Overview of N-BaIoT_ylabel_clustering

The **N-BaIoT dataset** consists of network traffic data from IoT devices in both benign (normal) and attack (malicious) scenarios. It’s used to develop models for detecting security threats like DDoS and port scans in IoT networks.

In this context, **ylabel** refers to the true labels in the dataset, indicating whether a traffic sample is normal (0) or an attack (1). **Clustering** involves grouping data points based on similarities without using labels.

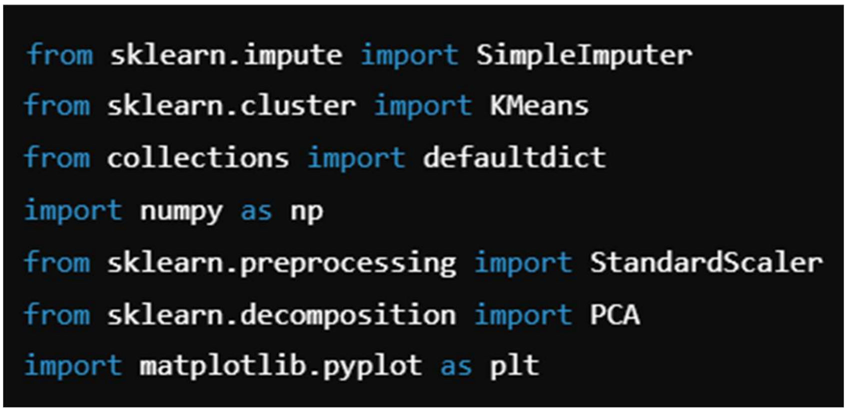
ylabel_clustering applies clustering techniques (e.g., k-means, DBSCAN) to the labelled data, aiming to group traffic patterns into clusters that reflect the true labels. It helps in:

1. **Anomaly Detection**: Identifying new or unknown attack patterns based on traffic behaviour.
2. **Evaluation**: Comparing clustering results with true labels to assess model performance.
3. **Hybrid Models**: Combining clustering and supervised methods to improve detection.

For example, clustering the traffic from smart home IoT devices can help differentiate normal and malicious behaviour, which can then be compared to the labels for validation.

Break down the of code for detailed understanding:

1. **Import Statements**



```
from sklearn.impute import SimpleImputer
from sklearn.cluster import KMeans
from collections import defaultdict
import numpy as np
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
```

- **from sklearn.impute import SimpleImputer**: Imports the SimpleImputer class from sklearn.impute for handling missing values. This line is included but not used in the provided code.
- **from sklearn.cluster import KMeans**: Imports the KMeans class from sklearn.cluster, which is used for performing K-means clustering.
- **from collections import defaultdict**: Imports defaultdict from the collections module. It is a type of dictionary that provides a default value for non-existent keys.

- **import numpy as np**: Imports the numpy library and aliases it as np. This library is used for numerical operations and array handling.
- **from sklearn.preprocessing import StandardScaler**: Imports StandardScaler from sklearn.preprocessing to standardize features by removing the mean and scaling to unit variance.
- **from sklearn.decomposition import PCA**: Imports PCA from sklearn.decomposition for dimensionality reduction.
- **import matplotlib.pyplot as plt**: Imports matplotlib.pyplot and aliases it as plt. This is used for creating visualizations like plots.

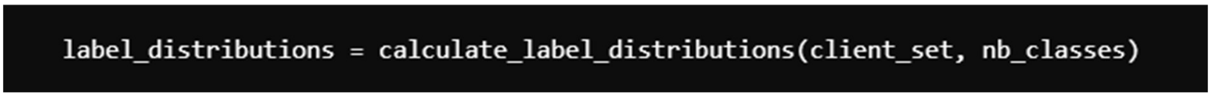
2. Function Definition

```
def perform_clustering_based_on_labels(client_set, nb_classes):
```

def perform_clustering_based_on_labels(client_set, nb_classes):: Defines a function named perform_clustering_based_on_labels that takes two arguments:

- client_set: A dictionary where each key is a client name and each value is client data.
- nb_classes: The number of classes in the data.

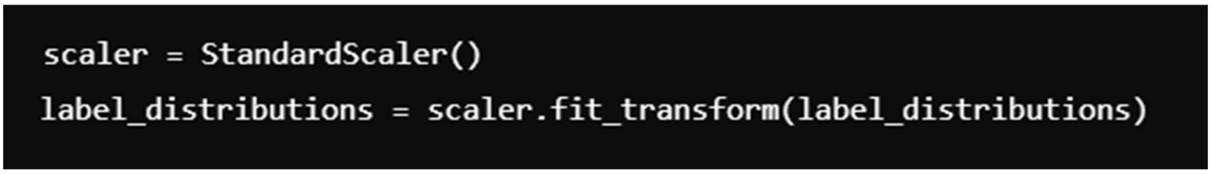
Calculate Label Distributions:



```
    label_distributions = calculate_label_distributions(client_set, nb_classes)
```

label_distributions: Calls a function calculate_label_distributions (assumed to be defined elsewhere) to compute the distribution of labels for each client. The result is stored in label_distributions.

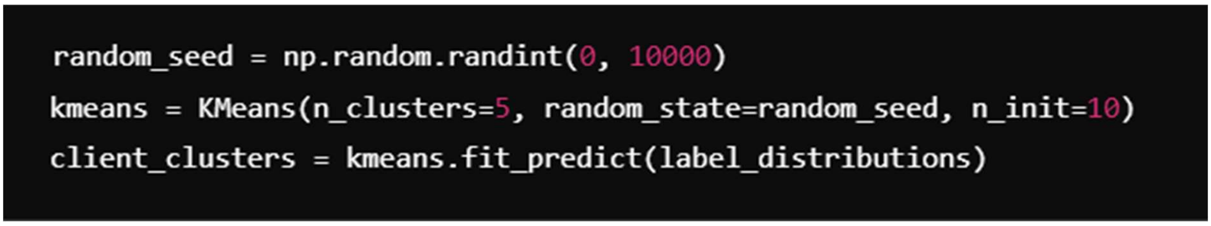
Standardizing the Data:



```
    scaler = StandardScaler()
    label_distributions = scaler.fit_transform(label_distributions)
```

- **scaler = StandardScaler()**: Initializes a StandardScaler object.
- **label_distributions = scaler.fit_transform(label_distributions)**: Standardizes the label_distributions by removing the mean and scaling to unit variance. The fit_transform method both fits the scaler and transforms the data.

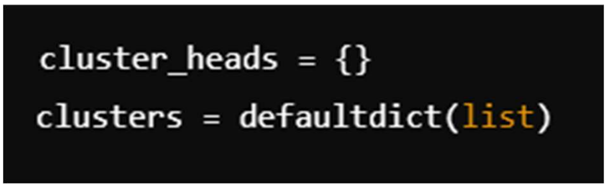
Performing K-means Clustering:



```
random_seed = np.random.randint(0, 10000)
kmeans = KMeans(n_clusters=5, random_state=random_seed, n_init=10)
client_clusters = kmeans.fit_predict(label_distributions)
```

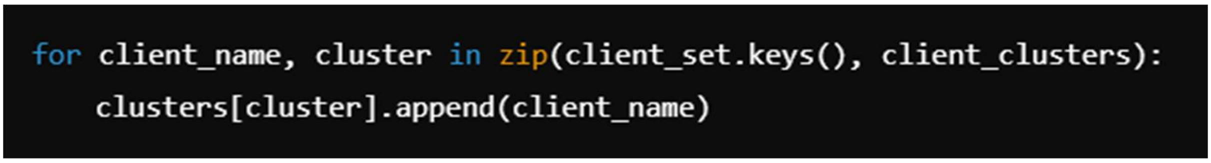
- **random_seed = np.random.randint(0, 10000)**: Generates a random integer between 0 and 10,000 to use as the seed for reproducibility.
- **kmeans = KMeans(n_clusters=5, random_state=random_seed, n_init=10)**: Initializes a KMeans object to perform clustering with 5 clusters. The random_state ensures reproducibility, and n_init=10 specifies the number of times the algorithm will be run with different centroid seeds.
- **client_clusters = kmeans.fit_predict(label_distributions)**: Fits the K-means algorithm to the standardized data and predicts the cluster assignments for each client. The result is an array of cluster labels.

Assigning Cluster Heads:



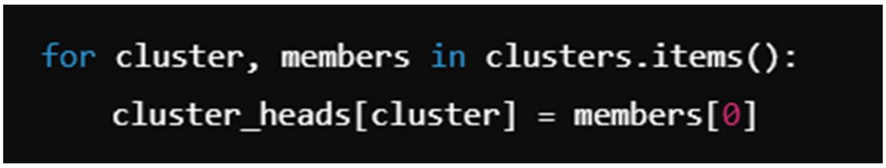
```
cluster_heads = {}
clusters = defaultdict(list)
```

- **cluster_heads = {}**: Initializes an empty dictionary to store the head (a representative) of each cluster.
- **clusters = defaultdict(list)**: Initializes a defaultdict of lists to collect clients into their respective clusters



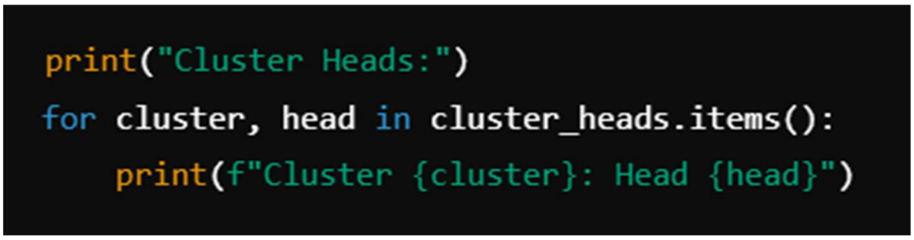
```
for client_name, cluster in zip(client_set.keys(), client_clusters):
    clusters[cluster].append(client_name)
```

- **for client_name, cluster in zip(client_set.keys(), client_clusters):**: Iterates over the client names and their corresponding cluster assignments.
- **clusters[cluster].append(client_name)**: Appends each client to the list corresponding to their assigned cluster.



```python
for cluster, members in clusters.items():
    cluster_heads[cluster] = members[0]
```

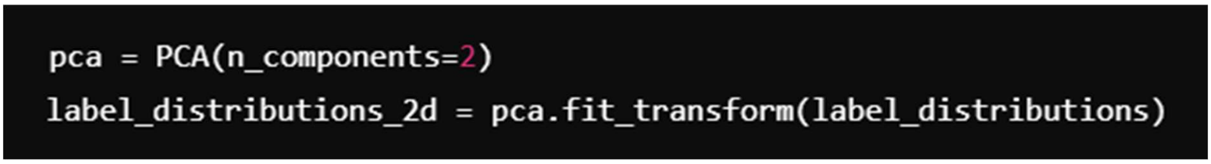
- **for cluster, members in clusters.items():**: Iterates over the clusters and their member clients.
- **cluster_heads[cluster] = members[0]**: Sets the first client in each cluster as the cluster head.



```python
print("Cluster Heads:")
for cluster, head in cluster_heads.items():
    print(f"Cluster {cluster}: Head {head}")
```

- **print("Cluster Heads:")**: Prints a header for the cluster heads information.
- **for cluster, head in cluster_heads.items():**: Iterates over the cluster heads.
- **print(f"Cluster {cluster}: Head {head}")**: Prints the cluster number and the name of the cluster head.

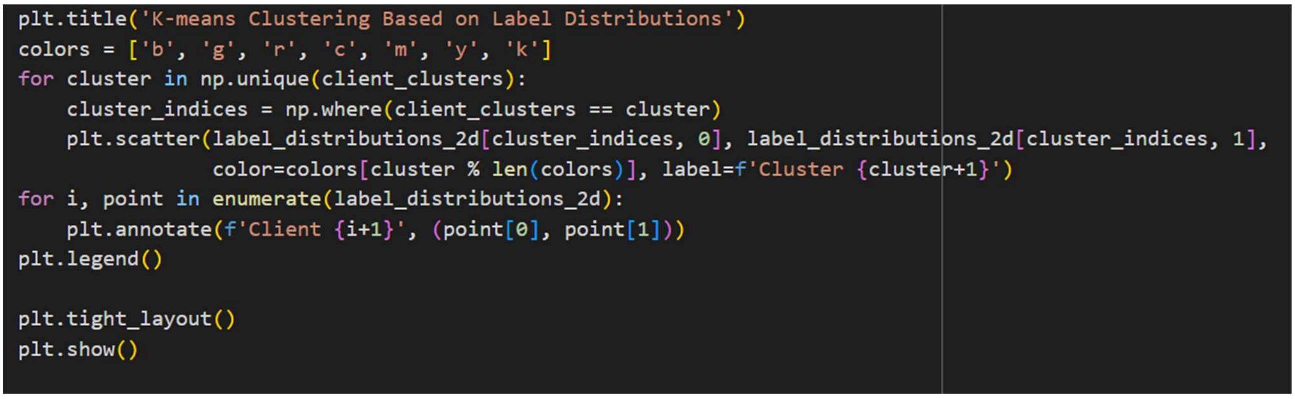
Dimensionality Reduction for Visualization:



```python
pca = PCA(n_components=2)
label_distributions_2d = pca.fit_transform(label_distributions)
```

- **pca = PCA(n_components=2)**: Initializes a PCA object to reduce the data to 2 dimensions.
- **label_distributions_2d = pca.fit_transform(label_distributions)**: Fits the PCA model and transforms the standardized data to 2D.

Visualization:

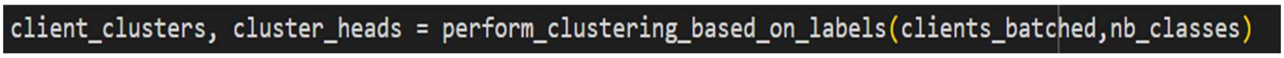


```python
plt.title('K-means Clustering Based on Label Distributions')
colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']
for cluster in np.unique(client_clusters):
    cluster_indices = np.where(client_clusters == cluster)
    plt.scatter(label_distributions_2d[cluster_indices, 0], label_distributions_2d[cluster_indices, 1],
                color=colors[cluster % len(colors)], label=f'Cluster {cluster+1}')
for i, point in enumerate(label_distributions_2d):
    plt.annotate(f'Client {i+1}', (point[0], point[1]))
plt.legend()

plt.tight_layout()
plt.show()
```

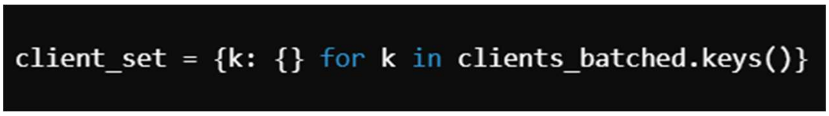
- **plt.figure(figsize=(16, 8))**: Creates a new figure for plotting with a specified size.
- **plt.title('K-means Clustering Based on Label Distributions')**: Sets the title for the plot.
- **colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k']**: Defines a list of colors for plotting.
- **for cluster in np.unique(client_clusters):**: Iterates over the unique cluster labels.
- **cluster_indices = np.where(client_clusters == cluster)**: Finds the indices of clients in the current cluster.
- **plt.scatter(label_distributions_2d[cluster_indices, 0], label_distributions_2d[cluster_indices, 1], color=colors[cluster % len(colors)], label=f'Cluster {cluster+1}')**: Plots the data points for the current cluster in a scatter plot with a specific color.
- **for i, point in enumerate(label_distributions_2d):**: Iterates over the 2D data points.
- **plt.annotate(f'Client {i+1}', (point[0], point[1]))**: Annotates each data point with the client number.
- **plt.legend()**: Adds a legend to the plot.
- **plt.tight_layout()**: Adjusts the layout to fit the plot elements neatly.
- **plt.show()**: Displays the plot.

3. Initial Clustering and Data Processing



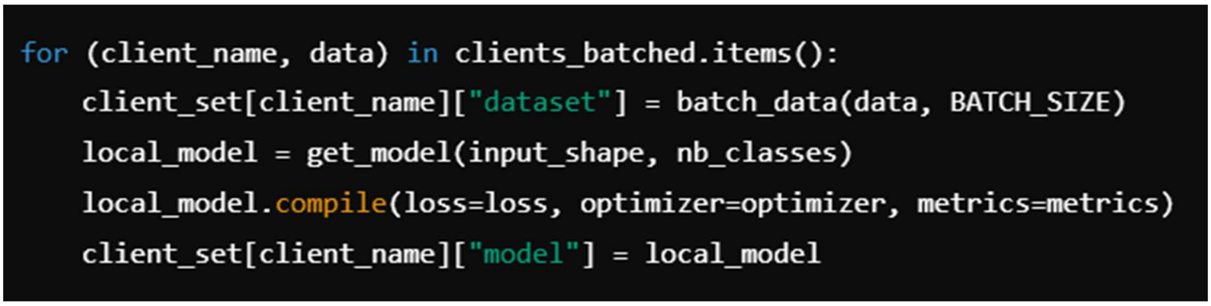
```python
client_clusters, cluster_heads = perform_clustering_based_on_labels(clients_batched,nb_classes)
```

client_clusters, cluster_heads = perform_clustering_based_on_labels(clients_batched, nb_classes): Calls the perform_clustering_based_on_labels function to get the cluster assignments and cluster heads.



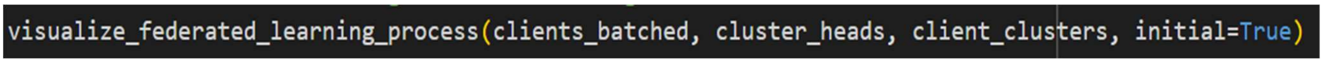
```python
client_set = {k: {} for k in clients_batched.keys()}
```

client_set = {k: {} for k in clients_batched.keys()}: Initializes an empty dictionary for each client in clients_batched.



```
for (client_name, data) in clients_batched.items():
    client_set[client_name]["dataset"] = batch_data(data, BATCH_SIZE)
    local_model = get_model(input_shape, nb_classes)
    local_model.compile(loss=loss, optimizer=optimizer, metrics=metrics)
    client_set[client_name]["model"] = local_model
```

- **for (client_name, data) in clients_batched.items():**: Iterates over each client's data in clients_batched.
- **client_set[client_name]["dataset"] = batch_data(data, BATCH_SIZE)**: Batches the data for each client using batch_data and assigns it to client_set.
- **local_model = get_model(input_shape, nb_classes)**: Gets a model instance using the get_model function.
- **local_model.compile(loss=loss, optimizer=optimizer, metrics=metrics)**: Compiles the model with the specified loss function, optimizer, and metrics.
- **client_set[client_name]["model"] = local_model**: Assigns the compiled model to client_set for each client.



```
visualize_federated_learning_process(clients_batched, cluster_heads, client_clusters, initial=True)
```

visualize_federated_learning_process(clients_batched, cluster_heads, client_clusters, initial=True): Calls a function (assumed to be defined elsewhere) to visualize the federated learning process, using the initial setup.