Classification
And
Clustering

Mohammad Hassan Heydari

Introduction

Classification and Clustering are two popular algorithms among AI researchers . In our project, we delve into the world of leaves, combining the power of classification and clustering algorithms to unlock hidden insights from leaf images and features .

Our dataset contains two major parts : Images and Features

Image dataset comprises high-resolution images of leaves, each representing a distinct species. These images serve as our canvas for experimentation.

Our features dataset includes the features of leaves , stored in a structural way.

Here's what we did in this ptoject :

Classification Algorithms:

- Deep Neural Networks (DNN): Armed with Keras and JAX, we harnessed the computational might of GPU[1]s to train deep neural networks. These models learned to distinguish between leaf species, leveraging both local and global features.

- Random Forest Classifier: In contrast, we opted for CPU-based training when implementing the random forest classifier. Its ensemble of decision trees provided robust predictions, even without the GPU's parallel processing muscle.

- Convolutional Neural Networks (CNN[2]): Our CNNs, inspired by the visual cortex's architecture, excelled at capturing spatial hierarchies within leaf images. Their convolutional layers learned intricate patterns, making them ideal for this task. We used the power of transfer learning algorithms and the light MobileNetV2 [1] model for training our image dataset . also we created a pipeline for training our model on augmented images, which leads to better performance

[1] Graphical Processing Unit
[2] Convolutional Neural Networks

Clustering with K-Means:

We didn't stop at classification. K-Means clustering stepped onto the stage, seeking underlying structures within our leaf dataset. By grouping similar leaves together, we aimed to reveal natural clusters—perhaps hinting at taxonomic relationships or environmental adaptations.

Dataset

Our features dataset is stored in a tabular manner in a CSV file . first column of the table is target and other columns are features , instead of the second column which relates the index of related leaf image , we accept other columns as features and ignore this column cause it is irrelevant to each leaf features .

Leaf images are stored in a directory . This very directory, includes sub-directories which contain leaf images related to each class .

Deep Fully Connected Neural Networks

Deep learning is increasingly gaining the researchers attention these days . we can apply these algorithms to solve almost any AI problems . Here , we use these algorithms to classify the labels of leaves, based on their features .

We Implemented this algorithm with Keras 3 [2] library which uses JAX [3] as its backend ; This approach allows us to train and evaluate our model on accelerators like GPU . Keras is a high level API library designed for implementing deep learning models as easier as possible .

First, we import necessary libraries , which includes Keras, JAX and Pandas . Note that we initialize *KERAS_BACKEND* environmental variable as JAX before importing Keras itself .

```
import os
os.environ['KERAS_BACKEND'] = 'jax'

import jax.numpy as jnp

import jax
import pandas as pd
import keras
```

Then we check our training platform and backend :

```
device = jax.devices()[0]
print(f'Training platform : {device.platform}')
print(f'Device Name : {device.device_kind}')
print(f'Keras backend: {keras.backend.backend()}')
```

which shows us this output :

```
Training platform : gpu
Device Name : NVIDIA GeForce MX450
Keras backend: jax
```

We load our dataset and shuffle it using Pandas library , Then we convert the data frame into a jax ndarray :

```
leaves_csv = pd.read_csv('Data/leaves.csv')

leaves_csv = leaves_csv.sample(frac=1).reset_index(drop=True)

leaves_np = jnp.asarray(leaves_csv)
```

Then we split our dataset inro *features* and *targets* :

```
# first column is targets, other columns are features, so :
targets = leaves_np[:, 0].astype(dtype=jnp.int32)

# Second column is irrelevant
features = leaves_np[:, 2:]
print(f'shape of features: {features.shape}')
print(f'shape of targets: {targets.shape}')
```

Also we split our dataset into Train and Validation sets :

```
X_train = features[:300]
X_test = features[300:]
y_train = targets[:300]
y_test = targets[300:]

print(f'shape of X_train : {X_train.shape}')
print(f'shape of X_test : {X_test.shape}')
print(f'shape of y_train : {y_train.shape}')
print(f'shape of y_test: {y_test.shape}')
```

The code above shows us some information about dataset, such as number of features and training examples :

```
shape of X_train : (300, 14)
shape of X_test : (39, 14)
shape of y_train : (300,)
shape of y_test: (39,)
```

After data preprocessing part, we initialize our model with Keras . We used simple *Dense* layers for learning the patterns between leaves features and *Dropout* layers for decreasing the probability of overfitting . We implement these concepts with this part of code :

```
model = keras.Sequential([
    keras.layers.Input(shape=(14,)),
    keras.layers.Dense(128, activation='elu', name='Dense_1'),
    keras.layers.Dropout(0.2, name='Dropout_1'),
    keras.layers.Dense(512, activation='elu', name='Dense_2'),
    keras.layers.Dropout(0.2, name='Dropout_2'),
    keras.layers.Dense(64, activation='elu', name='Dense_3'),
    keras.layers.Dense(36, activation='softmax', name='Output'),
])
```

We used 3 hidden layers with *ELU*[3] activation function ; which leads to learn the polynomial patterns from the dataset .

model architecture summary can be shown with *model.summary()* method :

[3] Exponential Linear Unit

Layer (type)	Output Shape	Param #
Dense_1 (Dense)	(None, 128)	1,920
Dropout_1 (Dropout)	(None, 128)	0
Dense_2 (Dense)	(None, 512)	66,048
Dropout_2 (Dropout)	(None, 512)	0
Dense_3 (Dense)	(None, 64)	32,832
Output (Dense)	(None, 36)	2,340

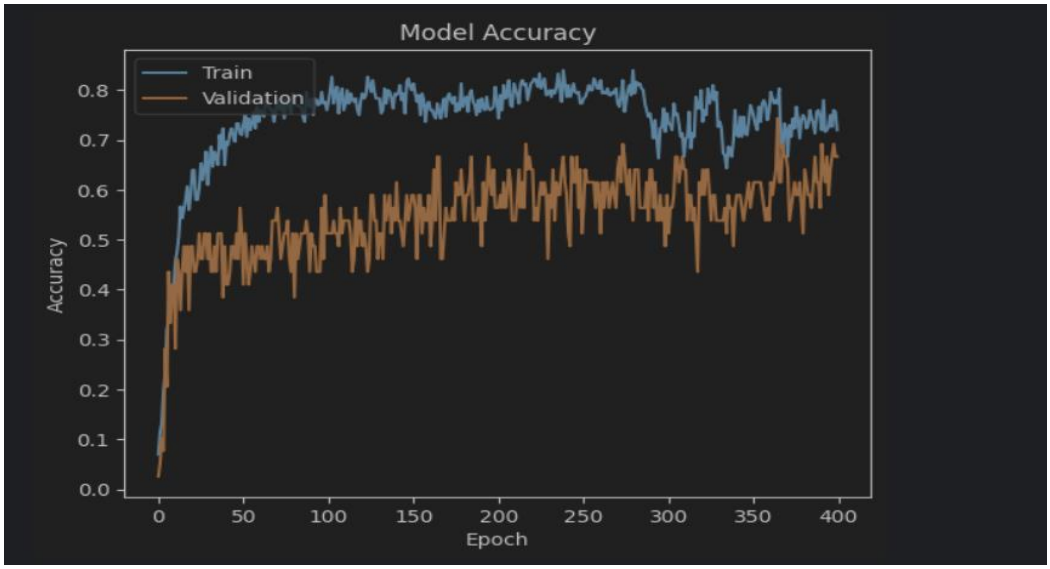
Total params: 103,140 (402.89 KB)

Trainable params: 103,140 (402.89 KB)

Non-trainable params: 0 (0.00 B)

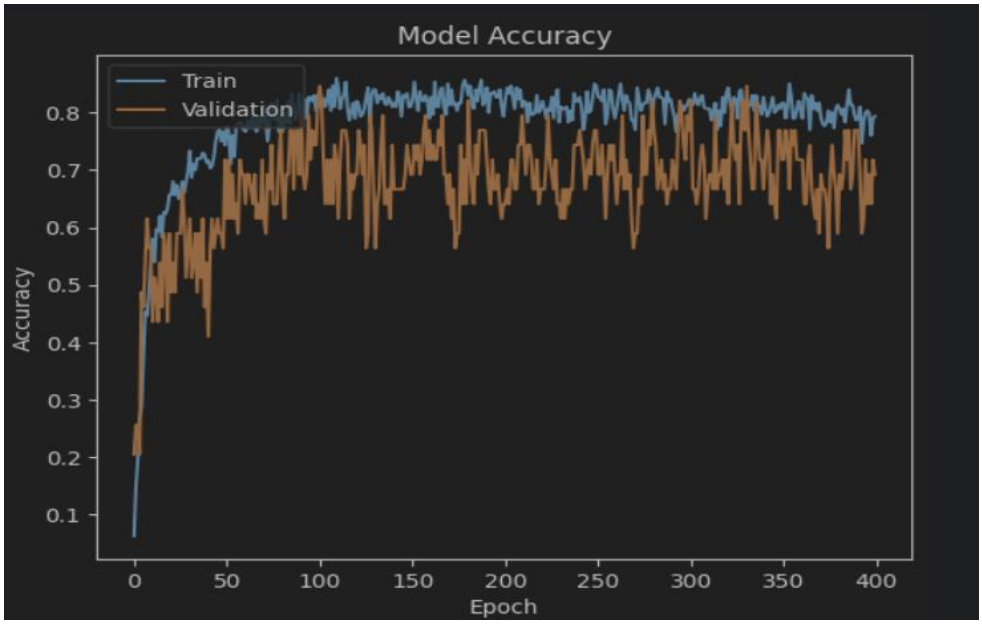
For training, we use Sparse Categorical Cross Entropy loss ; This is because our classification task is multi-class classification . We tried two different optimizers for training : SGD and Lion [4] . We trained our model for 400 epochs on each optimizer and compared the results . figure(1) shows that the SGD optimizer is heavily traped in the local minima , which leads to decreasing accuracy . Best validation accuracy on SGD optimizer was 74.4% :

Figure(1)



Then we trained our model on Lion optimizer with the same conditions . best model performance on validation set was 84.6% accuracy. Figure(2) shows the results :

Figure(2)



Results of training on Lion optimizer also indicate that this optimizer reduces the chance of overfitting , as shown in the figure(2) , train accuracy and validation accuracy closer to each other than SGD optimizer .

Figure(3) : Lion Optimiziation [4]

Algorithm 2 Lion Optimizer (ours)

given $\beta_1, \beta_2, \lambda, \eta, f$
initialize $\theta_0, m_0 \leftarrow 0$
while θ_t not converged **do**
  $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
  update model parameters
  $c_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$
  $\theta_t \leftarrow \theta_{t-1} - \eta_t(\text{sign}(c_t) + \lambda\theta_{t-1})$
  update EMA of g_t
  $m_t \leftarrow \beta_2 m_{t-1} + (1-\beta_2) g_t$
end while
return θ_t

We compile our model config with Keras *model.compile()* method . We initialize the loss function and optimizer of model in this method :

```
model.compile(loss='sparse_categorical_crossentropy',
optimizer='lion', metrics=['accuracy'], jit_compile= True)
```

jit_compile means that keras will use Just-In-Time compilation for numerical computations .

model training as also done with Keras *model.fit()* method . We also created a callback to save the best model with best performance on the validation set . all of this is implemented with this part of code :

```
checkpoint = keras.callbacks.ModelCheckpoint('model.keras',
monitor='val_accuracy', verbose=1, save_best_only=True)

history = model.fit(X_train, y_train, epochs=400,
validation_data=(X_test, y_test), verbose=2, callbacks=[checkpoint])
```

Random Forest Classifier

Random Forest is another classification algorithm that we implemented in our project . This algorithm is simply available in Scikit-Learn [5] library . First we load our dataset same as before but we apply polynomial features with degree = 3 on our dataset ; This leads to learn the complex patterns between dataset featers more acceptable :

```
from sklearn.preprocessing import PolynomialFeatures
def polynomial_features(X_train, X_test, degree):
    poly = PolynomialFeatures(degree)
    X_train_poly = poly.fit_transform(X_train)
    X_test_poly = poly.transform(X_test)
    return X_train_poly, X_test_poly

degree = 3
X_train, X_test = polynomial_features(X_train, X_test, degree= degree)
```

Then we train the model with Scikit-Learn model.fit() method with specified hyperparameters :

```
rf_classifier = RandomForestClassifier(n_estimators=200,
random_state=42)

rf_classifier.fit(X_train, y_train)
```

Finally we evaluate our model on training set and validation set :

```
y_test_pred = rf_classifier.predict(X_test)
y_train_pred = rf_classifier.predict(X_train)

accuracy_train = accuracy_score(y_train, y_train_pred)
accuracy_test = accuracy_score(y_test, y_test_pred)

print(f'Accuracy train : {accuracy_train}')
print(f'Accuracy test : {accuracy_test}')
```

This code shows us these results :

```
Accuracy train : 1.0
Accuracy test : 0.717948717948718
```

These results indicate that our random forest classifier perfectly converged on our training set , but the results on validation set is lower than a neural network which is train even on SGD optimizer .

So for better convergence , we apply Random Forest Classifier on our dataset but in the case of learning general patterns in the dataset , Neural Networks outperform random forests .

Table(10) : models details

Model	Poly-Degree	Epochs	Optimizer	Platform	Validation Accuracy	Train Accuracy
NN	1	400	Lion	GPU	0.846	0.862
NN	1	400	SGD	GPU	0.744	0.843
RF[4]	3	-	-	CPU	0.717	1.00

Convolutional Neural Networks

The classification which we we're doing on our dataset , was based on the dataset features of leaves ; but what if we had actual images of the leaves ?

In this section we implement image classification model based on convolutional neural networks and transfer learning techniques .

We first load our dataset with Keras utils.image_dataset_from_directory() function . This function takes a path directory contains of sub-directories and returns a Tensorflow data.Dataset object .

We load our images in the size of (128, 128) with 10 batches each . note that images are colorized :

```
image_size = (128, 128)
batch_size = 10

train_ds = keras.utils.image_dataset_from_directory(
    "leaves",
    seed=1337,
    image_size=image_size,
    batch_size=batch_size
)
```

[4] Random Forest

for better understanding , here we randomly plot 9 images of random leaves with their class numbers :

```
import numpy as np

plt.figure(figsize=(10, 10))
for images, labels in train_ds.take(1):
    print(labels)
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(np.array(images[i]).astype("uint8"))
        plt.title(int(labels[i]))
        plt.axis("off")
```

Result of this code is as follows :

We have only 402 images belonging to 36 classes , this can lead to underfitting or some other problems . We apply Data Augmentation techniques to increase the

number of training images . These techniques include random rotation, random flip, random brightness, random contrast and others .

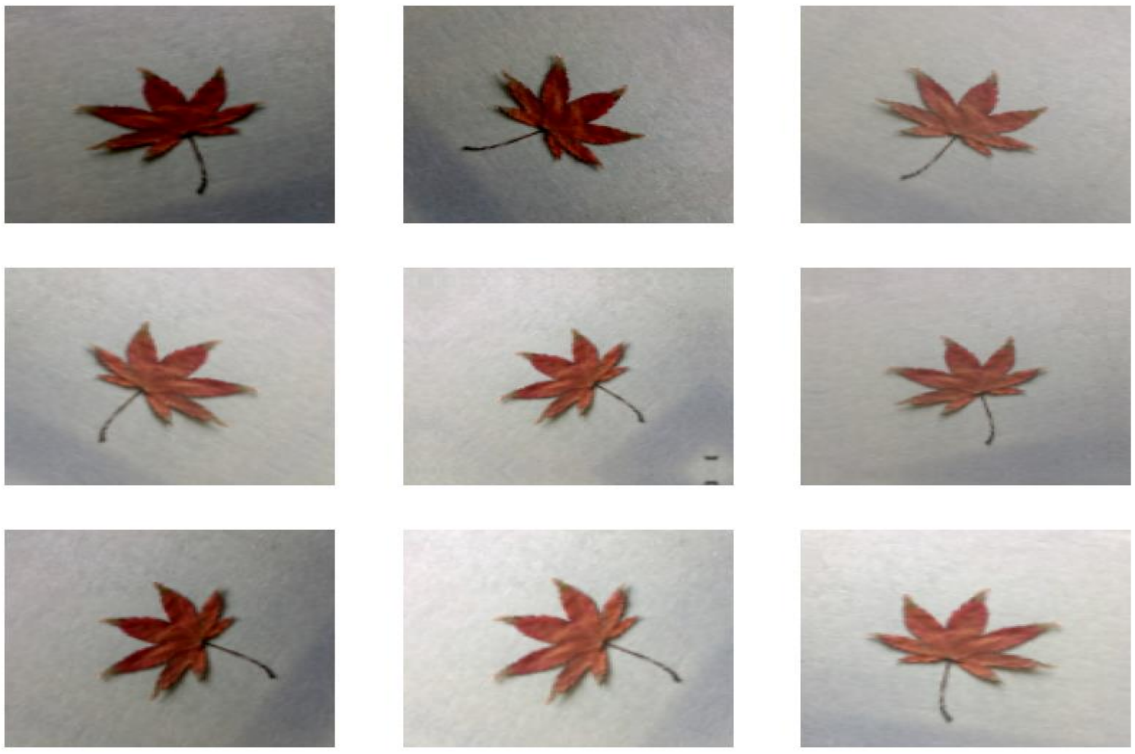
We implement these techniques with this part of Keras code :

```
from keras import layers

data_augmentation_layers = [
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.2),
    layers.RandomZoom(0.2),
    layers.RandomBrightness(0.2),
    layers.RandomContrast(0.2)
]


def data_augmentation(images):
    for layer in data_augmentation_layers:
        images = layer(images)
    return images
```

We randomly take one image from dataset and plot the augmentation results on it :



As you see , one image can convert to many images with some difference in rotation and brightness , but the label is the same . We apply these augmentations on all of our dataset and make a dataset with 5 times more images , including more than 2000 images :

```
new_dataset = []

for i in range(5):

    augmented_train_ds = train_ds.map(
        lambda x, y: (data_augmentation(x), y))

    new_dataset.append(augmented_train_ds)
```

After all the preprocessing parts, we initialize our model . we use MobileNetV2 [1] as our base model and fine tune it with Lion [4] optimizer . base model is followed by some fully connected heads to learn our datasets patterns more precise .

```
def make_model():

    model = keras.Sequential()
    base_model = keras.applications.MobileNetV2(input_shape=(128, 128,
3), include_top=False)
    base_model.trainable = True

    model.add(layers.Input(shape=(128, 128, 3)))
    model.add(base_model)
    model.add(layers.GlobalAveragePooling2D())
    model.add(layers.Dense(256, activation='elu'))
    model.add(layers.Dense(36, activation='softmax'))

    return model

model = make_model()
```

model summary is also available :

Layer (type)	Output Shape	Param #	Trai…
mobilenetv2_1.00_128 (Functional)	(None, 4, 4, 1280)	2,257,984	Y
global_average_pooling2d_6 (GlobalAveragePooling2D)	(None, 1280)	0	-
dense_19 (Dense)	(None, 256)	327,936	Y
dense_20 (Dense)	(None, 36)	9,252	Y

Total params: 2,595,172 (9.90 MB)

Trainable params: 2,561,060 (9.77 MB)

Non-trainable params: 34,112 (133.25 KB)

We Compile our model with sparse categorical cross entropy loss and Lion optimizer with a very small learning rate, hence we don't want to change the weights of MobileNet that much :

```
model.compile(
    optimizer=keras.optimizers.Lion(learning_rate=0.00005),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'],
)
```

Finally, we train our model on our new augmented dataset and save the training histories :

```
histories = []

for dataset in new_dataset:
    history = model.fit(dataset, epochs=20)
    histories.append(history)
```

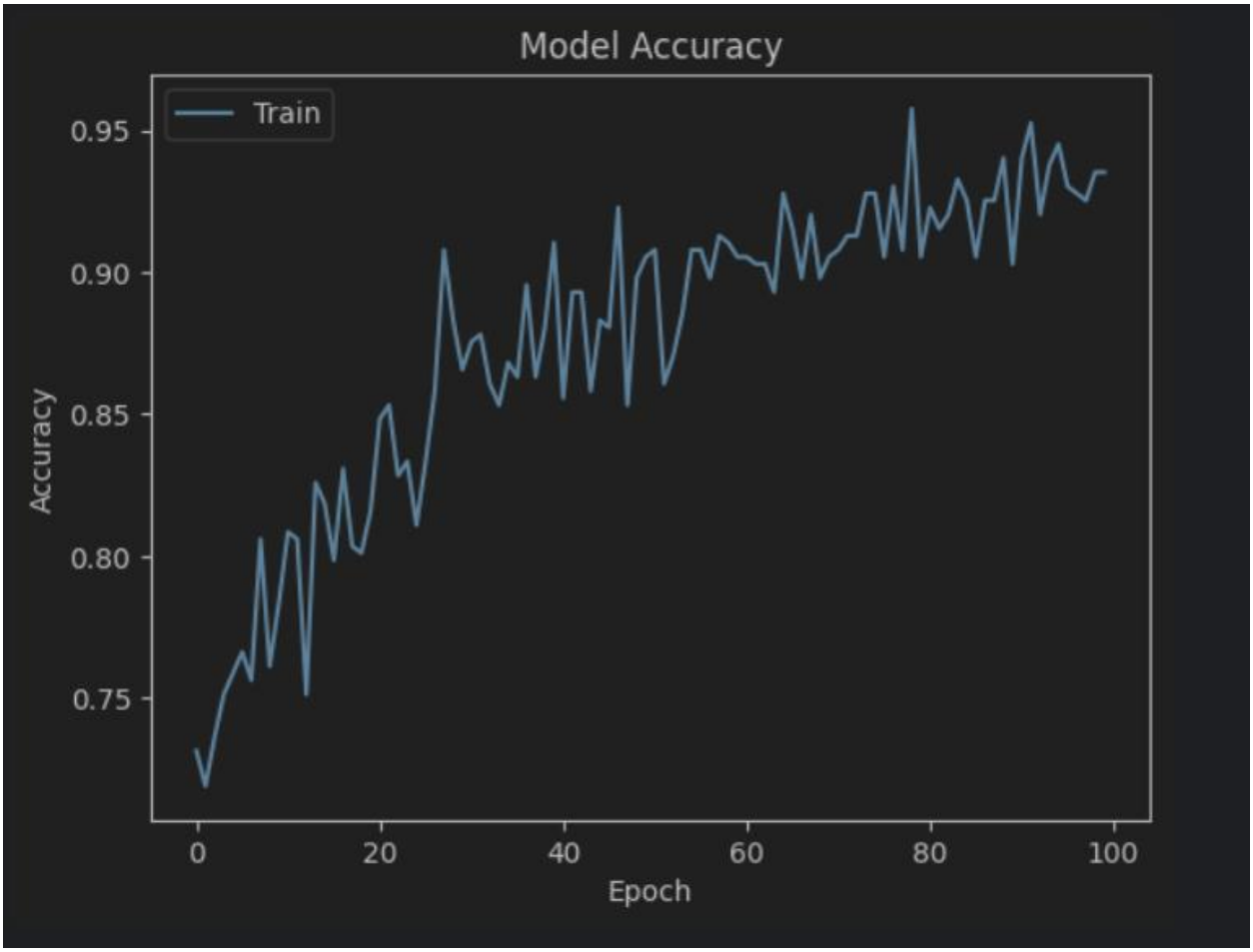
After all , we plot the accuracy results :

```
accuracies = []

for history in histories :
    for accuracy in history.history['accuracy']:
        accuracies.append(accuracy)

plt.plot(accuracies)

plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train'], loc='upper left')
plt.show()
```



Clustering

After classifying our dataset on 3 different models , its time to cluster our leaves but without any labels . We implement K-Means clustering algorithm with different number of centroids and show that without any labels, what happens to out leaves .

Loading our data is same as our previous implementations , but here we implemented some new approach to measure our performance of clustering .

We create a dictionary which maps all leaves with same labels into same target number . Keys of this dictionary contain of target numbers and values of this factionary include the features of leaves related to that target number . we ignore the targets with value 16 to 21 because the lack of them in dataset :

```
t = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11, 12, 13, 14, 15] + [22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34, 35, 36]
dataset_dict = {i : [] for i in t}

for i in range(len(targets)):

    dataset_dict[targets[i]].append(features[i])

for key, value in dataset_dict.items():
    print(f'Class {key} with {len(value)} images ')
```

this code returns this output :

Class 1 with 11 images

Class 2 with 10 images

Class 3 with 10 images

Class 4 with 8 images

Class 5 with 12 images

Class 6 with 8 images

Class 7 with 10 images

Class 8 with 11 images

Class 9 with 14 images

Class 10 with 13 images

Class 11 with 16 images

Class 12 with 12 images

Class 13 with 13 images

Class 14 with 12 images

Class 15 with 10 images

Class 22 with 12 images

Class 23 with 11 images

Class 24 with 13 images

Class 25 with 9 images

Class 26 with 12 images

Class 27 with 11 images

Class 28 with 12 images

Class 29 with 12 images

Class 30 with 12 images

Class 31 with 11 images

Class 32 with 11 images

Class 33 with 11 images

Class 34 with 11 images

Class 35 with 11 images

Class 36 with 10 images

Then we initialize K-Means model and train it with different number of clusters to show the impact of number of centroids on each class distribution . our metrics for evaluation is based on the number of leaves with same cluster centroid in each class . for example , if class 3 has 10 leaves and 7 of them have same cluster assignment , then the fitness of this class is 0.7 . we sum all the classes scores and store them to compare the results :

```python
from sklearn.cluster import KMeans
K = [2, 5, 10, 15, 20, 25, 30,  36]

for k in K :

    kmeans = KMeans(n_clusters=k, random_state=0, n_init=15, max_iter=1000)
    kmeans.fit(features)

    fitness = 0
    for key, value in dataset_dict.items():

        predictions = np.array(kmeans.predict(value))
        # Find frequency of each value
        values, counts = np.unique(predictions, return_counts=True)

        # Display value with highest frequency
        most_frequent_value = values[counts.argmax()]

        perecntage = list(predictions).count(most_frequent_value)/len(value)
        fitness += perecntage

        # If you want to see the progess, uncomment the line bellow
        # print(predictions, '--', most_frequent_value, '--', perecntage)

    print(f'K = {k}, fitness = {fitness}')
```

after training each k-means model with specific number of clusters , its results will be printed as the number of cluster of that model and its fitness :

```
K = 2, fitness = 30.0
K = 5, fitness = 28.687878787878788
K = 10, fitness = 24.41272893772894
K = 15, fitness = 22.23499000999001
K = 20, fitness = 20.801551226551226
K = 25, fitness = 18.87678016428016
K = 30, fitness = 19.142501942501937
K = 36, fitness = 17.291744366744364
```

As it is shown , with increasing the number of cluster centroids , leaves with same label will be assign to different centroids , it means that the distribution of each leaf even with same label as its co-labels are becoming more obvious .

For better explaination , out fitness method works as follows :

1- Calculate the most frequent cluster centroid in each leaf class
2- Find the percentage of leaves assigned to this centroid in that leaf class
3- Sum all of the percentages of each predicted class . this means that if all the leaves in a class are assigned to a same cluster centroid , then the fitness of model is the number of classes (which is 30 here)
4- Show the results and go on for the next model with different number of cluster centroids

Refrences

[1] Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.

[2] Keras Official Documentation

[3] JAX Official Documentation

[4] Chen, Xiangning, et al. "Symbolic discovery of optimization algorithms." *Advances in neural information processing systems* 36 (2024).

[5] SciKit-Learn Official Documentation