



You:

Create a python class called `PrincipalComponentAnalysis` that implements the Principal Component Analysis algorithm from scratch. It should have three methods, `fit`, `transform` and `fit_transform`. Two arguments should be able to be passed, first is `n_components` which specifies the number of components to use. If `n_components` is not specified it should default to all. Second is `decomposition_method`, which should be either 'eigen' or 'svd'. If not specified it should default to 'eigen'. 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 `PrincipalComponentAnalysis` class in Python that implements the Principal Component Analysis (PCA) algorithm from scratch involves several steps. The class will have three methods: `fit`, `transform`, and `fit_transform`. It will accept two arguments: `n_components` (defaulting to 'all' if not specified) and `decomposition_method` (defaulting to 'eigen' if not specified). Let's implement this class along with detailed docstrings and type hints:

```
python
import numpy as np
from typing import Optional, Tuple, Union

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

    Parameters
    -----
    n_components : int, optional
        Number of components to keep. If not specified, all components are kept.
    decomposition_method : str, optional
        Method to perform decomposition, 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,)
        Amount of variance explained by each of the selected components.
    mean_ : array, shape (n_features,)
        Per-feature empirical mean, estimated from the training set.
    """

    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.mean_ = None

    def fit(self, X: np.ndarray) -> 'PrincipalComponentAnalysis':
        """
        Fit the model with X.

        Parameters
        -----
        X : array-like, shape (n_samples, n_features)
            Training data.

        Returns
        -----
        self : object
            Returns the instance itself.
        """
        self.mean_ = np.mean(X, axis=0)
        X_centered = X - self.mean_

        if self.decomposition_method == 'eigen':
            cov_matrix = np.cov(X_centered, rowvar=False)
```

```

        eigenvalues, eigenvectors = np.linalg.eigh(cov_matrix)
    elif self.decomposition_method == 'svd':
        eigenvectors, singular_values, _ = np.linalg.svd(X_centered, full_matrices=False)
        eigenvalues = singular_values ** 2 / (X.shape[0] - 1)
    else:
        raise ValueError("decomposition_method should be either 'eigen' or 'svd'")

    # Sort eigenvectors by eigenvalues in descending order
    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    # Select the top n_components
    if self.n_components is not None:
        eigenvalues = eigenvalues[:self.n_components]
        eigenvectors = eigenvectors[:, :self.n_components]

    self.components_ = eigenvectors.T
    self.explained_variance_ = eigenvalues

    return self

def transform(self, X: np.ndarray) -> np.ndarray:
    """
    Apply dimensionality reduction to X.

    Parameters
    -----
    X : array-like, shape (n_samples, n_features)
        Data to transform.

    Returns
    -----
    X_new : array-like, shape (n_samples, n_components)
        Transformed data.
    """
    X_centered = X - self.mean_
    return np.dot(X_centered, self.components_.T)

def fit_transform(self, X: np.ndarray) -> np.ndarray:
    """
    Fit the model with X and apply the dimensionality reduction on X.

    Parameters
    -----
    X : array-like, shape (n_samples, n_features)
        Training data.

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

```

This class can now be used to perform PCA on a dataset. It's important to ensure that the input data (X) is properly preprocessed and standardized if necessary before applying PCA.
