CSE485

Deep Learning

Fall 2023

Deep Learning Major Task

Leaf Classification

Submitted to

Dr. Mahmoud Khalil

Eng. Mahmoud Soheil

Submitted by

Zakaria Sobhy AbdelSalam	19p2676
Mahmoud Mohamed Seddik Elnashar	19p3374
Omar Ashraf Mabrouk Abdelelwahab	19P8102

Table of Contents

1. Data Preparation

1.1 Data Exploration

Dataset Size and Structure:

- The dataset consists of 990 entries representing individual leaf samples.
- There are 196 columns in the dataset, with each column representing a different feature.

Features:

- The features are divided into three main categories: leaf margins, leaf shapes, and leaf textures.
- Leaf margin features are represented by columns 'margin1' through 'margin64'.
- Leaf shape features are represented by columns 'shape1' through 'shape64'.
- Leaf texture features are represented by columns 'texture1' through 'texture64'.

Target Variable:

- The 'species' column represents the species of the leaf.
- The 'species_num' column provides numerical labels for each species.

Data Types:

- The dataset includes columns of three different data types: float64, int64, and object.
- Most columns are of float64 type, representing numerical features.
- The 'id' column is of int64 type, and the 'species' and 'category' columns are of object type.

Memory Usage:

- The dataset has a memory usage of approximately 1.5 MB.

Categorical Labels:

- The 'category' column indicates whether a sample belongs to the training or validation set.
- This is not needed as we later divide the data set into training, validation and testing sets with 60%,20%,20% ratios respectively.

1.2 Data Cleaning

Desired Image Size

- To standardize the dataset, leaf images were resized to a common size of 256x256 pixels.

```
# Define your desired image size for resizing
desired_image_size = 256
```

Image Channel

- All images were converted to grayscale, resulting in a single-channel representation.

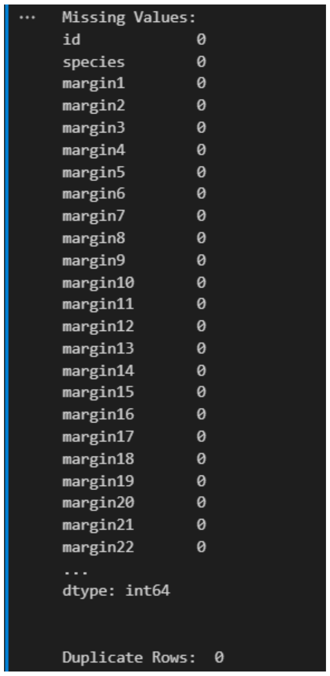
```
# Convert the image to grayscale
        img = Image.open(img_path).convert('L')
        img = self.image_transform(img)
```

1.3 Checking the data for missing values or duplicates

In this section, we ensure the quality of the dataset by examining for missing values and duplicates. The absence of missing values and duplicates is crucial for the reliability of our deep learning model.

```
wholeData.info()
# 3. Check for missing values or duplicates
print("Missing Values:")
print(wholeData.isnull().sum())
print("\n\nDuplicate Rows: ",wholeData.duplicated().sum())
```

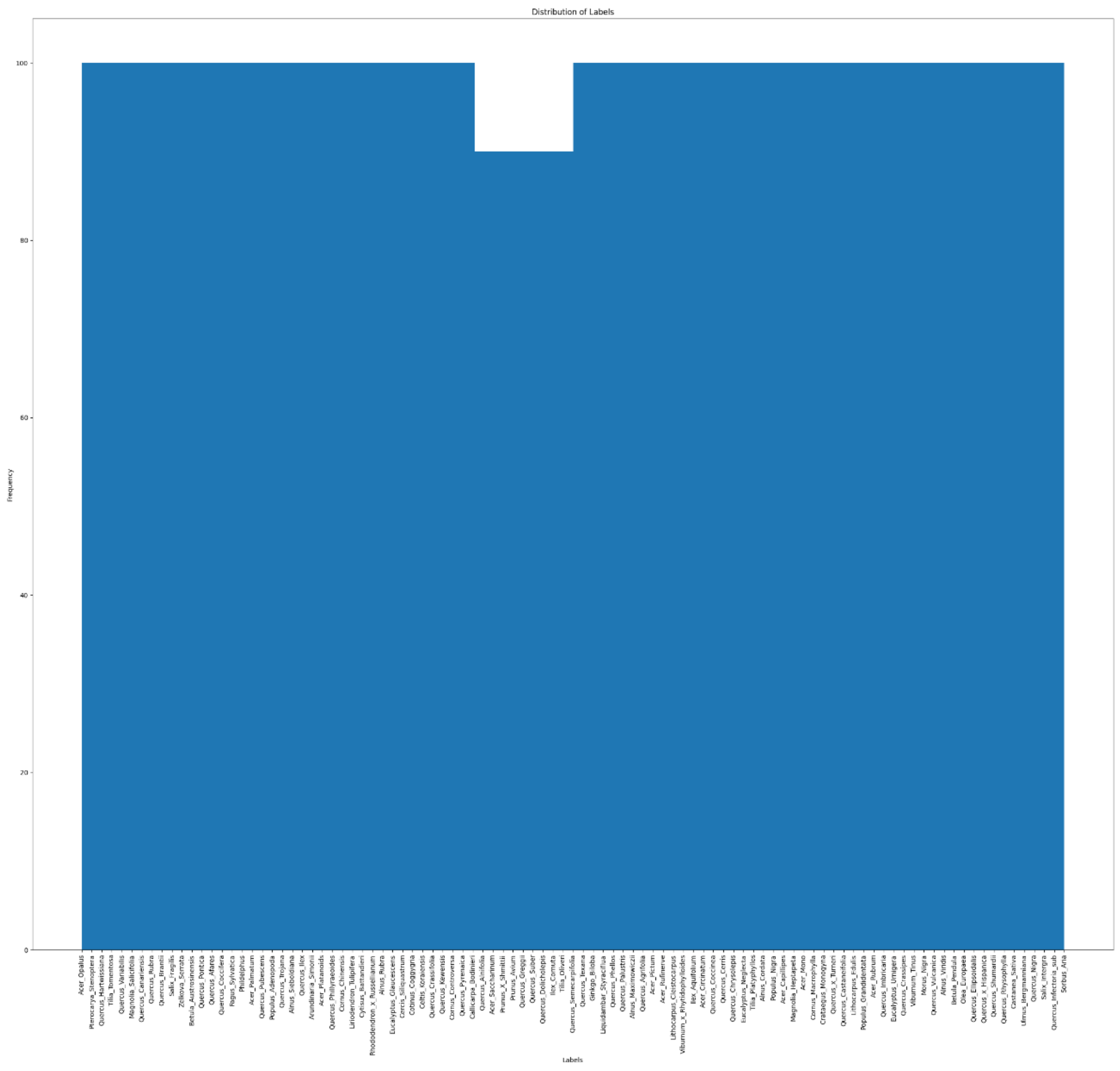
- The output of this code snippet shows that there are no missing values in any of the dataset columns. Each column has a count of 0, indicating a complete dataset without any undefined or absent data.
- The result also demonstrates that there are no duplicate rows in the dataset. Each data point is unique, contributing to the robustness of the dataset.



```
...  Missing Values:
     id          0
     species     0
     margin1     0
     margin2     0
     margin3     0
     margin4     0
     margin5     0
     margin6     0
     margin7     0
     margin8     0
     margin9     0
     margin10    0
     margin11    0
     margin12    0
     margin13    0
     margin14    0
     margin15    0
     margin16    0
     margin17    0
     margin18    0
     margin19    0
     margin20    0
     margin21    0
     margin22    0
     ...
     dtype: int64


     Duplicate Rows:  0
```

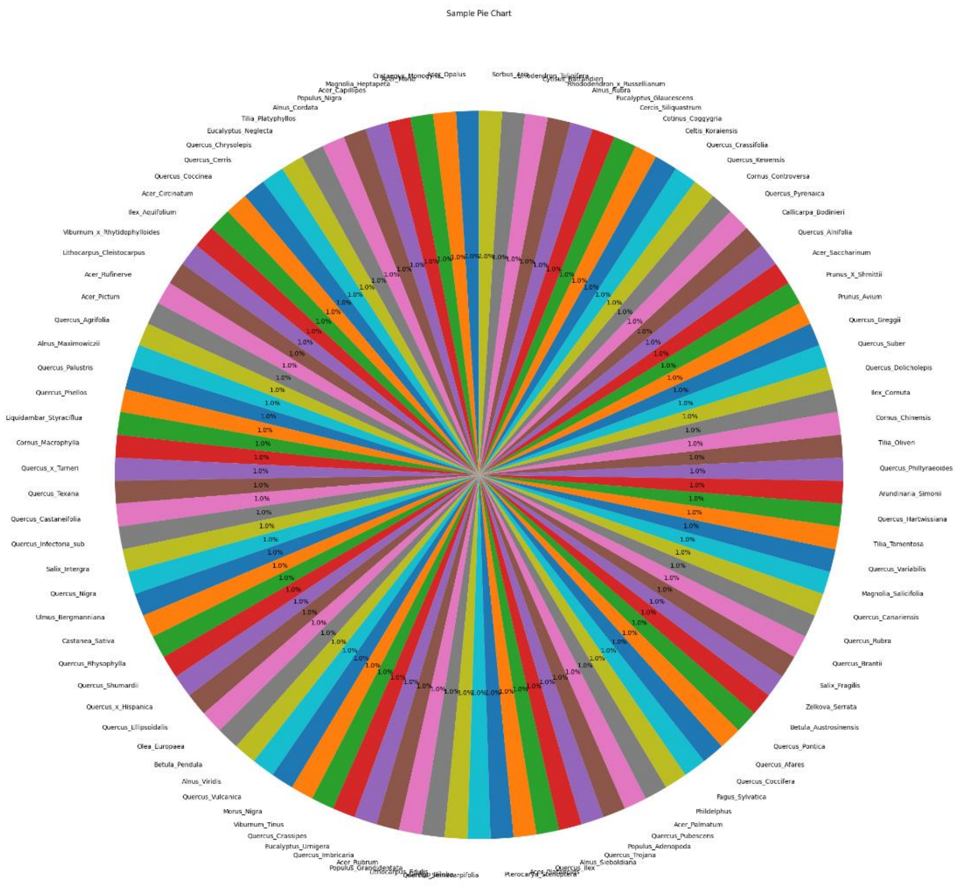
1.4 Visualizing the data using proper visualization methods

1.4.1 Histogram



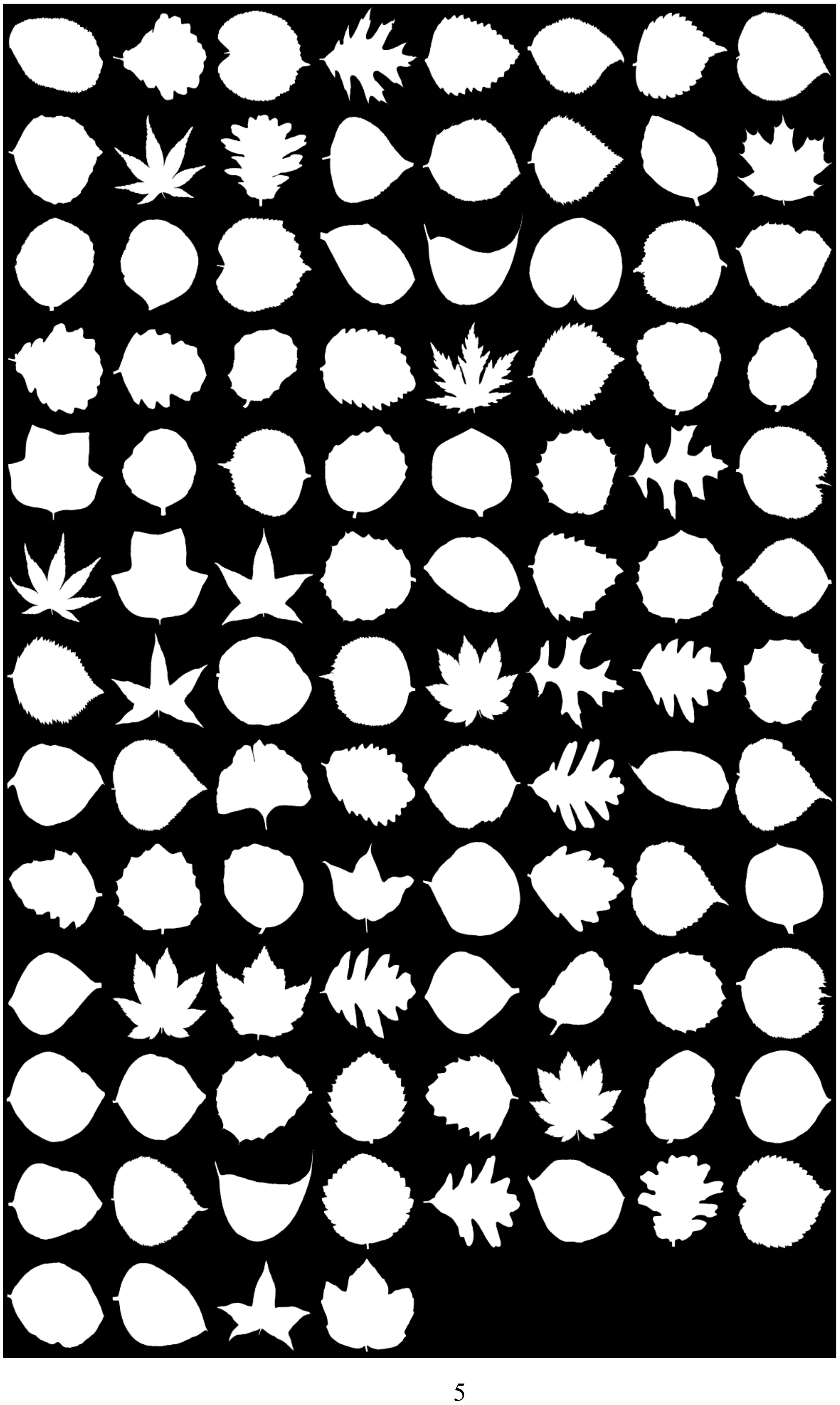
```
plt.figure(figsize=(30, 26))
plt.hist(wholeData['species'])
plt.xlabel('Labels')
plt.xticks(rotation=90)  # Rotating labels for better readability
plt.ylabel('Frequency')
plt.title('Distribution of Labels')
plt.savefig("./deep_data/generated/images/histogram.png")
plt.show()
```

1.4.2 Pie Chart

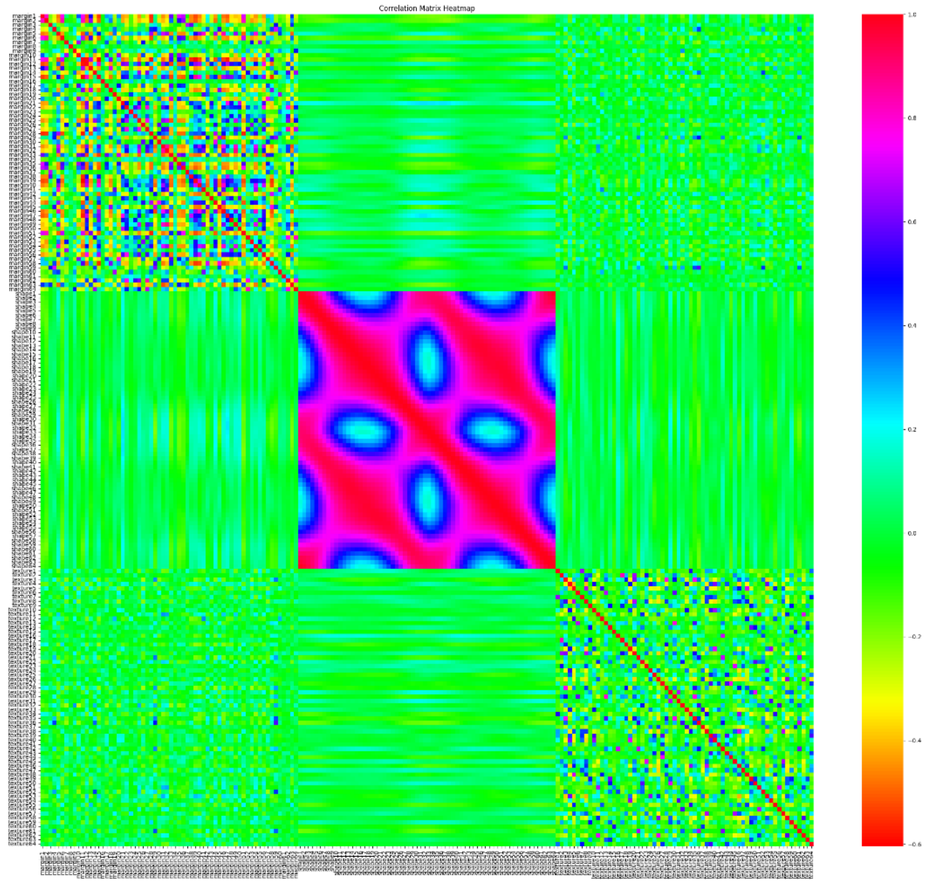


```python
# Create a pie chart
plt.figure(figsize=(30, 26))
# Calculate value counts for the 'species' column
category_counts = wholeData['species'].value_counts()
plt.pie(category_counts, labels=category_counts.index, autopct='%1.1f%%',
startangle=90)
plt.title('Sample Pie Chart')
# Save the pie chart as an image file (e.g., PNG, JPG, etc.)
plt.savefig('./deep_data/generated/images/pie_chart.png')  # Save the chart as
pie_chart.png
plt.show()
```

1.5 Drawing some of the images



1.6 Correlation Analysis



1.7 Dividing the data

- The 60-20-20 split strikes a balance between having enough data for training and having separate datasets for evaluation. A larger training set is beneficial for learning, while separate validation and testing sets help ensure unbiased evaluation.
- In this case of having a small dataset of only 990 samples, having a relatively larger validation and testing set becomes important to obtain more reliable performance estimates.
- If the data set was larger we would have considered 70-15-15 or 80-10-10.

```
target_column = 'species'
if "category" not in wholeData:
    # Splitting data into 80% train(60% train, 20% test) and 20% test
    train_data, test_data = train_test_split(wholeData, test_size=0.2,
stratify=wholeData[target_column], random_state=42)

    # Splitting the remaining 80% into 75% train and 25% validation (20%
validation)
    train_data, validation_data = train_test_split(train_data,
test_size=0.25, stratify=train_data[target_column], random_state=42)

    # Add a new column to each DataFrame indicating the attribute
('train', 'test', 'validate')
    train_data['category'] = 'train'
    test_data['category'] = 'test'
    validation_data['category'] = 'validate'

    # Concatenate the DataFrames together
    combined_df = pd.concat([train_data, test_data, validation_data],
ignore_index=True)

    # Sort the combined DataFrame by 'Attribute' and 'id'
    combined_df = combined_df.sort_values(by=['id'])

    # Write the sorted DataFrame to a CSV file
    try:
        combined_df.to_csv(CSV_FILE, index=False)
        print("Successful split into train_data, test_data,
validation_data\ntrain_data\ntest_data\nvalidation_data")
    except PermissionError as e:
        print("The csv file is opend please close the the file to allow
overwrite.\n"+str(e))
    except Exception as ex:
        # Handling any other exceptions that might occur
        print("An error occurred:", ex)
```

```
else:
    # Split DataFrame based on 'Category'
    train_data = wholeData[wholeData['category'] == 'train']
    test_data = wholeData[wholeData['category'] == 'test']
    validation_data = wholeData[wholeData['category'] == 'validate']
    print("Data was already
split\ntrain_data\ntest_data\nvalidation_data")
```

Check for Existing Categories:

- It checks if the column 'category' already exists in the DataFrame wholeData.

If 'category' Doesn't Exist:

- It splits the data into training (60%), testing (20%), and validation (20%) sets using train_test_split twice.
- Adds a new column 'category' to each DataFrame to indicate whether the row belongs to the training, testing, or validation set.
- Concatenates the DataFrames (train, test, validation) into a new DataFrame combined_df.
- Sorts combined_df by the columns 'Attribute' and 'id'.
- Writes the sorted DataFrame to a CSV file specified by CSV_FILE.

If 'category' Already Exists:

- It assumes that the data has already been split into training, testing, and validation sets based on the 'category' column.
- It extracts the subsets for training, testing, and validation from the existing 'category' column.

1.8 Encoding the labels

This encoding step is crucial, as it converts categorical labels into a format that can be used for numerical computations in the learning process. The numeric representation allows the model to learn relationships and patterns associated with each class during training.

```
      # generate mapping for lables (must be static because this will be
the learnt classification by our network)

target_column = 'species'
if "species_num" not in wholeData:
    # generate a mapping file
    labels_map = dict()
    i = 0
    for category in  wholeData['species'].unique().tolist():
        labels_map[category] = i
        i+=1
    output_map = {value: key for key, value in labels_map.items()}
    with open(LABLE2NUMBER, 'w') as json_file:
        json.dump(labels_map, json_file, indent=4)
    with open(NUMBER2LABLE, 'w') as json_file:
        json.dump(output_map, json_file, indent=4)

    wholeData["species_num"] = wholeData['species'].map(labels_map)
    # Write the sorted DataFrame to a CSV file
    try:
        wholeData.to_csv(CSV_FILE, index=False)
        print("Target classes have numeric mapping now.\nlabels_map :
dict(lable->numericvalue)\noutput_map : dict(numericvalue->lable)")
    except PermissionError as e:
        print("The csv file is opend please close the the file to allow
overwrite.\n"+str(e))
    except Exception as ex:
        # Handling any other exceptions that might occur
        print("An error occurred:", ex)
else:
    with open(LABLE2NUMBER, 'r') as json_file:
        labels_map = json.load(json_file)
    with open(NUMBER2LABLE, 'r') as json_file:
        output_map = json.load(json_file)
    print("Target classes already had numeric mapping.\nlabels_map :
dict(lable->numericvalue)\noutput_map : dict(numericvalue->lable)")
```

Check if Encoding Already Exists:

- The code checks if a column named "species_num" already exists in the DataFrame. If it does, it means that the labels have already been encoded, and the code reads the existing label-to-numeric mapping from JSON files.

Generate Label-to-Numeric Mapping:

- If the "species_num" column doesn't exist, the code generates a label-to-numeric mapping (labels_map) by iterating over unique labels in the "species" column. It assigns a unique numeric value to each label.

Generate Numeric-to-Label Mapping:

- It then creates an output mapping (output_map) for numeric-to-label mapping.

Save Mappings to JSON Files:

- The label-to-numeric mapping (labels_map) is saved to a JSON file (LABLE2NUMBER), and the numeric-to-label mapping (output_map) is saved to another JSON file (NUMBER2LABLE).

Map Labels to Numeric Values:

- The code adds a new column "species_num" to the DataFrame, where each label is replaced with its corresponding numeric value using the generated labels_map.

Save Updated DataFrame to CSV:

- The updated DataFrame with numeric labels is then saved to a CSV file (CSV_FILE).

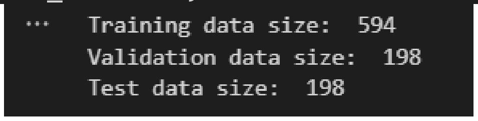
Read Existing Mappings:

- If the encoding already exists, the code reads the label-to-numeric mapping from the existing JSON file (LABLE2NUMBER) and the numeric-to-label mapping from the other JSON file (NUMBER2LABLE).

2. Training a neural network

2.1 Defining a custom data set class

```
class CustomDataset(Dataset):
    def __init__(self, dataframe, images_file_path, desired_image_size,
droped_features_list, external_features_mean, external_features_std_dev):
        self.dataframe = dataframe
        self.images_file_path = images_file_path
        self.desired_image_size = desired_image_size
        self.droped_features_list = droped_features_list
        self.image_transform = transforms.Compose([
            transforms.Resize((desired_image_size, desired_image_size)),
            transforms.ToTensor()
        ])
        self.external_features_transform = transforms.Compose([
            transforms.Normalize(external_features_mean,external_features_
std_dev, inplace=True)
        ])

    def __len__(self):
        return len(self.dataframe)

    def __getitem__(self, idx):
        img_path = self.images_file_path +
str(self.dataframe.iloc[idx]['id'])+'.jpg'
        # Convert the image to grayscale
        img = Image.open(img_path).convert('L')
        img = self.image_transform(img)

        lable =  self.dataframe.iloc[idx]['species_num']
        lable = torch.tensor(lable, dtype=torch.long)

        #convert other features to tensors
        columns_to_exclude = ['id', 'species', 'species_num', 'category']
+ self.droped_features_list
        other_features = self.dataframe.iloc[idx][[col for col in
self.dataframe.columns if col not in
columns_to_exclude]].values.astype(float)
        other_features = torch.tensor(other_features, dtype=torch.float32)

        return img, other_features, lable
```

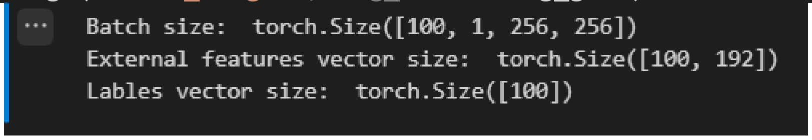


```
...   Training data size:  594
      Validation data size:  198
      Test data size:  198
```

2.2 Declaring dataset and dataloaders for training, validation, testing sets

```
# Define your desired image size for resizing
desired_image_size = 256
batch_size = 100
features_to_drop = []


columns_to_exclude = ['id', 'species', 'species_num', 'category'] +
features_to_drop
external_features_mean = train_data[[col for col in wholeData if col not
in columns_to_exclude]].mean().tolist()
external_features_std_dev = train_data[[col for col in wholeData if col
not in columns_to_exclude]].std().tolist()

# Create datasets and dataloaders
train_dataset = CustomDataset(train_data, IMAGES_FOLDER,
desired_image_size, features_to_drop, external_features_mean,
external_features_std_dev)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=False)

validation_dataset = CustomDataset(validation_data, IMAGES_FOLDER,
desired_image_size, features_to_drop, external_features_mean,
external_features_std_dev)
validation_dataloader = DataLoader(validation_dataset,
batch_size=batch_size, shuffle=False)

test_dataset = CustomDataset(test_data, IMAGES_FOLDER, desired_image_size,
features_to_drop, external_features_mean, external_features_std_dev)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
dataloaders = {'train':train_dataloader,
                'val': validation_dataloader}

dataset_sizes = {'train':len(train_data),
                 'val': len(validation_data)}


# Create an iterator from the DataLoader
train_iter = iter(train_dataloader)
# Get a batch of images without advancing the count
batch_images, external_features, batch_labels = next(train_iter)
# Perform operations with the batch of images
```

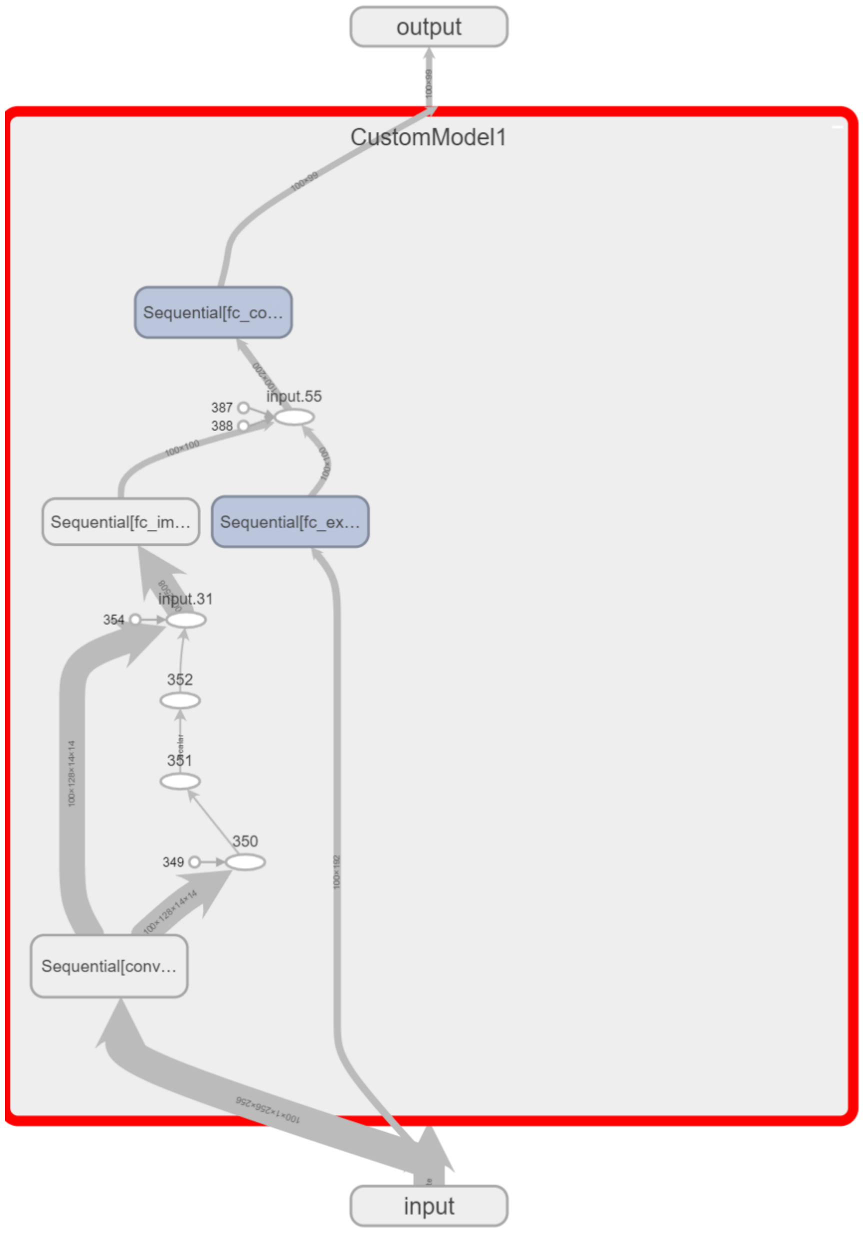
```python
print("Batch size: ", batch_images.size(), "\nExternal features vector
size: ",external_features.size(),
      "\nLables vector size: ", batch_labels.size())

img_grid = torchvision.utils.make_grid(batch_images)
writer.add_image('mnist_images', img_tensor=img_grid)
```



```
Batch size:  torch.Size([100, 1, 256, 256])
External features vector size:  torch.Size([100, 192])
Lables vector size:  torch.Size([100])
```

2.3 Deep Learning Model Architecture

The architecture employs convolutional layers to extract image features and fully connected layers to process both image and external features separately before concatenating them. The final fully connected layers produce the output of size output_size. The use of Batch Normalization, ReLU activation, and Dropout aims to improve the model's generalization and training stability. The model architecture is designed to handle both image and tabular data, making it suitable for multi-modal learning tasks.



2.3.1 Convolutional layers

The convolutional layers are responsible for extracting hierarchical features from the input image.

These layers include:

First Convolutional Layer:

- Input: N x 1 x 256 x 256 (assuming a grayscale image)
- Output: N x 16 x 254 x 254
- Operations: Conv2d (kernel size=3) -> BatchNorm2d -> ReLU -> MaxPool2d (kernel size=2, stride=2)

Second Convolutional Layer:

- Input: N x 16 x 127 x 127
- Output: N x 32 x 125 x 125

Operations: Conv2d (kernel size=3) -> BatchNorm2d -> ReLU -> MaxPool2d (kernel size=2, stride=2)

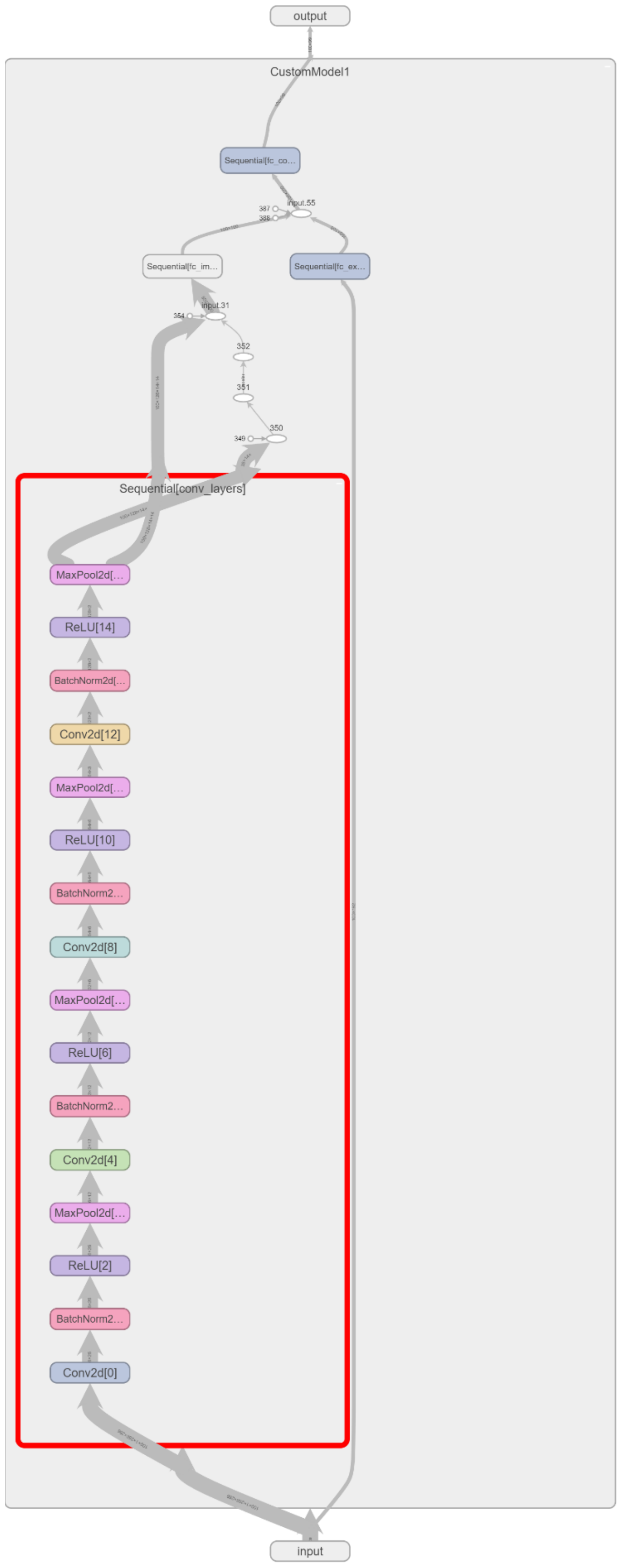
Third Convolutional Layer:

- Input: N x 32 x 62 x 62
- Output: N x 64 x 60 x 60

Operations: Conv2d (kernel size=3) -> BatchNorm2d -> ReLU -> MaxPool2d (kernel size=2, stride=2)

Fourth Convolutional Layer:

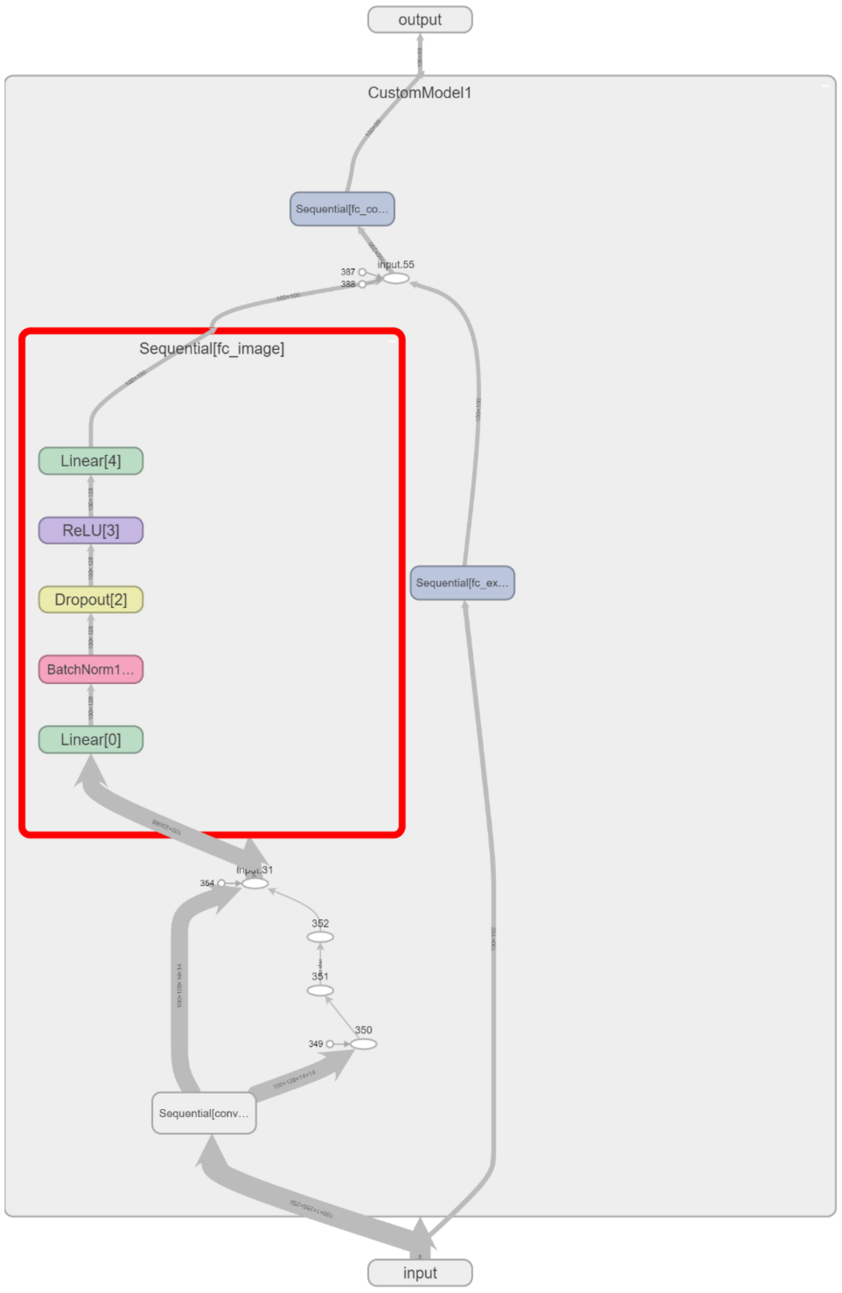
- Input: N x 64 x 30 x 30
- Output: N x 128 x 28 x 28
- Operations: Conv2d (kernel size=3) -> BatchNorm2d -> ReLU -> MaxPool2d (kernel size=2, stride=2)


output
CustomModel1
Sequential[fc_co...
input.55
387
388
Sequential[fc_im...
Sequential[fc_ex...
input.31
354
352
351
350
349
Sequential[conv_layers]
MaxPool2d[...
ReLU[14]
BatchNorm2d[...
Conv2d[12]
MaxPool2d[...
ReLU[10]
BatchNorm2...
Conv2d[8]
MaxPool2d[...
ReLU[6]
BatchNorm2...
Conv2d[4]
MaxPool2d[...
ReLU[2]
BatchNorm2...
Conv2d[0]
input

2.3.2 Image Fully connected Layers

After flattening the tensor from the convolutional layers, there are fully connected layers for processing image features:

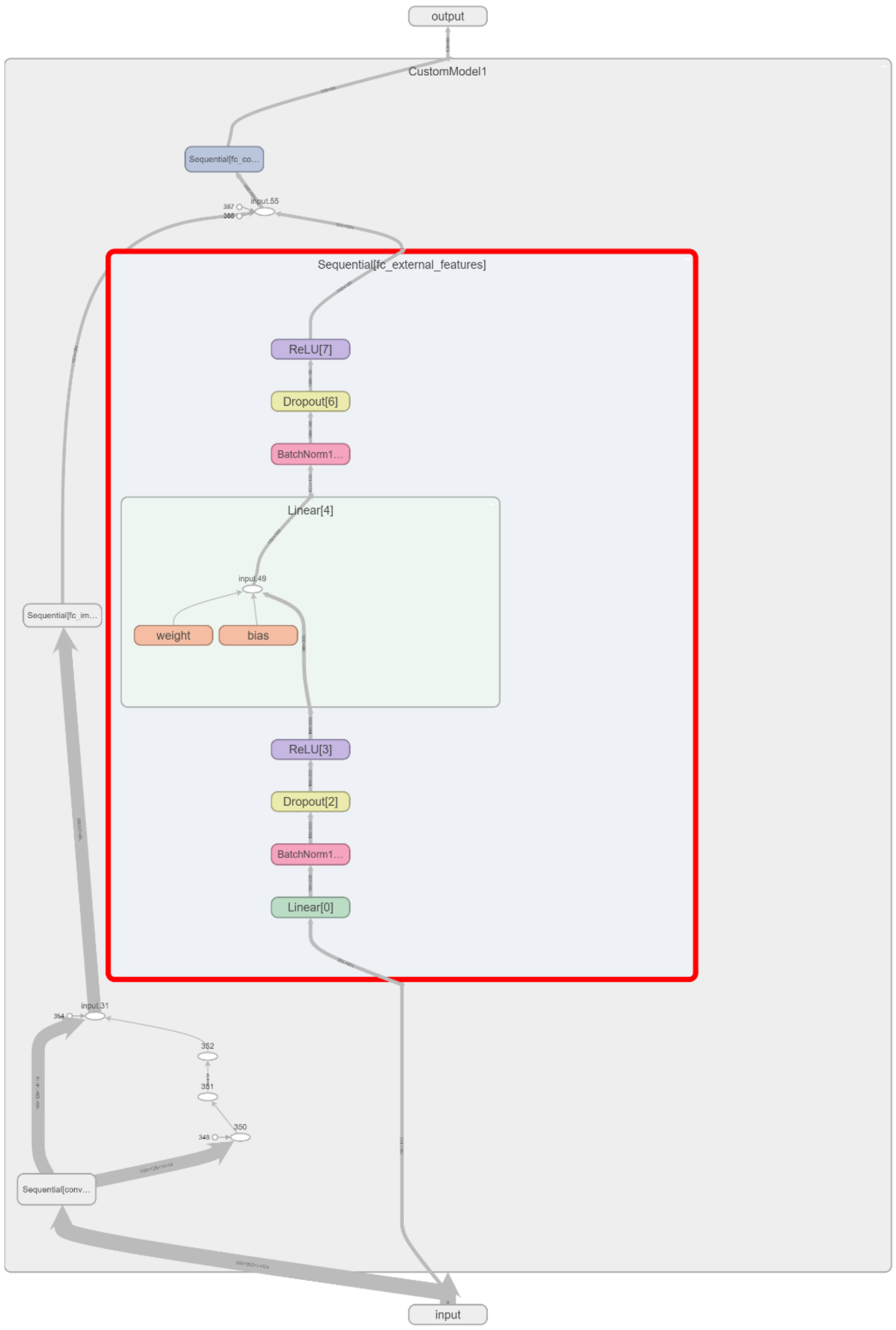
- Input: 128 * 14 * 14
- Operations: Linear (128 * 14 * 14 to 128) -> BatchNorm1d -> Dropout -> ReLU -> Linear (128 to 100)



2.3.3 External features fully connected Layers

These layers process the external features obtained from the CSV file:

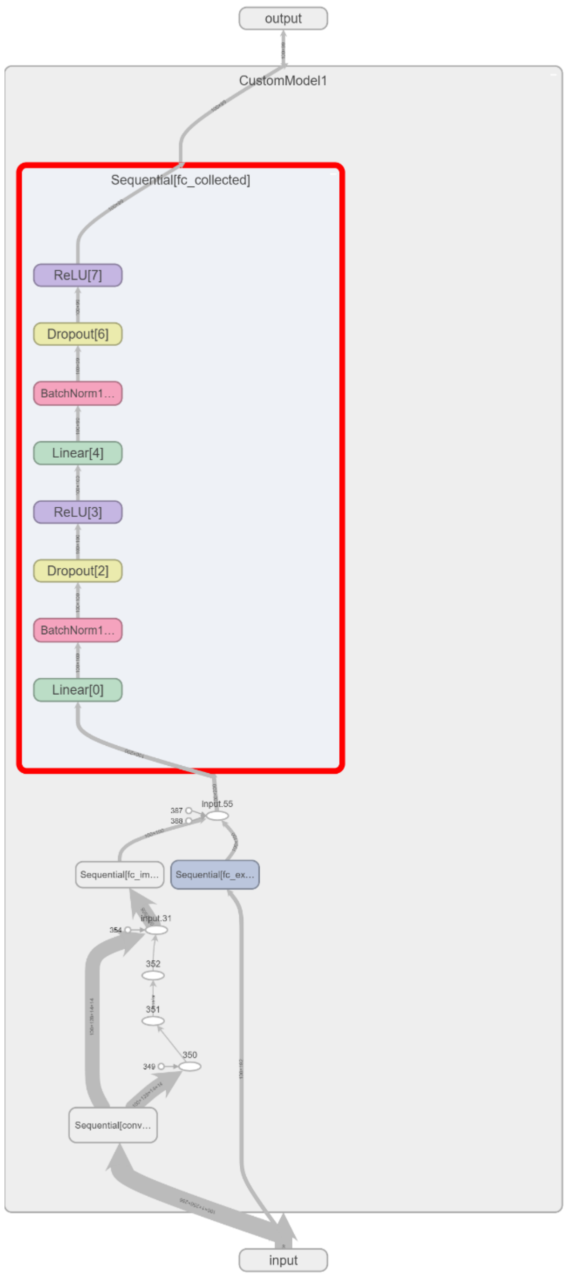
- Input: external_features_size (number of external features)
- Operations: Linear (external_features_size to 200) -> BatchNorm1d -> Dropout -> ReLU -> Linear (200 to 100) -> BatchNorm1d -> Dropout -> ReLU



2.3.4 Concatenation and finaly fully connected layers

The outputs from the image and external features processing are concatenated, and the final fully connected layers process this combined feature representation:

- Input: Concatenation of Image and External Features
- Operations: Linear (combined size to 100) -> BatchNorm1d -> Dropout -> ReLU -> Linear (100 to output_size)



```
class CustomModel1(nn.Module):
    def __init__(self, input_image_size, external_features_size,
output_size, dropout_rate):
        super(CustomModel1,self).__init__()
        self.conv_layers = nn.Sequential(
            # N x 1 x 256 x 256
            nn.Conv2d(input_image_size[0], 16, kernel_size=3), # N x 1 x
256 x 256 ->N x 16 x 254 x 254
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # N x 16 x 254 x 254 ->
N x 16 x 127 x 127

            nn.Conv2d(16, 32, kernel_size=3), # N x 16 x 127 x 127 -> N x
32 x 125 x 125
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # N x 32 x 125 x 125 ->
N x 32 x 62 x 62

            nn.Conv2d(32, 64, kernel_size=3), # N x 32 x 62 x 62 -> N x 64
x 60 x 60
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # N x 64 x 60 x 60 -> N
x 64 x 30 x 30

            nn.Conv2d(64, 128, kernel_size=3), #  N x 64 x 30 x 30 -> N x
128 x 28 x 28
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)  #  N x 128 x 28 x 28->
N x 128 x 14 x 14
        )
        # Calculate the output size after convolutions to define the fully
connected layers
        # Assuming input image size is square
        self.fc_image = nn.Sequential(
            nn.Linear(128*14*14, 128),
            nn.BatchNorm1d(128),
            nn.Dropout(dropout_rate),
            nn.ReLU(),
            nn.Linear(128, 100)
        )
        self.fc_external_features = nn.Sequential(
            nn.Linear(external_features_size, 200),
            nn.BatchNorm1d(200),
            nn.Dropout(dropout_rate),
            nn.ReLU(),

            nn.Linear(200, 100),
            nn.BatchNorm1d(100),
            nn.Dropout(dropout_rate),
            nn.ReLU(),
        )
        self.fc_collected = nn.Sequential(
            nn.Linear(200, 100),
            nn.BatchNorm1d(100),
            nn.Dropout(dropout_rate),
            nn.ReLU(),

            nn.Linear(100, output_size),
            nn.BatchNorm1d(output_size),
            nn.Dropout(dropout_rate),
            nn.ReLU(),
        )


    def forward(self, image, external_features_vector):
        x = self.conv_layers(image)
        x = x.view(x.size(0), -1)  # Flatten the tensor for fully
connected layers
        x = self.fc_image(x)

        x2 = self.fc_external_features(external_features_vector)

        x = torch.cat((x, x2), dim=1)
        x = self.fc_collected(x)
        return x
```

2.4 Definition of training and testing function

2.4.1 Training function

```
def train_model(model, criterion, optimizer, scheduler,dataloaders,
dataset_sizes, setup_description, num_epochs=100, print_frequency=100,
lr_scheduler_require_metric = False, lr_scheduler_no_step= False):
    print("*"*80, end='\n')
    print(setup_description)
    print("*"*80, end='\n')

    since = time.time() # ------------> to get the time taken by training-
---------------

    # ----------------- To save the weights of the model with the best
accuracy----------
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    n_total_steps = len(dataloaders["train"])
    for epoch in range(num_epochs):
        if (epoch+1) % print_frequency == 0:
            print(f'Epoch {epoch}/{num_epochs} ===== Best accuracy
reached: {best_acc*100:.4f}%')
            print('-' * 40)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:

            # 1. ------------------ Set the network mode------------------
---------------
            """
                be aware that some layers have different behavior during
train/evaluation
                (like BatchNorm, Dropout) so setting it matters.
            """
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode
            # ------------------------------------------------------------
---------------

            running_loss = 0.0
            running_corrects = 0
            # Iterate over data.
            for images, external_features, labels in dataloaders[phase]:
                images = images.to(device)
                external_features = external_features.to(device)
                labels = labels.to(device)

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(images, external_features)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        optimizer.zero_grad()
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * images.size(0)
                running_corrects += torch.sum(preds == labels.data)

            # For learning rate scheduling
            if phase == "train":
                if not lr_scheduler_no_step: #skip for CosineAnnealingLR
                    if lr_scheduler_require_metric: #set for
ReduceLROnPlateau
                        scheduler.step(running_loss)
                    else:#set for StepLR
                        scheduler.step()


            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            if (epoch+1) % print_frequency == 0:
                print(f'{phase} Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.4f}'+"%")


            if phase =="train":
                writer.add_scalar("training loss"+setup_description,
epoch_loss, epoch*n_total_steps)
```

```
                writer.add_scalar("training accuracy"+setup_description,
epoch_acc, epoch*n_total_steps)
            elif phase == "val":
                writer.add_scalar("validation loss"+setup_description,
epoch_loss, epoch*n_total_steps)
                writer.add_scalar("validation accuracy"+setup_description,
epoch_acc, epoch*n_total_steps)

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict()) # keep
copy of the best model weights
        if (epoch+1) % print_frequency == 0:
            print("="*80, end='\n')

    time_elapsed = time.time() - since
    print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed %
60:.0f}s')
    print(f'Best val Acc: {best_acc:4f}')
    writer.add_scalar("Best val Acc:"+setup_description, best_acc*100,
epoch*n_total_steps)
    writer.add_scalar("Training time"+setup_description, time_elapsed,
epoch*n_total_steps)

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model
```

2.4.2 Testing function

```
def test_model(model, setup_description, test_dataloader, num_epochs):
    confusion_mat = dict()
    class_labels = []
    class_preds = []
    with torch.no_grad():
        model.eval()   # Set model to evaluate mode
        n_correct = 0
        n_samples = 0
        for images, external_features, labels in test_dataloader:
            images = images.to(device)
            external_features = external_features.to(device)
            labels = labels.to(device)
            outputs = model(images, external_features)

            # value, index
            _, prediction = torch.max(outputs, 1)
            n_samples += labels.shape[0]
            n_correct += (prediction == labels).sum().item()

            # calculate the propability of each class from the output
            class_probs_batch = [torch.nn.functional.softmax(output,
dim=0) for output in outputs]

            class_preds.append(class_probs_batch)
            class_labels.append(labels)

        # 10000, 10, and 10000, 1
        # stack concatenates tensors along a new dimension
        # cat concatenates tensors in the given dimension
        class_preds = torch.cat([torch.stack(batch) for batch in
class_preds])
        class_labels = torch.cat(class_labels)
        accuracy = n_correct/n_samples *100
        print(f"Test accuracy is {accuracy:.2f}%")
        writer.add_scalar("Test accuracy "+ setup_description, accuracy,
num_epochs)
        _, class_preds = torch.max(class_preds, dim=1)
        confusion_mat["actual"]=class_labels
        confusion_mat["prediction"]=class_preds

    plt.figure(figsize=(30, 26))
    confusion_matrix =
metrics.confusion_matrix(confusion_mat["actual"].cpu(),
confusion_mat["prediction"].cpu())

    # Normalize the confusion matrix
    confusion_matrix = confusion_matrix.astype('float') /
confusion_matrix.sum(axis=1)[:, np.newaxis]
    sns.heatmap(confusion_matrix, annot=True, cmap='Blues', fmt='',
annot_kws={"size": 4})

    plt.title('Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('True')

   # Set x-axis and y-axis labels from the list of labels
    lables = output_map.values()
    plt.xticks(np.arange(len(list(lables))) + 0.5, lables, rotation=90)
    plt.yticks(np.arange(len(list(lables))) + 0.5, lables, rotation=0)


    plt.title(f'Confusion Matrix Test accuracy={accuracy:.2f}%')
    File_name = re.sub(r'[-:,]', '', setup_description)
    File_name = re.sub(r'[.]','_',File_name)
    File_name = re.sub(r'[|]',' ',File_name)
    PATH = "./deep_data/cnn_weights/" + File_name + ".pth"
    plt.savefig("./deep_data/generated/images/results/Confusion
Matrix"+File_name+".png")

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

2.5 Training and testing

Hyperparameter Tuning and Training Configuration:

Optimizers:

- Three optimizers are considered: Stochastic Gradient Descent (SGD), Adam, and RMSProp.
- The models are trained using each optimizer to compare their performance.

Learning Rates:

- Three learning rate scheduling strategies are employed:

Step Scheduler:

- Initial learning rate of 0.001.
- The learning rate is reduced by a factor of 0.1 every 20 epochs.

ReduceLROnPlateau:

- Monitors a specified metric (mode='min') and reduces the learning rate by a factor of 0.1 when the metric plateaus.

Cosine Annealing LR:

- Uses a cosine-shaped learning rate annealing with a maximum number of iterations (T_max) set to 20 epochs and a minimum learning rate (eta_min) of 0.

Dropout:

- Two dropout rates are experimented with: 0.2 and 0.8.
- Dropout is applied after certain layers to prevent overfitting during training.
- The dropout rate influences the probability of a neuron being dropped out during forward and backward passes.

Comparison of configurations:

- The 0.8 dropout rate yielded very poor results so the training on it was stopped as it wasn't effective.
- The following comparison is done on the 0.2 dropout rate.

Lr_scheduler/Optimizer	SGD	Adam	RMSprop
Step	Training complete in 10m 11s Best val Acc: 0.297980 Test accuracy is 24.24%	Training complete in 12m 3s Best val Acc: 0.969697 Test accuracy is 93.43%	Training complete in 11m 59s Best val Acc: 0.989899 Test accuracy is 96.97%
ReduceLRonPlateau	Training complete in 10m 1s Best val Acc: 0.303030 Test accuracy is 25.25%	Training complete in 12m 49s Best val Acc: 0.969697 Test accuracy is 92.42%	Training complete in 13m 37s Best val Acc: 0.989899 Test accuracy is 97.47%
Cosine	Training complete in 10m 2s Best val Acc: 0.297980 Test accuracy is 24.75%	Training complete in 13m 20s Best val Acc: 0.969697 Test accuracy is 92.42%	Training complete in 12m 11s Best val Acc: 0.989899 Test accuracy is 96.97%

Training Time:

- The training times vary for different combinations of learning rate schedulers and optimizers.
- The training times are reasonable, with slight variations based on the choice of optimizer and learning rate scheduler.

Validation Accuracy:

- The best validation accuracy varies across different combinations.
- Adam and RMSprop optimizers consistently achieve high validation accuracy, with RMSprop showing slightly better performance.

Test Accuracy:

- The test accuracies are generally high, indicating robust model performance on unseen data.
- Both Adam and RMSprop optimizers demonstrate superior performance, with RMSprop potentially outperforming Adam in terms of test accuracy.
- SGD appears to be less effective in terms of accuracy.

Learning Rate Schedulers:

- Learning rate schedulers influence training dynamics, yet their impact on final accuracy is not as prominent.
- The ReduceLROnPlateau scheduler demonstrates competence in dynamically adjusting the learning rate based on validation performance.

Optimizers:

- Adam and RMSprop optimizers consistently provide great results in terms of both validation and test accuracy.
- RMSprop, in particular, exhibits slightly better performance compared to Adam in this specific experiment.

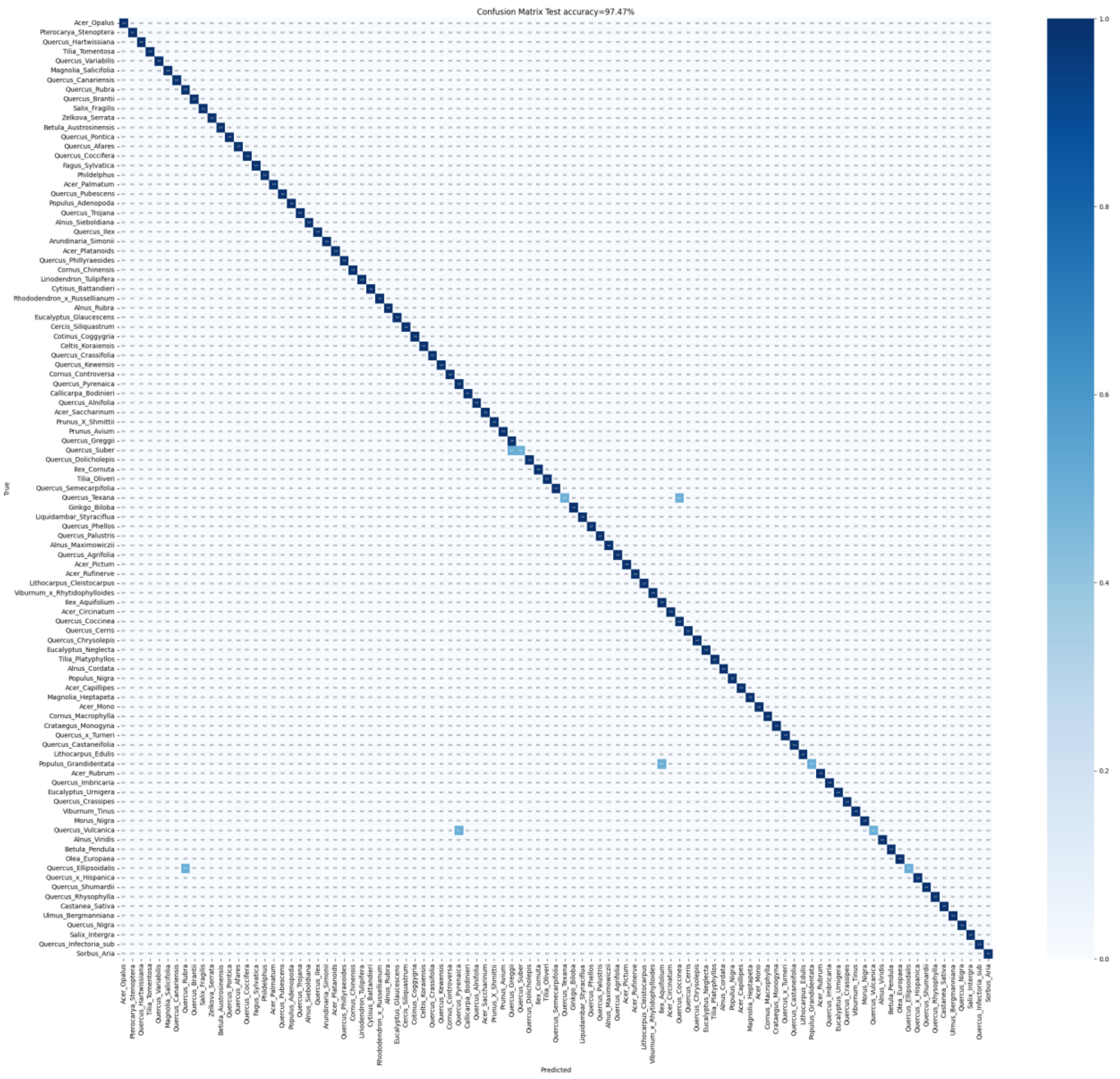
Best Hyper Parameters configuration:

Optimizer:RMSProb

LRScheduler:ReduceLROnPlateau

<u>Confusion matrix of the best result</u>

For all of the confusion matrices refer to the jupyter notebook



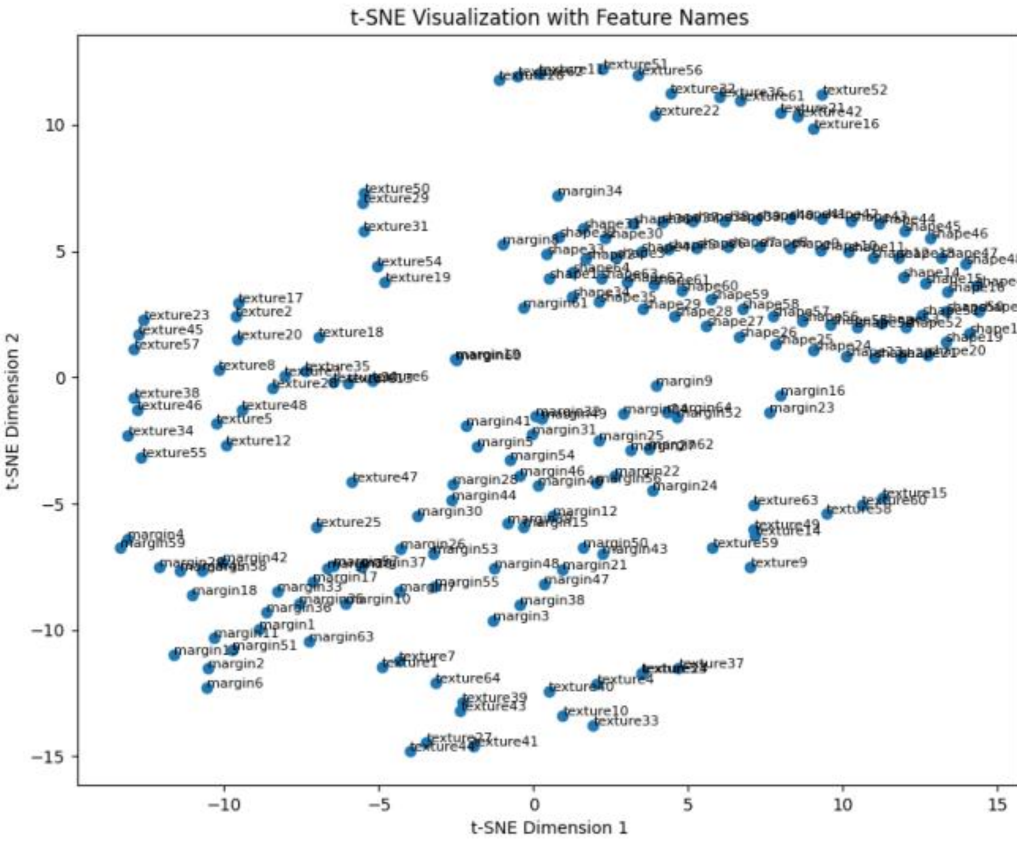
3. Extra part: Feature Engineering (Unsupervised learning)

3.0 Why feature engineering

- Enables the identification of clusters or groups of features that exhibit similar behavior.
- Assists in making informed decisions about which features to include or exclude in the modeling process.

3.1 Apply t-SNE to reduce feature data to 2D data

The code snippet demonstrates the application of t-Distributed Stochastic Neighbor Embedding (t-SNE) for visualizing high-dimensional feature data in a 2D space. This technique is particularly useful when dealing with complex datasets where traditional visualization methods struggle to capture the relationships between features.

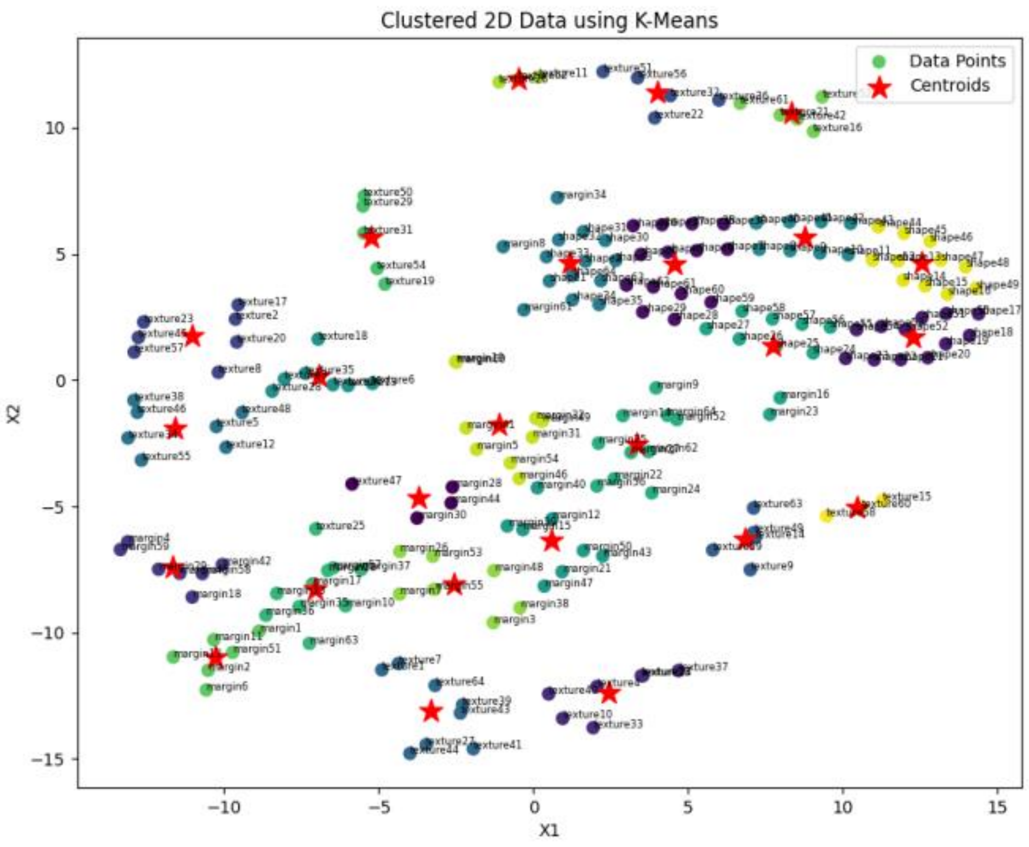


31

```
if FEATURE_EXTRACTION_MODE:
columns_to_exclude = ['id', 'species', 'species_num', 'category']
columns_to_keep =
wholeData.columns[~wholeData.columns.isin(columns_to_exclude)]
external_features = train_data[columns_to_keep]

feature_matrix = external_features.values.T  # Convert DataFrame to NumPy
array
print(feature_matrix.shape)
embedded_data = tsne.fit_transform(feature_matrix)
print(embedded_data.shape)

plt.figure(figsize=(10, 8))  # Adjust figure size as needed

# Plot the scatter plot
plt.scatter(embedded_data[:, 0], embedded_data[:, 1])

# Annotate points with feature names
for i, txt in enumerate(external_features):
plt.annotate(txt, (embedded_data[i, 0], embedded_data[i, 1]), fontsize=8)

plt.title('t-SNE Visualization with Feature Names')
plt.xlabel('t-SNE Dimension 1')
plt.ylabel('t-SNE Dimension 2')
plt.savefig("./deep_data/generated/images/FeatureEngineering/t-SNE.png")
plt.show()
else:
image_path = "./deep_data/generated/images/FeatureEngineering/t-SNE.png"
# Load the image using matplotlib's imread function
img = mpimg.imread(image_path)
# Display the image using imshow
plt.figure(figsize=(10, 8))  # Adjust figure size as needed
plt.imshow(img)
plt.axis('off')  # Turn off axis labels and ticks
plt.show()
```

3.2 Apply kmeans clustering to cluster similar features

applying the K-Means clustering algorithm to group similar features based on their patterns or characteristics. This process can uncover natural groupings within the feature space and enhance the understanding of relationships between features.

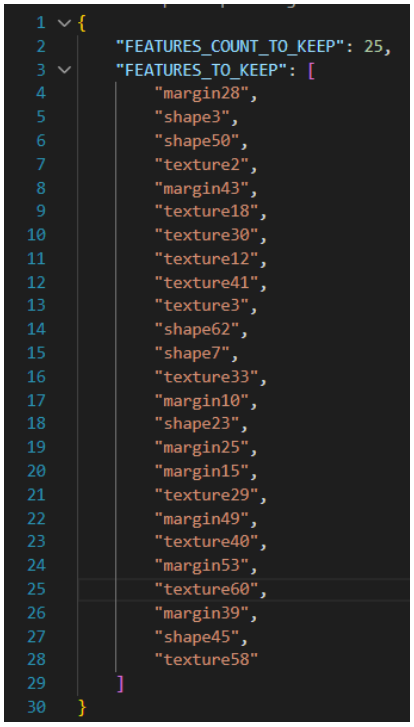


```python
if FEATURE_EXTRACTION_MODE:
    X=embedded_data

    # Apply K-Means clustering
    kmeans = KMeans(n_clusters=FEATURES_COUNT_TO_KEEP)
    kmeans.fit(X)

    # Get cluster labels and centroids
    cluster_labels = kmeans.labels_
    centroids = kmeans.cluster_centers_

    plt.figure(figsize=(10, 8))  # Adjust figure size as needed
    # Visualize the clustered data
    plt.scatter(X[:, 0], X[:, 1], c=cluster_labels, s=50, cmap='viridis', label='Data Points')
    plt.scatter(centroids[:, 0], centroids[:, 1], marker='*', s=200, c='red', label='Centroids')
    # Annotate points with feature names
    for i, txt in enumerate(external_features):
        plt.annotate(txt, (embedded_data[i, 0], embedded_data[i, 1]), fontsize=6)
    plt.title('Clustered 2D Data using K-Means')
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.legend()
    plt.savefig("./deep_data/generated/images/FeatureEngineering/kmeans.png")
    plt.show()
else:
    image_path = "./deep_data/generated/images/FeatureEngineering/kmeans.png"
    # Load the image using matplotlib's imread function
    img = mpimg.imread(image_path)
    # Display the image using imshow
    plt.figure(figsize=(10, 8))  # Adjust figure size as needed
    plt.imshow(img)
    plt.axis('off')  # Turn off axis labels and ticks
    plt.show()
```

3.3 Identifying Features Closest to Centroids in Projected Space

The aim is to find the data points in the original feature space that are closest to each centroid in the t-SNE projected feature space. This process helps in understanding the characteristics of features represented by each cluster.

- The identified features, along with the count to keep, are saved in a structured JSON format (FEATURES_TO_KEEP).



```
{
    "FEATURES_COUNT_TO_KEEP": 25,
    "FEATURES_TO_KEEP": [
        "margin28",
        "shape3",
        "shape50",
        "texture2",
        "margin43",
        "texture18",
        "texture30",
        "texture12",
        "texture41",
        "texture3",
        "shape62",
        "shape7",
        "texture33",
        "margin10",
        "shape23",
        "margin25",
        "margin15",
        "texture29",
        "margin49",
        "texture40",
        "margin53",
        "texture60",
        "margin39",
        "shape45",
        "texture58"
    ]
}
```

```
if FEATURE_EXTRACTION_MODE:
    # Calculate distances between data points and centroids
    distances = cdist(embedded_data, centroids, 'euclidean')
    # Find the closest point to each centroid
    closest_points_indices = np.argmin(distances, axis=0)
    # Get the closest points to each centroid
    closest_points = [embedded_data[closest_points_indices[i]] for i in
range(len(centroids))]
```

```
if FEATURE_EXTRACTION_MODE:
    # Finding indices of points from 'closest_points' in 'embedded_data'
    indices = []
    for point in closest_points:
        index = np.where(np.all(embedded_data == point, axis=1))[0]
        indices.extend(index)
    features_to_keep = list(train_data.columns[indices])
    f_keep = {  "FEATURES_COUNT_TO_KEEP": FEATURES_COUNT_TO_KEEP,
                "FEATURES_TO_KEEP": features_to_keep}
    with open(FEATURES_TO_KEEP, 'w') as json_file:
            json.dump(f_keep, json_file, indent=4)
```