

# Machine Learning Pipeline Documentation

Maciej Wielgosz

January 12, 2024

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Core dataset to be used for experiments</b>	<b>2</b>
<b>3</b>	<b>Data Preparation</b>	<b>3</b>
3.1	Cleaning File Names . . . . .	3
3.2	Preparing Train, Validation, and Test Sets . . . . .	4
<b>4</b>	<b>Data Loading</b>	<b>4</b>
<b>5</b>	<b>Data Visualization</b>	<b>4</b>
<b>6</b>	<b>Model Training</b>	<b>4</b>
<b>7</b>	<b>Model Evaluation</b>	<b>5</b>
<b>8</b>	<b>Confusion Matrix</b>	<b>5</b>
<b>9</b>	<b>Precision, recall and F1</b>	<b>6</b>
<b>10</b>	<b>Model activation visualization</b>	<b>6</b>

# 1 Introduction

This document provides a detailed overview of the machine learning pipeline implemented in the Python script `pipeline/run.py`. The pipeline is designed for a machine learning task and includes several key steps: data preparation, model training, and model evaluation.

## 2 Core dataset to be used for experiments

The basic dataset used is shortly described below.

### 1. Leaves of Broad Leaf Trees

- **Total Images:** 134
- **Types:**
  - Ash (25 images)
  - Beech (30 images)
  - Hornbeam (34 images)
  - Mountain Oak (22 images)
  - Sycamore Maple (23 images)
- **Details:**
  - Image Scale: 800 pixels height or 600 pixels width
  - Note: Ash leaves are compound, specifically pinnate

### 2. Bark of Trees

- **Total Images:** 1183
- **Types:**
  - Ash (34 images)
  - Beech (16 images)
  - Black Pine (166 images, divided into 3 age-based sub-classes)
  - Fir (127 images, divided into 3 age-based sub-classes)
  - Hornbeam (42 images)
  - Larch (200 images, divided into 3 age-based sub-classes)
  - Mountain Oak (77 images)
  - Scots Pine (190 images, divided into 3 age-based sub-classes)
  - Spruce (213 images, divided into 3 age-based sub-classes)
  - Swiss Stone Pine (96 images)
  - Sycamore Maple (22 images)
- **Details:**
  - Image Scale: 800 pixels height or 600 pixels width
  - Age Categories:
    - \* Less than 60 years
    - \* 60 to 80 years
    - \* More than 80 years

### 3. Needles of Conifers

- **Total Images:** 275
- **Types:**
  - Black Pine (107 images)
  - Fir (10 images)
  - Larch (114 images)
  - Scots Pine (10 images)
  - Spruce (13 images)
  - Swiss Stone Pine (21 images)
- **Details:**
  - Needle Classes:
    - \* Separate growth (Fir, Spruce)
    - \* Cluster growth (others)
  - Lighting:
    - \* Perfect conditions (Fir, Scots Pine, Spruce)
    - \* Natural conditions (others)



Figure 1: Black pine needles



Figure 2: Larch needles



Figure 3: Swiss stone pine

## 3 Data Preparation

Data preparation is the first and a critical phase of the machine learning pipeline. It involves organizing and pre-processing the data to make it suitable for training the model.

### 3.1 Cleaning File Names

The script starts by cleaning the file and directory names in the dataset. This step ensures that the file paths are standardized and free from irregularities which might cause errors during data loading.

```
1 from prepare_data.clean_file_names import clean_file_names
2
3 # RAW_DATA_PATH = "/home/nibio/mutable-outside-world/code/ml-department-workshop/ml-
4   department-workshop-dataset/simple-needles-2-class"
5 RAW_DATA_PATH = "/home/nibio/mutable-outside-world/code/ml-department-workshop/ml-
6   department-workshop-dataset/simple-needles-3-class"
7
8 # Clean file and directory names
9 clean_file_names(RAW_DATA_PATH)
```

## 3.2 Preparing Train, Validation, and Test Sets

This section of the script splits the dataset into training, validation, and test sets. Such a split is essential for training the model effectively and evaluating its performance accurately.

```
1 from prepare_data.prepare_train_val_test import PrepareTrainValTest
2 DATA_IN_PATH = RAW_DATA_PATH
3 DATA_OUT_PATH = "/home/nibio/mutable-outside-world/code/ml-department-workshop/datasets/
  data_splited"
4
5 # Create train, validation, and test data sets
6 prepare_data = PrepareTrainValTest(DATA_IN_PATH, DATA_OUT_PATH)
7
8 prepare_data.prepare_train_val_test()
```

## 4 Data Loading

Data loading is a process where the prepared data is loaded into the pipeline in a structured format, which can be easily accessed and manipulated during training and evaluation.

```
1 from pipeline.data_loader import create_data_loaders
2 DATA_PATH = DATA_OUT_PATH
3 BATCH_SIZE = 8
4 NUM_WORKERS = 4
5
6 # Create data loaders
7 train_loader, val_loader, test_loader = create_data_loaders(DATA_PATH, BATCH_SIZE,
  NUM_WORKERS)
```

## 5 Data Visualization

Visualizing data is a key aspect of understanding the dataset. This section shows sample images from the dataset, which helps in getting a visual understanding of the data on which the model will be trained.

```
1 from visualization.show_sample_images import show_sample_images
2
3 # Show sample images
4 show_sample_images(DATA_PATH, 'output_image.png')
```

## 6 Model Training

Model training is the core part of the pipeline. This phase includes setting up the neural network, defining the loss function and optimizer, and iterating over the training dataset to update model weights.

```
1 TRAIN = True
2
3 if TRAIN:
4     # Import necessary packages for training
5     import torch
6     import torch.nn as nn
7     import torch.nn.functional as F
8     import torch.optim as optim
9
10    # Import the model
11    from models.simple_cnn import SimpleCNN
12
13    # Create an instance of the model
14    model = SimpleCNN()
15
16    # Define the loss function and optimizer
17    criterion = nn.CrossEntropyLoss()
18
19    # Use Adam optimizer
20    optimizer = optim.Adam(model.parameters(), lr=0.001)
21
22    # Train the model
23    num_epochs = 5
```

```

24     for epoch in range(num_epochs):
25         running_loss = 0.0
26         for i, data in enumerate(train_loader):
27             # Get the inputs
28             inputs, labels = data
29
30             # Zero the parameter gradients
31             optimizer.zero_grad()
32
33             # Forward + backward + optimize
34             outputs = model(inputs)
35             loss = criterion(outputs, labels)
36             loss.backward()
37             optimizer.step()
38
39             # Print statistics
40             print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, loss))
41
42
43     # save the model
44     torch.save(model.state_dict(), 'simple_cnn.pth')

```

## 7 Model Evaluation

After training, the model is evaluated on a separate test dataset. This stage tests the model's performance and generalization on unseen data, which is crucial for understanding its real-world applicability.

```

1  # load the model
2  import torch
3
4  from models.simple_cnn import SimpleCNN
5
6  model = SimpleCNN()
7  model.load_state_dict(torch.load('simple_cnn.pth'))
8
9  # run the model on the test set and print the accuracy
10 correct = 0
11 total = 0
12
13 with torch.no_grad():
14     for data in test_loader:
15         images, labels = data
16         outputs = model(images)
17         _, predicted = torch.max(outputs.data, dim=1)
18         total += labels.size(0)
19         correct += (predicted == labels).sum().item()
20
21 print('Accuracy of the network on the test images: %d %%' % (100 * correct / total))

```

## 8 Confusion Matrix

The confusion matrix is a tool to visualize the performance of the classification model. It shows the true positive, true negative, false positive, and false negative predictions, providing insights into the type of errors made by the model.

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3  import seaborn as sns
4  from sklearn.metrics import confusion_matrix
5
6  # Get the predictions for the test data
7  y_pred = []
8  y_true = []
9
10 with torch.no_grad():
11     for data in test_loader:
12         images, labels = data
13         outputs = model(images)
14         _, predicted = torch.max(outputs.data, dim=1)

```

```

15     y_pred += predicted.tolist()
16     y_true += labels.tolist()
17
18 # Get the confusion matrix
19 cm = confusion_matrix(y_true, y_pred)
20
21 # Plot the confusion matrix
22 plt.figure(figsize=(10, 10))
23 sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
24 plt.xlabel('Predicted label')
25 plt.ylabel('True label')
26 # save the confusion matrix
27 plt.savefig('confusion_matrix.png')

```

## 9 Precision, recall and F1

Precision, recall, and the F1 score are critical metrics derived from the confusion matrix, a fundamental tool in evaluating the performance of classification models.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 from sklearn.metrics import precision_recall_fscore_support
4
5 # Get the predictions for the test data
6 y_pred = []
7 y_true = []
8
9 with torch.no_grad():
10     for data in test_loader:
11         images, labels = data
12         outputs = model(images)
13         _, predicted = torch.max(outputs.data, dim=1)
14         y_pred += predicted.tolist()
15         y_true += labels.tolist()
16
17 # Get the precision, recall, and F1-score
18 precision, recall, f1_score, _ = precision_recall_fscore_support(y_true, y_pred)
19
20 # Plot the precision, recall, and F1-score as a bar plot
21 plt.figure(figsize=(10, 10))
22 x = np.arange(len(precision))
23 width = 0.2
24 plt.bar(x, precision, width, label='Precision')
25 plt.bar(x + width, recall, width, label='Recall')
26 plt.bar(x + 2 * width, f1_score, width, label='F1-score')
27 plt.xlabel('Class')
28 plt.ylabel('Metric')
29 plt.title('Precision, Recall, and F1-score')
30 plt.xticks(x + width, range(len(precision)))
31 plt.legend()
32 # save the precision, recall, and F1-score
33 plt.savefig('precision_recall_f1_score.png')

```

## 10 Model activation visualization

Activations after several filters in CNN are visualized in this section. To show better how the model works.

```

1 ##### this section shows the activations after each layer of
   the model #####
2
3 import matplotlib.pyplot as plt
4 import os
5
6 def save_activations(activations, save_dir):
7     for name, act in activations.items():
8         num_features = act.size(1)
9         for i in range(num_features):
10             plt.figure()

```

```

11     plt.imshow(act[0, i].detach().numpy(), cmap='hot')
12     plt.axis('off')
13
14     # Save each channel's activation with a proper file name
15     filename = f"{save_dir}/{name}_channel_{i}.png"
16     plt.savefig(filename, bbox_inches='tight', pad_inches=0)
17     plt.close() # Close the plot to free up memory
18
19
20 # Assuming 'images' is a batch of images
21 # And 'model' is an instance of SimpleCNN
22 # create a folder to save the activations
23 save_dir = 'activations'
24 os.makedirs(save_dir, exist_ok=True)
25
26 outputs, activations = model(images, return_activations=True)
27 save_activations(activations, save_dir)

```

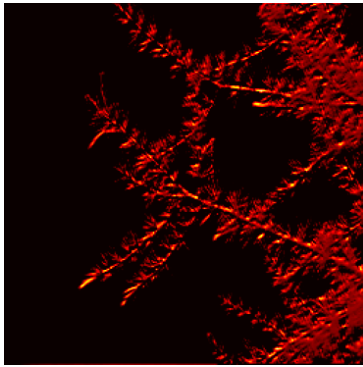


Figure 4: Conv 1 channel 7

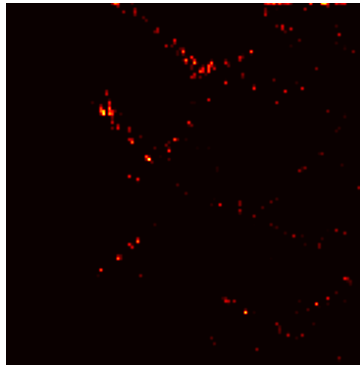


Figure 5: Conv 2 channel 15

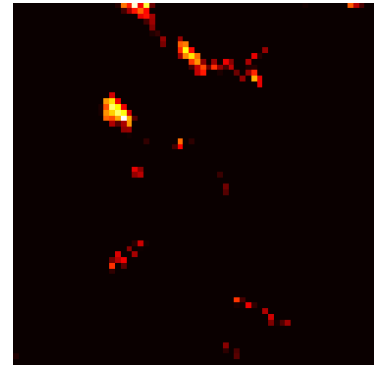


Figure 6: Conv 3 channel 9