```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from scipy import stats
from sklearn.metrics import silhouette_score


def read_clean_transpose_data(file_path):
    """
    Read original data from a CSV file, clean the data by handling NaN values,
    and transpose the data for analysis.

    Parameters:
    - file_path (str): Path to the CSV file containing the data.

    Returns:
    - original_data (pd.DataFrame): Original data read from the CSV file.
    - cleaned_data (pd.DataFrame): Cleaned data with NaN values handled.
    - transposed_data (pd.DataFrame): Transposed version of the cleaned data.
    """
    # Read original data
    original_data = pd.read_csv(file_path)

    # Select relevant columns for cleaning and analysis
    columns_for_analysis = ['Time' ,
                            'Water productivity, total (constant 2015 US$ GDP per cubic
meter of total freshwater withdrawal) [ER.GDP.FWTL.M3.KD]' ,
                            'Adjusted net national income (annual % growth)
[NY.ADJ.NNTY.KD.ZG]' ,
                            'Adjusted savings: education expenditure (% of GNI)
[NY.ADJ.AEDU.GN.ZS]' ,
                            'Adjusted savings: natural resources depletion (% of GNI)
[NY.ADJ.DRES.GN.ZS]']

    # Replace non-numeric values with NaN
    original_data[columns_for_analysis] = \
        original_data[columns_for_analysis].apply(pd.to_numeric , errors =
'coerce')

    # Drop rows with missing values in selected columns
    cleaned_data = original_data.dropna(subset = columns_for_analysis).copy()

    # Transpose the cleaned data for analysis
    transposed_data = cleaned_data.transpose()
```

```
    return original_data , cleaned_data , transposed_data


def FittingModel(x , a , b):
    """
        Fit an exponential model to the input data.

        Parameters:
        - x (array-like): Independent variable values.
        - a (float): Amplitude parameter of the exponential model.
        - b (float): Decay or growth parameter of the exponential model.

        Returns:
        - array-like: Modeled values based on the exponential model.
        """
    return a * np.exp(b * x)


def perform_clustering(data , columns_for_clustering):
    """
    Perform clustering on the provided data using k-means.

    Parameters:
    - data (pd.DataFrame): Input data for clustering.
    - columns_for_clustering (list): List of columns used for clustering.

    Returns:
    - clustered_data (pd.DataFrame): Data with cluster labels assigned.
    - kmeans (KMeans): Fitted k-means model.
    """
    # Normalize data for clustering
    normalized_data = \
        (data[columns_for_clustering] - data[columns_for_clustering].mean()) /
data[columns_for_clustering].std()

    # Perform clustering (example with k-means)
    kmeans = KMeans(n_clusters = 4)
    data['Cluster'] = kmeans.fit_predict(normalized_data)

    return data, kmeans


def confidenceInterval(predicted_values , std_dev , confidence = 0.95):
    """
        Calculate the confidence interval for predicted values.
```

```
        Parameters:
        - predicted_values (array-like): The predicted values for which confidence
intervals are calculated.
        - std_dev (array-like): Standard deviation of the predicted values.
        - confidence (float, optional): Confidence level for the interval. Default is
0.95.

        Returns:
        - tuple: Lower and upper bounds of the confidence interval.
        """
    z_score = np.abs(stats.norm.ppf((1 - confidence) / 2))
    lower = predicted_values - z_score * std_dev
    upper = predicted_values + z_score * std_dev
    return lower , upper


def fit_curve_and_visualize(original_data , x_column , y_column):
    """
    Fit a curve to the data and visualize the fitted model with confidence interval.

    Parameters:
    - original_data (pd.DataFrame): Original data.
    - x_column (str): Column representing the x-axis variable.
    - y_column (str): Column representing the y-axis variable.
    """
    # Handle NaN or infinite values in the data for curve fitting
    mask = ~np.isnan(original_data[x_column]) &
~np.isnan(original_data[y_column])
    x_data = original_data[x_column][mask].astype(float)
    y_data = original_data[y_column][mask].astype(float)

    # Provide initial parameter guesses
    initial_guesses = [1.0 , 0.01]

    # Fit the model to the data with initial guesses and increased maxfev
    params , covariance = curve_fit(FittingModel , x_data , y_data , p0 =
initial_guesses , maxfev = 2000)

    # Predict future values
    future_years = np.arange(2000 , 2042 , 1)
    predicted_values = FittingModel(future_years , *params)

    # Estimate confidence intervals
    std_dev = np.sqrt(np.diag(covariance))
    lower , upper = confidenceInterval(predicted_values , std_dev[0] ,
confidence = 0.95)
```

```
    # Visualize the fitted model and confidence range
    plt.scatter(original_data[x_column] , original_data[y_column] , label = 'Actual
Data')
    plt.plot(future_years , predicted_values , label = 'Fitted Model' , color = 'red')
    plt.fill_between(future_years , lower , upper , color = 'red' , alpha = 0.2 ,
                    linewidth = 10 , label = 'Confidence Interval')
    plt.xlabel(x_column)
    plt.ylabel(y_column)
    plt.title(f'Fitted Model with Confidence Interval for {y_column}')
    plt.legend()
    plt.show()


# Load the data using the new function
original_data , cleaned_data , transposed_data = \

read_clean_transpose_data('caa036dd-4f78-4dd4-81bd-6e9fbfd3c9b8_Data.
csv')

columns_for_clustering = ['Water productivity, total (constant 2015 US$ GDP
per cubic meter of total freshwater withdrawal) [ER.GDP.FWTL.M3.KD]' ,
                          'Adjusted net national income (annual % growth)
[NY.ADJ.NNTY.KD.ZG]' ,
                          'Adjusted savings: education expenditure (% of GNI)
[NY.ADJ.AEDU.GN.ZS]' ,
                          'Adjusted savings: natural resources depletion (% of GNI)
[NY.ADJ.DRES.GN.ZS]']

# Perform clustering
cleaned_data , kmeans_model = perform_clustering(cleaned_data ,
columns_for_clustering)

# Calculate and print silhouette score
silhouette_avg = \
    silhouette_score((cleaned_data[columns_for_clustering] -
cleaned_data[columns_for_clustering].mean()) /
cleaned_data[columns_for_clustering].std() ,
                    cleaned_data['Cluster'])
print(f"Silhouette Score: {silhouette_avg}")

# Visualize cluster centers
plt.scatter(cleaned_data[columns_for_clustering[0]] ,
            cleaned_data[columns_for_clustering[1]] ,
            c = cleaned_data['Cluster'] , cmap = 'viridis')
plt.scatter(kmeans_model.cluster_centers_[: , 0] ,
kmeans_model.cluster_centers_[: , 1] , marker = 'x' ,
```

```
            s = 200 , linewidths = 3 , color = 'red')
plt.xlabel('Water productivity')
plt.ylabel('Adjusted net national income')
plt.title('Clustering of Countries with Cluster Centers')
plt.show()

# Fit curve and visualize for 'Water productivity'
fit_curve_and_visualize(original_data , 'Time' ,
                        'Water productivity, total (constant 2015 US$ GDP per cubic
meter of total freshwater withdrawal) [ER.GDP.FWTL.M3.KD]')
```