```
from sklearn.datasets import fetch_openml
from sklearn.decomposition import PCA
import pandas as pd
import numpy as np
import warnings

warnings.filterwarnings("ignore")
```

```
def get_original_pairwise_affinities(X: np.ndarray, perplexity: int = 10
    """
    Function to obtain affinities matrix.

    Parameters:
    X (np.ndarray): The input data array.
    perplexity (int): The perplexity value for the grid search.

    Returns:
    np.ndarray: The pairwise affinities matrix.
    """

    n = len(X)

    print("Computing Pairwise Affinities....")

    p_ij = np.zeros(shape=(n, n))
    for i in range(0, n):
        # Equation 1 numerator
        diff = X[i] - X
        σ_i = grid_search(diff, i, perplexity)  # Grid Search for σ_i
        norm = np.linalg.norm(diff, axis=1)
        p_ij[i, :] = np.exp(-(norm**2) / (2 * σ_i**2))

        # Set p = 0 when j = i
        np.fill_diagonal(p_ij, 0)

        # Equation 1
        p_ij[i, :] = p_ij[i, :] / np.sum(p_ij[i, :])

    # Set 0 values to minimum numpy value (ε approx. = 0)
    ε = np.nextafter(0, 1)
    p_ij = np.maximum(p_ij, ε)

    print("Completed Pairwise Affinities Matrix. \n")

    return p_ij
```

```python
def grid_search(diff_i: np.ndarray, i: int, perplexity: int) -> float:
    """
    Helper function to obtain σ's based on user-specified perplexity.

    Parameters:
        diff_i (np.ndarray): Array containing the pairwise differences b
        i (int): Index of the current data point.
        perplexity (int): User-specified perplexity value.

    Returns:
        float: The value of σ that satisfies the perplexity condition.
    """

    result = np.inf  # Set first result to be infinity

    norm = np.linalg.norm(diff_i, axis=1)
    std_norm = np.std(norm)  # Use standard deviation of norms to define

    for σ_search in np.linspace(0.01 * std_norm, 5 * std_norm, 200):
        # Equation 1 Numerator
        p = np.exp(-(norm**2) / (2 * σ_search**2))

        # Set p = 0 when i = j
        p[i] = 0

        # Equation 1 (ε -> 0)
        ε = np.nextafter(0, 1)
        p_new = np.maximum(p / np.sum(p), ε)

        # Shannon Entropy
        H = -np.sum(p_new * np.log2(p_new))

        # Get log(perplexity equation) as close to equality
        if np.abs(np.log(perplexity) - H * np.log(2)) < np.abs(result):
            result = np.log(perplexity) - H * np.log(2)
            σ = σ_search

    return σ
```

```python
def get_symmetric_p_ij(p_ij: np.ndarray) -> np.ndarray:
    """
    Function to obtain symmetric affinities matrix utilized in t-SNE.

    Parameters:
    p_ij (np.ndarray): The input affinity matrix.

    Returns:
    np.ndarray: The symmetric affinities matrix.

    """
    print("Computing Symmetric p_ij matrix....")

    n = len(p_ij)
    p_ij_symmetric = np.zeros(shape=(n, n))
    for i in range(0, n):
        for j in range(0, n):
            p_ij_symmetric[i, j] = (p_ij[i, j] + p_ij[j, i]) / (2 * n)

    # Set 0 values to minimum numpy value (ε approx. = 0)
    ε = np.nextafter(0, 1)
    p_ij_symmetric = np.maximum(p_ij_symmetric, ε)

    print("Completed Symmetric p_ij Matrix. \n")

    return p_ij_symmetric
```

```python
def initialization(
    X: np.ndarray, n_dimensions: int = 2, initialization: str = "random"
) -> np.ndarray:
    """
    Obtain initial solution for t-SNE either randomly or using PCA.

    Parameters:
        X (np.ndarray): The input data array.
        n_dimensions (int): The number of dimensions for the output solu
        initialization (str): The initialization method. Can be 'random'

    Returns:
        np.ndarray: The initial solution for t-SNE.

    Raises:
        ValueError: If the initialization method is neither 'random' nor
    """

    # Sample Initial Solution
    if initialization == "random" or initialization != "PCA":
        y0 = np.random.normal(loc=0, scale=1e-4, size=(len(X), n_dimensi
    elif initialization == "PCA":
        X_centered = X - X.mean(axis=0)
        _, _, Vt = np.linalg.svd(X_centered)
        y0 = X_centered @ Vt.T[:, :n_dimensions]
    else:
        raise ValueError("Initialization must be 'random' or 'PCA'")

    return y0
```

```python
def get_low_dimensional_affinities(Y: np.ndarray) -> np.ndarray:
    """
    Obtain low-dimensional affinities.

    Parameters:
    Y (np.ndarray): The low-dimensional representation of the data point

    Returns:
    np.ndarray: The low-dimensional affinities matrix.
    """

    n = len(Y)
    q_ij = np.zeros(shape=(n, n))

    for i in range(0, n):
        # Equation 4 Numerator
        diff = Y[i] - Y
        norm = np.linalg.norm(diff, axis=1)
        q_ij[i, :] = (1 + norm**2) ** (-1)

    # Set p = 0 when j = i
    np.fill_diagonal(q_ij, 0)

    # Equation 4
    q_ij = q_ij / q_ij.sum()

    # Set 0 values to minimum numpy value (ε approx. = 0)
    ε = np.nextafter(0, 1)
    q_ij = np.maximum(q_ij, ε)

    return q_ij
```

```
def get_gradient(p_ij: np.ndarray, q_ij: np.ndarray, Y: np.ndarray) -> n
    """
    Obtain gradient of cost function at current point Y.

    Parameters:
    p_ij (np.ndarray): The joint probability distribution matrix.
    q_ij (np.ndarray): The Student's t-distribution matrix.
    Y (np.ndarray): The current point in the low-dimensional space.

    Returns:
    np.ndarray: The gradient of the cost function at the current point Y
    """

    n = len(p_ij)

    # Compute gradient
    gradient = np.zeros(shape=(n, Y.shape[1]))
    for i in range(0, n):
        # Equation 5
        diff = Y[i] - Y
        A = np.array([(p_ij[i, :] - q_ij[i, :])])
        B = np.array([(1 + np.linalg.norm(diff, axis=1)) ** (-1)])
        C = diff
        gradient[i] = 4 * np.sum((A * B).T * C, axis=0)

    return gradient
```

```python
def tsne(
    X: np.ndarray,
    perplexity: int = 10,
    T: int = 1000,
    η: int = 200,
    early_exaggeration: int = 4,
    n_dimensions: int = 2,
):
    """
    t-SNE (t-Distributed Stochastic Neighbor Embedding) algorithm impl

    Args:
        X (np.ndarray): The input data matrix of shape (n_samples, n_f
        perplexity (int, optional): The perplexity parameter. Default
        T (int, optional): The number of iterations for optimization.
        η (int, optional): The learning rate for updating the low-dime
        early_exaggeration (int, optional): The factor by which the pa
            during the early iterations of optimization. Default is 4.
        n_dimensions (int, optional): The number of dimensions of the

    Returns:
        list[np.ndarray, np.ndarray]: A list containing the final low-
            of embeddings at each iteration.

    """
    n = len(X)

    # Get original affinities matrix
    p_ij = get_original_pairwise_affinities(X, perplexity)
    p_ij_symmetric = get_symmetric_p_ij(p_ij)

    # Initialization
    Y = np.zeros(shape=(T, n, n_dimensions))
    Y_minus1 = np.zeros(shape=(n, n_dimensions))
    Y[0] = Y_minus1
    Y1 = initialization(X, n_dimensions)
    Y[1] = np.array(Y1)

    print("Optimizing Low Dimensional Embedding....")
    # Optimization
    for t in range(1, T - 1):
        # Momentum & Early Exaggeration
        if t < 250:
            α = 0.5
            early_exaggeration = early_exaggeration
        else:
            α = 0.8
            early_exaggeration = 1

        # Get Low Dimensional Affinities
        q_ij = get_low_dimensional_affinities(Y[t])

        # Get Gradient of Cost Function
```

```
        gradient = get_gradient(early_exaggeration * p_ij_symmetric, q

        # Update Rule
        Y[t + 1] = Y[t] - η * gradient + α * (Y[t] - Y[t - 1])  # Use

        # Compute current value of cost function
        if t % 50 == 0 or t == 1:
            cost = np.sum(p_ij_symmetric * np.log(p_ij_symmetric / q_i
            print(f"Iteration {t}: Value of Cost Function is {cost}")

    print(
        f"Completed Low Dimensional Embedding: Final Value of Cost Fun
    )
    solution = Y[-1]

    return solution, Y
```

```
# Fetch MNIST data
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
mnist.target = mnist.target.astype(np.uint8)

X_total = pd.DataFrame(mnist["data"])
y_total = pd.DataFrame(mnist["target"])

X_reduced = X_total.sample(n=1000)
y_reduced = y_total.loc[X_reduced.index]

# PCA to keep 30 components
X = PCA(n_components=30).fit_transform(X_reduced)
```

```
pd.DataFrame(X).head()
```

	0	1	2	3	4	5	6	7
0	251.341518	-172.337170	646.036607	-604.508676	-731.343394	-394.328252	140.586490	-213.8402
1	-314.851523	-260.686369	43.224837	-274.154795	-11.658326	-400.286229	619.657685	-473.6459
2	821.801836	-119.450767	-806.741011	970.782659	117.791691	-334.742575	243.026331	168.21878
3	-510.040645	62.922287	307.400985	165.162939	624.308344	-231.669371	-305.342884	97.953858
4	-14.882274	-1092.338219	-128.897715	-91.210132	-359.246382	276.104464	-110.414882	303.12374

```
X = np.array(X)

p_ij = get_original_pairwise_affinities(X)
```

```
Computing Pairwise Affinities....
Completed Pairwise Affinities Matrix.
```

```
pd.DataFrame(p_ij).head()
```

	0	1	2	3	4	5	6	7	8
0	4.940656e-324	1.821657e-07	1.037453e-12	1.014894e-09	1.117579e-07	3.180475e-08	5.791328e-12	6.567848e-08	1.30359 09
1	5.839298e-05	4.940656e-324	7.097552e-09	1.597654e-06	1.853781e-07	2.413588e-06	2.571279e-11	2.722544e-09	2.37096 04
2	2.525843e-08	2.063356e-07	4.940656e-324	2.985402e-06	6.375862e-06	1.414581e-05	1.077387e-07	1.502563e-06	5.86119 08
3	6.204282e-11	3.330044e-10	3.847147e-11	4.940656e-324	1.203690e-08	1.585765e-04	4.940876e-10	8.526250e-11	2.68843 05
4	2.696440e-07	6.902787e-10	2.903788e-09	2.739484e-07	4.940656e-324	2.587361e-06	4.634820e-11	5.112489e-09	4.75372 09

```
p_ij_symmetric = get_symmetric_p_ij(p_ij)
```

```
Computing Symmetric p_ij matrix....
Completed Symmetric p_ij Matrix.
```

```
pd.DataFrame(p_ij_symmetric).head()
```

	0	1	2	3	4	5	6	7	8
0	4.940656e-324	2.928757e-08	1.262973e-11	5.384684e-13	1.907009e-10	1.858131e-11	1.554646e-11	2.973141e-09	6.51798 13
1	2.928757e-08	4.940656e-324	1.067166e-10	7.989937e-10	9.303418e-11	1.207318e-09	5.313887e-13	1.533034e-12	1.18548 07
2	1.262973e-11	1.067166e-10	4.940656e-324	1.492720e-09	3.189383e-09	7.073305e-09	7.245097e-11	7.596714e-10	2.93059 11
3	5.384684e-13	7.989937e-10	1.492720e-09	4.940656e-324	1.429926e-10	1.765006e-07	2.690509e-09	4.785229e-11	1.34421 08
4	1.907009e-10	9.303418e-11	3.189383e-09	1.429926e-10	4.940656e-324	1.387253e-09	6.839854e-11	1.173887e-10	2.37686 12

```
y0 = initialization(X, 2, "random")
pd.DataFrame(y0).head()
```

	0	1
0	-0.000025	-0.000051
1	0.000034	0.000169
2	-0.000146	-0.000128
3	0.000018	-0.000113
4	-0.000106	-0.000102

```
q_ij = get_low_dimensional_affinities(y0)
pd.DataFrame(q_ij).head()
```

	0	1	2	3	4	5	6	7	8	9
0	4.940656e-324	1.001001e-06	1.001001e-06	1.001001e-06	1.001001e-06	0.000001	0.000001	0.000001	0.000001	0.00
1	1.001001e-06	4.940656e-324	1.001001e-06	1.001001e-06	1.001001e-06	0.000001	0.000001	0.000001	0.000001	0.00
2	1.001001e-06	1.001001e-06	4.940656e-324	1.001001e-06	1.001001e-06	0.000001	0.000001	0.000001	0.000001	0.00
3	1.001001e-06	1.001001e-06	1.001001e-06	4.940656e-324	1.001001e-06	0.000001	0.000001	0.000001	0.000001	0.00
4	1.001001e-06	1.001001e-06	1.001001e-06	1.001001e-06	4.940656e-324	0.000001	0.000001	0.000001	0.000001	0.00

```
gradient = get_gradient(p_ij_symmetric, q_ij, y0)
pd.DataFrame(gradient).head()
```

	0	1
0	3.440979e-08	7.813820e-09
1	-2.469521e-07	-5.151329e-07
2	3.244037e-07	9.484072e-08
3	4.419449e-07	-4.531208e-07
4	7.304593e-08	2.699164e-07

```
solution, Y = tsne(
    X, perplexity=10, T=1000, η=200, early_exaggeration=4, n_dimensions=
)
```

```
Computing Pairwise Affinities....
Completed Pairwise Affinities Matrix.
```

```
Computing Symmetric p_ij matrix....
Completed Symmetric p_ij Matrix.

Optimizing Low Dimensional Embedding....
Iteration 1: Value of Cost Function is 4.445604299609592
Iteration 50: Value of Cost Function is 2.7049394197024554
Iteration 100: Value of Cost Function is 2.411386294688672
Iteration 150: Value of Cost Function is 2.370099941722107
Iteration 200: Value of Cost Function is 2.362135171508304
Iteration 250: Value of Cost Function is 2.35928319350489
Iteration 300: Value of Cost Function is 1.2494410859712
Iteration 350: Value of Cost Function is 1.157905121298321
Iteration 400: Value of Cost Function is 1.12892053939353
Iteration 450: Value of Cost Function is 1.1142709861063884
Iteration 500: Value of Cost Function is 1.1046409694887442
Iteration 550: Value of Cost Function is 1.0978103425607435
```

```
pd.DataFrame(solution)
```

	0	1
0	-150.759538	-177.678641
1	74.161478	34.137389
2	98.119584	-164.411579
3	-54.590937	-35.019846
4	-184.648020	85.079192
...	...	...
995	165.020130	128.444057
996	191.895674	-122.030675
997	-190.773637	121.281332
998	-123.115617	2.636375
999	-116.233125	-148.644828

1000 rows × 2 columns

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
g1 = ax.scatter(solution[:, 0], solution[:, 1], c=y_reduced, cmap="tab10
ax.axis("off")
ax.set_title("MNIST t-SNE")
plt.colorbar(g1, ax=ax)
plt.show()
```

Download