You:

Create a Python class called PrincipalComponentAnalysis that implements the Principal Component Analysis algorithm from scratch. During initialization, the class should take two arguments: * n_components, if not specified should default to all. * decomposition_method should be either 'eigen' or 'svd'. If not specified it should default to 'eigen'. Add a fit method that takes one input, X, that is either a numpy array, pandas dataframe or a list of lists as input. The fit method should compute the eigen values and eigen vectors (components) for the inputted data, and stores them on the object. If n_components is specified the fit method should store only top n_components eigen values and eigen vectors. In the fit, the method should also compute the explained variance ratio, and cumulative sum of explained variance ratio for each component and store it on the object. Also add a transform method that takes one input, X, that is either a numpy array, pandas dataframe or a list of lists as input. The transform method should project the inputted data onto the components and return the projected data. The transform method should also take an optional argument, n_components, that specifies the number of components to project onto. If n_components is not specified it should default to all. The last method to add is a fit_transform method that takes one input, X, that is either a numpy array, pandas dataframe or a list of lists as input. Like transform, this method should also take an optional n_components, specifying n_components, otherwise use all. The fit_transform method should be called the fit method and then the transform method. Make sure to add documentation to the class and methods in the form of docstrings and comments. Also, make sure to add type hints to the methods.

ChatGPT:

Creating a Python class for Principal Component Analysis (PCA) from scratch involves implementing the algorithm, typically using eigenvalue decomposition or singular value decomposition (SVD). Below is an implementation in Python using the eigenvalue decomposition approach as default, with the flexibility to use SVD as well.

```
python
import numpy as np
import pandas as pd
from typing import Union, Optional, List

class PrincipalComponentAnalysis:
    """
    Principal Component Analysis (PCA) implementation.

    Parameters:
    n_components: int, optional
        Number of components to keep. If n_components is not set or None,
        all components are kept.
    decomposition_method: str, optional
        The method used for decomposition. Can be either 'eigen' or 'svd'.
        Defaults to 'eigen'.

    Attributes:
    components_: array, shape (n_components, n_features)
        Principal axes in feature space.
    explained_variance_: array, shape (n_components,)
        The amount of variance explained by each of the selected components.
    explained_variance_ratio_: array, shape (n_components,)
        Percentage of variance explained by each of the selected components.
    cumulative_explained_variance_ratio_: array, shape (n_components,)
        Cumulative sum of explained variance ratio.
    """

    def __init__(self, n_components: Optional[int] = None, decomposition_method: str = 'eigen'):
        self.n_components = n_components
        self.decomposition_method = decomposition_method
        self.components_ = None
        self.explained_variance_ = None
        self.explained_variance_ratio_ = None
        self.cumulative_explained_variance_ratio_ = None

    def fit(self, X: Union[np.ndarray, pd.DataFrame, List[List[float]]]):
        """
        Fit the model with X.

        Parameters:
        X: array-like, shape (n_samples, n_features)
            Training data, where n_samples is the number of samples
            and n_features is the number of features.
        """
        if isinstance(X, list):
            X = np.array(X)
        elif isinstance(X, pd.DataFrame):
            X = X.to_numpy()

        # Standardize data
        X = X - np.mean(X, axis=0)

        # Compute covariance matrix
        cov_matrix = np.cov(X, rowvar=False)
        # Eigenvalue decomposition or SVD
        if self.decomposition_method == 'eigen':
            eigen_values, eigen_vectors = np.linalg.eigh(cov_matrix)
        elif self.decomposition_method == 'svd':
            U, S, V = np.linalg.svd(X.T)
            eigen_values = S ** 2 / (len(X) - 1)
            eigen_vectors = U
        else:
            raise ValueError("Decomposition method must be either 'eigen' or 'svd'.")

        # Sort eigenvalues and eigenvectors
        idx = np.argsort(eigen_values)[::-1]
        eigen_values = eigen_values[idx]
        eigen_vectors = eigen_vectors[:, idx]

        # Select n_components
        if self.n_components is not None:
            eigen_values = eigen_values[:self.n_components]
            eigen_vectors = eigen_vectors[:, :self.n_components]

        self.components_ = eigen_vectors.T
        self.explained_variance_ = eigen_values
        self.explained_variance_ratio_ = eigen_values / eigen_values.sum()
        self.cumulative_explained_variance_ratio_ = np.cumsum(self.explained_variance_ratio_)

    def transform(self, X: Union[np.ndarray, pd.DataFrame, List[List[float]]], n_components: Optional[int] = N
        """
        Apply dimensionality reduction to X.

        X is projected on the first principal components previously extracted
        from a training set.

        Parameters:
        X: array-like, shape (n_samples, n_features)
            New data, where n_samples is the number of samples
            and n_features is the number of features.
        n_components: int, optional
            Number of components to project onto. If not specified, use all.

        Returns:
        X_new: array-like, shape (n_samples, n_components)
            Transformed values.
        """
        if isinstance(X, list):
            X = np.array(X)
        elif isinstance(X, pd.DataFrame):
            X = X.to_numpy()

        X = X - np.mean(X, axis=0)

        if n_components is None:
            n_components = self.n_components

        X_transformed = np.dot(X, self.components_[:n_components].T)
        return X_transformed

    def fit_transform(self, X: Union[np.ndarray, pd.DataFrame, List[List[float]]], n_components: Optional[int]
        """
        Fit the model with X and apply the dimensionality reduction on X.

        Parameters:
        X: array-like, shape (n_samples, n_features)
            Training data, where n_samples is the number of samples
            and n_features is the number of features.
        n_components: int, optional
            Number of components to project onto. If not specified, use all.

        Returns:
        X_new: array-like, shape (n_samples, n_components)
            Transformed values.
        """
        self.fit(X)
        return self.transform(X, n_components)
```

This implementation provides a basic structure for PCA. The `fit` method computes the principal components, explained variance, and explained variance ratio. The `transform` method projects new data onto these components, and `fit_transform` combines these two steps. This class handles input in the form of numpy arrays, pandas dataframes, or lists of lists, providing flexibility for different data types.
