Dimensionality Reduction Programming Practice Questions

Question 1: "Using `.toarray()` means that the numbers in the matrix remain the same, but only the shape changes, right?"

Answer:

Yes, that's exactly right! When you use `.toarray()`, the numbers themselves do not change, but the **way the data is stored** does. What's important here is that the **structure or format of the data** can affect the **efficiency of computations** and the **applicability of different algorithms**.

Differences Between Sparse Matrix and Dense Matrix

1. **Sparse Matrix** is a special format used for storing data with many zeroes, where only the non-zero values are stored. This allows for efficient memory and computational management, which is particularly useful for very large datasets. For instance, in text data where many words do not appear in a given document, the corresponding values will be zero, making sparse matrix storage more suitable.
2. On the other hand, a **Dense Matrix** stores all the values in memory, including the zeros. This is the standard format used in general matrix operations.

Why Use `.toarray()`?

`.toarray()` is a function that converts a **sparse matrix** to a **dense matrix**. Algorithms like PCA are designed to work with **dense matrices**, so if the data is stored as a sparse matrix, **the algorithm might not function properly**. Therefore, before applying PCA, `.toarray()` is used to **convert the sparse matrix to a dense matrix**.

However, the numeric values do not change. What changes is the **storage method**. In a sparse matrix, zeros are omitted, while in a dense matrix, all values including zeros are stored.

- **Sparse Matrix:**

- In a sparse matrix, many values are zeros, and only the non-zero values are stored along with their positions. For example, consider the following matrix:

  ```
  Original matrix:
  [ [0, 0, 3],
    [4, 0, 0],
    [0, 0, 0] ]
  ```

  As a sparse matrix, it would be represented as:

  ```
  [(0, 2, 3), (1, 0, 4
  ```

  Here, each tuple represents `(row index, column index, value)`, storing only the positions and values of the non-zero elements.

- **Dense Matrix:**
  - In a dense matrix, all values are stored, including the zeros. The same matrix as a dense matrix would look like this:

    ```
    [ [0, 0, 3],
      [4, 0, 0],
      [0, 0, 0] ]
    ```

In summary, a dense matrix stores every value, while a sparse matrix efficiently stores only the non-zero values and their locations. When using `.toarray()`, a sparse matrix is converted to a dense matrix, storing all values, including zeros, in memory.

```
from sklearn.manifold import TSNE

tfidf = TfidfVectorizer(tokenizer=tokenizer)
X_train_tfidf = tfidf.fit_transform(X_train)  # Transform the
X_test_tfidf = tfidf.transform(X_test)  # Transform the test

tsne = TSNE(n_components=2, random_state=7, init='random')
tsne_tfidf = tsne.fit_transform(X_train_tfidf)
print('TSNE dimension:', tsne_tfidf.shape)
```

```
def tsne_graph(tsne_2, label, lim=None):
    colors = {0:'blue', 1:'red', 2:'green', 3:'purple'}
    x = tsne_2[:,0]  # Use the first compressed dimension as
    y = tsne_2[:,1]  # Use the second compressed dimension as
    plt.figure(figsize=(15,10))
    if lim == None:
        lim = [min(x), max(x), min(y), max(y)]
    plt.xlim(lim[0], lim[1])
    plt.ylim(lim[2], lim[3])
    # for i in range(500):
    for i in range(len(x)):
        # Output text on the graph with color based on the y
        if (lim[0] < x[i] < lim[1]) and (lim[2] < y[i] < lim[
            plt.text(x[i], y[i], label[i], color=colors[label
    plt.show()

tsne_graph(tsne_tfidf, y_train)
```

```
svd = TruncatedSVD(n_components=100, random_state=1)  # Speci
X_train_lsa = svd.fit_transform(X_train_tfidf)
X_test_lsa = svd.transform(X_test_tfidf)

print('LSA Converted X shape:', X_train_lsa.shape)
# Print the sum of the explained variance ratio to check how
print('Sum of explained variance ratio: {:.3f}'.format(svd.ex

tsne_lsa = tsne.fit_transform(X_train_lsa)
print('TSNE dimension:', tsne_lsa.shape)
print('#Selected categories:', newsgroups_train.target_names)

tsne_graph(tsne_lsa, y_train)
```

Question 2-1: Are the number of data points the same in both codes?

The number of data points remains the same in both codes.

Both codes perform dimensionality reduction on the same training dataset `X_train`. The **number of data points (samples)** does not change across the codes since the same text data is being used. The difference lies in the **dimensionality reduction approach** applied.

- **First Code**: t-SNE is applied directly to `X_train_tfidf` to reduce the high-dimensional data to two dimensions.
- **Second Code**: LSA (SVD) is first used to reduce the data to 100 dimensions, and then t-SNE is applied to further reduce it to two dimensions.

Thus, although both codes use the same dataset with the same number of data points, the approach to dimensionality reduction differs.

Question 2-2: Why does the distribution of data points look different in the visualizations? What is the difference?

The reason the visualized distributions of the data points look different is due to the characteristics of t-SNE and the differences in the dimensionality reduction techniques used.

Differences and Causes:

1. **Dimensionality Reduction Approach**
    - **First Code**: t-SNE is applied directly to the TF-IDF vectors, **reducing the high-dimensional data to two dimensions**. As a result, some data points may be **densely clustered or overlapped**, as t-SNE tries to preserve the complex structure of the original high-dimensional space. Thus, not all data points may be evenly distributed in the visualization.
    - **Second Code**: The data is first **reduced to 100 dimensions using LSA (SVD)** before applying t-SNE, which reduces it to two dimensions. Since LSA simplifies the data's complexity beforehand, t-SNE processes a less complex dataset, allowing **important patterns to be emphasized**. This can make the relationships between data points **more clearly visible**.
2. **t-SNE Characteristics**
    - **t-SNE** aims to maintain the proximity of data points in the high-dimensional space when projecting them into lower dimensions. However, during this process, **clusters may appear denser** or the **distribution may look uneven**.

- **First Code** deals with the complex structure of high-dimensional data directly, resulting in **larger variations in the distance between points or dense regions**.
- In contrast, **Second Code** uses LSA to pre-reduce the data's dimensionality, making it easier for t-SNE to **produce a more evenly distributed visualization**.

Summary:

- **First Code** applies t-SNE directly to high-dimensional data, which may result in **denser clusters or overlapping data points**.
- **Second Code** uses LSA for dimensionality reduction before t-SNE, resulting in **a more evenly distributed and easily distinguishable visualization**.

Conclusion: Both codes use the same number of data points, but the distribution of points in the visualizations differs due to the differences in the dimensionality reduction methods. **Applying LSA before t-SNE** simplifies the data, leading to a more **evenly distributed pattern**, while **the first code deals directly with the high-dimensional complexity**, potentially resulting in **denser or less visible clusters**.