Lecture 20 - Non-Linear Dimensionality Reduction with Manifold Learning

Manifold Learning (Non-Linear Dimensionality Reduction)

As we have already noted, many natural sources of data correspond to low-dimensional, possibly noisy, non-linear manifolds embedded within the higher dimensional observed data space.Capturing this property explicitly can lead to improved density modeling compared with more general methods.

PCA and LDA are often used to project a data set onto a lower-dimensional space. However both of them assume that the data samples live in an underlying linear manifold.

There are other dimensionality reduction techniques that do not assume the manifold is linear. They include:

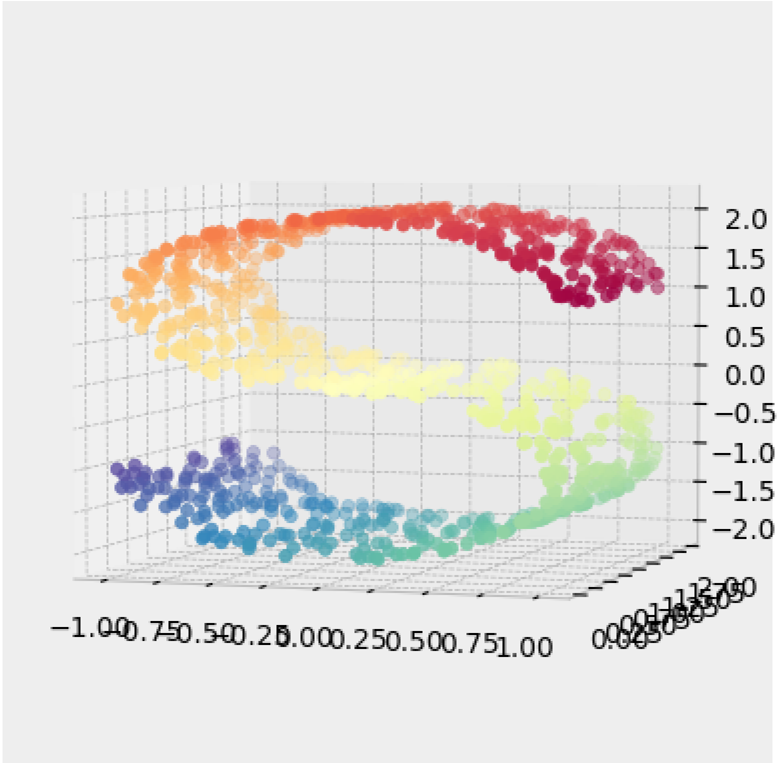
1. Multi-Dimensional Scaling (MDS)
2. Isometric Mapping (ISOMAP)
3. Locally Linear Embedding (LLE)
4. t-Distributed Stochastic Neighbor Embedding (t-SNE)

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('bmh')
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
```

In [2]:

```
from sklearn.datasets import make_s_curve

n_points = 1000
X_scurve, color = make_s_curve(n_points, random_state=0)
n_neighbors = 10
n_components = 2

fig=plt.figure(figsize=(15, 5))
ax=fig.add_subplot(111, projection='3d')
ax.scatter(X_scurve[:, 0], X_scurve[:, 1], X_scurve[:, 2], c=color, cmap=plt.cm.Spe
ax.view_init(4, -72)
```



Here is the **simplest possible explanation** of **MDS, ISOMAP, LLE, and t-SNE** — with intuition only, no math, and baby-level examples.

🌈 First: What are these?

All four are **manifold learning / nonlinear dimensionality reduction** algorithms. They take **high-dimensional data** and make it **lower-dimensional (usually 2D)** while keeping the *shape* of the data.

You can think of them as **ways to make a map** of complicated data.

🌍 1) MDS — Multi-Dimensional Scaling

Simple Meaning

👉 "Place the points in 2D so that the distances between them stay the same."

Analogy

You know the distances between cities (Delhi–Mumbai, Mumbai–Chennai). MDS creates a *map* that keeps those distances consistent.

Works Well For

- Data where distances matter
- Global structure

Weakness

- Doesn't understand curved manifolds
- Only uses distances → not local structure

🛣️ 2) ISOMAP

Simple Meaning

👉 "Like MDS, but uses *road distances* instead of straight-line distances."

Analogy

Imagine a mountain. Two towns on opposite sides are **close in straight line**, but **far by road**.

ISOMAP uses **road distance** (geodesic distance), not straight distance.

Why?

Because data may lie on a curved surface → straight distance is misleading.

Works Well For

- Curved shapes (Swiss roll)
- Nonlinear data

Weakness

- Breaks if distances between far points are wrong
- Sensitive to number of neighbors

🧩 3) LLE — Locally Linear Embedding

Simple Meaning

👉 “Every point can be reconstructed from its closest neighbors; keep that relationship in the low-dimensional space.”

Analogy

Imagine each point has 5 friends. Your position is defined by your friends’ positions.

LLE says:

> If you keep friend relationships the same, you keep the shape of the manifold.

What It Preserves

- Local structure (small neighborhoods)
- Curved surfaces

Weakness

- Doesn’t preserve global shape
- Can produce weird holes or distortions

🎨 4) t-SNE

Simple Meaning

👉 “Make similar points stay very close, and push very different points far apart.”

Analogy

Imagine compressing a crowd:

- Close friends stay together
- Strangers get pushed away This creates clear **clusters**.

Why People Love t-SNE

- Beautiful visualizations
- Perfect for clustering
- Works incredibly well with images, embeddings, etc.

Weakness

- Cannot preserve global structure
- Plots can look random if parameters are wrong

- You cannot use it for downstream ML (only visualization)

🎯 Summary Table (Super Easy)

Method	What it preserves	Intuition	Good for
MDS	Distances	Make a map by keeping distances	Simple global structures
ISOMAP	Road distance (geodesic)	Map curved surfaces	Manifolds (Swiss roll)
LLE	Local neighborhoods	Keep friend relations	Local structure, curved data
t-SNE	Clusters	Make similar points very close	Visualization of clusters

---

🍼 Baby-Level Summary

- **MDS:** Keep distances same
- **ISOMAP:** Keep *road* distances same
- **LLE:** Keep neighborhoods same
- **t-SNE:** Make clusters look clean and separated

If you want, I can also explain: 👉 PCA vs MDS vs ISOMAP vs LLE vs t-SNE (very simple) 👉 Which one to use in which situation 👉 Visual demo with Swiss roll or digits dataset

Just tell me!

Multi-Dimensional Scaling (MDS)

Another linear technique with a similar aim is **multidimensional scaling**, or **MDS**. It finds a low-dimensional projection of the data such as to **preserve the pairwise distances between data points**, and involves finding the eigenvectors of the distance matrix.

Consider a set of mean-centered observations $X = \{x_1, x_2, \ldots, x_N\}$ where $x_i \in \mathrm{R}^D$. By mean-centered samples X, I mean that $\mu_j = \sum_{i=1}^{N} x_{ij} = 0, \forall j = 1, 2, \ldots, D$.

Consider the **proximity matrix** D that stores pairwise distances of data points $d_{ij} = \mathrm{distance}(x_i, x_j)$:

$$D = \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1N} \\ d_{21} & d_{22} & \cdots & d_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ d_{N1} & d_{N2} & \cdots & d_{NN} \end{bmatrix}$$

Note that D is an $N \times N$ symmetric matrix.

Given an assumed Euclidean proximity matrix, D, the **goal** of MDS is to find a set of points, Y, that have the same proximity matrix in an M-dimensional space, where $M < D$.

- MDS preserves the global data structure.
- MDS can use any distance metric to compute the pairwise distances between points.
- There is the need to store the proximity matrix (half of it, since it is symmetric). Thus requiring a significant computational and storage resources for large datasets.
    - There are $N(N-1)/2$ distance computations, where N is the number of samples.
- **Classical MDS** refers to MDS when using Euclidean distances.
- In the case where the pairwise distances are computed with the Euclidean distance metric, MDS gives equivalent results to PCA. Therefore, MDS is a generalization of PCA.

---

euclidean proximity It is a table of distances between every pair of points, where the distance is Euclidean distance (the normal straight-line distance).

That's it.

Deciding Number of Dimensions

Interpretability of the MDS solution is often important, and lower dimensional solutions will typically be easier to interpret and visualize. However, dimension selection is also an issue of balancing underfitting and overfitting. Lower dimensional solutions may underfit by leaving out important dimensions of the dissimilarity data. Higher dimensional solutions may overfit to noise in the dissimilarity measurements.

- How would you decide on the number of dimensions?

Model selection tools like cross-validation on the subsequent module in the pipeline (e.g. classification) can thus be useful to select the dimensionality that balances underfitting and overfitting.

Test the Results for Reliability and Validity

Compute R-squared to determine what proportion of variance of the scaled data can be accounted for by the MDS procedure. An R-square of 0.6 is considered the minimum acceptable level. An R-square of 0.8 is considered good for metric scaling and .9 is considered good for non-metric scaling.

In practice, there are other algorithms other than eigendecomposition that can be used to find MDS solutions. This can be viewed as a tunable parameter as well.

In [3]:
```
from sklearn.manifold import MDS

MDS?
```

```
Init signature:
MDS(
    n_components=2,
    *,
    metric=True,
    n_init=4,
    max_iter=300,
    verbose=0,
    eps=0.001,
    n_jobs=None,
    random_state=None,
    dissimilarity='euclidean',
    normalized_stress='auto',
)
Docstring:
Multidimensional scaling.

Read more in the :ref:`User Guide <multidimensional_scaling>`.

Parameters
----------
n_components : int, default=2
    Number of dimensions in which to immerse the dissimilarities.

metric : bool, default=True
    If ``True``, perform metric MDS; otherwise, perform nonmetric MDS.
    When ``False`` (i.e. non-metric MDS), dissimilarities with 0 are considered as
    missing values.

n_init : int, default=4
    Number of times the SMACOF algorithm will be run with different
    initializations. The final results will be the best output of the runs,
    determined by the run with the smallest final stress.

max_iter : int, default=300
    Maximum number of iterations of the SMACOF algorithm for a single run.

verbose : int, default=0
    Level of verbosity.

eps : float, default=1e-3
    Relative tolerance with respect to stress at which to declare
    convergence. The value of `eps` should be tuned separately depending
    on whether or not `normalized_stress` is being used.

n_jobs : int, default=None
    The number of jobs to use for the computation. If multiple
    initializations are used (``n_init``), each run of the algorithm is
    computed in parallel.

    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.

random_state : int, RandomState instance or None, default=None
    Determines the random number generator used to initialize the centers.
```

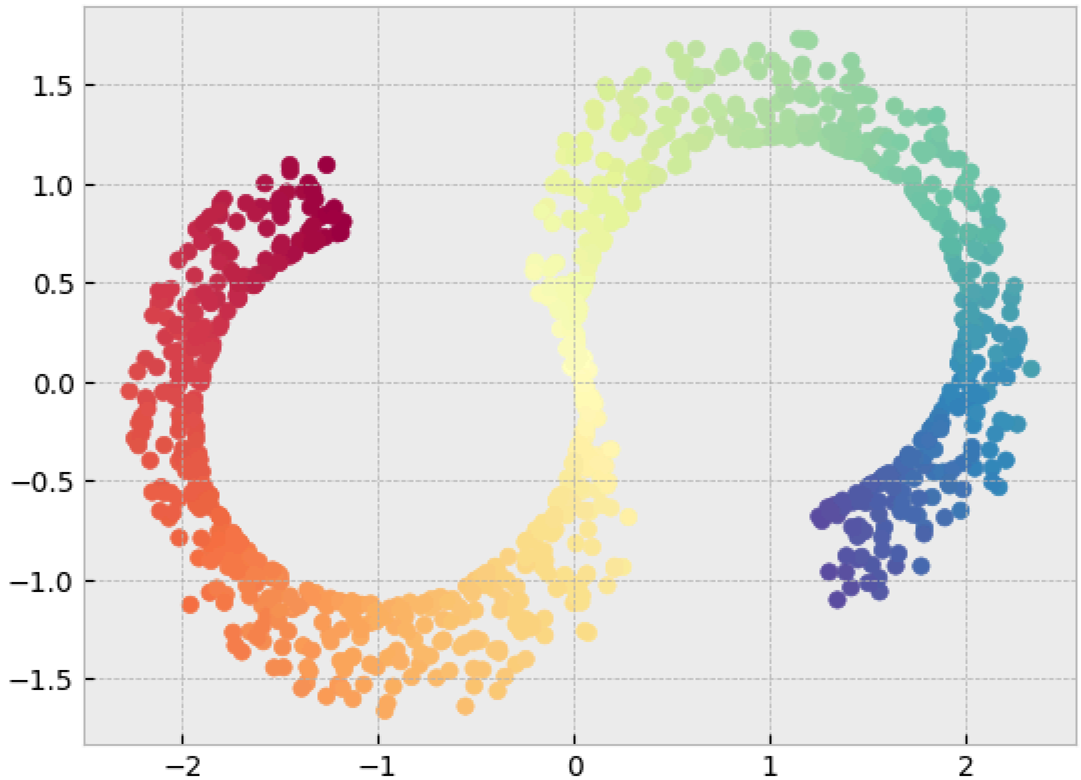
```
    Pass an int for reproducible results across multiple function calls.
    See :term:`Glossary <random_state>`.

dissimilarity : {'euclidean', 'precomputed'}, default='euclidean'
    Dissimilarity measure to use:

    - 'euclidean':
        Pairwise Euclidean distances between points in the dataset.

    - 'precomputed':
        Pre-computed dissimilarities are passed directly to ``fit`` and
        ``fit_transform``.

normalized_stress : bool or "auto" default="auto"
    Whether use and return normed stress value (Stress-1) instead of raw
    stress calculated by default. Only supported in non-metric MDS.

    .. versionadded:: 1.2

    .. versionchanged:: 1.4
       The default value changed from `False` to `"auto"` in version 1.4.

Attributes
----------
embedding_ : ndarray of shape (n_samples, n_components)
    Stores the position of the dataset in the embedding space.

stress_ : float
    The final value of the stress (sum of squared distance of the
    disparities and the distances for all constrained points).
    If `normalized_stress=True`, and `metric=False` returns Stress-1.
    A value of 0 indicates "perfect" fit, 0.025 excellent, 0.05 good,
    0.1 fair, and 0.2 poor [1]_.

dissimilarity_matrix_ : ndarray of shape (n_samples, n_samples)
    Pairwise dissimilarities between the points. Symmetric matrix that:

    - either uses a custom dissimilarity matrix by setting `dissimilarity`
      to 'precomputed';
    - or constructs a dissimilarity matrix from data using
      Euclidean distances.

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

n_iter_ : int
    The number of iterations corresponding to the best stress.
```

```
See Also
--------
sklearn.decomposition.PCA : Principal component analysis that is a linear
    dimensionality reduction method.
sklearn.decomposition.KernelPCA : Non-linear dimensionality reduction using
    kernels and PCA.
TSNE : T-distributed Stochastic Neighbor Embedding.
Isomap : Manifold learning based on Isometric Mapping.
LocallyLinearEmbedding : Manifold learning using Locally Linear Embedding.
SpectralEmbedding : Spectral embedding for non-linear dimensionality.

References
----------
.. [1] "Nonmetric multidimensional scaling: a numerical method" Kruskal, J.
   Psychometrika, 29 (1964)

.. [2] "Multidimensional scaling by optimizing goodness of fit to a nonmetric
   hypothesis" Kruskal, J. Psychometrika, 29, (1964)

.. [3] "Modern Multidimensional Scaling - Theory and Applications" Borg, I.;
   Groenen P. Springer Series in Statistics (1997)

Examples
--------
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import MDS
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = MDS(n_components=2, normalized_stress='auto')
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)

For a more detailed example of usage, see:
:ref:`sphx_glr_auto_examples_manifold_plot_mds.py`
File:           c:\users\hp\appdata\local\programs\python\python312\lib\site-package
s\sklearn\manifold\_mds.py
Type:           type
Subclasses:
```

In [4]:
```
mds = MDS(n_components=2, dissimilarity='euclidean')

Y = mds.fit_transform(X_scurve)

Y.shape
```

Out[4]: (1000, 2)

In [5]:
```
plt.scatter(Y[:,0], Y[:,1], c=color, cmap=plt.cm.Spectral);
```



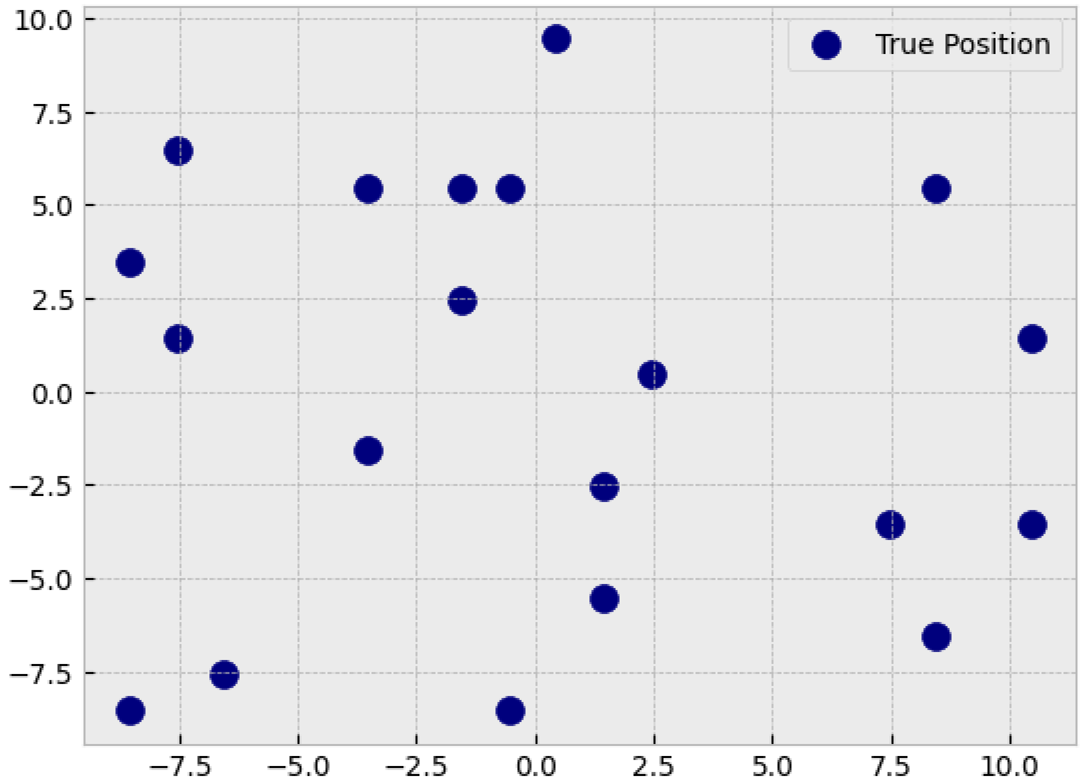
Application Example

The data produced by laser range scanning systems is typically a rectangular grid of distances from the sensor to the object being scanned. If the sensor and object are fixed, only objects that are "point viewable" can be fully digitized. More sophisticated systems are capable of digitizing objects by rotating either the sensor or the object.

- However, the scanning of topologically more complex objects (those with "holes") cannot be accomplished by either of these methods. To adequately scan these objects, multiple view points must be used. Merging the data generated from multiple view points to reconstruct a polyhedral surface representation is a non-trivial task.

In [6]:

```
n_samples = 20
seed = np.random.RandomState(seed=3)
X_true = seed.randint(0, 20, 2 * n_samples).astype(float)
X_true = X_true.reshape((n_samples, 2))
# Center the data
X_true -= X_true.mean()

s = 100
plt.scatter(X_true[:, 0], X_true[:, 1], color='navy', s=s, label='True Position')
plt.legend();
```



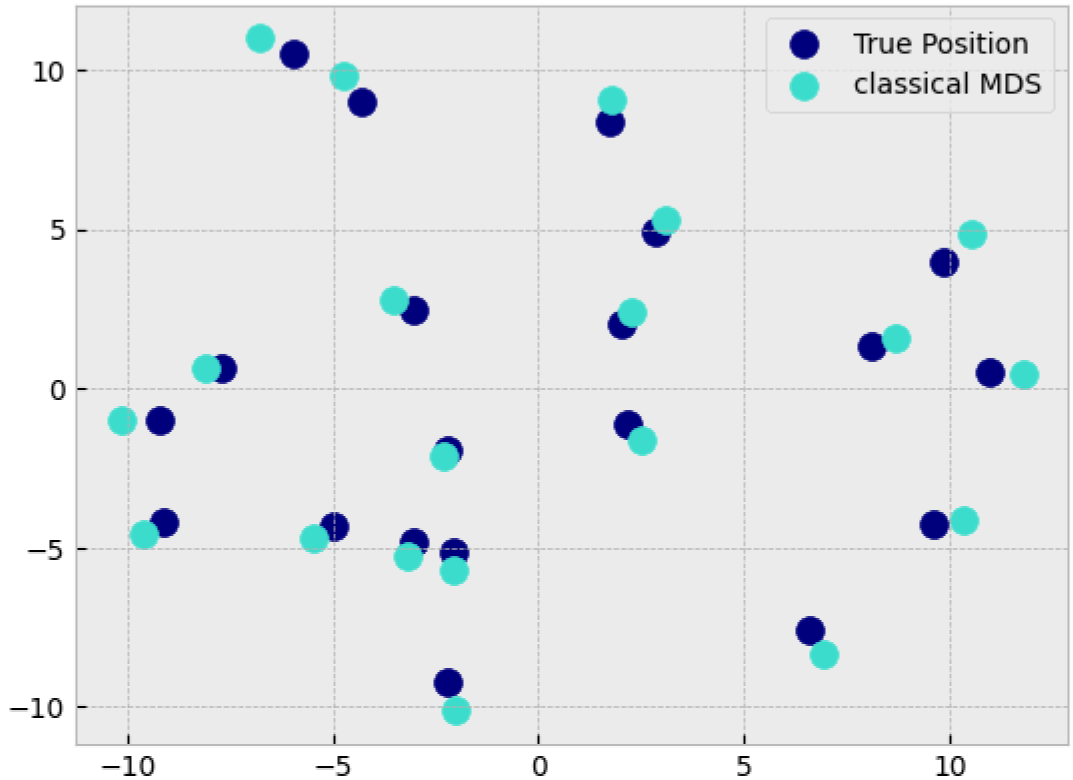
In [7]:
```
from scipy.spatial.distance import squareform
from scipy.spatial.distance import pdist

# Distances between pairs of points
distances = squareform(pdist(X_true, metric='euclidean'))

# Corrupting distance by adding additive Uniform(0,1) noise
noise = np.random.rand(n_samples, n_samples)
noise = noise + noise.T
# ensuring no noise is added to the diagonal of the distance matrix
noise[np.arange(noise.shape[0]), np.arange(noise.shape[0])] = 0
distances += noise

distances.shape
```

Out[7]: (20, 20)

In [8]:
```
mds = MDS(n_components=2,
          dissimilarity='precomputed',
          random_state=seed,
         normalized_stress='auto')

Y_mds = mds.fit(distances).embedding_
```

In [9]:
```
# Rotate the data
pca = PCA(n_components=2)
X_true = pca.fit_transform(X_true)

Y_mds = pca.fit_transform(Y_mds)
```

In [10]:
```
plt.scatter(X_true[:, 0], X_true[:, 1], color='navy', s=s, label='True Position')
plt.scatter(Y_mds[:, 0], Y_mds[:, 1], color='turquoise', s=s, label='classical MDS'
plt.legend(scatterpoints=1, loc="best", shadow=False);
```



In [11]:
```
from sklearn.metrics import r2_score

r2_score(X_true, Y_mds)
```

Out[11]:
```
0.9923934390812481
```

Isometric feature Mapping (ISOMAP)

In **Isometric feature Mapping**, or **ISOMAP**, the goal is to project the data to a lower-dimensional space using MDS, but where the distance/dissimilarities are defined in terms of the **geodesic distances** measured along the manifold.

ISOMAP was introduced in 2000 in a Science issue paper and approaches dimensionality reduction for data that lies on a manifold.

- It builds on MDS by using a specific distance: geodesic distances.
- The contribution of ISOMAP is that the proximity matrix, D, is constructed using **geodesic distances**.

Steps for Implementing MDS

1. **Construct neighborhood graph:** Identify the neighbors of each point on the manifold (e.g., K nearest neighbors, ϵ-ball approach). The edges between neighbors are weighted using their distance in the input space.

2. **Compute shortest paths:** Compute the distance between all pairs of points on the manifold by computing their shortest path distance on the graph created in step 1 (e.g. Floyd-Warshall's algorithm). This will produce the distance matrix D_G.

Floys-warshall's algorithm is an algorithm that finds the shortest distance between every pair of nodes in a graph.

Not just one pair. All pairs.

It gives you a distance table telling:

shortest A → B

shortest A → C

shortest B → D

every possible pair

3. **Construct M-dimensional embedding:** Apply MDS to the matrix of graph distances computed in step 2.

Floyd-Warshall Algorithm

The Foyd-Warshall alorithm to compute graph-based distance:

1. Initialize $d_G(i,j) = d_X(i,j)$ if i, j are identified as neighbors. Otherwise, set $d_G(i,j) = \infty$.
    - This means that, if data points are considered neighbors of a particular point, then Euclidean distance between that point and its neighbors is sufficient and assumed to be close enough to the geodesic distance. (Note that this is true for a reasonably small number of neighbors.)
2. For each value $k = 1, 2, \ldots, N$, replace all entries of $d_G(i,j)$ by $\min\left(d_G(i,j), d_G(i,k) + d_G(k,j)\right)$.

The matrix of final values $D_G = \{d_G(i,j)\}$ will contain the **shortest path distances** between all pairs of points in G.

Shortcomings of ISOMAP

If the data matrix $X \in \mathrm{R}^{N \times D}$ is sufficiently dense, then graph shortest path distance will approximate closely the original geodesic distance.

- ISOMAP may suffer from *non-convexity* such as holes on manifols.
- We need to compute pairwise shortest distance path between **all** sample pairs (i, j). This matrix is a global matrix, non-sparse and it requires cubic complexity $O(N^3)$.

In [12]:
```
from IPython.display import Image
Image('figures/isomap.png', width=900)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
File ~\AppData\Roaming\Python\Python312\site-packages\IPython\core\display.py:1045,
in Image._data_and_metadata(self, always_both)
   1044 try:
-> 1045     b64_data = b2a_base64(self.data, newline=False).decode("ascii")
   1046 except TypeError as e:

TypeError: a bytes-like object is required, not 'str'

The above exception was the direct cause of the following exception:

FileNotFoundError                         Traceback (most recent call last)
File ~\AppData\Roaming\Python\Python312\site-packages\IPython\core\formatters.py:97
7, in MimeBundleFormatter.__call__(self, obj, include, exclude)
    974     method = get_real_method(obj, self.print_method)
    976     if method is not None:
--> 977         return method(include=include, exclude=exclude)
    978     return None
    979 else:

File ~\AppData\Roaming\Python\Python312\site-packages\IPython\core\display.py:1035,
in Image._repr_mimebundle_(self, include, exclude)
   1033 if self.embed:
   1034     mimetype = self._mimetype
-> 1035     data, metadata = self._data_and_metadata(always_both=True)
   1036     if metadata:
   1037         metadata = {mimetype: metadata}

File ~\AppData\Roaming\Python\Python312\site-packages\IPython\core\display.py:1047,
in Image._data_and_metadata(self, always_both)
   1045     b64_data = b2a_base64(self.data, newline=False).decode("ascii")
   1046 except TypeError as e:
-> 1047     raise FileNotFoundError(
   1048         "No such file or directory: '%s'" % (self.data)) from e
   1049 md = {}
   1050 if self.metadata:

FileNotFoundError: No such file or directory: 'figures/isomap.png'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
File ~\AppData\Roaming\Python\Python312\site-packages\IPython\core\display.py:1045,
in Image._data_and_metadata(self, always_both)
   1044 try:
-> 1045     b64_data = b2a_base64(self.data, newline=False).decode("ascii")
   1046 except TypeError as e:

TypeError: a bytes-like object is required, not 'str'

The above exception was the direct cause of the following exception:

FileNotFoundError                         Traceback (most recent call last)
File ~\AppData\Roaming\Python\Python312\site-packages\IPython\core\formatters.py:34
7, in BaseFormatter.__call__(self, obj)
    345     method = get_real_method(obj, self.print_method)
    346     if method is not None:
--> 347         return method()
    348     return None
    349 else:

File ~\AppData\Roaming\Python\Python312\site-packages\IPython\core\display.py:1067,
in Image._repr_png_(self)
   1065 def _repr_png_(self):
   1066     if self.embed and self.format == self._FMT_PNG:
-> 1067         return self._data_and_metadata()

File ~\AppData\Roaming\Python\Python312\site-packages\IPython\core\display.py:1047,
in Image._data_and_metadata(self, always_both)
   1045     b64_data = b2a_base64(self.data, newline=False).decode("ascii")
   1046 except TypeError as e:
-> 1047     raise FileNotFoundError(
   1048         "No such file or directory: '%s'" % (self.data)) from e
   1049 md = {}
   1050 if self.metadata:

FileNotFoundError: No such file or directory: 'figures/isomap.png'
```

Out[12]: <IPython.core.display.Image object>

In [13]:
```python
from sklearn.manifold import Isomap

Isomap?
```

```
Init signature:
Isomap(
    *,
    n_neighbors=5,
    radius=None,
    n_components=2,
    eigen_solver='auto',
    tol=0,
    max_iter=None,
    path_method='auto',
    neighbors_algorithm='auto',
    n_jobs=None,
    metric='minkowski',
    p=2,
    metric_params=None,
)
Docstring:
Isomap Embedding.

Non-linear dimensionality reduction through Isometric Mapping

Read more in the :ref:`User Guide <isomap>`.

Parameters
----------
n_neighbors : int or None, default=5
    Number of neighbors to consider for each point. If `n_neighbors` is an int,
    then `radius` must be `None`.

radius : float or None, default=None
    Limiting distance of neighbors to return. If `radius` is a float,
    then `n_neighbors` must be set to `None`.

    .. versionadded:: 1.1

n_components : int, default=2
    Number of coordinates for the manifold.

eigen_solver : {'auto', 'arpack', 'dense'}, default='auto'
    'auto' : Attempt to choose the most efficient solver
    for the given problem.

    'arpack' : Use Arnoldi decomposition to find the eigenvalues
    and eigenvectors.

    'dense' : Use a direct solver (i.e. LAPACK)
    for the eigenvalue decomposition.

tol : float, default=0
    Convergence tolerance passed to arpack or lobpcg.
    not used if eigen_solver == 'dense'.

max_iter : int, default=None
    Maximum number of iterations for the arpack solver.
    not used if eigen_solver == 'dense'.
```

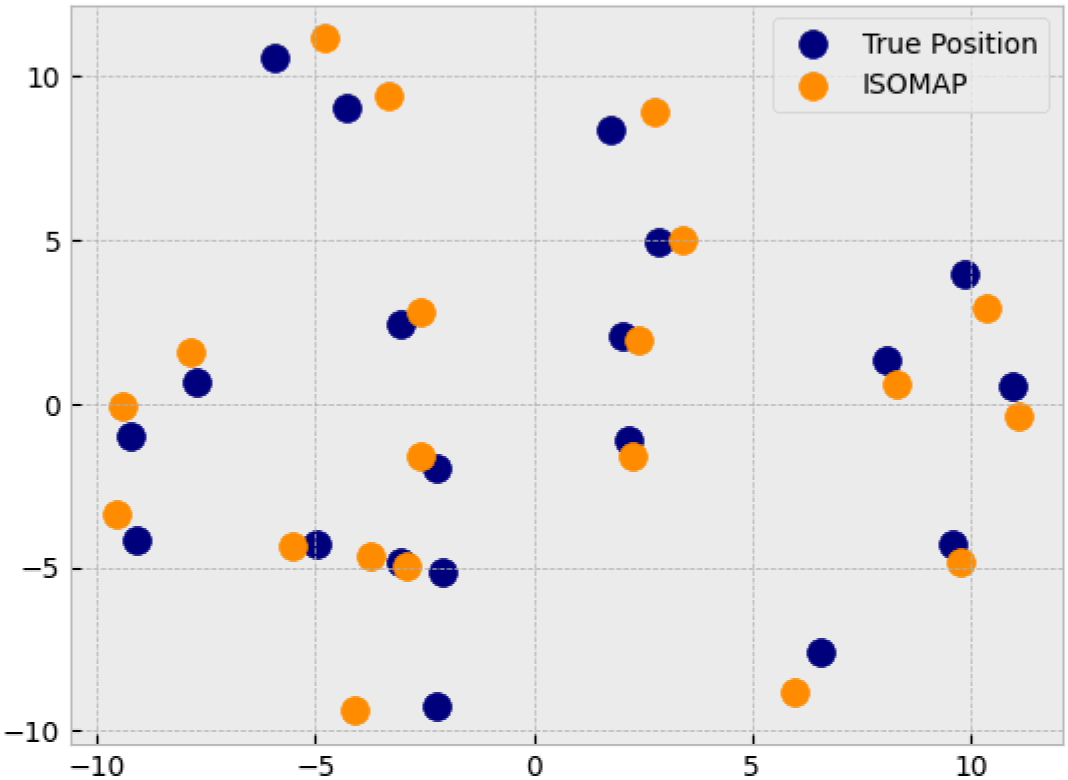
```
path_method : {'auto', 'FW', 'D'}, default='auto'
    Method to use in finding shortest path.

    'auto' : attempt to choose the best algorithm automatically.

    'FW' : Floyd-Warshall algorithm.

    'D' : Dijkstra's algorithm.

neighbors_algorithm : {'auto', 'brute', 'kd_tree', 'ball_tree'},
default='auto'
    Algorithm to use for nearest neighbors search,
    passed to neighbors.NearestNeighbors instance.

n_jobs : int or None, default=None
    The number of parallel jobs to run.
    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.

metric : str, or callable, default="minkowski"
    The metric to use when calculating distance between instances in a
    feature array. If metric is a string or callable, it must be one of
    the options allowed by :func:`sklearn.metrics.pairwise_distances` for
    its metric parameter.
    If metric is "precomputed", X is assumed to be a distance matrix and
    must be square. X may be a :term:`Glossary <sparse graph>`.

    .. versionadded:: 0.22

p : float, default=2
    Parameter for the Minkowski metric from
    sklearn.metrics.pairwise.pairwise_distances. When p = 1, this is
    equivalent to using manhattan_distance (l1), and euclidean_distance
    (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

    .. versionadded:: 0.22

metric_params : dict, default=None
    Additional keyword arguments for the metric function.

    .. versionadded:: 0.22

Attributes
----------
embedding_ : array-like, shape (n_samples, n_components)
    Stores the embedding vectors.

kernel_pca_ : object
    :class:`~sklearn.decomposition.KernelPCA` object used to implement the
    embedding.

nbrs_ : sklearn.neighbors.NearestNeighbors instance
    Stores nearest neighbors instance, including BallTree or KDtree
    if applicable.
```

```
dist_matrix_ : array-like, shape (n_samples, n_samples)
    Stores the geodesic distance matrix of training data.

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

See Also
--------
sklearn.decomposition.PCA : Principal component analysis that is a linear
    dimensionality reduction method.
sklearn.decomposition.KernelPCA : Non-linear dimensionality reduction using
    kernels and PCA.
MDS : Manifold learning using multidimensional scaling.
TSNE : T-distributed Stochastic Neighbor Embedding.
LocallyLinearEmbedding : Manifold learning using Locally Linear Embedding.
SpectralEmbedding : Spectral embedding for non-linear dimensionality.

References
----------

.. [1] Tenenbaum, J.B.; De Silva, V.; & Langford, J.C. A global geometric
       framework for nonlinear dimensionality reduction. Science 290 (5500)

Examples
--------
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import Isomap
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = Isomap(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
File:           c:\users\hp\appdata\local\programs\python\python312\lib\site-package
s\sklearn\manifold\_isomap.py
Type:           type
Subclasses:
```

In [14]:

```
isomap = Isomap(n_components=2, n_neighbors=6)
Y_isomap = isomap.fit_transform(X_true)

# Rotate the data
pca = PCA(n_components=2)
X_true = pca.fit_transform(X_true)

Y_isomap = pca.fit_transform(Y_isomap)
```

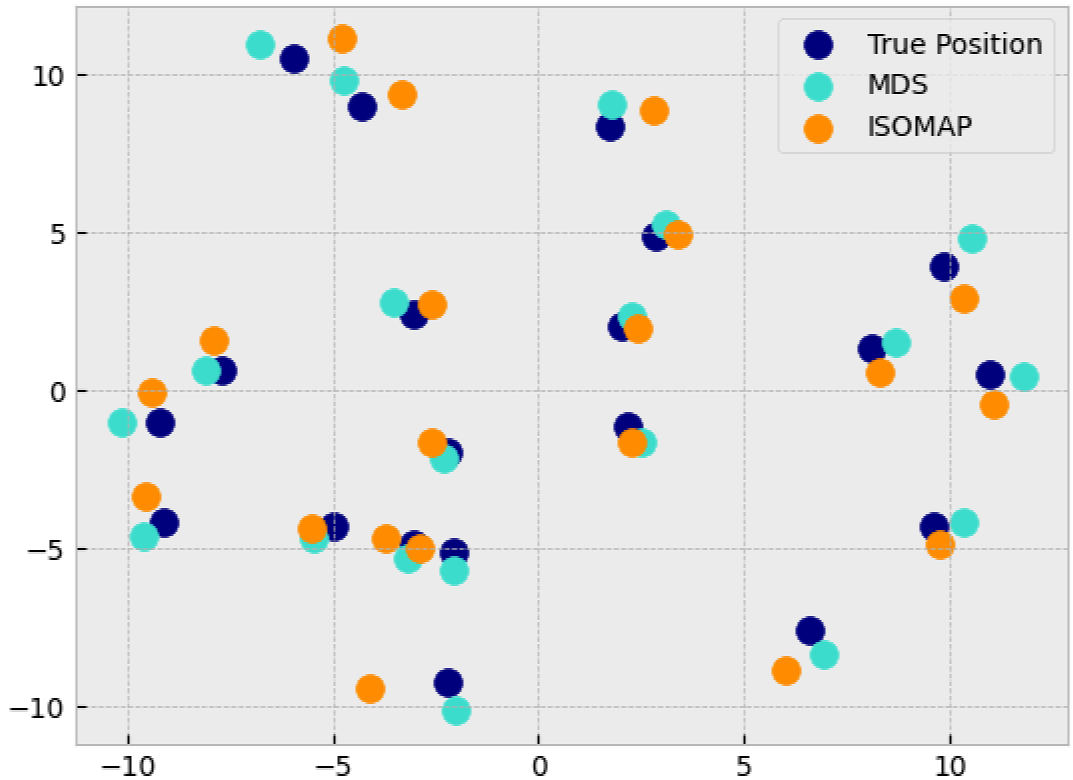
In [15]:
```
plt.scatter(X_true[:, 0], X_true[:, 1], color='navy', s=s, label='True Position')
plt.scatter(Y_isomap[:, 0], Y_isomap[:, 1], color='darkorange', s=s, label='ISOMAP'
plt.legend(scatterpoints=1, loc="best", shadow=False);
```



In [16]:
```
r2_score(X_true, Y_isomap)
```

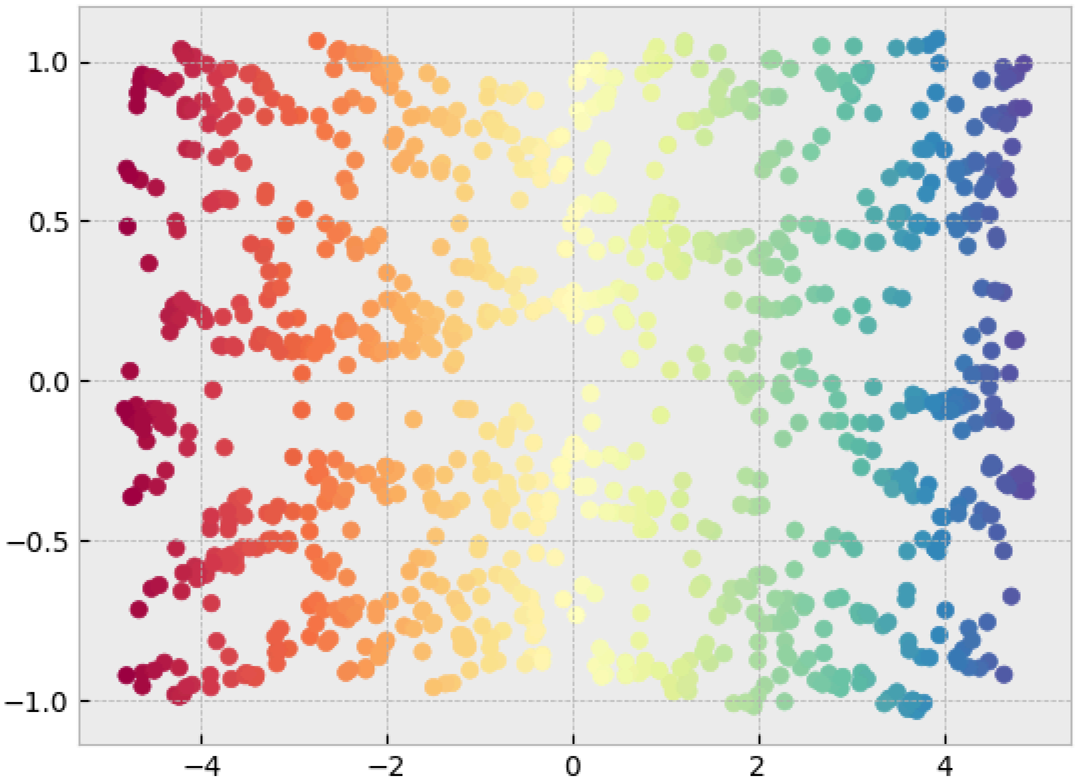
Out[16]: 0.9863031587621671

In [17]:
```
plt.scatter(X_true[:, 0], X_true[:, 1], color='navy', s=s, label='True Position')
plt.scatter(Y_mds[:, 0], Y_mds[:, 1], color='turquoise', s=s, label='MDS')
plt.scatter(Y_isomap[:, 0], Y_isomap[:, 1], color='darkorange', s=s, label='ISOMAP'
plt.legend(scatterpoints=1, loc="best", shadow=False);
```



In [18]:

```
# S-curve manifold learning with ISOMAP

isomap = Isomap(n_components=2, n_neighbors=10)

Y = isomap.fit_transform(X_scurve)

plt.scatter(Y[:,0], Y[:,1], c=color, cmap=plt.cm.Spectral);
```



Locally Linear Embedding (LLE)

Locally linear embedding, or **LLE** first computes the set of coefficients that best reconstructs each data point from its neighbors. These coefficients are arranged to be invariant to rotations, translations, and scalings of that data point and its neighbors, and hence they characterize the local geometrical properties of the neighborhood.

- LLE was also introduced in 2000 in the same Science issue as ISOMAP.
- The intuition behind LLE is that each data point and its close neighbors lie on or close to a *locally linear* patch of the manifold.
  - We can characterize the local geometry of these patches by linear coefficients that reconstruct each data point from its neighbors.
- Each point can be written as a linear combination of its neighbors.
- The lower-dimensional projection will be a combination of locally linear patches.

Informally, imagine taking a pair of scissors, cutting out locally linear patches of the underlying manifold, and placing them in the low dimensional embedding space. Assume further that this operation is done in a way that preserves the angles formed by each data point to its nearest neighbors. In this case, the transplantation of each patch involves no more than a translation, rotation, and rescaling of its data, exactly the operations to which the weights are invariant. Thus, when the patch arrives at its low dimensional destination, we expect the same weights to reconstruct each data point from its neighbors.

Steps to Implement LLE

1. **Construct neighborhood graph:** Identify the neighbors of each point on the manifold (e.g., K nearest neighbors, ϵ-ball approach). The edges between neighbors are weighted using their distance in the input space.

2. **Find a set of weights $W \in \mathbb{R}^{D \times K}$ such that each point $x_i \in \mathbb{R}^{D \times 1}$ can be reconstructed by its K neighbors.**

$$x_i = \sum_{j=1}^{K} w_{ij} x_{i(j)}$$

where $x_{i(j)}$ is the j-th neighbor of x_i.

The reconstruction errors are measured by the cost function:

$$\epsilon(W) = \sum_{i=1}^{N} \| x_i - \sum_{j=1}^{K} w_{ij} x_{i(j)} \|_2^2$$

which adds up the squared distances between all the data points and their reconstructions. The weights w_{ij} summarize the contribution of the j-th data point to the i-th reconstruction. To compute the weights w_{ij}, we minimize the cost function $\epsilon(W)$ subject to two constraints:

Contraint 1: Each data point x_i is reconstructed only from its neighbors, enforcing $w_{ij} = 0$ if x_j is not a neighbor of x_i.

Constraint 2: The rows of the weight matrix sum to one: $\sum_{j=1}^{K} w_{ij} = 1$. The optimal weights subject to these constraints are found by solving a least-squares problem.

3. **Each high-dimensional observation $x_i \in \mathbb{R}^{D \times 1}$ is mapped to a low-dimensional vector $x_i \in \mathbb{R}^{M \times 1}$ representing global internal coordinates on the manifold.** This is done by choosing M-dimensional coordinates y_i to minimize the embedding cost function:

$$\Phi(Y) = \sum_{i=1}^{N} \| y_i - \sum_{j=1}^{K} w_{ij} y_j \|_2^2$$

with fixed weights W. To compute the projection Y, we minimize the cost function $\Phi(Y)$ subject to two constraints:

Contraint 1: The coordinates are centered on the origin: $\sum_{i=1}^{N} y_i = 0$

Contraint 2: Also, to avoid degenerate solutions, we constrain the embedding vectors to have unit covariance, with outer products that satisfy $\frac{1}{N}YY^T = I$

- More details on the mathematical derivations can be found in the original published work (here).

In [19]:
```
from sklearn.manifold import LocallyLinearEmbedding as LLE

LLE?
```

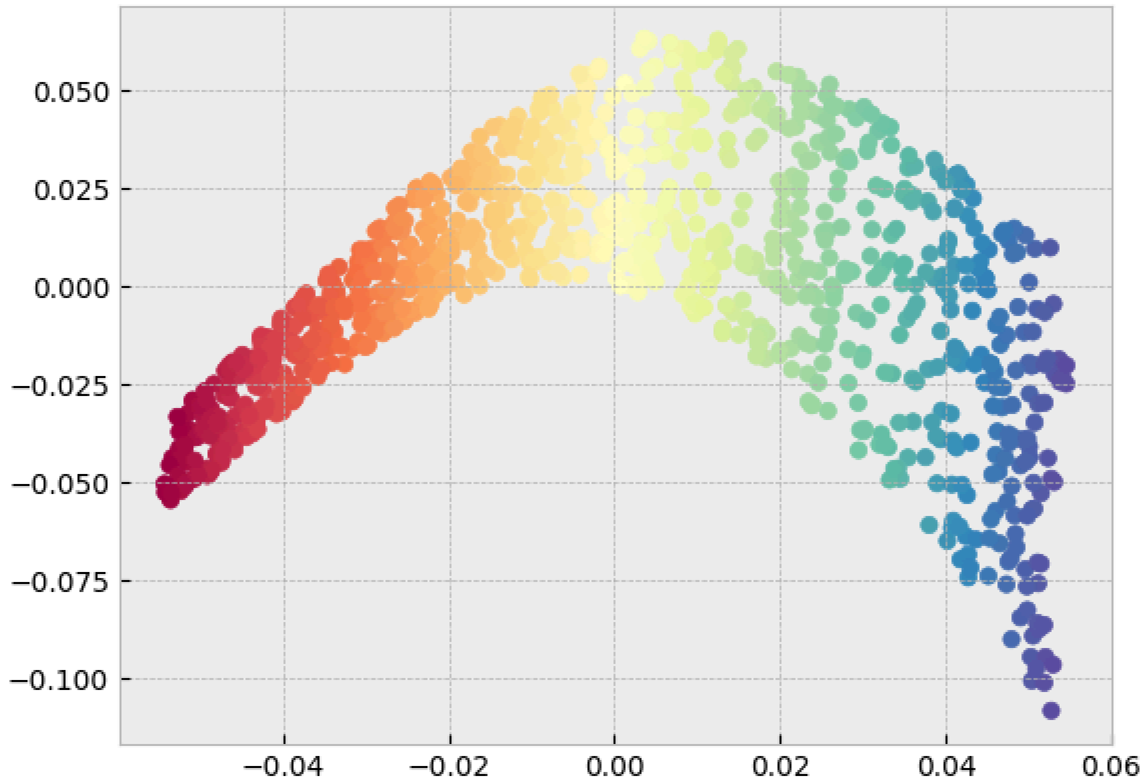
```
Init signature:
LLE(
    *,
    n_neighbors=5,
    n_components=2,
    reg=0.001,
    eigen_solver='auto',
    tol=1e-06,
    max_iter=100,
    method='standard',
    hessian_tol=0.0001,
    modified_tol=1e-12,
    neighbors_algorithm='auto',
    random_state=None,
    n_jobs=None,
)
Docstring:
Locally Linear Embedding.

Read more in the :ref:`User Guide <locally_linear_embedding>`.

Parameters
----------
n_neighbors : int, default=5
    Number of neighbors to consider for each point.

n_components : int, default=2
    Number of coordinates for the manifold.

reg : float, default=1e-3
    Regularization constant, multiplies the trace of the local covariance
    matrix of the distances.

eigen_solver : {'auto', 'arpack', 'dense'}, default='auto'
    The solver used to compute the eigenvectors. The available options are:

    - `'auto'` : algorithm will attempt to choose the best method for input
      data.
    - `'arpack'` : use arnoldi iteration in shift-invert mode. For this
      method, M may be a dense matrix, sparse matrix, or general linear
      operator.
    - `'dense'`  : use standard dense matrix operations for the eigenvalue
      decomposition. For this method, M must be an array or matrix type.
      This method should be avoided for large problems.

    .. warning::
       ARPACK can be unstable for some problems.  It is best to try several
       random seeds in order to check results.

tol : float, default=1e-6
    Tolerance for 'arpack' method
    Not used if eigen_solver=='dense'.

max_iter : int, default=100
    Maximum number of iterations for the arpack solver.
    Not used if eigen_solver=='dense'.
```

```
method : {'standard', 'hessian', 'modified', 'ltsa'}, default='standard'
    - `standard`: use the standard locally linear embedding algorithm. see
      reference [1]_
    - `hessian`: use the Hessian eigenmap method. This method requires
      ``n_neighbors > n_components * (1 + (n_components + 1) / 2``. see
      reference [2]_
    - `modified`: use the modified locally linear embedding algorithm.
      see reference [3]_
    - `ltsa`: use local tangent space alignment algorithm. see
      reference [4]_

hessian_tol : float, default=1e-4
    Tolerance for Hessian eigenmapping method.
    Only used if ``method == 'hessian'``.

modified_tol : float, default=1e-12
    Tolerance for modified LLE method.
    Only used if ``method == 'modified'``.

neighbors_algorithm : {'auto', 'brute', 'kd_tree', 'ball_tree'},
default='auto'
    Algorithm to use for nearest neighbors search, passed to
    :class:`~sklearn.neighbors.NearestNeighbors` instance.

random_state : int, RandomState instance, default=None
    Determines the random number generator when
    ``eigen_solver`` == 'arpack'. Pass an int for reproducible results
    across multiple function calls. See :term:`Glossary <random_state>`.

n_jobs : int or None, default=None
    The number of parallel jobs to run.
    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.

Attributes
----------
embedding_ : array-like, shape [n_samples, n_components]
    Stores the embedding vectors

reconstruction_error_ : float
    Reconstruction error associated with `embedding_`

n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

nbrs_ : NearestNeighbors object
```

```
    Stores nearest neighbors instance, including BallTree or KDtree
    if applicable.

See Also
--------
SpectralEmbedding : Spectral embedding for non-linear dimensionality
    reduction.
TSNE : Distributed Stochastic Neighbor Embedding.

References
----------

.. [1] Roweis, S. & Saul, L. Nonlinear dimensionality reduction
    by locally linear embedding.  Science 290:2323 (2000).
.. [2] Donoho, D. & Grimes, C. Hessian eigenmaps: Locally
    linear embedding techniques for high-dimensional data.
    Proc Natl Acad Sci U S A.  100:5591 (2003).
.. [3] `Zhang, Z. & Wang, J. MLLE: Modified Locally Linear
    Embedding Using Multiple Weights.
    <https://citeseerx.ist.psu.edu/doc_view/pid/0b060fdbd92cbcc66b383bcaa9ba5e5e624d
7ee3>`_
.. [4] Zhang, Z. & Zha, H. Principal manifolds and nonlinear
    dimensionality reduction via tangent space alignment.
    Journal of Shanghai Univ.  8:406 (2004)

Examples
--------
>>> from sklearn.datasets import load_digits
>>> from sklearn.manifold import LocallyLinearEmbedding
>>> X, _ = load_digits(return_X_y=True)
>>> X.shape
(1797, 64)
>>> embedding = LocallyLinearEmbedding(n_components=2)
>>> X_transformed = embedding.fit_transform(X[:100])
>>> X_transformed.shape
(100, 2)
File:           c:\users\hp\appdata\local\programs\python\python312\lib\site-package
s\sklearn\manifold\_locally_linear.py
Type:           type
Subclasses:
```

In [20]:

```python
# S-curve manifold learning with ISOMAP
# K nearest neighbors approachfor defining neighborhood
lle = LLE(n_components=2, n_neighbors=10)

Y = lle.fit_transform(X_scurve)

plt.scatter(Y[:,0], Y[:,1], c=color, cmap=plt.cm.Spectral);
```



t-Distributed Stochastic Neighbor Embedding (t-SNE) - tee-snee

Stochastic Neighbor Embedding (SNE) starts by converting the high-dimensional Euclidean distances between data points into conditional probabilities that represent similarities. The similarity of data point x_j to data point x_i is the conditional probability, $p_{j|i}$, that x_i would pick x_j as its neighbor if neighbors were picked in proportion to their probability density under a Gaussian centered at x_i.

For nearby data points, $p_{j|i}$ is relatively high, whereas for widely separated data points, $p_{j|i}$ will be almost infinitesimal (for reasonable values of the variance of the Gaussian, σ_i). Mathematically, the conditional probability $p_{j|i}$ is given by

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{k \neq i}\exp(-\|x_i - x_k\|^2/2\sigma_i^2)}$$

where σ_i is the variance of the Gaussian that is centered on data point x_i. Because we are only interested in modeling pairwise similarities, we set the value of $p_{i|i}$ to zero. For the low-dimensional counterparts y_i and y_j of the high-dimensional data points x_i and x_j, it is possible to compute a similar conditional probability, which we denote by $q_{j|i}$.

We set the variance of the Gaussian that is employed in the computation of the conditional probabilities $q_{j|i}$ to $\frac{1}{\sqrt{2}}$. (Note that setting the variance in the low-dimensional Gaussians to another value only results in a rescaled version of the final.) Hence, we model the similarity of map point y_j to map point y_i:

$$q_{j|i} = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_{k \neq i} \exp(-\|y_i - y_k\|^2)}$$

Again, since we are only interested in modeling pairwise similarities, we set $q_{i|i} = 0$.

```
In [ ]:
```

If the map points y_i and y_j correctly model the similarity between the high-dimensional data points x_i and x_j, the conditional probabilities $p_{j|i}$ and $q_{j|i}$ will be equal. Motivated by this observation, SNE aims to find a low-dimensional data representation that minimizes the mismatch between $p_{j|i}$ and $q_{j|i}$.

A natural measure of the faithfulness with which $q_{j|i}$ models $p_{j|i}$ is the **Kullback-Leibler (KL) divergence** (which is in this case equal to the cross-entropy up to an additive constant). **SNE minimizes the sum of Kullback-Leibler divergences over all data points** using a **gradient descent** method. The objective function is as follows:

$$J(y) = \sum_i \mathrm{KL}(P_i \parallel Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

in which P_i represents the conditional probability distribution over all other data points given data point x_i, and Q_i represents the conditional probability distribution over all other map points given map point y_i. Because the Kullback-Leibler (KL) divergence is not symmetric, different types of error in the pairwise distances in the low-dimensional map are not weighted equally. In particular, there is a large cost for using widely separated map points to represent nearby data points (i.e., for using a small $q_{j|i}$ to model a large $p_{j|i}$), but there is only a small cost for using nearby map points to represent widely separated data points.

This small cost comes from wasting some of the probability mass in the relevant Q distributions. In other words, the SNE cost function focuses on retaining the local structure of the data in the map (for reasonable values of the variance of the Gaussian in the high-dimensional space, σ_i).

The remaining parameter to be selected is the variance σ_i of the Gaussian that is centered over each high-dimensional data point, x_i. It is not likely that there is a single value of σ_i that is optimal for all data points in the data set because the density of the data is likely to vary.

In dense regions, a smaller value of σ_i is usually more appropriate than in sparser regions. Any particular value of σ_i induces a probability distribution, P_i, over all of the other data points. This distribution has an entropy which increases as σ_i increases. SNE performs a binary search for the value of σ_i that produces a P_i with a **fixed perplexity** that is specified by the user.

The perplexity is defined as

$$\mathrm{Perp}(P_i) = 2^{H(P_i)}$$

where $H(P_i)$ is the **Shannon entropy** of P_i measured in bits

$$H(P_i) = -\sum_j p_{j|i} \log_2(p_{j|i})$$

The perplexity can be interpreted as a smooth measure of the effective number of neighbors. The performance of SNE is fairly robust to changes in the perplexity, and **typical values are between 5 and 50**.

The minimization of the objective function is performed using a gradient descent method. The gradient has a surprisingly simple form

$$\frac{\partial J}{\partial y_i} = 2\sum_j \left(p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j}\right)(y_i - y_j)$$

Here is **t-SNE explained in super easy, baby-level language** using the picture you gave.

🌈 t-SNE in Easy Language

Imagine you have **many high-dimensional data points** (each point has many features, like 100 or 500 numbers). We want to **draw them nicely in 2D** so we can visualize clusters.

But simply plotting them will not work because 100-D → 2-D loses information.

So t-SNE uses the idea:

> **"If two points are close in high-dimensional space, keep them close in the 2-D plot." If they are far, keep them far."**

⭐ Step-by-Step SUPER SIMPLE Explanation

1️⃣ Think of each point as a person looking for neighbors

For each point (x_i):

- Look at all other points (x_j)
- If (x_j) is **close**, probability is **high**
- If (x_j) is **far**, probability is **very small**

This probability is:

$$p_{j|i} = \frac{\exp\left(-|x_i - x_j|^2 / 2\sigma_i^2\right)}{\sum_{k \ne i} \exp\left(-|x_i - x_k|^2 / 2\sigma_i^2\right)}$$

💡 **This just means:** "Points near me should look very likely to be my neighbors. Far points should look unlikely."

2️⃣ σ_i (sigma) controls how many neighbors we consider

- A **big σ** = everyone looks close
- A **small σ** = only very close neighbors matter

(Perplexity controls this.)

3️⃣ In the 2D map (y_i and y_j), we do the same thing

We compute another probability:

$$q_{j|i} = \frac{\exp(-|y_i - y_j|^2)}{\sum_{k \ne i} \exp(-|y_i - y_k|^2)}$$

💡 This is the **2D neighborhood probability**.

4️⃣ Goal: Make p's and q's look the same

t-SNE tries to place the 2D points (y_i) so that:

- If two points were close in high-D, they stay close in 2-D
- If far in high-D, they remain far in 2-D

Mathematically:

> “Make ($q_{j|i}$) (2D similarity) look like ($p_{j|i}$) (high-D similarity).”

It does this by minimizing a loss function (KL divergence).

🌟 EASIEST SUMMARY (1–liner)

High-dimensional nearby points → keep them close. High-dimensional far points → push them far.

This creates beautiful clusters.

🌈 Super Simple Analogy

Imagine a class of students standing in a large field (100-D):

- Students close to each other = friends
- Students far apart = not friends

You want to make a **2D seating plan** (like a bench) where:

- Friends sit close
- Non-friends sit far

t-SNE looks at friendships (probabilities) and arranges the seats.

If you want, I can also explain:

✅ What perplexity is ✅ Why t-SNE creates clusters ✅ How it compares to PCA, LLE, Isomap — all in easy language.

🌟 KL Divergence in Super Easy Language

KL Divergence answers one question:

👉 “How different are two probability distributions?”

Or even simpler:

👉 “How surprised am I if I think the world works like A, but the real world actually works like B?"

If A and B are the same → KL = 0 (no surprise)

If A and B are different → KL is large (big surprise)

Here is the **easiest possible explanation** of the continuation you posted. No maths, no fancy terms — **just what it means in real life**.

🌈 1️⃣ What is the big idea here?

t-SNE wants the **2D map** to behave **exactly like** the high-dimensional world:

- If two points were close → keep them close in 2D
- If two points were far → keep them far in 2D

To check "how well" the 2D map is doing this matching, t-SNE uses **KL Divergence**.

⭐ 2️⃣ KL Divergence (super easy meaning)

KL Divergence = **"How different are the two probability distributions?"**

Here:

- ($p_{j|i}$) = probability that (x_j) is neighbor of (x_i) in high-D
- ($q_{j|i}$) = probability that (y_j) is neighbor of (y_i) in 2-D

If they match perfectly → KL = 0 If they don't match → KL gets bigger

So the goal:

🧠 Make q look like p as much as possible.

⭐ 3️⃣ The cost function J(y)

This is just:

> "Add up ALL KL divergences for ALL points"

$$J(y)=\sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

💡 **Meaning:** If two points were very close in high-D but you placed them far in 2-D → BIG penalty. If they were far in high-D but you placed them close → small penalty.

This is why t-SNE preserves **local structure** so well.

🌟 4️⃣ Why small errors are okay?

In the text:

> “There is a large cost for using widely separated map points to represent nearby high-dimensional points.”

Means: **If two points are actually close but you put them far → very bad!**

But:

> “There is only a small cost for using nearby map points to represent widely separated data points.”

Means: **If two points are far but you put them close → not too bad.**

That’s why t-SNE forms tight clusters — it avoids splitting close points.

🌈 5️⃣ What is σ_i and why do we tune it?

Each point has its own “spread” σ_i, deciding:

- **Small σ → only a few nearby neighbors matter**
- **Large σ → many neighbors matter**

Different areas of the dataset have different densities, so each point needs its own σ_i.

t-SNE finds the right σ_i using a **binary search** so that something called **perplexity** matches a fixed value (given by the user).

⭐ 6️⃣ What is Perplexity? (VERY EASY meaning)

Perplexity ≈

👉 “How many neighbors should each point care about?”

Examples:

- Perplexity 5 → each point tries to consider ~5 close neighbors
- Perplexity 30 → each point tries to consider ~30 neighbors

Common values: **5 to 50**

High perplexity → global structure Low perplexity → small, tight clusters

---

⭐ 7️⃣ Entropy ($H(P_i)$)

Don't worry about the formula.

Entropy $H(P_i)$ just measures:

> "How spread out are my neighbors?"

If entropy is high → neighbors are spread wide If entropy is low → neighbors are very close

Perplexity = (2^{H(P_i)})

You don't need to compute this; just understand:

✔️ entropy decides perplexity ✔️ perplexity tells model how many neighbors to consider

---

🌈 8️⃣ Gradient Descent (VERY simple meaning)

The last part gives the formula:

[\frac{\partial J}{\partial y_i}=2\sum_j(p_{j|i}-q_{j|i} + p_{i|j}-q_{i|j})(y_i-y_j)]

But easy meaning:

> "Move points around so the 2D similarities (q) start matching the high-D similarities (p)."

Think of it like:

- If two points should be close → **pull them together**
- If they are too close but should be far → **push them apart**

It keeps adjusting the 2D points until the map becomes "correct."

🦋 FINAL SUPER SIMPLE SUMMARY

Here is everything in one tiny paragraph:

t-SNE compares how close points are in high-D (p) vs 2-D (q). It uses KL divergence to measure this mismatch and adjusts the 2-D map so q becomes like p. Perplexity controls how many neighbors each point cares about. Gradient descent moves points to reduce mismatches, pulling close neighbors together and pushing far ones apart, creating a clean clustered visualization.

If you want, I can now also explain:

✅ How t-SNE differs from PCA, LLE, Isomap ✅ Why t-SNE often makes clusters ✅ What perplexity visually changes — all in baby-level language.

In [27]:
```
from sklearn.manifold import TSNE

TSNE?
```

```
Init signature:
TSNE(
    n_components=2,
    *,
    perplexity=30.0,
    early_exaggeration=12.0,
    learning_rate='auto',
    n_iter=1000,
    n_iter_without_progress=300,
    min_grad_norm=1e-07,
    metric='euclidean',
    metric_params=None,
    init='pca',
    verbose=0,
    random_state=None,
    method='barnes_hut',
    angle=0.5,
    n_jobs=None,
)
Docstring:
T-distributed Stochastic Neighbor Embedding.

t-SNE [1] is a tool to visualize high-dimensional data. It converts
similarities between data points to joint probabilities and tries
to minimize the Kullback-Leibler divergence between the joint
probabilities of the low-dimensional embedding and the
high-dimensional data. t-SNE has a cost function that is not convex,
i.e. with different initializations we can get different results.

It is highly recommended to use another dimensionality reduction
method (e.g. PCA for dense data or TruncatedSVD for sparse data)
to reduce the number of dimensions to a reasonable amount (e.g. 50)
if the number of features is very high. This will suppress some
noise and speed up the computation of pairwise distances between
samples. For more tips see Laurens van der Maaten's FAQ [2].

Read more in the :ref:`User Guide <t_sne>`.

Parameters
----------
n_components : int, default=2
    Dimension of the embedded space.

perplexity : float, default=30.0
    The perplexity is related to the number of nearest neighbors that
    is used in other manifold learning algorithms. Larger datasets
    usually require a larger perplexity. Consider selecting a value
    between 5 and 50. Different values can result in significantly
    different results. The perplexity must be less than the number
    of samples.

early_exaggeration : float, default=12.0
    Controls how tight natural clusters in the original space are in
    the embedded space and how much space will be between them. For
    larger values, the space between natural clusters will be larger
    in the embedded space. Again, the choice of this parameter is not
```

```
    very critical. If the cost function increases during initial
    optimization, the early exaggeration factor or the learning rate
    might be too high.

learning_rate : float or "auto", default="auto"
    The learning rate for t-SNE is usually in the range [10.0, 1000.0]. If
    the learning rate is too high, the data may look like a 'ball' with any
    point approximately equidistant from its nearest neighbours. If the
    learning rate is too low, most points may look compressed in a dense
    cloud with few outliers. If the cost function gets stuck in a bad local
    minimum increasing the learning rate may help.
    Note that many other t-SNE implementations (bhtsne, FIt-SNE, openTSNE,
    etc.) use a definition of learning_rate that is 4 times smaller than
    ours. So our learning_rate=200 corresponds to learning_rate=800 in
    those other implementations. The 'auto' option sets the learning_rate
    to `max(N / early_exaggeration / 4, 50)` where N is the sample size,
    following [4] and [5].

    .. versionchanged:: 1.2
       The default value changed to `"auto"`.

n_iter : int, default=1000
    Maximum number of iterations for the optimization. Should be at
    least 250.

n_iter_without_progress : int, default=300
    Maximum number of iterations without progress before we abort the
    optimization, used after 250 initial iterations with early
    exaggeration. Note that progress is only checked every 50 iterations so
    this value is rounded to the next multiple of 50.

    .. versionadded:: 0.17
       parameter *n_iter_without_progress* to control stopping criteria.

min_grad_norm : float, default=1e-7
    If the gradient norm is below this threshold, the optimization will
    be stopped.

metric : str or callable, default='euclidean'
    The metric to use when calculating distance between instances in a
    feature array. If metric is a string, it must be one of the options
    allowed by scipy.spatial.distance.pdist for its metric parameter, or
    a metric listed in pairwise.PAIRWISE_DISTANCE_FUNCTIONS.
    If metric is "precomputed", X is assumed to be a distance matrix.
    Alternatively, if metric is a callable function, it is called on each
    pair of instances (rows) and the resulting value recorded. The callable
    should take two arrays from X as input and return a value indicating
    the distance between them. The default is "euclidean" which is
    interpreted as squared euclidean distance.

metric_params : dict, default=None
    Additional keyword arguments for the metric function.

    .. versionadded:: 1.1

init : {"random", "pca"} or ndarray of shape (n_samples, n_components),
```

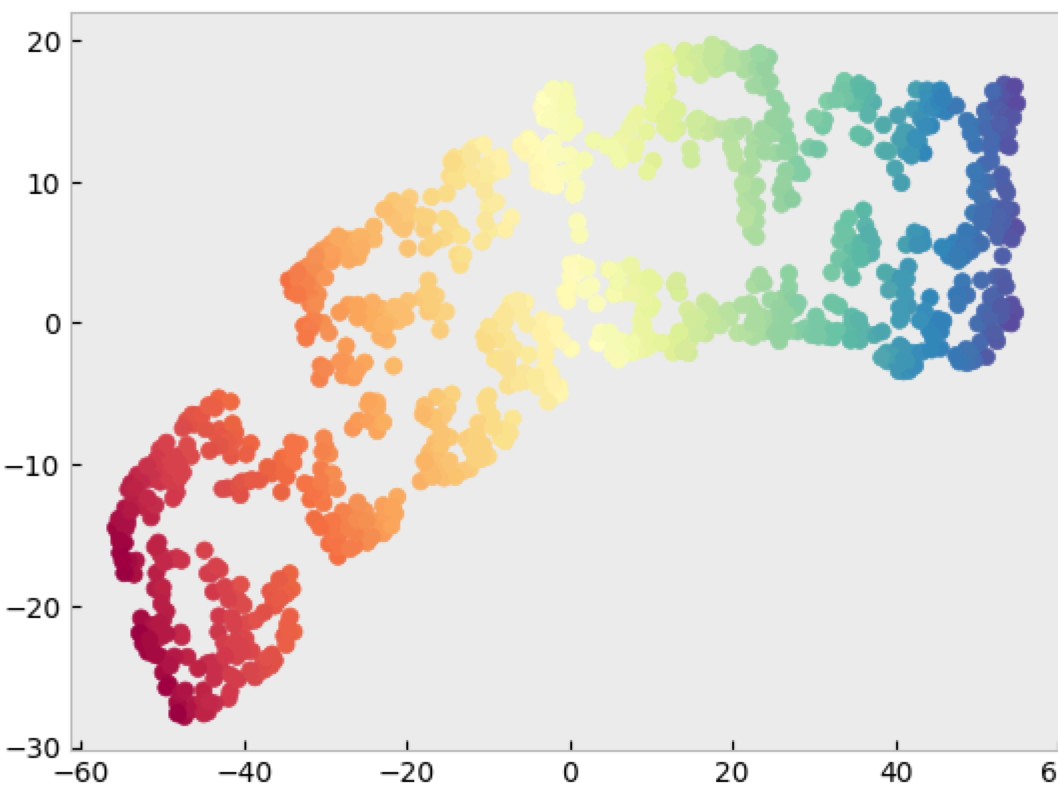
```
default="pca"
    Initialization of embedding.
    PCA initialization cannot be used with precomputed distances and is
    usually more globally stable than random initialization.

    .. versionchanged:: 1.2
       The default value changed to `"pca"`.

verbose : int, default=0
    Verbosity level.

random_state : int, RandomState instance or None, default=None
    Determines the random number generator. Pass an int for reproducible
    results across multiple function calls. Note that different
    initializations might result in different local minima of the cost
    function. See :term:`Glossary <random_state>`.

method : {'barnes_hut', 'exact'}, default='barnes_hut'
    By default the gradient calculation algorithm uses Barnes-Hut
    approximation running in O(NlogN) time. method='exact'
    will run on the slower, but exact, algorithm in O(N^2) time. The
    exact algorithm should be used when nearest-neighbor errors need
    to be better than 3%. However, the exact method cannot scale to
    millions of examples.

    .. versionadded:: 0.17
       Approximate optimization *method* via the Barnes-Hut.

angle : float, default=0.5
    Only used if method='barnes_hut'
    This is the trade-off between speed and accuracy for Barnes-Hut T-SNE.
    'angle' is the angular size (referred to as theta in [3]) of a distant
    node as measured from a point. If this size is below 'angle' then it is
    used as a summary node of all points contained within it.
    This method is not very sensitive to changes in this parameter
    in the range of 0.2 - 0.8. Angle less than 0.2 has quickly increasing
    computation time and angle greater 0.8 has quickly increasing error.

n_jobs : int, default=None
    The number of parallel jobs to run for neighbors search. This parameter
    has no impact when ``metric="precomputed"`` or
    (``metric="euclidean"`` and ``method="exact"``).
    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.

    .. versionadded:: 0.22

Attributes
----------
embedding_ : array-like of shape (n_samples, n_components)
    Stores the embedding vectors.

kl_divergence_ : float
    Kullback-Leibler divergence after optimization.
```

```
n_features_in_ : int
    Number of features seen during :term:`fit`.

    .. versionadded:: 0.24

feature_names_in_ : ndarray of shape (`n_features_in_`,)
    Names of features seen during :term:`fit`. Defined only when `X`
    has feature names that are all strings.

    .. versionadded:: 1.0

learning_rate_ : float
    Effective learning rate.

    .. versionadded:: 1.2

n_iter_ : int
    Number of iterations run.

See Also
--------
sklearn.decomposition.PCA : Principal component analysis that is a linear
    dimensionality reduction method.
sklearn.decomposition.KernelPCA : Non-linear dimensionality reduction using
    kernels and PCA.
MDS : Manifold learning using multidimensional scaling.
Isomap : Manifold learning based on Isometric Mapping.
LocallyLinearEmbedding : Manifold learning using Locally Linear Embedding.
SpectralEmbedding : Spectral embedding for non-linear dimensionality.

Notes
-----
For an example of using :class:`~sklearn.manifold.TSNE` in combination with
:class:`~sklearn.neighbors.KNeighborsTransformer` see
:ref:`sphx_glr_auto_examples_neighbors_approximate_nearest_neighbors.py`.

References
----------

[1] van der Maaten, L.J.P.; Hinton, G.E. Visualizing High-Dimensional Data
    Using t-SNE. Journal of Machine Learning Research 9:2579-2605, 2008.

[2] van der Maaten, L.J.P. t-Distributed Stochastic Neighbor Embedding
    https://lvdmaaten.github.io/tsne/

[3] L.J.P. van der Maaten. Accelerating t-SNE using Tree-Based Algorithms.
    Journal of Machine Learning Research 15(Oct):3221-3245, 2014.
    https://lvdmaaten.github.io/publications/papers/JMLR_2014.pdf

[4] Belkina, A. C., Ciccolella, C. O., Anno, R., Halpert, R., Spidlen, J.,
    & Snyder-Cappione, J. E. (2019). Automated optimized parameters for
    T-distributed stochastic neighbor embedding improve visualization
    and analysis of large datasets. Nature Communications, 10(1), 1-12.

[5] Kobak, D., & Berens, P. (2019). The art of using t-SNE for single-cell
    transcriptomics. Nature Communications, 10(1), 1-14.
```

```
Examples
--------
>>> import numpy as np
>>> from sklearn.manifold import TSNE
>>> X = np.array([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]])
>>> X_embedded = TSNE(n_components=2, learning_rate='auto',
...                   init='random', perplexity=3).fit_transform(X)
>>> X_embedded.shape
(4, 2)
File:           c:\users\hp\appdata\local\programs\python\python312\lib\site-package
s\sklearn\manifold\_t_sne.py
Type:           type
Subclasses:
```

In [29]:

```python
# S-curve manifold learning with ISOMAP

tsne = TSNE(n_components=2,learning_rate='auto', init='random')

Y = tsne.fit_transform(X_scurve)

plt.scatter(Y[:,0], Y[:,1], c=color, cmap=plt.cm.Spectral);
```



Interpretation and Visualization Example - MNIST Dataset

In [30]:
```
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.manifold import Isomap, MDS, LocallyLinearEmbedding as LLE, TSNE
plt.rcParams['axes.grid'] = False
import warnings
warnings.filterwarnings("ignore")
```

In [24]:
```
# Loading MNIST data set
image_size = 28 # width and length
no_of_different_labels = 10 #  i.e. 0, 1, 2, 3, ..., 9
image_pixels = image_size * image_size

# Loading Training Samples
train_data = np.loadtxt("mnist_train.csv", delimiter=",")
data = train_data[:,1:]
target = train_data[:,0]

# Visualizing examples per class
plt.figure(figsize=(8,8))
grid_loc=1
for i in range(10): # for each class label
    idx_locations = np.where(target==i)[0] # identify index location where labels a
    idx = np.random.choice(range(len(idx_locations)),replace=False,size=9) # select
    for j in range(9):
        plt.subplot(10,9,grid_loc) # grid labels counts left to right, top to botto
        # reshaping randomly selected image as 28x28 and displaying it
        plt.imshow(data[idx_locations[j]].reshape((28,28)), cmap='gray')
        plt.axis('off')
        grid_loc+=1
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[24], line 7
      4 image_pixels = image_size * image_size
      6 # Loading Training Samples
----> 7 train_data = np.loadtxt("mnist_train.csv", delimiter=",")
      8 data = train_data[:,1:]
      9 target = train_data[:,0]

File c:\Users\HP\AppData\Local\Programs\Python\Python312\Lib\site-packages\numpy\lib
\npyio.py:1373, in loadtxt(fname, dtype, comments, delimiter, converters, skiprows,
usecols, unpack, ndmin, encoding, max_rows, quotechar, like)
   1370 if isinstance(delimiter, bytes):
   1371     delimiter = delimiter.decode('latin1')
-> 1373 arr = _read(fname, dtype=dtype, comment=comment, delimiter=delimiter,
   1374             converters=converters, skiplines=skiprows, usecols=usecols,
   1375             unpack=unpack, ndmin=ndmin, encoding=encoding,
   1376             max_rows=max_rows, quote=quotechar)
   1378 return arr

File c:\Users\HP\AppData\Local\Programs\Python\Python312\Lib\site-packages\numpy\lib
\npyio.py:992, in _read(fname, delimiter, comment, quote, imaginary_unit, usecols, s
kiplines, max_rows, converters, ndmin, unpack, dtype, encoding)
    990     fname = os.fspath(fname)
    991 if isinstance(fname, str):
--> 992     fh = np.lib._datasource.open(fname, 'rt', encoding=encoding)
    993     if encoding is None:
    994         encoding = getattr(fh, 'encoding', 'latin1')

File c:\Users\HP\AppData\Local\Programs\Python\Python312\Lib\site-packages\numpy\lib
\_datasource.py:193, in open(path, mode, destpath, encoding, newline)
    156 """
    157 Open `path` with `mode` and return the file object.
    158 
   (...)
    189 
    190 """
    192 ds = DataSource(destpath)
--> 193 return ds.open(path, mode, encoding=encoding, newline=newline)

File c:\Users\HP\AppData\Local\Programs\Python\Python312\Lib\site-packages\numpy\lib
\_datasource.py:533, in DataSource.open(self, path, mode, encoding, newline)
    530     return _file_openers[ext](found, mode=mode,
    531                               encoding=encoding, newline=newline)
    532 else:
--> 533     raise FileNotFoundError(f"{path} not found.")

FileNotFoundError: mnist_train.csv not found.
```

In [ ]:

```
# use only 1/40 of the data: full dataset takes a long time!

X_train = data[::40]
t_train = target[::40]
```

In [ ]:
```
#PCA
model = PCA(n_components=2)
proj_pca = model.fit_transform(X_train)

#PCA
model = LDA(n_components=2)
proj_lda = model.fit_transform(X_train, t_train)
```

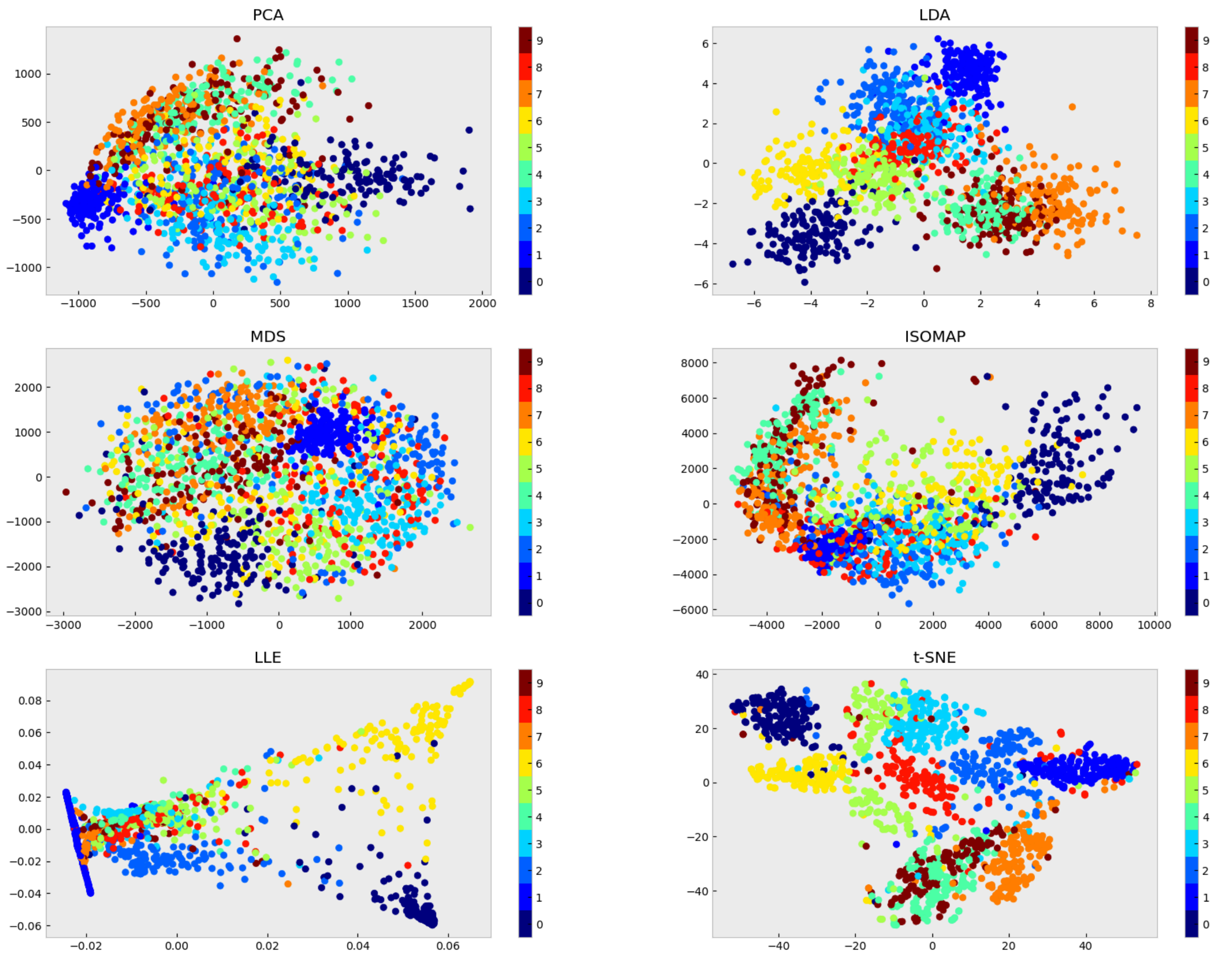
In [ ]:
```
#MDS
model = MDS(n_components=2, normalized_stress='auto')
proj_mds = model.fit_transform(X_train)
```

In [ ]:
```
# IsoMap
model = Isomap(n_components=2)
proj_isomap = model.fit_transform(X_train)
```

In [ ]:
```
# LLE
model = LLE(n_components=2)
proj_lle = model.fit_transform(X_train)
```

In [ ]:
```
# t-SNE
model = TSNE(n_components=2,learning_rate='auto', init='random')
proj_tsne = model.fit_transform(X_train)
```

In [ ]:
```
plt.figure(figsize=(20,15))
plt.subplot(3,2,1)
plt.scatter(proj_pca[:, 0], proj_pca[:, 1], c=t_train, cmap=plt.cm.get_cmap('jet',
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5); plt.title('PCA')

plt.subplot(3,2,2)
plt.scatter(proj_lda[:, 0], proj_lda[:, 1], c=t_train, cmap=plt.cm.get_cmap('jet',
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5); plt.title('LDA')

plt.subplot(3,2,3)
plt.scatter(proj_mds[:, 0], proj_mds[:, 1], c=t_train, cmap=plt.cm.get_cmap('jet',
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5); plt.title('MDS')

plt.subplot(3,2,4)
plt.scatter(proj_isomap[:, 0], proj_isomap[:, 1], c=t_train, cmap=plt.cm.get_cmap('
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5); plt.title('ISOMAP')

plt.subplot(3,2,5)
plt.scatter(proj_lle[:, 0], proj_lle[:, 1], c=t_train, cmap=plt.cm.get_cmap('jet',
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5); plt.title('LLE')

plt.subplot(3,2,6)
plt.scatter(proj_tsne[:, 0], proj_tsne[:, 1], c=t_train, cmap=plt.cm.get_cmap('jet'
plt.colorbar(ticks=range(10))
plt.clim(-0.5, 9.5); plt.title('t-SNE');
```

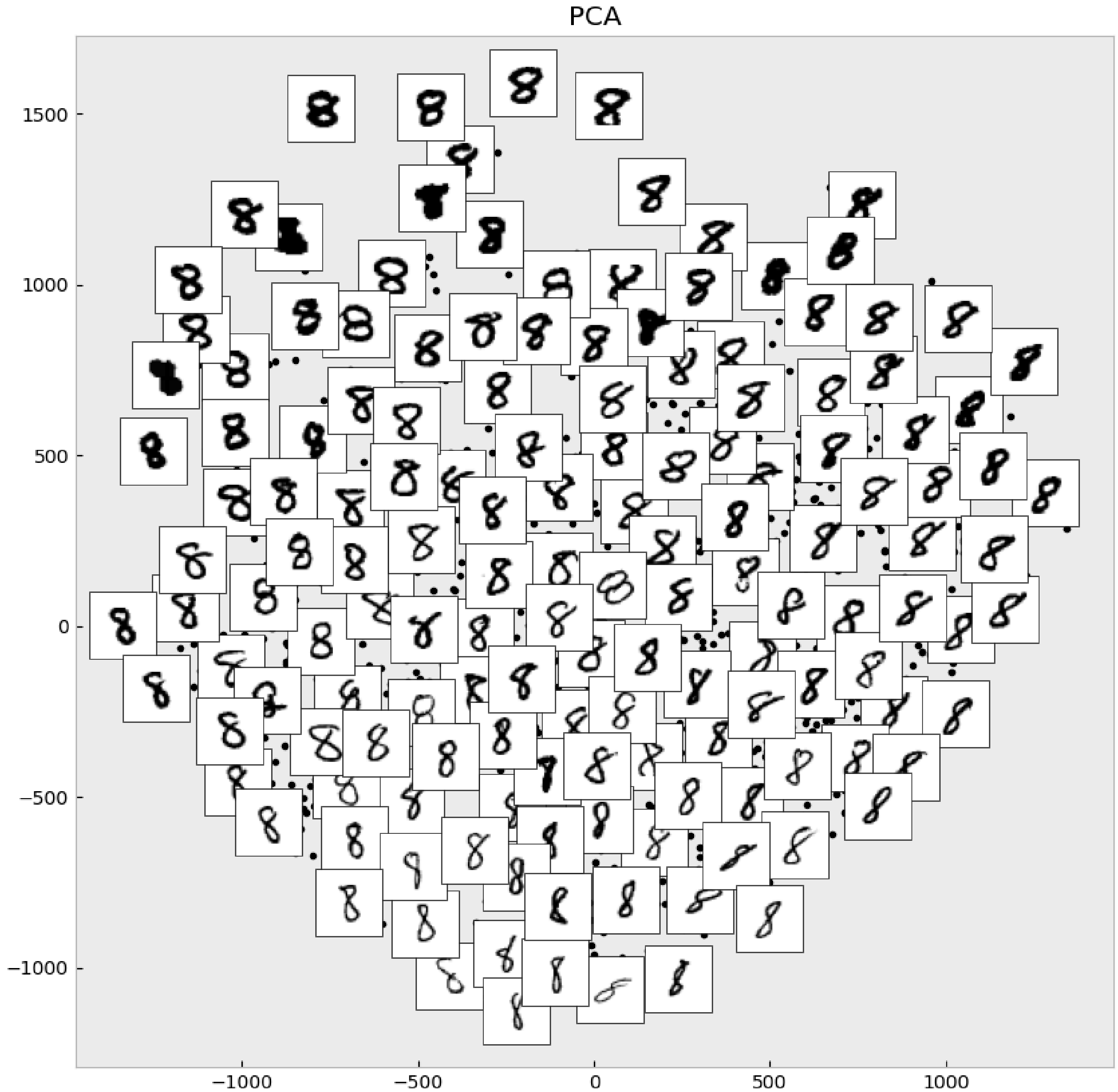


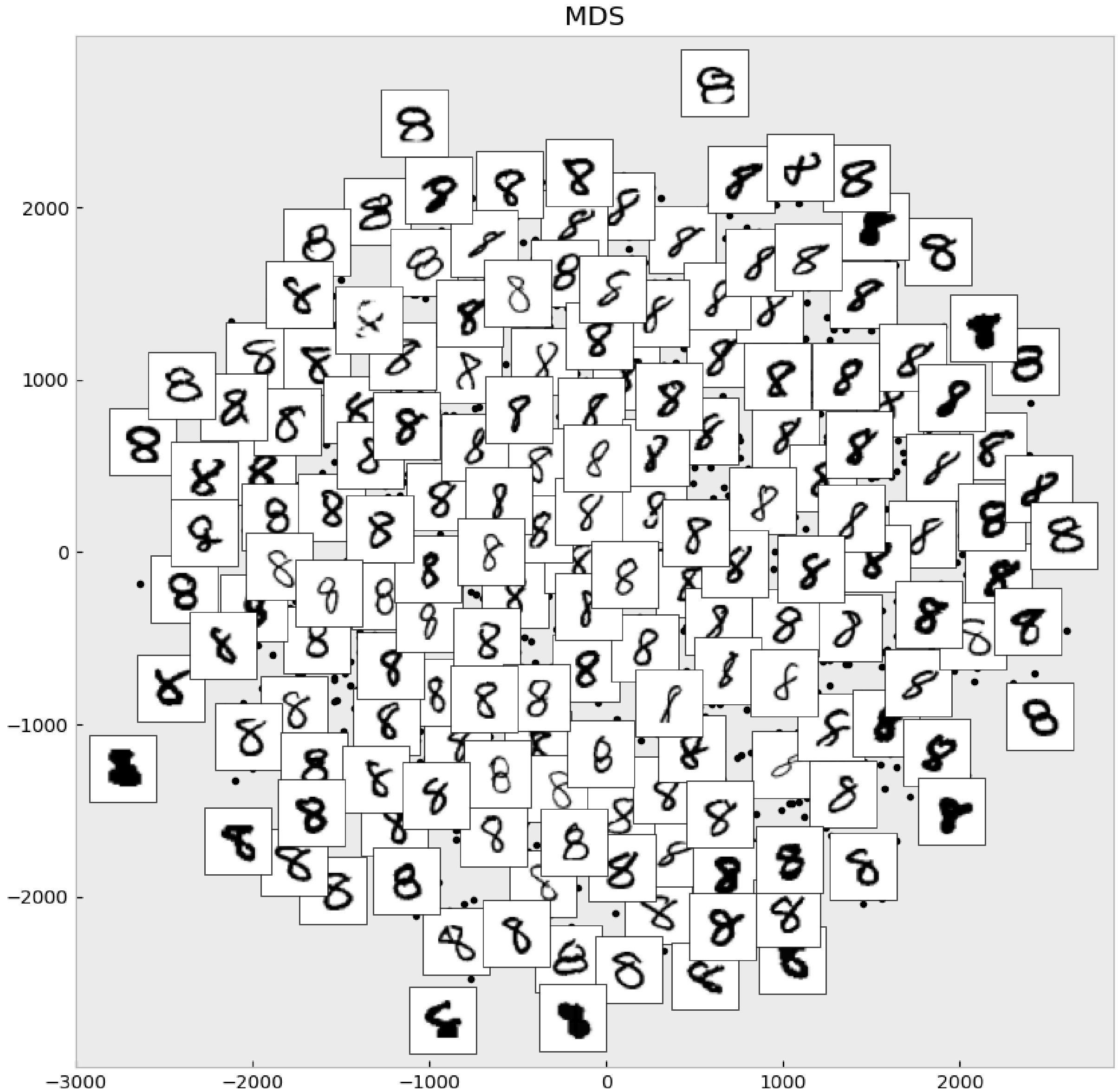
Let's now interpret the new embedding dimensions via visualization:

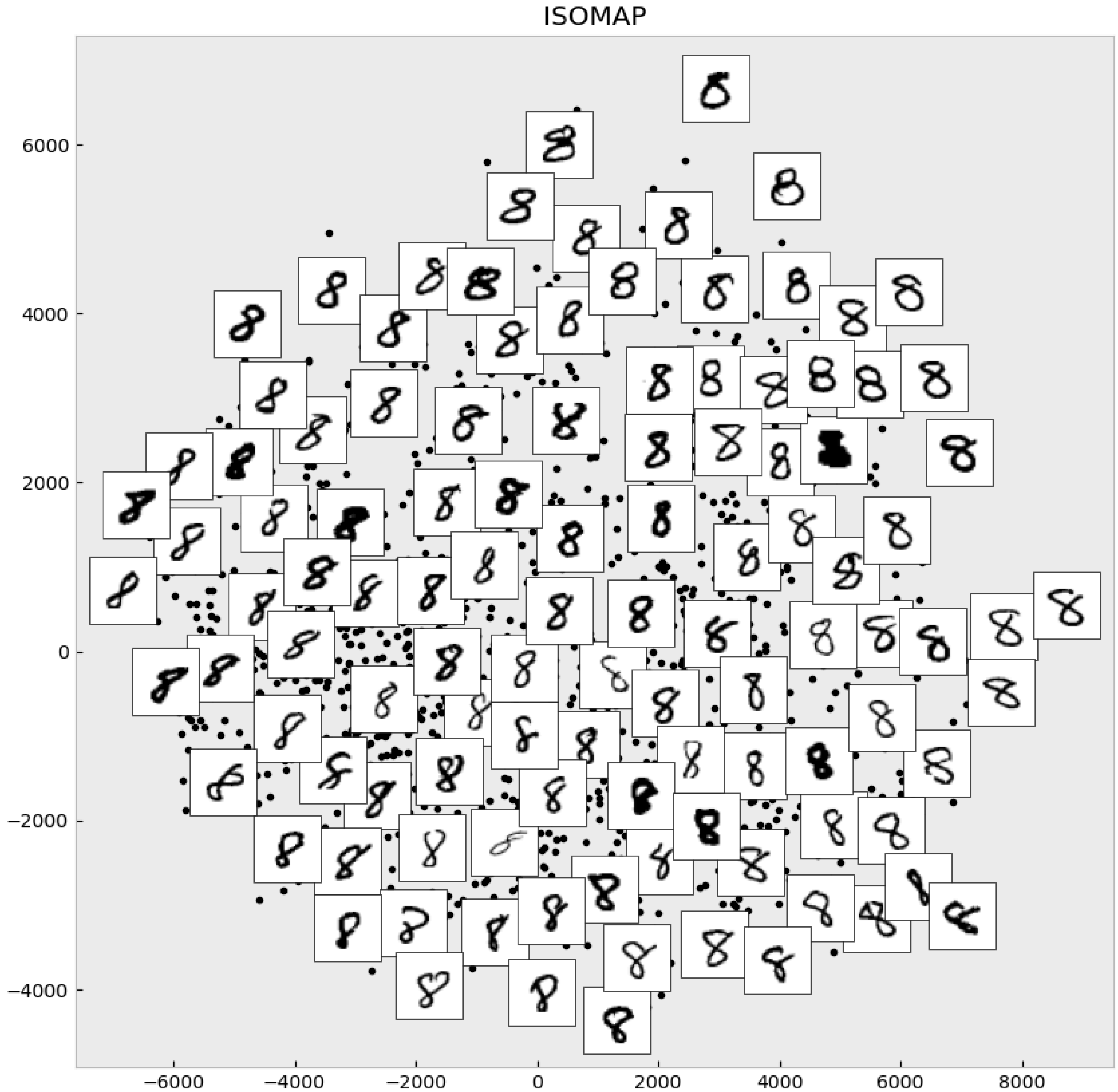
In [ ]:
```
from matplotlib import offsetbox
def plot_components(data, model, images=None, ax=None,
                    thumb_frac=0.05, cmap='gray'):
    ax = ax or plt.gca()

    proj = model.fit_transform(data, t)
    ax.plot(proj[:, 0], proj[:, 1], '.k')

    if images is not None:
        min_dist_2 = (thumb_frac * max(proj.max(0) - proj.min(0))) ** 2
        shown_images = np.array([2 * proj.max(0)])
        for i in range(data.shape[0]):
            dist = np.sum((proj[i] - shown_images) ** 2, 1)
            if np.min(dist) < min_dist_2:
                # don't show points that are too close
                continue
            shown_images = np.vstack([shown_images, proj[i]])
            imagebox = offsetbox.AnnotationBbox(
                offsetbox.OffsetImage(images[i], cmap=cmap),
                                      proj[i])
            ax.add_artist(imagebox)
```

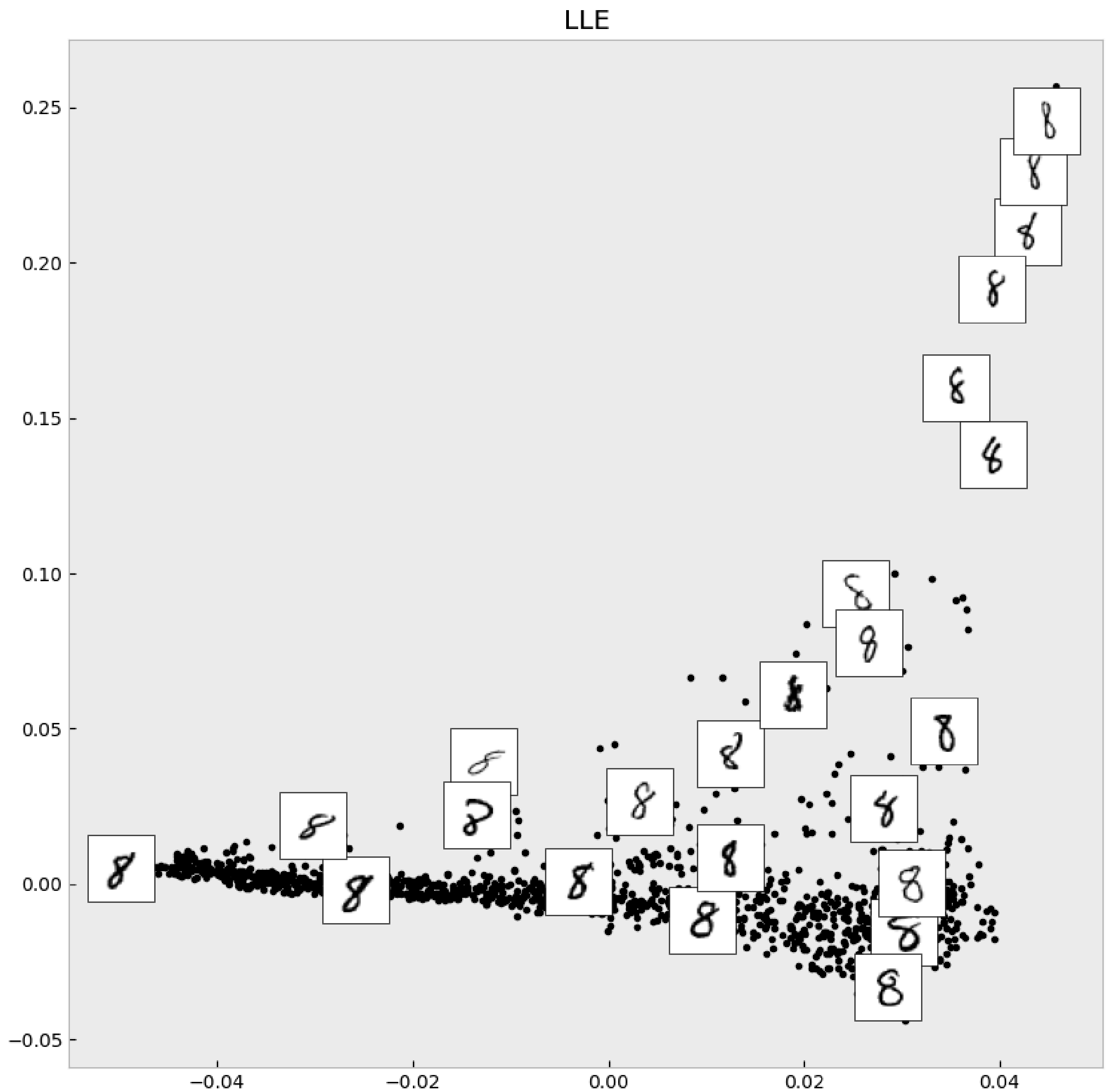
In [ ]:
```
# Choose 1/4 of a digit to project
X = train_data[:,1:]
```

```python
t = train_data[:,0]
N,D = data.shape

# Visualizing a specific digit
digit = 8
data = X[t == digit,:][::4]

# PCA
fig, ax = plt.subplots(figsize=(10, 10))
model = PCA(n_components=2)
plot_components(data, model, images=data.reshape((-1, 28, 28)),
                ax=ax, thumb_frac=0.05, cmap='gray_r')
plt.title('PCA')

# MDS
fig, ax = plt.subplots(figsize=(10, 10))
model2 = MDS(n_components=2, max_iter=100, n_init=1)
plot_components(data, model2, images=data.reshape((-1, 28, 28)),
                ax=ax, thumb_frac=0.05, cmap='gray_r')
plt.title('MDS')

# ISOMAP
fig, ax = plt.subplots(figsize=(10, 10))
model3 = Isomap(n_components=2)
plot_components(data, model3, images=data.reshape((-1, 28, 28)),
                ax=ax, thumb_frac=0.05, cmap='gray_r')
plt.title('ISOMAP');

# LLE
fig, ax = plt.subplots(figsize=(10, 10))
model3 = LLE(n_components=2, n_neighbors=5)
plot_components(data, model3, images=data.reshape((-1, 28, 28)),
                ax=ax, thumb_frac=0.05, cmap='gray_r')
plt.title('LLE')

# t-SNE
fig, ax = plt.subplots(figsize=(10, 10))
model3 = TSNE(n_components=2,learning_rate='auto', init='random')
plot_components(data, model3, images=data.reshape((-1, 28, 28)),
                ax=ax, thumb_frac=0.05, cmap='gray_r')
plt.title('t-SNE');
```


PCA
1500
1000
500
0
−500
−1000
−1000
−500
0
500
1000


MDS
2000
1000
0
−1000
−2000
−3000
−2000
−1000
0
1000
2000


ISOMAP
6000
4000
2000
0
−2000
−4000
−6000
−4000
−2000
0
2000
4000
6000
8000


LLE
0.25
0.20
0.15
0.10
0.05
0.00
−0.05
−0.04
−0.02
0.00
0.02
0.04

t-SNE