0.5498	0.8324
1	0.6469	0.8607
999	0.5347	0.8480
159	0.5061	0.8508
61161	0.5711	0.8514
51661513	0.6113	0.8598
1234556	0.5311	0.8511
Average	**0.5683**	**0.8488**

Particle Swarm Optimization (PSO)

Seed	Silhouette Score	Trustworthiness
1	0.5000	0.8400
42	0.5200	0.8100
768	0.5000	0.8400
6895	0.5000	0.8400
74445	0.5000	0.8400
Average	**0.5040**	**0.8340**

Bat Algorithm (BA)

Seed	Silhouette Score	Trustworthiness
1	0.4236	0.8652
42	0.3210	0.8725
576	0.4664	0.8642
212	0.3725	0.8682
989	0.3543	0.8651
Average	**0.3876**	**0.8670**

Artificial Bee Colony (ABC)

Seed	Silhouette Score	Trustworthiness
1234556	0.2708	0.7735
1231	0.3258	0.7641
42	0.2357	0.7538
50	0.2999	0.7430
100	0.2795	0.7496
Average	**0.2823**	**0.7568**

Firefly Algorithm (FA)

Seed	Silhouette Score	Trustworthiness
1234556	0.0316	0.7363
100	-0.0353	0.6867
42	0.1290	0.6656
50	0.0326	0.7252
1231	0.0420	0.7622
Average	**0.0400**	**0.7152**

Summary of Average Performance

Algorithm	Average Silhouette Score	Average Trustworthiness
SOM	-0.0007	0.9149
ACO	0.5683	0.8488
PSO	0.5040	0.8340
BA	0.3876	0.8670
ABC	0.2823	0.7568
FA	0.0400	0.7152

Original Comparison Table:

Algorithm	Silhouette Score	Trustworthiness	Time Complexity	Space Complexity
PCA	Varies by dataset	Moderate	$O(\min(n^2d, nd^2))$	$O(nd)$
t-SNE	Often high	High	$O(n^2 \log n)$	$O(n^2)$
SOM	Moderate	Moderate	$O(n \times \text{iterations} \times \text{grid\_size}^2 \times d)$	$O(\text{grid\_size}^2 \times d)$
ACO	~0.27-0.33	~0.74-0.77	$O(\text{iterations} \times \text{ants} \times n^2 \times k)$	$O(n \times d\_low)$
PSO	Varies by features	Varies	$O(\text{iterations} \times \text{particles} \times n \times d)$	$O(\text{particles} \times d)$
BA	~0.24-0.31	~0.74-0.77	$O(\text{iterations} \times \text{bats} \times n \times (d + k))$	$O(\text{bats} \times d \times d\_low)$
ABC	~0.27-0.33	~0.74-0.77	$O(\text{iterations} \times \text{bees} \times n^2 \times k)$	$O(\text{bees} \times n \times d\_low)$
FA	~0.03-0.04	~0.73-0.74	$O(\text{iterations} \times \text{fireflies}^2 \times n^2 \times k)$	$O(\text{fireflies} \times n \times d\_low)$

Conclusion

Nature-inspired algorithms offer interesting alternatives to traditional dimensionality reduction techniques like PCA and t-SNE. While they generally have higher computational complexity, they provide different trade-offs in terms of structure preservation and visualization quality.

Key Observations:

Trustworthiness:

- SOM achieves the highest trustworthiness score (0.9149), significantly outperforming all other algorithms in preserving local structure.
- BA also performs well (0.8670), followed by ACO and PSO.
- ABC and FA have relatively lower trustworthiness scores but still maintain reasonable preservation of local structure.

Silhouette Score:

- ACO demonstrates the best clustering performance with the highest average silhouette score (0.5683).
- PSO shows remarkable consistency in silhouette scores across different seeds.
- SOM and FA have poor silhouette scores, suggesting they don't create well-defined clusters.

Computational Complexity:

- Nature-inspired algorithms generally have higher computational complexity than PCA but can be competitive with t-SNE for smaller datasets.
- FA has the highest computational complexity due to its quadratic dependence on the number of fireflies.
- PSO offers a good balance between performance and computational efficiency.

Parameter Sensitivity:

- These algorithms require careful parameter tuning, which can be both a challenge and an opportunity for customization.
- Different random seeds can significantly affect performance, particularly for silhouette scores.

Visualization Quality:

- The nature-inspired approaches often produce visually distinctive projections that can reveal different aspects of the data structure.
- The trade-off between trustworthiness and silhouette score suggests these algorithms have different strengths in preserving global versus local structure.

The choice of algorithm should depend on the specific requirements of the visualization task, dataset characteristics, and computational constraints. SOM is recommended when preserving local structure is paramount, while ACO provides better cluster separation. For balanced performance, BA offers a good compromise between trustworthiness and silhouette score.