Ministry of Education, Culture, and Research of the Republic of Moldova

Technical University of Moldova

Department: Software Engineering and Automatic Control

Study Program: Software Engineering

Report

At *Data Analysis and Visualisation*

Laboratory 2

Done by: st. Dana Speianu, IS-211 M

Checked by: prof.univ. Nistor Grozavu

Chişinău, 2022

Laboratory 2
Theme: Data visualization: Projection models (Manifold learning)

Manifold learning is an approach to non-linear dimensionality reduction. Algorithms for this task are based on the idea that the dimensionality of many data sets is only artificially high.

High-dimensional datasets can be very difficult to visualize. While data in two or three dimensions can be plotted to show the inherent structure of the data, equivalent high-dimensional plots are much less intuitive. To aid visualization of the structure of a dataset, the dimension must be reduced in some way.

The simplest way to accomplish this dimensionality reduction is by taking a random projection of the data. Though this allows some degree of visualization of the data structure, the randomness of the choice leaves much to be desired. In a random projection, it is likely that the more interesting structure within the data will be lost.

To address this concern, a number of supervised and unsupervised linear dimensionality reduction frameworks have been designed, such as Principal Component Analysis (PCA), Independent Component Analysis, Linear Discriminant Analysis, and others. These algorithms define specific rubrics to choose an “interesting” linear projection of the data. These methods can be powerful, but often miss important non-linear structures in the data.

Manifold Learning can be thought of as an attempt to generalize linear frameworks like PCA to be sensitive to non-linear structure in data. Though supervised variants exist, the typical manifold learning problem is unsupervised: it learns the high-dimensional structure of the data from the data itself, without the use of predetermined classifications.

Isomap

One of the earliest approaches to manifold learning is the Isomap algorithm, short for Isometric Mapping. Isomap can be viewed as an extension of Multi-dimensional Scaling (MDS) or Kernel PCA. Isomap seeks a lower-dimensional embedding which maintains geodesic distances between all points. Isomap can be performed with the object Isomap.

Locally Linear Embedding

Locally linear embedding (LLE) seeks a lower-dimensional projection of the data which preserves distances within local neighborhoods. It can be thought of as a series of local Principal Component Analyses which are globally compared to find the best non-linear embedding.

Locally linear embedding can be performed with function locally_linear_embedding or its object-oriented counterpart LocallyLinearEmbedding.

Modified Locally Linear Embedding

One well-known issue with LLE is the regularization problem. When the number of neighbors is greater than the number of input dimensions, the matrix defining each local neighborhood is rank-deficient. To address this, standard LLE applies an arbitrary regularization parameter, which is chosen relative to the trace of the local weight matrix. Though it can be shown formally that as the solution converges to the desired embedding, there is no guarantee that the optimal solution will be found. This problem manifests itself in embeddings that distort the underlying geometry of the manifold.

One method to address the regularization problem is to use multiple weight vectors in each neighborhood. This is the essence of modified locally linear embedding (MLLE). MLLE can be performed with function locally_linear_embedding or its object-oriented counterpart LocallyLinearEmbedding, with the keyword method = 'modified'. It requires n_neighbors > n_components.

Hessian Eigenmapping

Hessian Eigenmapping (also known as Hessian-based LLE: HLLE) is another method of solving the regularization problem of LLE. It revolves around a hessian-based quadratic form at each neighborhood which is used to recover the locally linear structure. Though other implementations note its poor scaling with data size, sklearn implements some algorithmic improvements which make its cost comparable to that of other LLE variants for small output dimensions. HLLE can be performed with function locally_linear_embedding or its object-oriented counterpart LocallyLinearEmbedding, with the keyword method = 'hessian'. It requires n_neighbors > n_components * (n_components + 3) / 2.

t-distributed Stochastic Neighbor Embedding (t-SNE)

t-SNE (TSNE) converts affinities of data points to probabilities. The affinities in the original space are represented by Gaussian joint probabilities and the affinities in the embedded space are represented by Student's t-distributions. This allows t-SNE to be particularly sensitive to local structure and has a few other advantages over existing techniques:

- Revealing the structure at many scales on a single map
- Revealing data that lie in multiple, different, manifolds or clusters
- Reducing the tendency to crowd points together at the center

While Isomap, LLE and variants are best suited to unfold a single continuous low dimensional manifold, t-SNE will focus on the local structure of the data and will tend to extract clustered local groups of samples as highlighted on the S-curve example. This ability to group

samples based on the local structure might be beneficial to visually disentangle a dataset that comprises several manifolds at once as is the case in the digits dataset.

The Kullback-Leibler (KL) divergence of the joint probabilities in the original space and the embedded space will be minimized by gradient descent. Note that the KL divergence is not convex, i.e. multiple restarts with different initializations will end up in local minima of the KL divergence. Hence, it is sometimes useful to try different seeds and select the embedding with the lowest KL divergence.

The disadvantages to using t-SNE are rough:

- t-SNE is computationally expensive and can take several hours on million-sample datasets where PCA will finish in seconds or minutes
- The Barnes-Hut t-SNE method is limited to two or three-dimensional embeddings.
- The algorithm is stochastic and multiple restarts with different seeds can yield different embeddings. However, it is perfectly legitimate to pick the embedding with the least error.
- Global structure is not explicitly preserved. This problem is mitigated by initializing points with PCA (using init='pca').

Multidimensional Scaling (MDS)

Multidimensional scaling (MDS) seeks a low-dimensional representation of the data in which the distances respect well the distances in the original high-dimensional space.

In general, MDS is a technique used for analyzing similarity or dissimilarity data. It attempts to model similarity or dissimilarity data as distances in a geometric space. The data can be ratings of similarity between objects, interaction frequencies of molecules, or trade indices between countries.

There exist two types of MDS algorithms: metric and non-metric. In the scikit-learn, the class MDS implements both. In Metric MDS, the input similarity matrix arises from a metric (and thus respects the triangular inequality), the distances between output two points are then set to be as close as possible to the similarity or dissimilarity data. In the non-metric version, the algorithms will try to preserve the order of the distances and hence seek a monotonic relationship between the distances in the embedded space and the similarities/dissimilarities.

Laboratory 2

Data visualisation: Projection models (Manifold learning)

Part I. Swiss-roll dataset

Swiss Roll: data is essentially a rectangle, but has been "rolled up" like a swiss roll in three dimensional space. Ideally a dimension reduction technique should be able to "unroll" it. The data has been coloured according to one dimension of the rectangle, so should form a rectangle of smooth color variation.

1. Generate the swis-roll dataset with 2000 points using the function datasets.make_swiss_roll

In [18]:
```
from sklearn.datasets import make_swiss_roll
swissroll, swissroll_labels = make_swiss_roll(
    n_samples=1000, noise=0.1, random_state=42
)
```

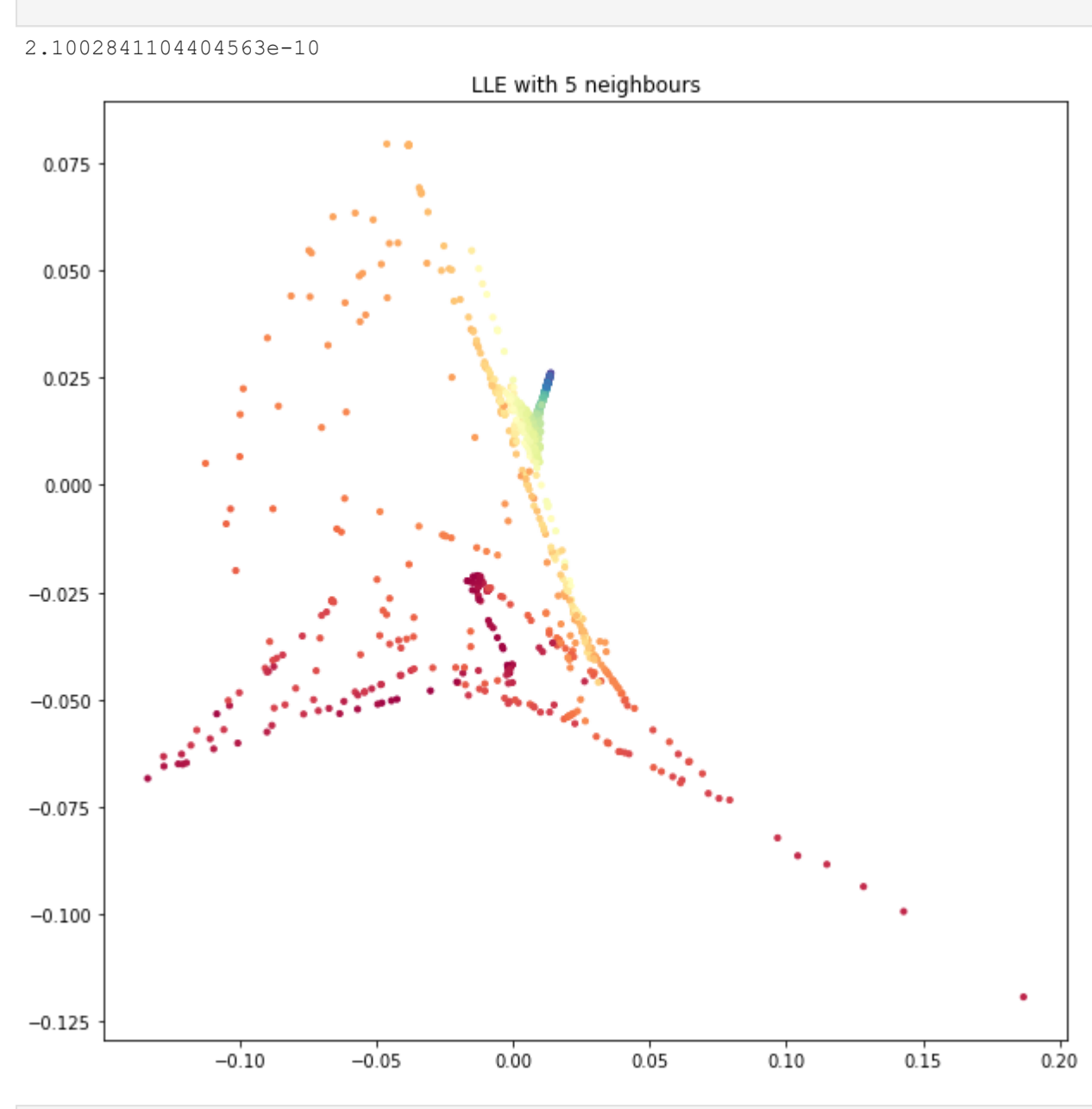
1. Apply the PCA and plot the data.

In [19]:
```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns
import time
import numpy as np


reducers = [(PCA, {})]
test_data = [(swissroll, swissroll_labels)]
dataset_names = ["Swiss Roll"]
n_rows = len(test_data)
n_cols = len(reducers)
ax_index = 1
ax_list = []

plt.figure(figsize=(10, 8))
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)
for data, labels in test_data:
    for reducer, args in reducers:
        start_time = time.time()
        embedding = reducer(n_components=2, **args).fit_transform(data)
        elapsed_time = time.time() - start_time
        ax = plt.subplot(n_rows, n_cols, ax_index)
        if isinstance(labels[0], tuple):
            ax.scatter(*embedding.T, s=10, c=labels, alpha=0.5)
        else:
            ax.scatter(*embedding.T, s=10, c=labels, cmap="Spectral", alpha=0.5)
        ax.text(
            0.99,
            0.01,
            "{:.2f} s".format(elapsed_time),
            transform=ax.transAxes,
            size=14,
            horizontalalignment="right",
        )
        ax_list.append(ax)
        ax_index += 1
plt.setp(ax_list, xticks=[], yticks=[])

for i in np.arange(n_rows) * n_cols:
    ax_list[i].set_ylabel(dataset_names[i // n_cols], size=16)
for i in range(n_cols):
    ax_list[i].set_xlabel(repr(reducers[i][0]()).split("(")[0], size=16)
    ax_list[i].xaxis.set_label_position("top")

plt.tight_layout()
plt.show()
```

1. Apply LLE (Local Linear Embedding) with 5 neighbours (manifold.locally_linear_embedding) by printing the error. Change the number of neighbours from 2 to 15 and plot the error line. Which is the best number of neighbours ?

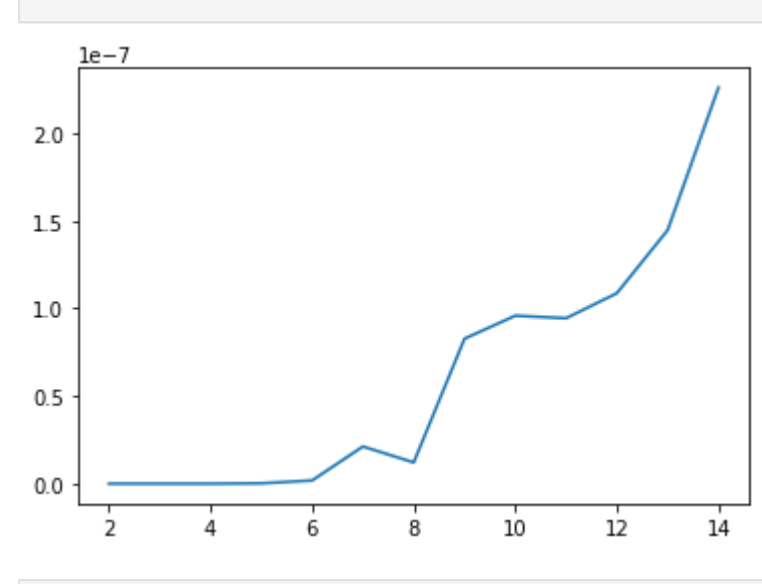
In [20]:
```
from sklearn.manifold import LocallyLinearEmbedding
embedding = LocallyLinearEmbedding(n_neighbors=5, n_components=2)
X_transformed = embedding.fit_transform(swissroll)
print(embedding.reconstruction_error_)
plt.figure(figsize=(10, 10))
plt.scatter(X_transformed[:, 0], X_transformed[:, 1], c=swissroll_labels, s=10, cmap=plt.cm.get_cmap('Spectral'
plt.title("LLE with 5 neighbours")
plt.show()
```

```
2.1002841104404563e-10
```



In [21]:
```
error =np.Infinity
error_list = []
nr_of_neighbors = []
for n_neighbors in range(2, 15):
    embedding = LocallyLinearEmbedding(n_neighbors=n_neighbors, eigen_solver='dense')
    X_lle = embedding.fit_transform(swissroll)
    if embedding.reconstruction_error_ < error:
        error = embedding.reconstruction_error_
    error_list.append(embedding.reconstruction_error_)
    nr_of_neighbors.append(n_neighbors)
```

In [22]:
```
import matplotlib.pyplot as plt
import numpy as np

plt.plot(nr_of_neighbors, error_list)
plt.show()
```



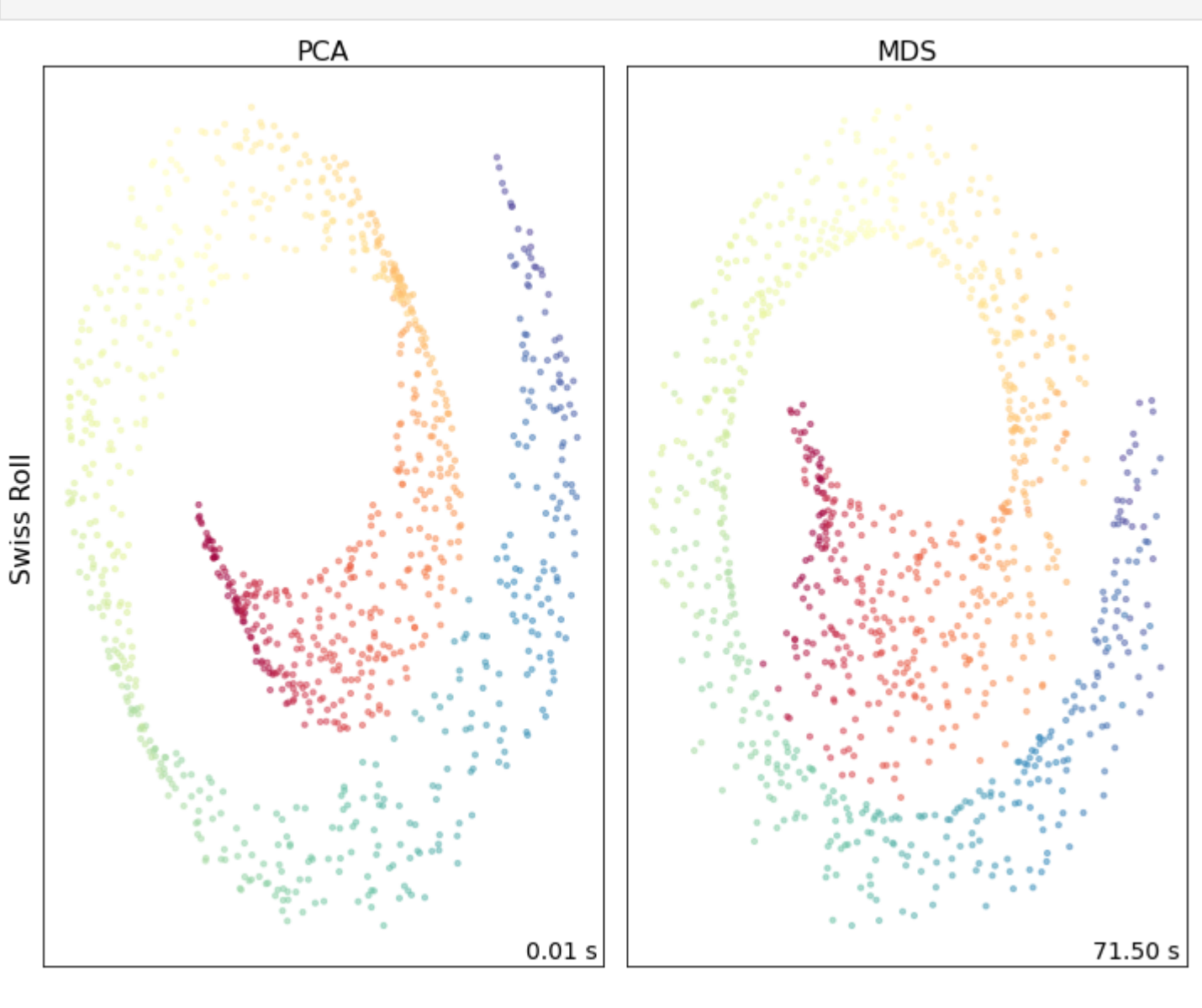
In [23]:
```
print(error_list)
print(nr_of_neighbors)
print("The smallest error is ",min(error_list))
```

```
[8.687158318203206e-16, -4.916067096633607e-15, -1.6463563334324976e-15, 2.1003380582384244e-10, 1.832064877865
863e-09, 2.1224234923719744e-08, 1.2123302673409451e-08, 8.276932292443444e-08, 9.58917127448764e-08, 9.4455516
02821132e-08, 1.0873843040993794e-07, 1.4476951301413483e-07, 2.260884450224833e-07]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
The smallest error is  -4.916067096633607e-15
```

The best number of neighbors is 3.

1. Use Multi Dimensional Scaling with manifold.MDS and visualize the dataset in 2 dimension.

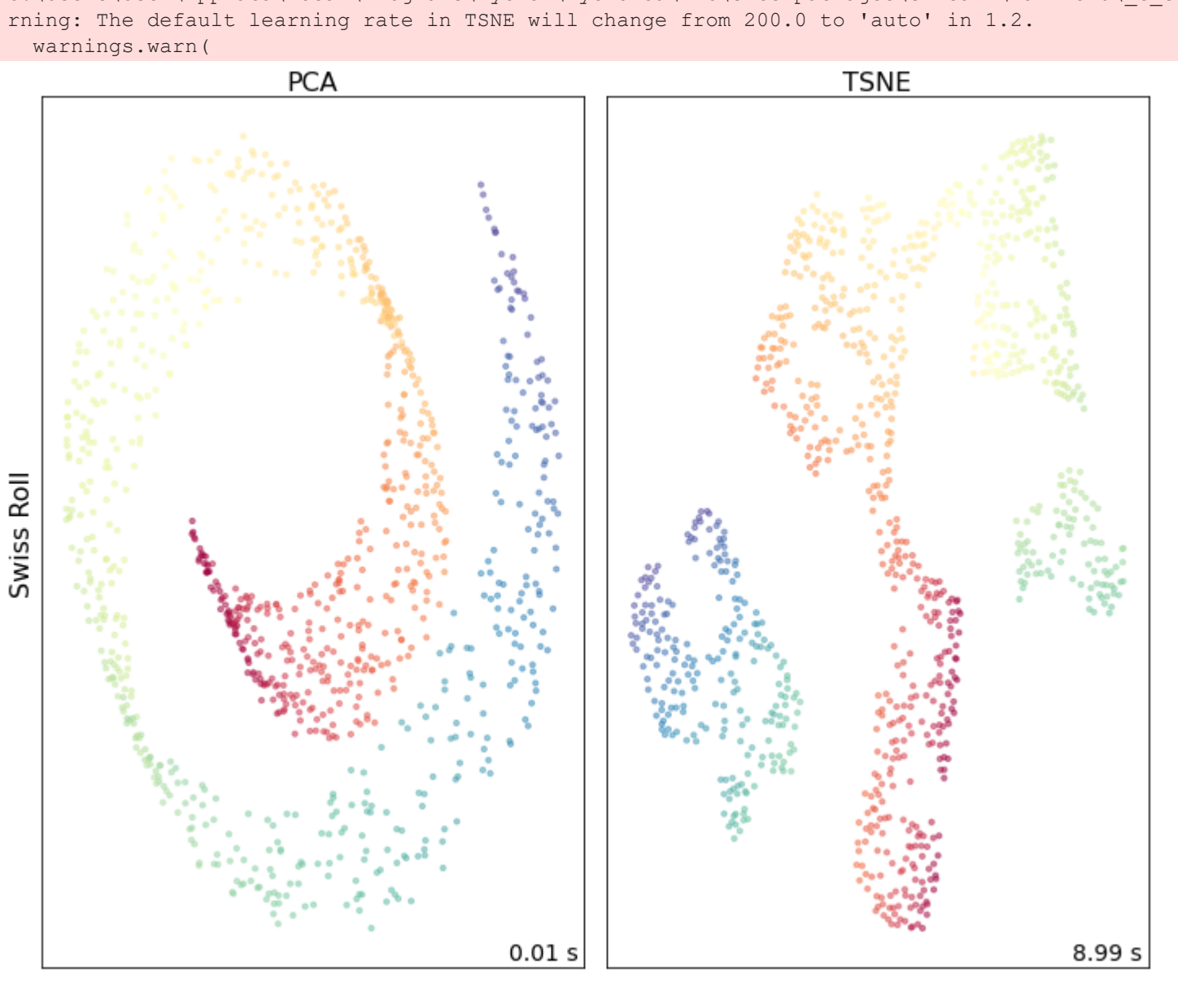
In [24]:
```
from sklearn.manifold import MDS

reducers = [(PCA, {}),
        (MDS, {}),
]
test_data = [(swissroll, swissroll_labels)]
dataset_names = ["Swiss Roll"]
n_rows = len(test_data)
n_cols = len(reducers)
ax_index = 1
ax_list = []

plt.figure(figsize=(10, 8))
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)
for data, labels in test_data:
    for reducer, args in reducers:
        start_time = time.time()
        embedding = reducer(n_components=2, **args).fit_transform(data)
        elapsed_time = time.time() - start_time
        ax = plt.subplot(n_rows, n_cols, ax_index)
        if isinstance(labels[0], tuple):
            ax.scatter(*embedding.T, s=10, c=labels, alpha=0.5)
        else:
            ax.scatter(*embedding.T, s=10, c=labels, cmap="Spectral", alpha=0.5)
        ax.text(
            0.99,
            0.01,
            "{:.2f} s".format(elapsed_time),
            transform=ax.transAxes,
            size=14,
            horizontalalignment="right",
        )
        ax_list.append(ax)
        ax_index += 1
plt.setp(ax_list, xticks=[], yticks=[])

for i in np.arange(n_rows) * n_cols:
    ax_list[i].set_ylabel(dataset_names[i // n_cols], size=16)
for i in range(n_cols):
    ax_list[i].set_xlabel(repr(reducers[i][0]()).split("(")[0], size=16)
    ax_list[i].xaxis.set_label_position("top")

plt.tight_layout()
plt.show()
```



1. Apply t-SNE model to the same dataset with manifold.TSNE. Visualize the dataset.

In [25]:
```
from sklearn.manifold import TSNE

reducers = [(PCA, {}),
        (TSNE, {"perplexity": 50}),
]
test_data = [(swissroll, swissroll_labels)]
dataset_names = ["Swiss Roll"]
n_rows = len(test_data)
n_cols = len(reducers)
ax_index = 1
ax_list = []

plt.figure(figsize=(10, 8))
plt.subplots_adjust(
    left=0.02, right=0.98, bottom=0.001, top=0.96, wspace=0.05, hspace=0.01
)
for data, labels in test_data:
    for reducer, args in reducers:
        start_time = time.time()
        embedding = reducer(n_components=2, **args).fit_transform(data)
        elapsed_time = time.time() - start_time
        ax = plt.subplot(n_rows, n_cols, ax_index)
        if isinstance(labels[0], tuple):
            ax.scatter(*embedding.T, s=10, c=labels, alpha=0.5)
        else:
            ax.scatter(*embedding.T, s=10, c=labels, cmap="Spectral", alpha=0.5)
        ax.text(
            0.99,
            0.01,
            "{:.2f} s".format(elapsed_time),
            transform=ax.transAxes,
            size=14,
            horizontalalignment="right",
        )
        ax_list.append(ax)
        ax_index += 1
plt.setp(ax_list, xticks=[], yticks=[])

for i in np.arange(n_rows) * n_cols:
    ax_list[i].set_ylabel(dataset_names[i // n_cols], size=16)
for i in range(n_cols):
    ax_list[i].set_xlabel(repr(reducers[i][0]()).split("(")[0], size=16)
    ax_list[i].xaxis.set_label_position("top")

plt.tight_layout()
plt.show()
```

```
C:\Users\User\AppData\Local\Programs\Python\Python39\lib\site-packages\sklearn\manifold\_t_sne.py:780: FutureWa
rning: The default initialization in TSNE will change from 'random' to 'pca' in 1.2.
  warnings.warn(
C:\Users\User\AppData\Local\Programs\Python\Python39\lib\site-packages\sklearn\manifold\_t_sne.py:790: FutureWa
rning: The default learning rate in TSNE will change from 200.0 to 'auto' in 1.2.
  warnings.warn(
```



1. Conclude and give the details of the best model.

The best model is t-SNE as according to the images above it grouped the points closer to each other that are of the same color but every shape that is contained in the image contains more clusters.

Part II. Digit dataset

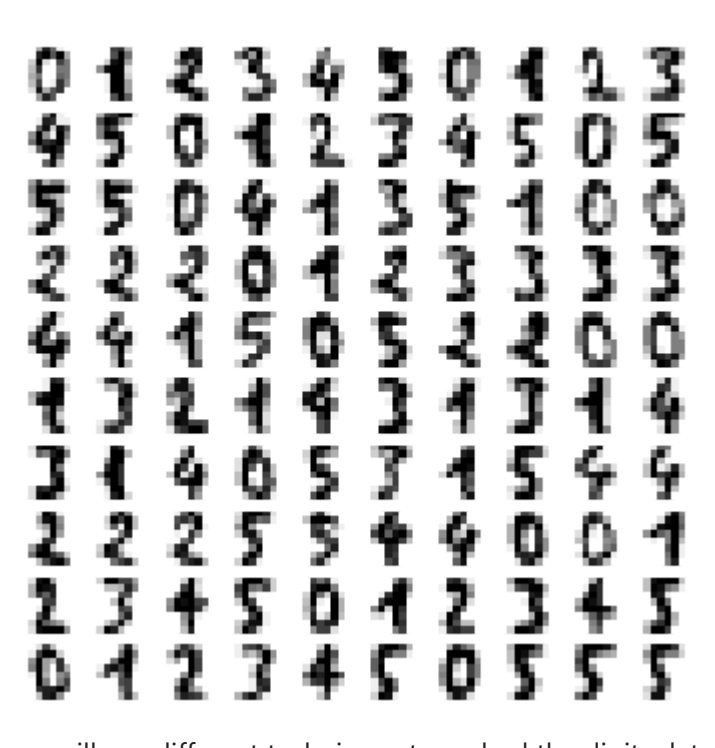
Import the digit dataset containing only 6 classes : digits = datasets.load_digits(n_class=6) Follow the tutorial : https://scikit-learn.org/dev/auto_examples/manifold/plot_lle_digits.html

In [26]:
```
from sklearn.datasets import load_digits
digits = load_digits(n_class=6)
```

In [27]:
```
X, y = digits.data, digits.target
n_samples, n_features = X.shape
n_neighbors = 30
```

In [28]:
```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(nrows=10, ncols=10, figsize=(6, 6))
for idx, ax in enumerate(axs.ravel()):
    ax.imshow(X[idx].reshape((8, 8)), cmap=plt.cm.binary)
    ax.axis("off")
_ = fig.suptitle("A selection from the 64-dimensional digits dataset", fontsize=16)
```

A selection from the 64-dimensional digits dataset



Below, we will use different techniques to embed the digits dataset. We will plot the projection of the original data onto each embedding. It will allow us to check whether or digits are grouped together in the embedding space, or scattered across it.

In [29]:
```
import numpy as np
from matplotlib import offsetbox
from sklearn.preprocessing import MinMaxScaler


def plot_embedding(X, title, ax):
    X = MinMaxScaler().fit_transform(X)
    for digit in digits.target_names:
        ax.scatter(
            *X[y == digit].T,
            marker=f"${digit}$",
            s=60,
            color=plt.cm.Dark2(digit),
            alpha=0.425,
            zorder=2,
        )
    shown_images = np.array([[1.0, 1.0]])  # just something big
    for i in range(X.shape[0]):
        # plot every digit on the embedding
        # show an annotation box for a group of digits
        dist = np.sum((X[i] - shown_images) ** 2, 1)
        if np.min(dist) < 4e-3:
            # don't show points that are too close
            continue
        shown_images = np.concatenate([shown_images, [X[i]]], axis=0)
        imagebox = offsetbox.AnnotationBbox(
            offsetbox.OffsetImage(digits.images[i], cmap=plt.cm.gray_r), X[i]
        )
        imagebox.set(zorder=1)
        ax.add_artist(imagebox)

    ax.set_title(title)
    ax.axis("off")
```

Below, we compare different techniques. However, there are a couple of things to note:

the RandomTreesEmbedding is not technically a manifold embedding method, as it learn a high-dimensional representation on which we apply a dimensionality reduction method. However, it is often useful to cast a dataset into a representation in which the classes are linearly-separable.

the LinearDiscriminantAnalysis and the NeighborhoodComponentsAnalysis, are supervised dimensionality reduction method, i.e. they make use of the provided labels, contrary to other methods.

the TSNE is initialized with the embedding that is generated by PCA in this example. It ensures global stability of the embedding, i.e., the embedding does not depend on random initialization.

In [30]:
```
from sklearn.decomposition import TruncatedSVD
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomTreesEmbedding
from sklearn.manifold import (
    Isomap,
    LocallyLinearEmbedding,
    MDS,
    SpectralEmbedding,
    TSNE,
)
from sklearn.neighbors import NeighborhoodComponentsAnalysis
from sklearn.pipeline import make_pipeline
from sklearn.random_projection import SparseRandomProjection

embeddings = {
    "Random projection embedding": SparseRandomProjection(
        n_components=2, random_state=42
    ),
    "Truncated SVD embedding": TruncatedSVD(n_components=2),
    "Linear Discriminant Analysis embedding": LinearDiscriminantAnalysis(
        n_components=2
    ),
    "Isomap embedding": Isomap(n_neighbors=n_neighbors, n_components=2),
    "Standard LLE embedding": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="standard"
    ),
    "Modified LLE embedding": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="modified"
    ),
    "Hessian LLE embedding": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="hessian"
    ),
    "LTSA LLE embedding": LocallyLinearEmbedding(
        n_neighbors=n_neighbors, n_components=2, method="ltsa"
    ),
    "MDS embedding": MDS(n_components=2, n_init=1, max_iter=120, n_jobs=2),
    "Random Trees embedding": make_pipeline(
        RandomTreesEmbedding(n_estimators=200, max_depth=5, random_state=0),
        TruncatedSVD(n_components=2),
    ),
    "Spectral embedding": SpectralEmbedding(
        n_components=2, random_state=0, eigen_solver="arpack"
    ),
    "t-SNE embeedding": TSNE(
        n_components=2,
        init="pca",
        learning_rate="auto",
        n_iter=500,
        n_iter_without_progress=150,
        n_jobs=2,
        random_state=0,
    ),
    "NCA embedding": NeighborhoodComponentsAnalysis(
        n_components=2, init="pca", random_state=0
    ),
}
```

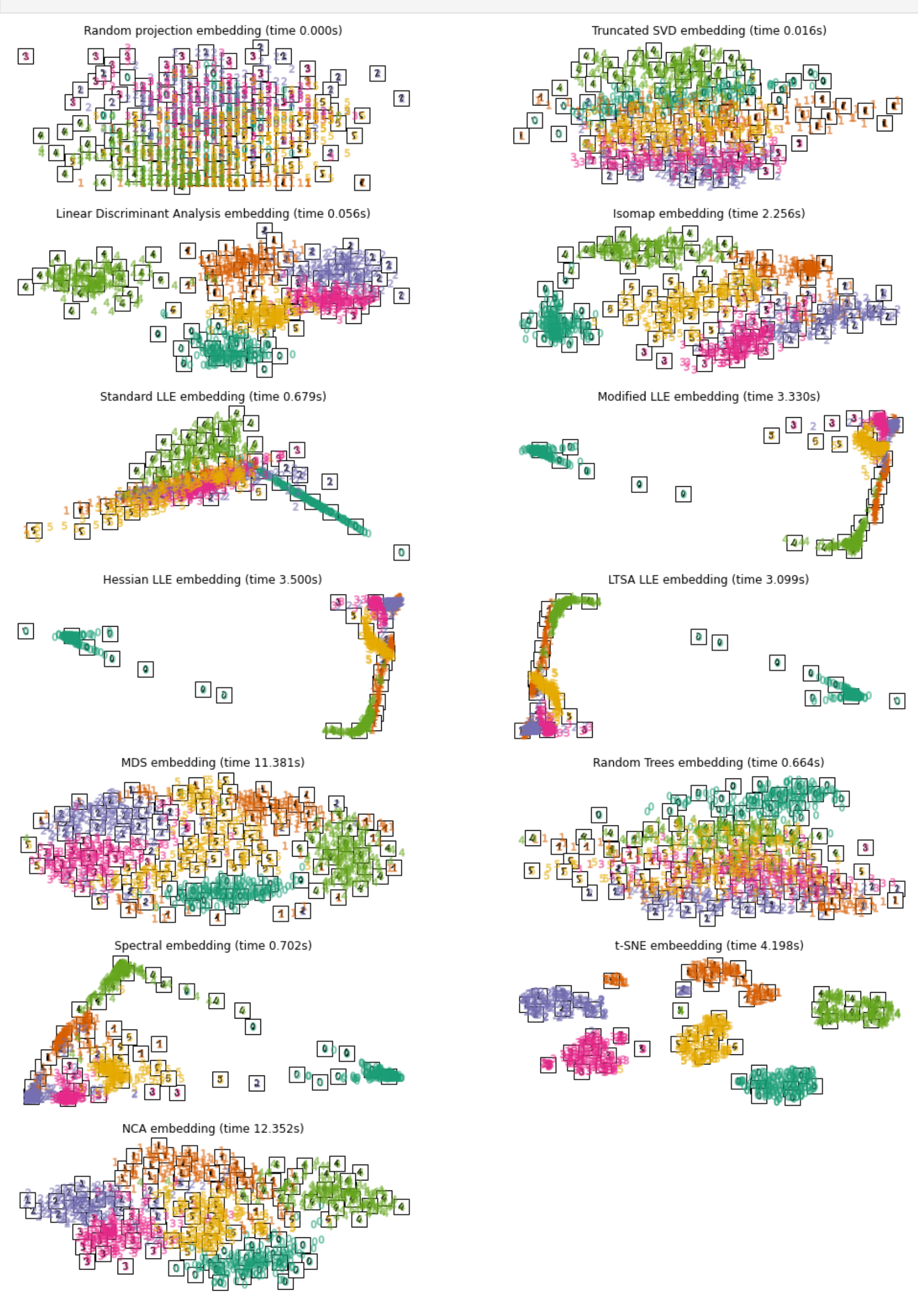
Once we declared all the methodes of interest, we can run and perform the projection of the original data. We will store the projected data as well as the computational time needed to perform each projection.

In [31]:
```
from time import time

projections, timing = {}, {}
for name, transformer in embeddings.items():
    if name.startswith("Linear Discriminant Analysis"):
        data = X.copy()
        data.flat[:: X.shape[1] + 1] += 0.01  # Make X invertible
    else:
        data = X

    print(f"Computing {name}...")
    start_time = time()
    projections[name] = transformer.fit_transform(data, y)
    timing[name] = time() - start_time
```

```
Computing Random projection embedding...
Computing Truncated SVD embedding...
Computing Linear Discriminant Analysis embedding...
Computing Isomap embedding...
Computing Standard LLE embedding...
Computing Modified LLE embedding...
Computing Hessian LLE embedding...
Computing LTSA LLE embedding...
Computing MDS embedding...
Computing Random Trees embedding...
Computing Spectral embedding...
Computing t-SNE embeedding...
C:\Users\User\AppData\Local\Programs\Python\Python39\lib\site-packages\sklearn\manifold\_t_sne.py:982: FutureWa
rning: The PCA initialization in TSNE will change to have the standard deviation of PC1 equal to 1e-4 in 1.2. T
his will ensure better convergence.
  warnings.warn(
Computing NCA embedding...
```

Finally, we can plot the resulting projection given by each method.

In [32]:
```
from itertools import zip_longest

fig, axs = plt.subplots(nrows=7, ncols=2, figsize=(17, 24))

for name, ax in zip_longest(timing, axs.ravel()):
    if name is None:
        ax.axis("off")
        continue
    title = f"{name} (time {timing[name]:.3f}s)"
    plot_embedding(projections[name], title, ax)

plt.show()
```



1. Analyse the results by explaining the models

1. Random Projection embedding took the smallest amount of time and did ot very good job as the points are very spread.
2. The Truncated SVD embedding has a better result as the grouping are better observable than for Random Projection embedding and took time that the first.
3. Linear Discriminant Analysis embedding worked better as there are visible clusters and the points are close to each other. The Isomap model results looks almost similar with the Linear Discriminant Analysis embedding, but it was executed with 36 times more time.
4. Hessian and LTSA LLE embedding looks the same and the clusters in these models are very concentrated and points are ver close to each other. Almost all clusters are located in the left side of the graph.
5. The t-NSE results are the best than any other model. The clusters are very good isolated from other clusters and we can easily distinguish them.

1. Use a classification model (Decision Tree for example) on all the projections and compute the errors.

In [35]:
```
from sklearn.tree import DecisionTreeClassifier
# for each projection

for name, projection in projections.items():
    clf = DecisionTreeClassifier(random_state=0)
    clf.fit(projection, y)
    score = clf.score(projection, y)
    print("{}: Score of the model = {:.3f} %".format(name, score*100))
```

```
Random projection embedding: Score of the model = 71.837 %
Truncated SVD embedding: Score of the model = 100.000 %
Linear Discriminant Analysis embedding: Score of the model = 100.000 %
Isomap embedding: Score of the model = 100.000 %
Standard LLE embedding: Score of the model = 100.000 %
Modified LLE embedding: Score of the model = 100.000 %
Hessian LLE embedding: Score of the model = 100.000 %
LTSA LLE embedding: Score of the model = 100.000 %
MDS embedding: Score of the model = 100.000 %
Random Trees embedding: Score of the model = 100.000 %
Spectral embedding: Score of the model = 100.000 %
t-SNE embeedding: Score of the model = 100.000 %
NCA embedding: Score of the model = 100.000 %
```