ML Second Project Documentation

Professor: Dr. Kiani

Ilya Shabanpour Fouladi

4003613041

GitHub Rep. link:

https://github.com/ilya-shabanpour/ML_Classification-Clustering

1402 – 1403 second semester

Table of Contents

Foreword

In this Documentation I want to elaborate on the implementation of my second Machine Learning Project.

Our task is to find a good classification and clustering model based on some features that were given to us in the form of a csv dataset.

Also, we had to extract features from about 400 leaf pictures and add those features to the first csv file.

In my implementation I've used these Python Libraries:

- Numpy
- Pandas
- Sklearn
- MatPlotLib
- Seaborn
- Skimage

Feature Extraction

Our original dataset has 14 features. In order to add other features based on the pictures of leaves given to us in leaves.rar file, we need to use specific algorithms.

The First one is Hog (Histogram of Oriented Gradients). This method "captures edge orientations and intensity gradients within localized portions of an image to represent its structural features".

The second one is LBP (Local Binary Pattern). This method "encodes local texture information by thresholding neighborhood pixel values and generating a binary pattern for each pixel".

I used these two methods to extract features from the pictures.

Here is the code:

```python
import os
import pandas as pd
from skimage.feature import hog, local_binary_pattern
from skimage.io import imread
from skimage import img_as_ubyte, transform
import numpy as np
import re

# Function to extract HOG features
def extract_hog_features(image):
    # Increase pixels_per_cell and cells_per_block to reduce the number of
features
    features, hog_image = hog(image, pixels_per_cell=(32, 32),
cells_per_block=(2, 2), visualize=True, multichannel=False)
    return features

# Function to extract LBP features
def extract_lbp_features(image):
    # Parameters for LBP
    radius = 3
    n_points = 8 * radius
    lbp = local_binary_pattern(image, n_points, radius, method='uniform')
    # Calculate the histogram of the LBP
    lbp_hist, _ = np.histogram(lbp.ravel(), bins=np.arange(0, n_points + 3),
range=(0, n_points + 2))
    # Normalize the histogram
    lbp_hist = lbp_hist.astype("float")
    lbp_hist /= (lbp_hist.sum() + 1e-6)
    return lbp_hist

# Function to extract both HOG and LBP features
def extract_features(image_path):
    # Read image using skimage
    img = imread(image_path, as_gray=True)

    # Reduce image resolution to reduce the number of HOG features
    img_resized = transform.resize(img, (128, 128))

    # Convert image to unsigned byte (0-255)
    img_uint8 = img_as_ubyte(img_resized)

    # Extract HOG features
    hog_features = extract_hog_features(img_uint8)

    # Extract LBP features
    lbp_features = extract_lbp_features(img_uint8)

    # Combine HOG and LBP features
    combined_features = np.hstack((hog_features, lbp_features))

    return combined_features

# Directory containing images
image_dir = '/content/drive/MyDrive/All_Leaves'

# Initialize lists to store features and corresponding labels
all_features = []
all_labels = []
all_ex_labels = []
pattern = r'C(\d+)'
pattern2 = r'(\d+)'

# Iterate over each image in the directory
for filename in os.listdir(image_dir):
    if filename.endswith('.JPG'):
        # Extract label from filename (format is 'C01.jpg', 'C02.png', etc.)
        split = filename.split('_')
        label = split[1]
        ex_label = split[2]

        label = re.findall(pattern, label)[0]
        label = int(label)

        match = re.search(pattern2, ex_label)
        if match:
            ex_label = match.group(0)
            ex_label = int(ex_label)

        # Construct image path
        image_path = os.path.join(image_dir, filename)

        # Extract features
        features = extract_features(image_path)

        # Append features and label to lists
        all_features.append(features)
        all_labels.append(label)
        all_ex_labels.append(ex_label)

# Determine the number of features dynamically
```

```
num_features = len(all_features[0])
feature_columns = [f'feature_{i}' for i in range(num_features)]

# Create a pandas DataFrame for features and labels
leaf_data = pd.DataFrame(all_features, columns=feature_columns)
leaf_data['label'] = all_labels
leaf_data['ex_label'] = all_ex_labels

# Save DataFrame to CSV
csv_file = '/content/drive/MyDrive/leaf_hog_lbp_features.csv'
leaf_data.to_csv(csv_file, index=False)

print(f'HOG and LBP features extracted and saved to {csv_file}.')
```

I used Google Colab and uploaded the images in my drive. Then for each picture I extracted Hog and LBP features. In each image's name there is the number of the class it belongs to and the number of the sample it is.

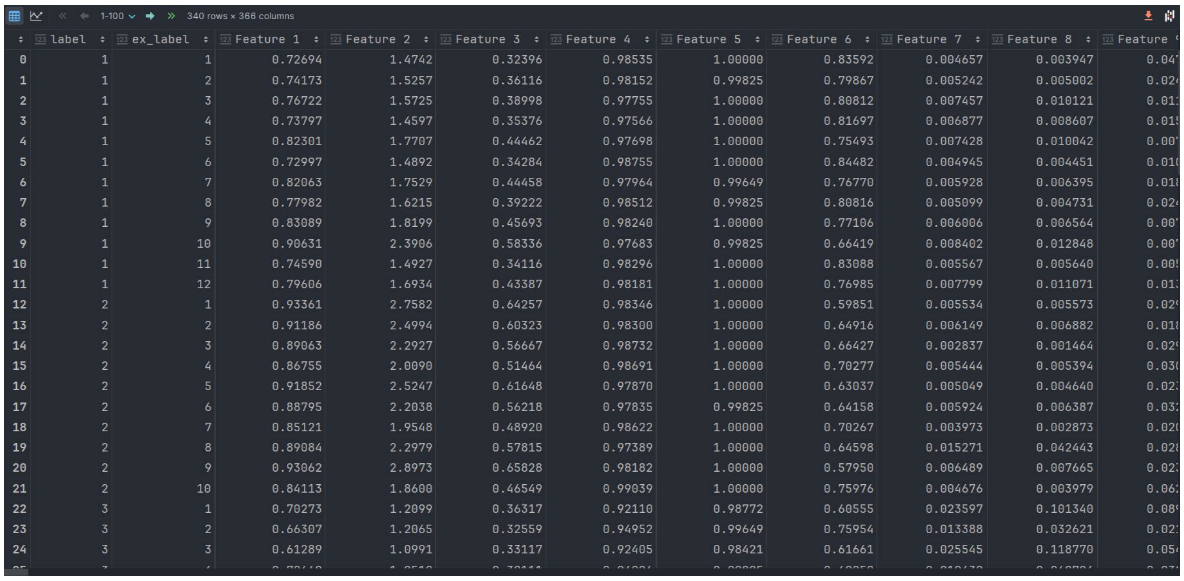
So, I added two columns for 'Label' and 'sample' two each row so that later it would be possible to merge the first dataset and this new one.

In the end I put the extracted features and labels in a .csv file.

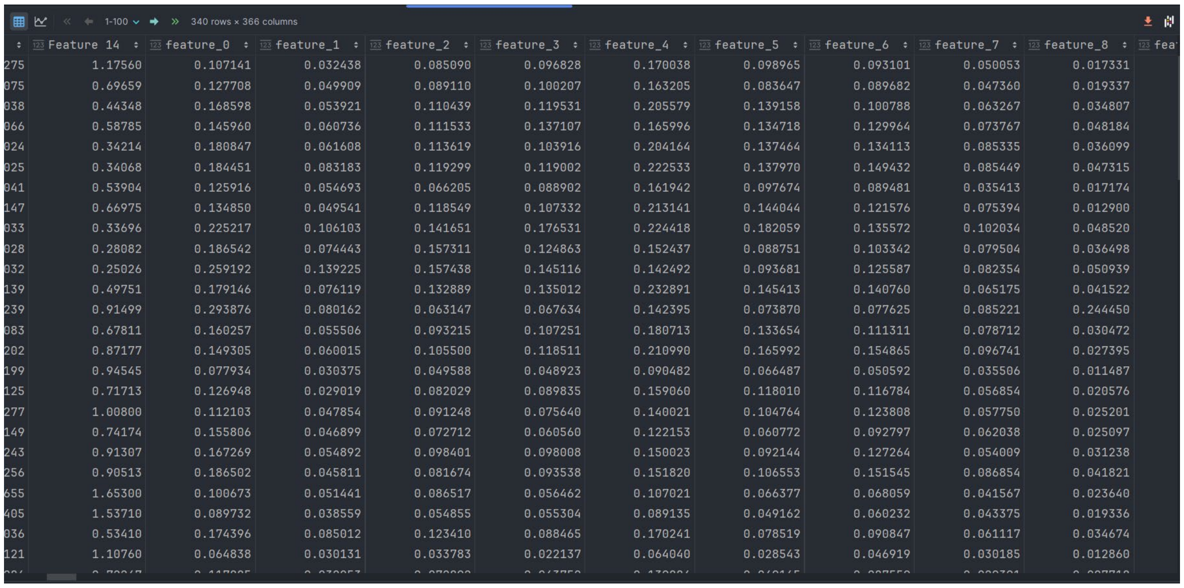
Data Preprocessing

We should preprocess our data so that it will be ready to be given to our models.

At first let’s look at our dataset:


1-100
340 rows × 366 columns

	label	ex_label	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5	Feature 6	Feature 7	Feature 8	Feature
0	1	1	0.72694	1.4742	0.32396	0.98535	1.00000	0.83592	0.004657	0.003947	0.04
1	1	2	0.74173	1.5257	0.36116	0.98152	0.99825	0.79867	0.005242	0.005002	0.02
2	1	3	0.76722	1.5725	0.38998	0.97755	1.00000	0.80812	0.007457	0.010121	0.01
3	1	4	0.73797	1.4597	0.35376	0.97566	1.00000	0.81697	0.006877	0.008607	0.01
4	1	5	0.82301	1.7707	0.44462	0.97698	1.00000	0.75493	0.007428	0.010042	0.00
5	1	6	0.72997	1.4892	0.34284	0.98755	1.00000	0.84482	0.004945	0.004451	0.01
6	1	7	0.82063	1.7529	0.44458	0.97964	0.99649	0.76770	0.005928	0.006395	0.01
7	1	8	0.77982	1.6215	0.39222	0.98512	0.99825	0.80816	0.005099	0.004731	0.02
8	1	9	0.83089	1.8199	0.45693	0.98240	1.00000	0.77106	0.006006	0.006564	0.00
9	1	10	0.90631	2.3906	0.58336	0.97683	0.99825	0.66419	0.008402	0.012848	0.00
10	1	11	0.74590	1.4927	0.34116	0.98296	1.00000	0.83088	0.005567	0.005640	0.00
11	1	12	0.79606	1.6934	0.43387	0.98181	1.00000	0.76985	0.007799	0.011071	0.01
12	2	1	0.93361	2.7582	0.64257	0.98346	1.00000	0.59851	0.005534	0.005573	0.02
13	2	2	0.91186	2.4994	0.60323	0.98300	1.00000	0.64916	0.006149	0.006882	0.01
14	2	3	0.89063	2.2927	0.56667	0.98732	1.00000	0.66427	0.002837	0.001464	0.02
15	2	4	0.86755	2.0090	0.51464	0.98691	1.00000	0.70277	0.005444	0.005394	0.03
16	2	5	0.91852	2.5247	0.61648	0.97870	1.00000	0.63037	0.005049	0.004640	0.02
17	2	6	0.88795	2.2038	0.56218	0.97835	0.99825	0.64158	0.005924	0.006387	0.03
18	2	7	0.85121	1.9548	0.48920	0.98622	1.00000	0.70267	0.003973	0.002873	0.02
19	2	8	0.89084	2.2979	0.57815	0.97389	1.00000	0.64598	0.015271	0.042443	0.02
20	2	9	0.93062	2.8973	0.65828	0.98182	1.00000	0.57950	0.006489	0.007665	0.02
21	2	10	0.84113	1.8600	0.46549	0.99039	1.00000	0.75976	0.004676	0.003979	0.06
22	3	1	0.70273	1.2099	0.36317	0.92110	0.98772	0.60555	0.023597	0.101340	0.08
23	3	2	0.66307	1.2065	0.32559	0.94952	0.99649	0.75954	0.013388	0.032621	0.02
24	3	3	0.61289	1.0991	0.33117	0.92405	0.98421	0.61661	0.025545	0.118770	0.05


1-100
340 rows × 366 columns

	Feature 14	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8
275	1.17560	0.107141	0.032438	0.085090	0.096828	0.170038	0.098965	0.093101	0.050053	0.017331
075	0.69659	0.127708	0.049909	0.089110	0.100207	0.163205	0.083647	0.089682	0.047360	0.019337
038	0.44348	0.168598	0.053921	0.110439	0.119531	0.205579	0.139158	0.100788	0.063267	0.034807
066	0.58785	0.145960	0.060736	0.111533	0.137107	0.165996	0.134718	0.129964	0.073767	0.048184
024	0.34214	0.180847	0.061608	0.113619	0.103916	0.204164	0.137464	0.134113	0.085335	0.036099
025	0.34068	0.184451	0.083183	0.119299	0.119002	0.222533	0.137970	0.149432	0.085449	0.047315
041	0.53904	0.125916	0.054693	0.066205	0.088902	0.161942	0.097674	0.089481	0.035413	0.017174
147	0.66975	0.134850	0.049541	0.118549	0.107332	0.213141	0.144044	0.121576	0.075394	0.012900
033	0.33696	0.225217	0.106103	0.141651	0.176531	0.224418	0.182059	0.135572	0.102034	0.048520
028	0.28082	0.186542	0.074443	0.157311	0.124863	0.152437	0.088751	0.103342	0.079504	0.036498
032	0.25026	0.259192	0.139225	0.157438	0.145116	0.142492	0.093681	0.125587	0.082354	0.050939
139	0.49751	0.179146	0.076119	0.132889	0.135012	0.232891	0.145413	0.140760	0.065175	0.041522
239	0.91499	0.293876	0.080162	0.063147	0.067634	0.142395	0.073870	0.077625	0.085221	0.244450
083	0.67811	0.160257	0.055506	0.093215	0.107251	0.180713	0.133654	0.111311	0.078712	0.030472
202	0.87177	0.149305	0.060015	0.105500	0.118511	0.210990	0.165992	0.154865	0.096741	0.027395
199	0.94545	0.077934	0.030375	0.049588	0.048923	0.090482	0.066487	0.050592	0.035506	0.011487
125	0.71713	0.126948	0.029019	0.082029	0.089835	0.159060	0.118010	0.116784	0.056854	0.020576
277	1.00800	0.112103	0.047854	0.091248	0.075640	0.140021	0.104764	0.123808	0.057750	0.025201
149	0.74174	0.155806	0.046899	0.072712	0.060560	0.122153	0.060772	0.092797	0.062038	0.025097
243	0.91307	0.167269	0.054892	0.098401	0.098008	0.150023	0.092144	0.127264	0.054009	0.031238
256	0.90513	0.186502	0.045811	0.081674	0.093538	0.151820	0.106553	0.151545	0.086854	0.041821
655	1.65300	0.100673	0.051441	0.086517	0.056462	0.107021	0.066377	0.068059	0.041567	0.023640
405	1.53710	0.089732	0.038559	0.054855	0.055304	0.089135	0.049162	0.060232	0.043375	0.019336
036	0.53410	0.174396	0.085012	0.123410	0.088465	0.170241	0.078519	0.090847	0.061117	0.034674
121	1.10760	0.064838	0.030131	0.033783	0.022137	0.064040	0.028543	0.046919	0.030185	0.012860

As you can see the first two columns are related to labels and 364 columns are features, 14 of which are given in the original dataset and 350 columns are the extracted features. There are no missing values or NaNs in this dataset.

One Thing about this dataset is that it only has 30 unique labels. But the numbers are from 1 to 36 so we have to map each label to make it in 0 to 29 order. This code does exactly that:

```
y = df.pop("label")
y = np.array(y)
class_samples = df.pop("ex_label")

unique_labels = np.unique(y)

label_mapping = {old_label: new_label for new_label, old_label in
enumerate(unique_labels)}
y = np.vectorize(label_mapping.get)(y)
```

Classification

First, we split the dataset into train and test (with test_size = 0.1):

```
x_train, x_test, y_train, y_test = train_test_split(x, y, shuffle=True,
test_size=0.1)
```

then we scale our features using Standard Scaling:

```
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)
```

we could make Polynomial Features and add them to the features:

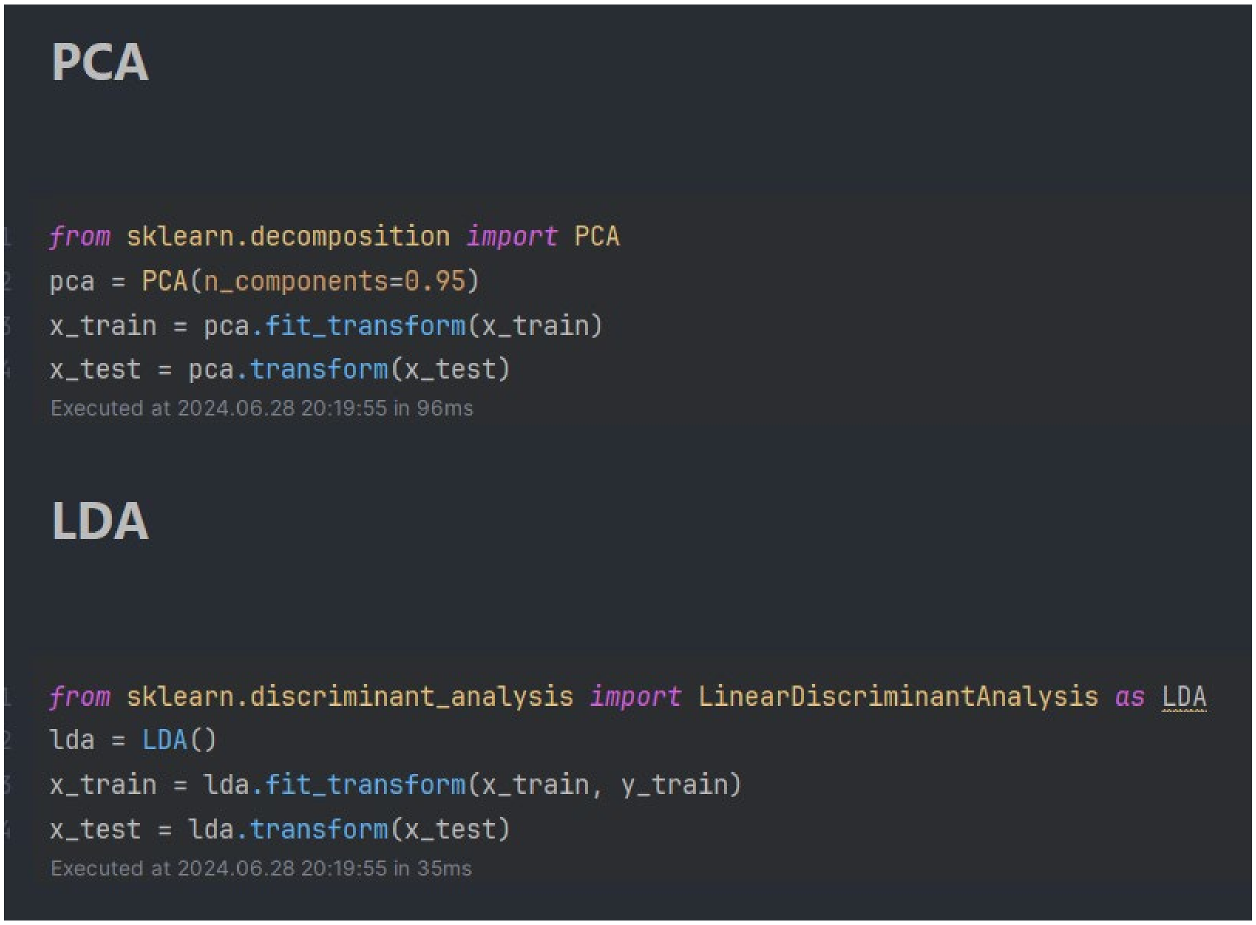
```
pf = PolynomialFeatures(degree=2)
x_train_poly = pf.fit_transform(x_train)
x_test_poly = pf.transform(x_test)

x_train_poly = scaler.fit_transform(x_train_poly)
x_test_poly = scaler.transform(x_test_poly)
```

*** but after checking the test accuracies I found out that the Polynomial features are reducing my test accuracy, so I deleted them.

I used PCA and LDA for Dimension Reduction. PCA with n_components = 0.95

Keeps 95% of the most important created features. And LDA reduces the number of features to the number of classes minus 1. So in the end we have 29 refined features.



PCA

```python
from sklearn.decomposition import PCA
pca = PCA(n_components=0.95)
x_train = pca.fit_transform(x_train)
x_test = pca.transform(x_test)
```

Executed at 2024.06.28 20:19:55 in 96ms

LDA

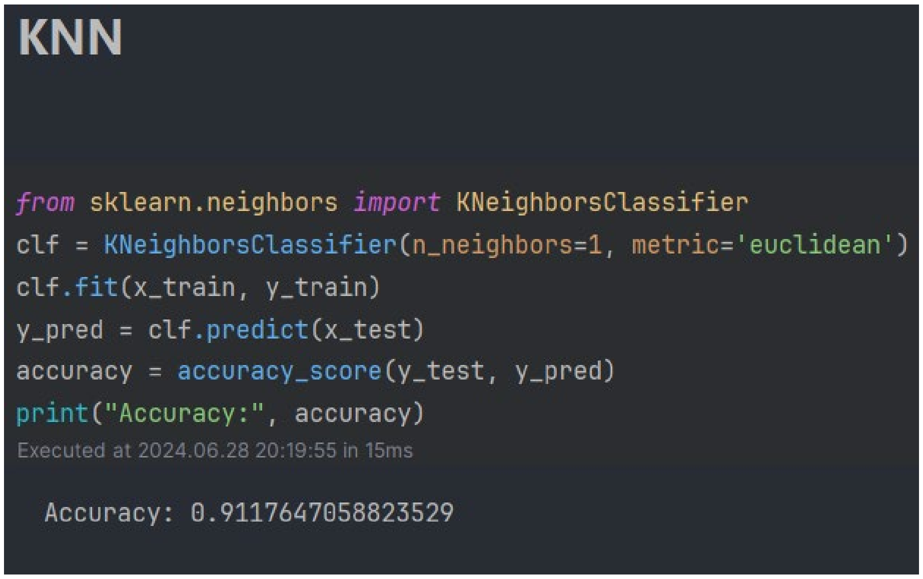
```python
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda = LDA()
x_train = lda.fit_transform(x_train, y_train)
x_test = lda.transform(x_test)
```

Executed at 2024.06.28 20:19:55 in 35ms

Models

I Used KNN, GNB, Adaboost, Random Forest and SVM.

KNN:



KNN

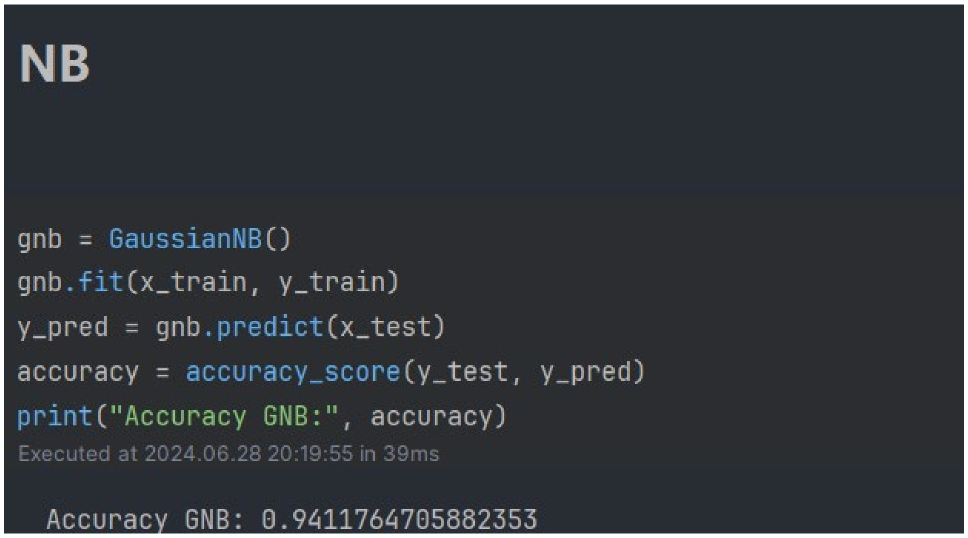
```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=1, metric='euclidean')
clf.fit(x_train, y_train)
y_pred = clf.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Executed at 2024.06.28 20:19:55 in 15ms

```
  Accuracy: 0.9117647058823529
```

KNN finds the K Nearest neighbors of a given test data and based on the label of the neighbors judges the label of the test data. I’ve set the number of neighbors to 1 and distance metric to ‘Euclidean’ because after testing these parameters get the best result in my dataset.

GNB:



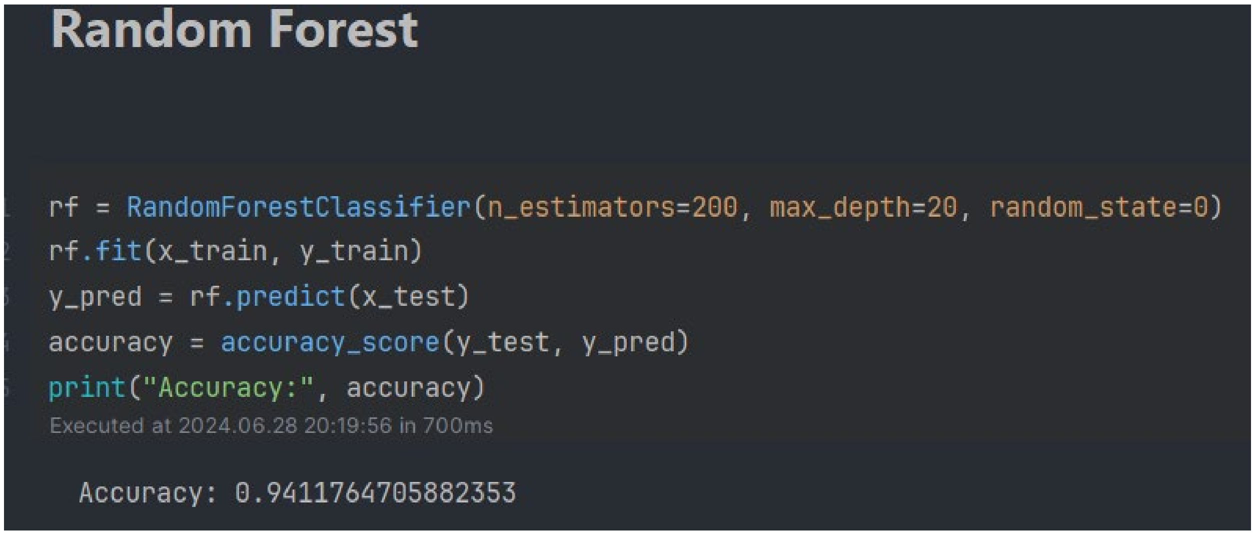
NB

```
gnb = GaussianNB()
gnb.fit(x_train, y_train)
y_pred = gnb.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy GNB:", accuracy)
```

Executed at 2024.06.28 20:19:55 in 39ms

```
  Accuracy GNB: 0.9411764705882353
```

Random Forest:


Random Forest

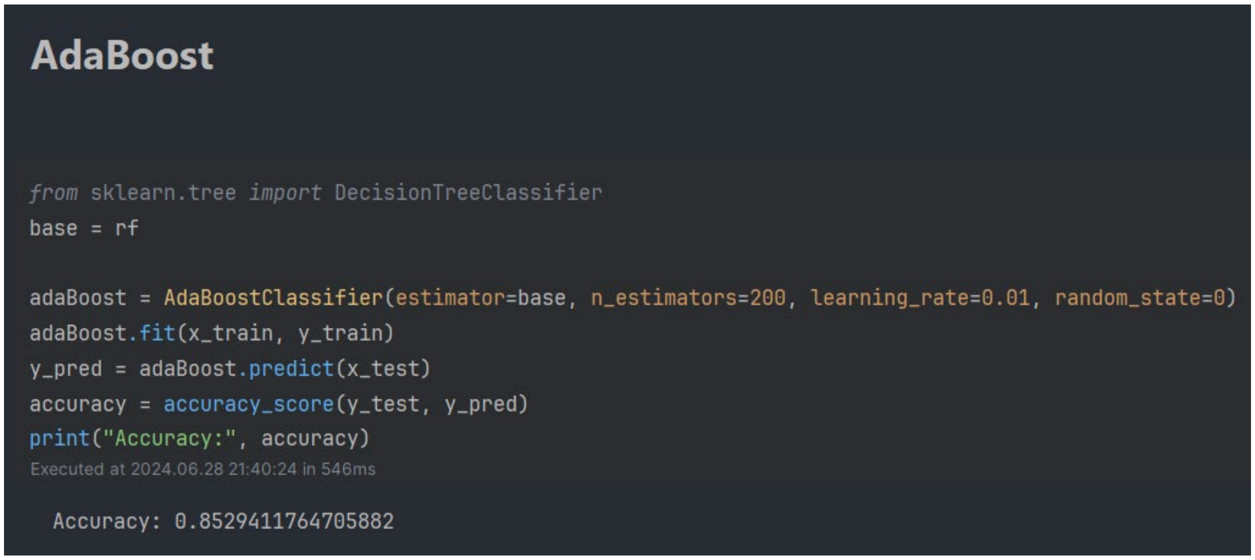
```
rf = RandomForestClassifier(n_estimators=200, max_depth=20, random_state=0)
rf.fit(x_train, y_train)
y_pred = rf.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
Executed at 2024.06.28 20:19:56 in 700ms

  Accuracy: 0.9411764705882353
```

Random Forest is a bagging method for decision trees. It checks many different decision trees and returns the one with the highest accuracy.

Max_depth shows how deep the tree search goes.

AdaBoost:


AdaBoost

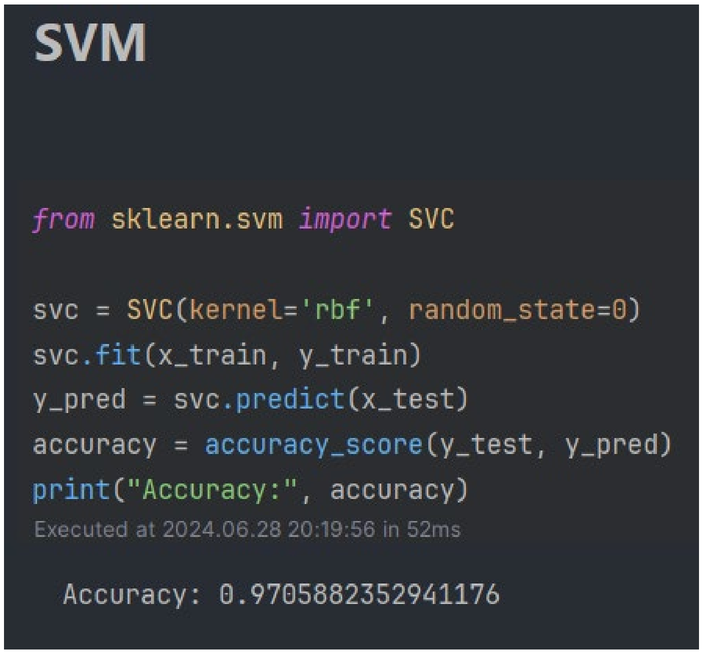
```
from sklearn.tree import DecisionTreeClassifier
base = rf

adaBoost = AdaBoostClassifier(estimator=base, n_estimators=200, learning_rate=0.01, random_state=0)
adaBoost.fit(x_train, y_train)
y_pred = adaBoost.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
Executed at 2024.06.28 21:40:24 in 546ms

  Accuracy: 0.8529411764705882
```

AdaBoost takes a base Model like decision trees or Random Forest and tries to improve the learning by focusing on the data with error in the last epoch. I gave it Random Forest as the base model. There is a chance that this method leads to Overfit.

SVM:



SVM

```
from sklearn.svm import SVC

svc = SVC(kernel='rbf', random_state=0)
svc.fit(x_train, y_train)
y_pred = svc.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Executed at 2024.06.28 20:19:56 in 52ms

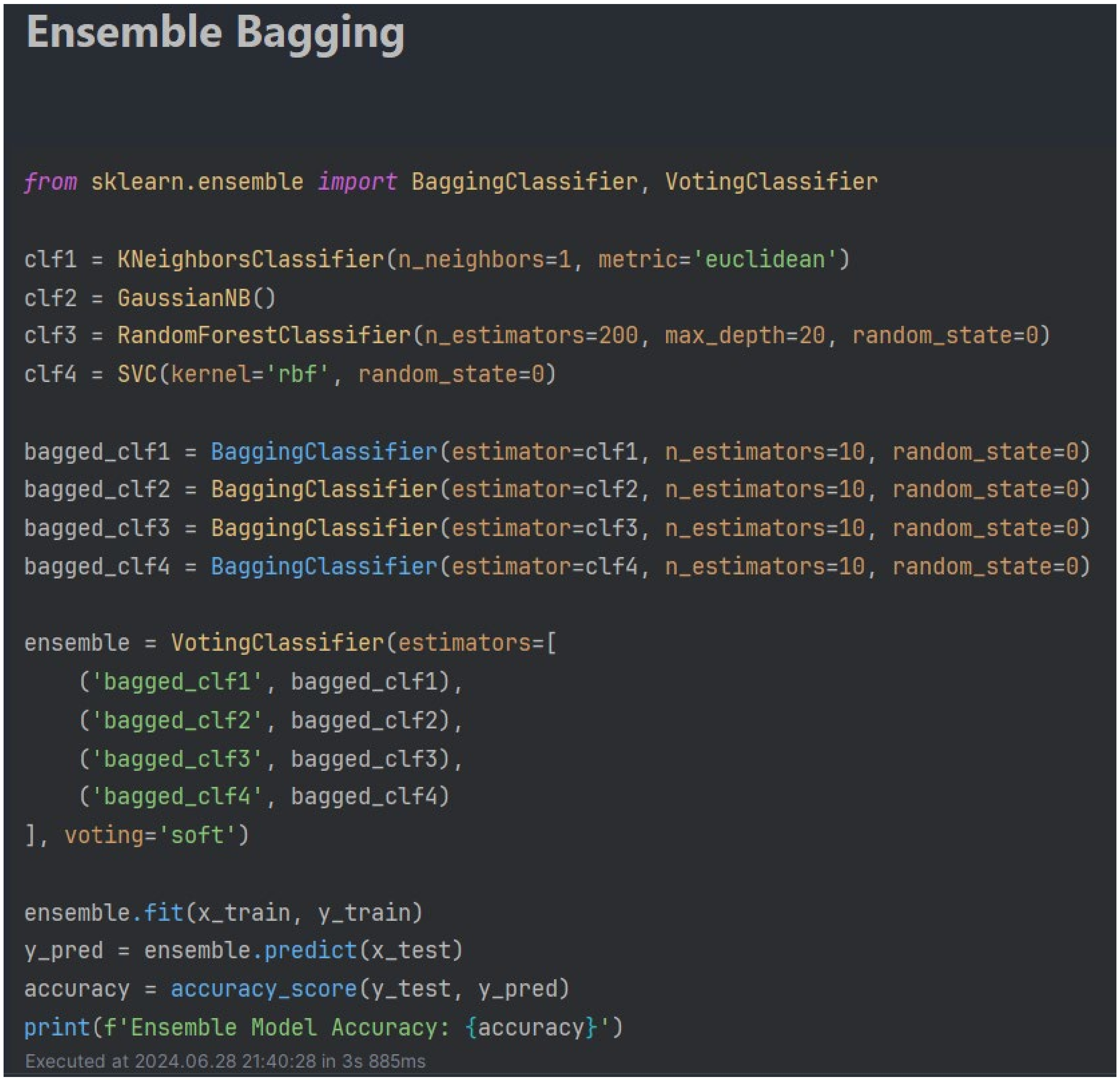
```
Accuracy: 0.9705882352941176
```

Support Vector Machine Model uses one with one method for more than 2 classes. The kernel is set to 'rbf' to detect non-linear relationships.

This method and Random Forest had the best test accuracies in my dataset.

Assembling Models Using Bagging and Voting

After defining the different Models, we can use Bagging and Voting for getting a better final result:



Ensemble Bagging

```python
from sklearn.ensemble import BaggingClassifier, VotingClassifier

clf1 = KNeighborsClassifier(n_neighbors=1, metric='euclidean')
clf2 = GaussianNB()
clf3 = RandomForestClassifier(n_estimators=200, max_depth=20, random_state=0)
clf4 = SVC(kernel='rbf', random_state=0)

bagged_clf1 = BaggingClassifier(estimator=clf1, n_estimators=10, random_state=0)
bagged_clf2 = BaggingClassifier(estimator=clf2, n_estimators=10, random_state=0)
bagged_clf3 = BaggingClassifier(estimator=clf3, n_estimators=10, random_state=0)
bagged_clf4 = BaggingClassifier(estimator=clf4, n_estimators=10, random_state=0)

ensemble = VotingClassifier(estimators=[
    ('bagged_clf1', bagged_clf1),
    ('bagged_clf2', bagged_clf2),
    ('bagged_clf3', bagged_clf3),
    ('bagged_clf4', bagged_clf4)
], voting='soft')

ensemble.fit(x_train, y_train)
y_pred = ensemble.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)
print(f'Ensemble Model Accuracy: {accuracy}')
```

Executed at 2024.06.28 21:40:28 in 3s 885ms

Ensemble Model Accuracy: 0.8823529411764706

First, we redefine the Previous models and bag them. Then we create a Voting Classifier with those bagged models and I used “soft” voting method. Soft voting means that a probability based on each models accuracy is calculated and then the weighted vote leads to the decision.

Clustering

For Clustering I used the extracted and preprocessed features as I mentioned earlier.

In clustering there is no need for train and test splitting because we need all the data and their features.

However, we should scale the data using Standard Scaling.

Defining Algorithms

First, we define the algorithms we want to use. I used K-Means, Agglomerative Clustering, DBSCAN and Gaussian Mixture Model:

```
clustering_algorithms = [
    KMeans(n_clusters=30, random_state=42, n_init=20),
    AgglomerativeClustering(n_clusters=30),
    DBSCAN(eps=2.5, min_samples=5),
    GaussianMixture(n_components=30, random_state=42)
]
```

the n_components shows the K clusters we want in each algorithm. I've used the number of true labels for the number of clusters.

Esp in DBSCAN show the distance range that two points can have to form a cluster and Min_Sample shows the number of points each cluster needs to become one.

Predicting Labels and Silhouette Score

For each algorithm we compute the clusters and their members, then we calculate their Silhouette Score:

```
for algorithm in clustering_algorithms:
    name = type(algorithm).__name__

    if name == 'GaussianMixture':
        algorithm.fit(X_scaled)
        y_pred = np.argmax(algorithm.predict_proba(X_scaled), axis=1)
    else:
        algorithm.fit(X_scaled)
        if hasattr(algorithm, 'labels_'):
            y_pred = algorithm.labels_
        else:
            y_pred = algorithm.predict(X_scaled)

    if len(np.unique(y_pred)) == 1:
        print(f'{name} did not find any clusters.')
        continue

    if len(np.unique(y_pred)) > 1:
        silhouette = silhouette_score(X_scaled, y_pred)
    else:
        silhouette = -1
```

Mapping Clusters to Labels

Then we map the clusters to the true labels of our dataset. The label of each cluster is the label of its most repeated member label. For example, if there are 8 samples of class 4 in a cluster and it is the highest rank, then the label of this cluster is 4:

```
def map_clusters_to_labels(y_true, y_pred):
    labels = np.unique(y_true)
    clusters = np.unique(y_pred)
    matrix = np.zeros((len(labels), len(clusters)))

    for i, label in enumerate(labels):
        for j, cluster in enumerate(clusters):
            matrix[i, j] = np.sum((y_true == label) & (y_pred == cluster))

    return np.argmax(matrix, axis=0)
```

Then we create the confusion matrix of that algorithm and store the values in a dictionary:

```
conf_matrix = confusion_matrix(y, y_pred_mapped)
accuracy = accuracy_score(y, y_pred_mapped)

results[name] = {
    'confusion_matrix': conf_matrix,
    'accuracy': accuracy,
    'silhouette_score': silhouette
}
```

Finally, we display the accuracy, Silhouette Score and Confusion Matrix for each Clustering Algorithm using MatPlotLib and Seaborn:

```
for name, result in results.items():
    print(f'{name} - Accuracy: {result["accuracy"]}, Silhouette Score:
{result["silhouette_score"]}')
    plt.figure(figsize=(10, 8))
    sns.heatmap(result['confusion_matrix'], annot=True, fmt='d',
cmap='Blues')
    plt.title(f'{name} Confusion Matrix')
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.show()
```

These are the final results:

DBSCAN did not find any clusters.

K-Means - Accuracy: 0.6352941176470588, Silhouette Score: 0.08869227933873192

Agglomerative Clustering - Accuracy: 0.6852941176470588, Silhouette Score: 0.09290624713834027

Gaussian Mixture - Accuracy: 0.6058823529411764, Silhouette Score: 0.0786121524800283

DBSCAN doesn't find any clusters. DBSCAN is a Density based clustering method and it can't create clusters based on my scaled data.

The other algorithms have a low Silhouette Score but good accuracies. This is because this high dimensional multifeatured and densely close dataset of mine, makes the distance metric less meaningful and therefore gives a low Silhouette Score.

The accuracy of each algorithm is based on their matching with the true labels. This will be shown in the confusion matrix.

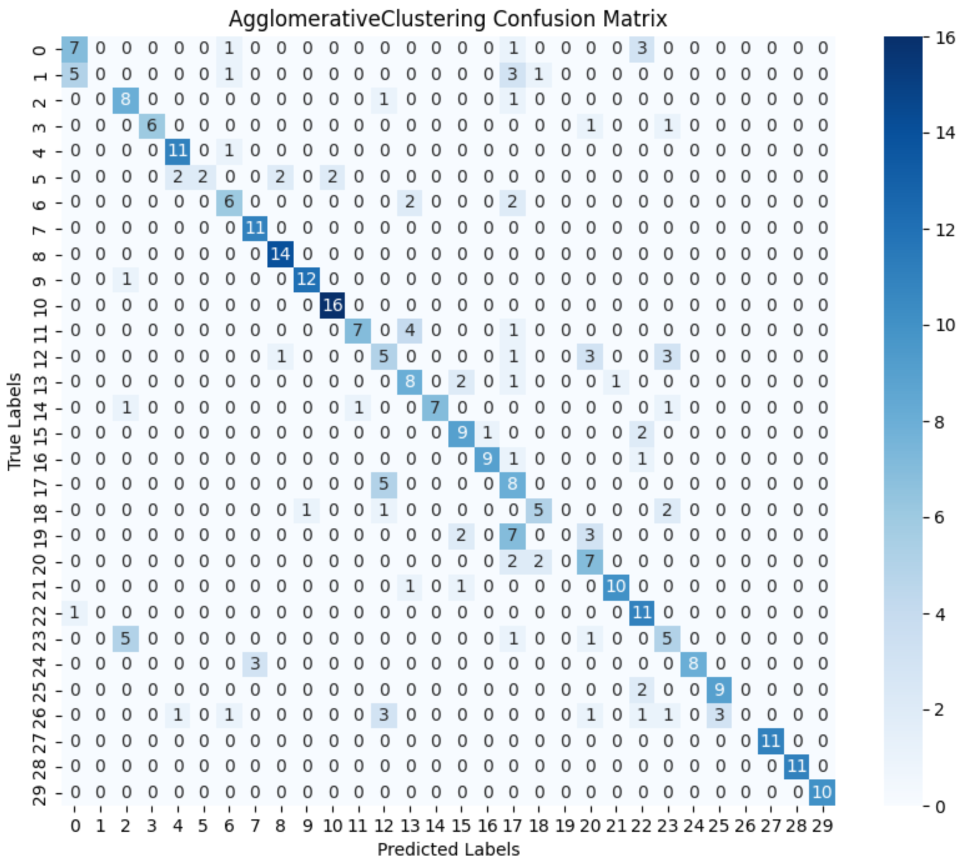
The best algorithm based on accuracy was:

1 – Agglomerative Clustering

2 – K-Means

3 – GMM

confusion matrix

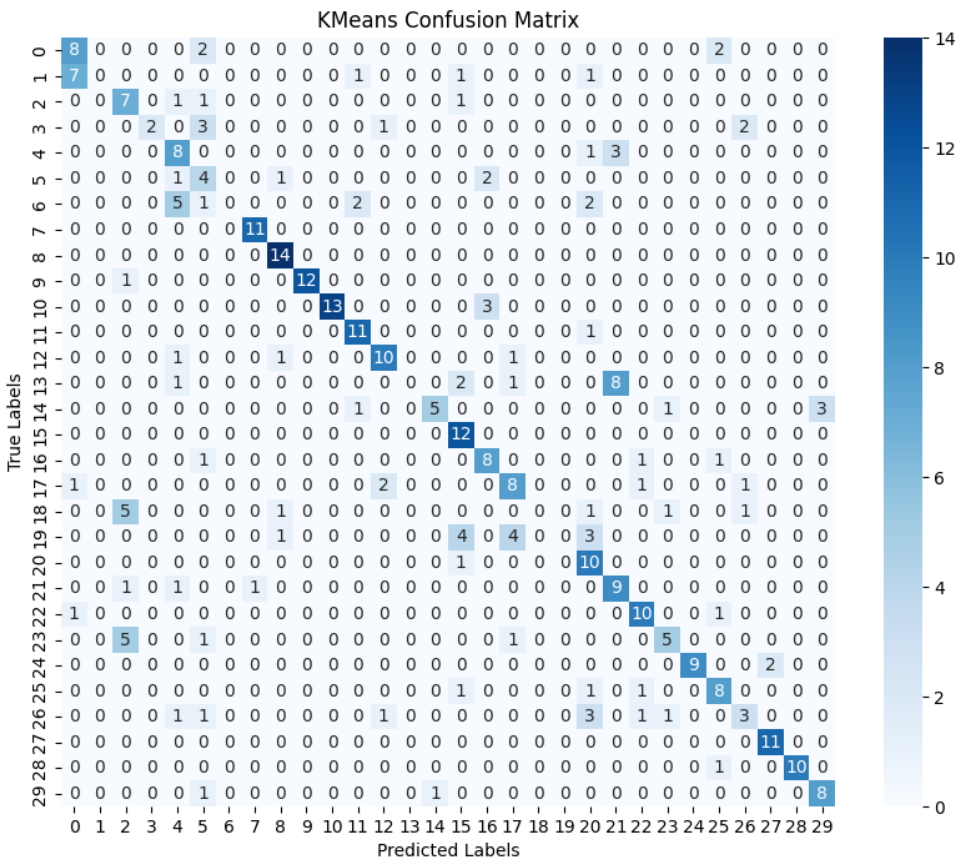


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	7	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	3	0	0	0	0	0	0	0
1	5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	3	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	8	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
4	0	0	0	0	11	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	2	2	0	0	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	6	0	0	0	0	0	0	2	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	1	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	7	0	4	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	1	0	0	0	5	0	0	0	0	1	0	0	3	0	0	3	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	2	0	1	0	0	0	1	0	0	0	0	0	0	0	0
14	0	0	1	0	0	0	0	0	0	0	0	1	0	0	7	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	1	0	0	0	0	0	2	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	1	0	0	0	0	1	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	5	0	0	0	0	2	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	7	0	0	3	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	2	0	7	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	10	0	0	0	0	0	0	0	0
22	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	0	0	0	0	0	0
23	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	5	0	0	0	0	0	0
24	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	9	0	0	0	0
26	0	0	0	0	1	0	1	0	0	0	0	0	3	0	0	0	0	0	0	0	1	0	1	1	0	3	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10

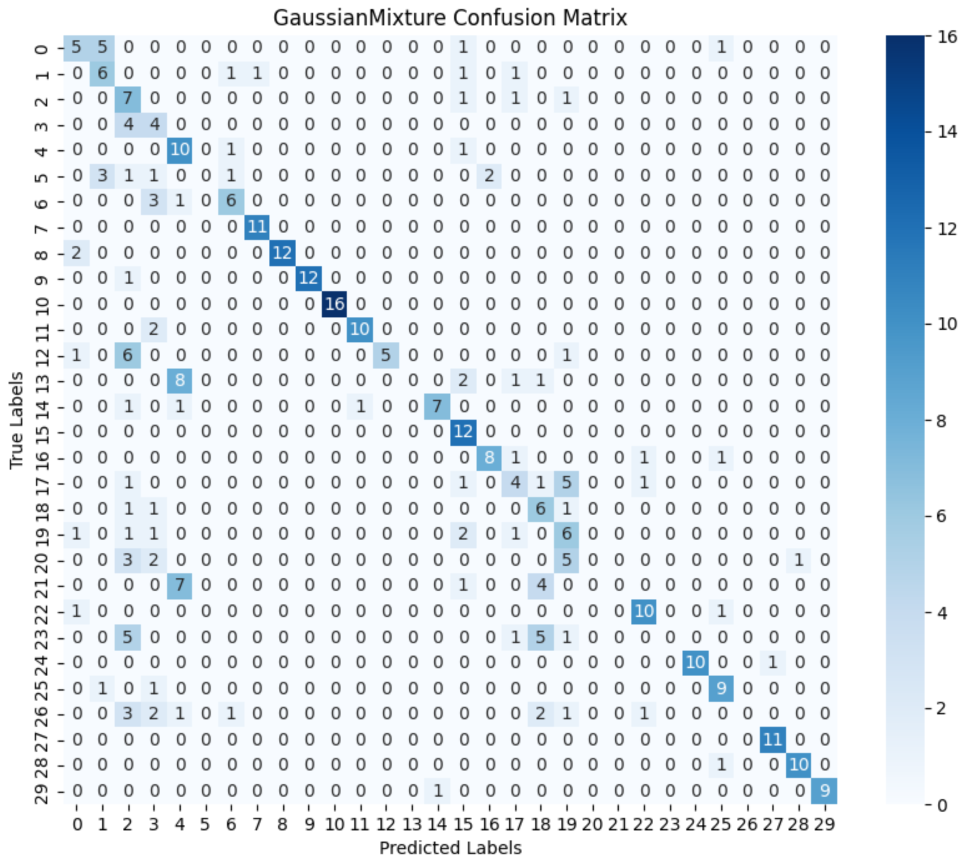
Agglomerative Clustering - Accuracy: 0.685

The main diameter of confusion matrix shows the truly predicted clusters. For instance, all 16 samples of the 11th class of leaves (the 10th label here), are in a single cluster and this raises the total accuracy. But the samples of 13th class of leaves (the 12th class here) have been scattered across multiple clusters and that lowers the total accuracy.

But generally, for the majority of classes, most of their samples are in one cluster and this shows a good clustering.



K-Means - Accuracy: 0.635



True \ Predicted	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	6	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0
3	0	0	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	10	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	3	1	1	0	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	3	1	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	2	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	1	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	2	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	1	0	6	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	2	0	1	1	0	0	0	0	0	0	0	0	0	0	0
14	0	0	1	0	1	0	0	0	0	0	0	1	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	1	0	0	0	0	1	0	0	1	0	0	0	0
17	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	4	1	5	0	0	1	0	0	0	0	0	0	0
18	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	1	0	0	0	0	0	0	0	0	0	0
19	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	2	0	1	0	6	0	0	0	0	0	0	0	0	0	0
20	0	0	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	1	0
21	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	1	0	0	4	0	0	0	0	0	0	0	0	0	0	0
22	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	1	0	0	0	0
23	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5	1	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	1	0	0
25	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	0
26	0	0	3	2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	1	0	0	0	0	0	0	0
27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	0
28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	10	0
29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9

Gaussian Mixture - Accuracy: 0.605