```
import numpy as np

class FeatureReduction():

    def __init__(self, data=None):
        self.model_params = {'Projection Matrix': None}
        self.data = data
    def fit(self, data):
        pass

    def predict(self, data):
        pass


class PrincipleComponentAnalysis(FeatureReduction):
    '''self.model_params is where you will save your principle components (up to 
LoV)'''
    ''' Its useful to use a projection matrix as your only param'''

    def __init__(self, data=None):
        super().__init__(data)

    def fit(self, thresh=0.95, plot_var = True):
       '''Find the principle components of your data'''
       dataset = self.data

       target_pandas = dataset.iloc[:, -1]
       data_pandas = dataset.iloc[:, :-1]

       target = target_pandas.values
       data = data_pandas.values

       covariance_matrix = self.covariance(data)

       eigen_values, eigen_vectors = np.linalg.eig(covariance_matrix)

       variance_explained = self.calc_variance_explained(eigen_values)
       # Calculates the cumulative sum of the explained_variance_ratio vector aka 
the normalized and sorted eigen values
       cumulative_variance_ratio = np.cumsum(variance_explained)

       n_components = np.argmax(cumulative_variance_ratio >= thresh) + 1

       number_cols = np.argsort(eigen_values)[::-1][:n_components]

       #Selected the most important classes
       selected_vectors = eigen_vectors[:, number_cols]

    #    standardized_data = self.standardize_dataset(data)
```

```
#    projected = np.dot(standardized_data, selected_vectors)

   self.model_params['Projection Matrix'] = selected_vectors

   return self.model_params['Projection Matrix']


def predict(self, data):
    ''' You can change this function if you want'''
    standardized_data = self.standardize_dataset(data)
    return np.dot(standardized_data, self.model_params['Projection Matrix'])


def standardize_dataset(self, data):
    centered_data = data - np.mean(data, axis = 0)
    std_data = np.std(data, axis=0)

    return centered_data/std_data

def recurr_np_sample_mean(self, np_array):
    if(np_array.ndim == 1):
        return np_array
    elif (isinstance(np_array, np.ndarray)  and np_array.ndim > 1):
        return np.sum(self.recurr_np_sample_mean(elem) for elem in np_array)
    else:
        raise ValueError("Unsupported Data Type")

def sampleMean(self, np_array, axis=0):
    ''' Each column represents a feature'''
    if not isinstance(np_array, (np.ndarray)):
        raise Exception('Wrong Data Type. Required np.ndarray')

    if axis == 1:
        np_array = np_array.T
    # concat_np_array = np.empty((np_array.size))
    ans = self.recurr_np_sample_mean(np_array)

    ans = ans/np_array.shape[0]

    if axis == 1:
        ans = ans.T

    return ans

def covariance(self, np_array):
    ''' Each column represents a feature'''
    if not isinstance(np_array, (np.ndarray)):
        raise Exception('Wrong Data Type. Required np.ndarray')

    sample_mean_array = self.sampleMean(np_array, axis=0)
```

```
        centered_data = np_array - sample_mean_array

        ans = np.dot(centered_data.T, centered_data) / (centered_data.shape[0] - 1)

        return ans

    def calc_variance_explained(self, eigen_values):
        '''Input: list of eigen values
           Output: list of normalized values corresponding to percentage of
information an eigen value contains'''
        '''Your code here'''
        sorted_indices = np.argsort(eigen_values)[::-1]
        sorted_eigen_values = eigen_values[sorted_indices]

        # Computes the normalized eigen values
        variance_explained = sorted_eigen_values / np.sum(sorted_eigen_values)

        '''Stop Coding here'''
        return variance_explained
```